HITACHI
Inspire the Next

Nonstop Database

HiRDB Version 9

SQL Reference

3020-6-457-10(E)

■ Relevant program products

List of program products:

For the Red Hat Enterprise Linux AS 4 (AMD64 & Intel EM64T), Red Hat Enterprise Linux ES 4 (AMD64 & Intel EM64T), or Linux 5 (AMD/Intel 64) operating system:

P-9W62-3592 HiRDB Server Version 9 09-01

P-F9W62-11925 HiRDB Non Recover Front End Server Version 9 09-00

P-F9W62-11926 HiRDB Advanced High Availability Version 9 09-00

This edition of the manual is released for the preceding program products, which have been developed under a quality management system that has been certified to comply with ISO9001 and TickIT. This manual may also apply to other program products; for details, see *Before Installing* or *Readme file* (for the UNIX version, see *Software Information* or *Before Installing*).

■ Trademarks

■ Restrictions

■ Issued

Dec. 2011: 3020-6-457-10(E)

■ Copyright

Preface

This manual explains the syntax of SQL statements used to manipulate the databases that you manage with the HiRDB Version 9 Nonstop Database program product.

Intended readers

This manual is intended for users who will be using HiRDB Version 9 (referred to hereafter as *HiRDB*) to design and create tables or to create and execute UAPs.

It is assumed that readers of this manual have the following:

- For Windows systems, a basic knowledge of managing Windows
- For UNIX Systems, a basic knowledge of managing UNIX, or Linux
- Basic knowledge of SQL
- A basic knowledge of programming in C language, COBOL, or Java

This manual is based on the following manuals, which we recommend you read before reading this manual:

- *HiRDB Version 9 Installation and Design Guide*
- *HiRDB Version 9 UAP Development Guide*

Organization of this manual

This manual consists of the following chapters and appendixes:

1. Basics

Chapter 1 explains the basics of SQL.

2. Details of Constituent Elements

Chapter 2 explains in detail the constituent elements of SQL.

3. Definition SQL

Chapter 3 explains the syntax and structure of the definition SQL.

4. Data Manipulation SQL

Chapter 4 explains the syntax and structure of the data manipulation SQL.

5. Control SQL

Chapter 5 explains the syntax and structure of the control SQL.

6. Embedded Language Syntax

Chapter 6 explains the syntax and structure of the embedded language.

7. Routine Control SQL

Chapter 7 explains the syntax and structure of the routine control SQL.

A. Reserved Words

Appendix A provides lists of the HiRDB and SQL reserved words.

B. List of SQLs

Appendix B provides a list of the SQL statements.

C. Example Database

Appendix C explains the example database used in this manual.

Related publications

This manual is related to the following manuals, which should be read as required.

HiRDB (for UNIX)

- *For UNIX Systems HiRDB Version 9 Description* (3000-6-451)#
- *For UNIX Systems HiRDB Version 9 Installation and Design Guide* (3000-6-452(E))
- *For UNIX Systems HiRDB Version 9 System Definition* (3000-6-453(E))
- *For UNIX Systems HiRDB Version 9 System Operation Guide* (3000-6-454(E))
- *For UNIX Systems HiRDB Version 9 Command Reference* (3000-6-455(E))
- *For UNIX Systems HiRDB Version 9 Staticizer Option Description and User's Guide* (3000-6-463)#
- *For UNIX Systems HiRDB Version 9 Disaster Recovery System Configuration and Operation Guide* (3000-6-464(E))
- *For UNIX Systems HiRDB Version 9 Memory Database Installation and Operation Guide* (3020-6-469)#
- *For UNIX Systems HiRDB First Step Guide* (3000-6-254)#

HiRDB (for both Windows and UNIX)

- *HiRDB Version 9 UAP Development Guide* (3020-6-456(E))
- *HiRDB Version 9 Messages* (3020-6-458(E))
- *HiRDB Version 9 XDM/RD E2 Connection Facility* (3020-6-465)#

- *HiRDB Version 9 Batch Job Accelerator* (3020-6-468)#
- *HiRDB Version 9 XML Extension* (3020-6-480)#
- *HiRDB Version 9 Text Search Plug-in* (3020-6-481)#
- *HiRDB Version 8 Security Guide* (3020-6-359)#
- *HiRDB Datareplicator Version 8 Description, User's Guide and Operator's Guide* (3020-6-360(E))
- *HiRDB Datareplicator Extension Version 8* (3020-6-361)#
- *HiRDB Dataextractor Version 8 Description, User's Guide and Operator's Guide* (3020-6-362(E))

In references to HiRDB Version 9 manuals, this manual omits the phrases *for UNIX systems* and *for Windows systems*. Refer to either the UNIX or Windows HiRDB manual, whichever is appropriate for your platform.

For related products

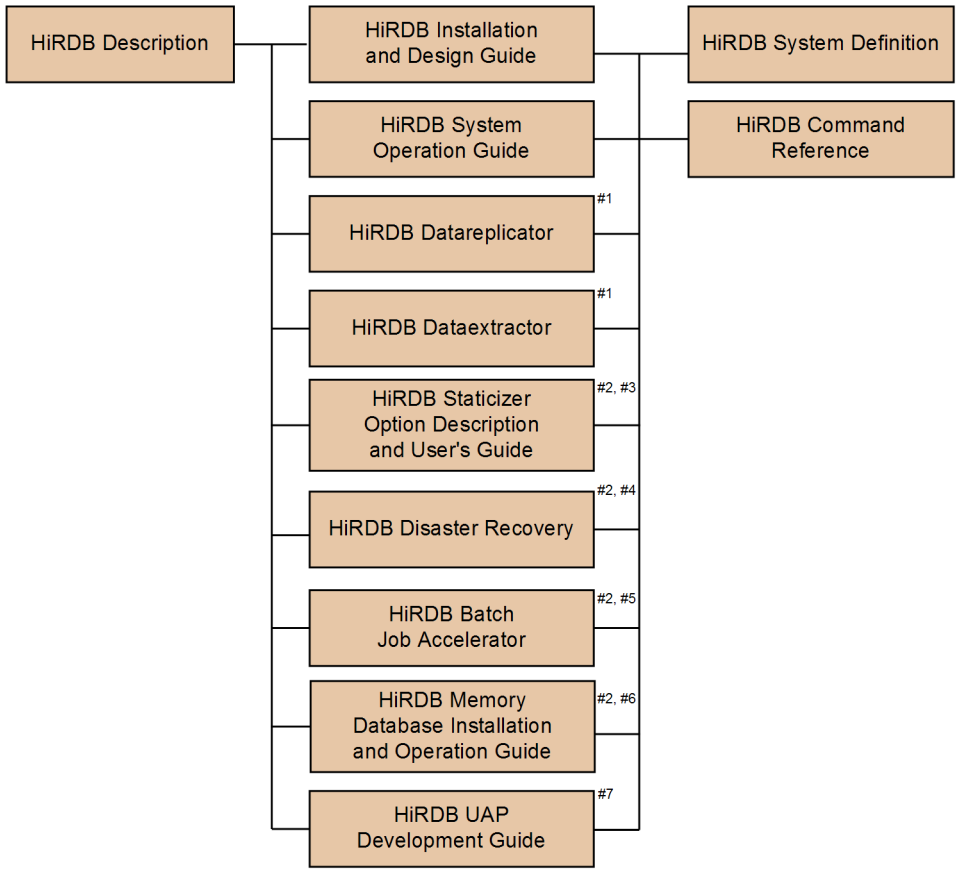
- *VOS3 XDM/RD E2 SQL Reference* (6190-6-656)#

#: This manual has been published in Japanese only; it is not available in English.

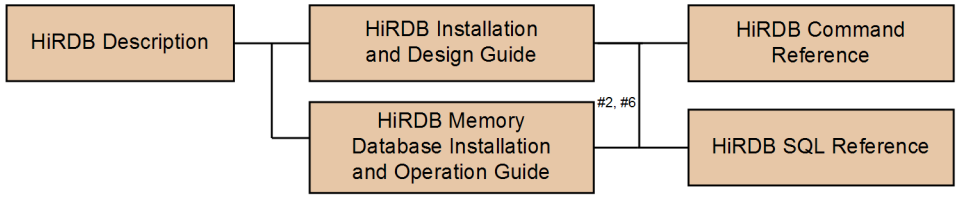
Organization of HiRDB manuals

The HiRDB manuals are organized as shown below. For the most efficient use of these manuals, it is suggested that they be read in the order they are shown, going from left to right.

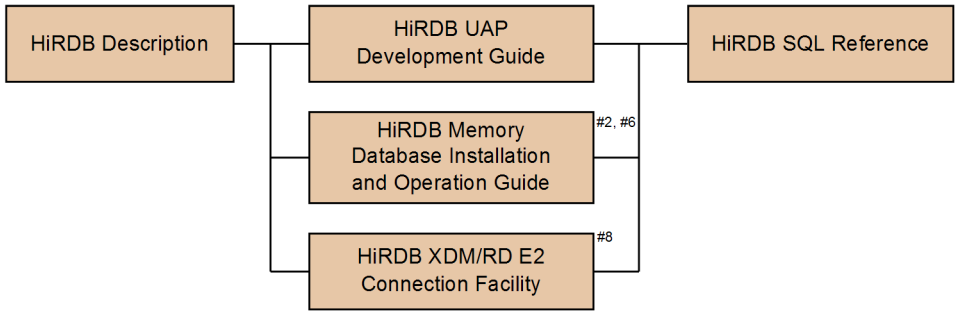
For system administrators:



For users who create tables:



For users who create or execute UAPs:



#1: Read if you intend to use the replication facility to link data.
#2: Published for UNIX only. There is no corresponding Windows manual.
#3: Read if you intend to use the inner replica facility.
#4: Read if you intend to configure a disaster recovery system.
#5: Read if you intend to use in-memory data processing to accelerate batch operations.
#6: Read if you intend to use the memory database facility.
#7: Read if you intend to link HiRDB to an OLTP system.
#8: Read if you intend to use the XDM/RD E2 connection facility to perform operations on XDM/RD E2 databases.

Conventions: Abbreviations for product names

This manual uses the following abbreviations for product names:

<table>
<tr><th>Full name or meaning</th><th colspan="2">Abbreviation</th></tr>
<tr><td rowspan="2">HiRDB Server Version 9</td><td>HiRDB/Single Server</td><td rowspan="2">HiRDB or HiRDB Server</td></tr>
<tr><td>HiRDB/Parallel Server</td></tr>
<tr><td>HiRDB/Developer's Kit Version 9</td><td rowspan="2">HiRDB/ Developer's Kit</td><td rowspan="4">HiRDB Client</td></tr>
<tr><td>HiRDB/Developer's Kit Version 9 (64)</td></tr>
<tr><td>HiRDB/Run Time Version 9</td><td rowspan="2">HiRDB/Run Time</td></tr>
<tr><td>HiRDB/Run Time Version 9 (64)</td></tr>
<tr><td>HiRDB Advanced High Availability Version 9</td><td colspan="2">HiRDB Advanced High Availability</td></tr>
<tr><td>HiRDB Accelerator Version 8</td><td colspan="2" rowspan="2">HiRDB Accelerator</td></tr>
<tr><td>HiRDB Accelerator Version 9</td></tr>
<tr><td>HiRDB Non Recover Front End Server Version 9</td><td colspan="2">HiRDB Non Recover FES</td></tr>
</table>

Full name or meaning	Abbreviation
HiRDB Staticizer Option Version 9	HiRDB Staticizer Option
HiRDB Disaster Recovery Light Edition Version 9	HiRDB Disaster Recovery Light Edition
HiRDB Text Search Plug-in Version 9	HiRDB Text Search Plug-in
HiRDB XML Extension Version 9	HiRDB XML Extension
HiRDB Datareplicator Version 8	HiRDB Datareplicator
HiRDB Dataextractor Version 8	HiRDB Dataextractor
HiRDB Adapter for XML - Standard Edition	HiRDB Adapter for XML
HiRDB Adapter for XML - Enterprise Edition	
HiRDB Control Manager	HiRDB CM
HiRDB Control Manager Agent	HiRDB CM Agent
Hitachi TrueCopy	TrueCopy
Hitachi TrueCopy basic	
TrueCopy	
TrueCopy remote replicator	
JP1/Automatic Job Management System 3	JP1/AJS3
JP1/Automatic Job Management System 2	
JP1/Automatic Job Management System 2 - Scenario Operation	JP1/AJS2-SO
JP1/Cm2/Extensible SNMP Agent	JP1/ESA
JP1/Cm2/Extensible SNMP Agent for Mib Runtime	
JP1/Cm2/Network Node Manager	JP1/NNM
JP1/Integrated Management - Manager	JP1/Integrated Management or JP1/IM
JP1/Integrated Management - View	
JP1/Magnetic Tape Access	EasyMT
EasyMT	
JP1/Magnetic Tape Library	MTguide
JP1/NETM/Audit - Manager	JP1/NETM/Audit

Full name or meaning	Abbreviation	
JP1/NETM/DM	JP1/NETM/DM	
JP1/NETM/DM Manager		
JP1/Performance Management	JP1/PFM	
JP1/Performance Management - Agent Option for HiRDB	JP1/PFM-Agent for HiRDB	
JP1/Performance Management - Agent Option for Platform	JP1/PFM-Agent for Platform	
JP1/Performance Management/SNMP System Observer	JP1/SSO	
JP1/VERITAS NetBackup BS v4.5	NetBackup	
JP1/VERITAS NetBackup v4.5		
JP1/VERITAS NetBackup BS V4.5 Agent for HiRDB License	JP1/VERITAS NetBackup Agent for HiRDB License	
JP1/VERITAS NetBackup V4.5 Agent for HiRDB License		
JP1/VERITAS NetBackup 5 Agent for HiRDB License		
OpenTP1/Server Base Enterprise Option	TP1/EE	
Virtual-storage Operating System 3/Forefront System Product	VOS3/FS	VOS3
Virtual-storage Operating System 3/Leading System Product	VOS3/LS	
Virtual-storage Operating System 3/Unific System Product	VOS3/US	
Extensible Data Manager/Base Extended Version 2 XDM Basic Program XDM/BASE E2	XDM/BASE E2	
XDM/Data Communication and Control Manager 3 XDM Data Communication Management System XDM/DCCM3	XDM/DCCM3	
XDM/Relational Database Relational Database System XDM/RD	XDM/RD	XDM/RD
XDM/Relational Database Extended Version 2 Relational Database System XDM/RD E2	XDM/RD E2	
VOS3 Database Connection Server	DB Connection Server	
Oracle WebLogic Server	WebLogic Server	
DB2 Universal Database for OS/390 Version 6	DB2	
DNCWARE ClusterPerfect (Linux Edition)	ClusterPerfect	
Java™	Java	

Full name or meaning	Abbreviation	
Microsoft(R) Office Excel	Microsoft Excel or Excel	
Microsoft(R) Visual C++(R)	Visual C++ or C++ language	
PowerHA for AIX, V5.5	PowerHA	
PowerHA SystemMirror V6.1		
HP-UX 11i V2 (IPF)	HP-UX or HP-UX (IPF)	
HP-UX 11i V3 (IPF)		
AIX 5L V5.2	AIX 5L	AIX
AIX 5L V5.3		
AIX V6.1	AIX V6.1	
Linux(R)	Linux	
Red Hat Enterprise Linux AS 4 (AMD64 & Intel EM64T)	Linux AS 4	Linux
Red Hat Enterprise Linux AS 4 (x86)		
Red Hat Enterprise Linux ES 4 (AMD64 & Intel EM64T)	Linux ES 4	
Red Hat Enterprise Linux ES 4 (x86)		
Red Hat Enterprise Linux 5.1 Advanced Platform (x86)	Linux 5.1	
Red Hat Enterprise Linux 5.1 (x86)		
Red Hat Enterprise Linux 5.1 Advanced Platform (AMD/Intel 64)		
Red Hat Enterprise Linux 5.1 (AMD/Intel 64)		
Red Hat Enterprise Linux 5.2 Advanced Platform (AMD/Intel 64)	Linux 5.2	
Red Hat Enterprise Linux 5.2 (AMD/Intel 64)		
Red Hat Enterprise Linux 5.3 Advanced Platform (AMD/Intel 64)	Linux 5.3	
Red Hat Enterprise Linux 5.3 (AMD/Intel 64)		
Red Hat Enterprise Linux 5.4 Advanced Platform (AMD/Intel 64)	Linux 5.4	
Red Hat Enterprise Linux 5.4 (AMD/Intel 64)		
Red Hat Enterprise Linux AS 4 (AMD64 & Intel EM64T)	Linux (EM64T)	
Red Hat Enterprise Linux ES 4 (AMD64 & Intel EM64T)		
Red Hat Enterprise Linux 5.1 Advanced Platform (AMD/Intel 64)		

Full name or meaning	Abbreviation	
Red Hat Enterprise Linux 5.1 (AMD/Intel 64)		
Red Hat Enterprise Linux 5.2 Advanced Platform (AMD/Intel 64)		
Red Hat Enterprise Linux 5.2 (AMD/Intel 64)		
Red Hat Enterprise Linux 5.3 Advanced Platform (AMD/Intel 64)		
Red Hat Enterprise Linux 5.3 (AMD/Intel 64)		
Red Hat Enterprise Linux 5.4 Advanced Platform (AMD/Intel 64)		
Red Hat Enterprise Linux 5.4 (AMD/Intel 64)		
Red Hat Enterprise Linux 5.1 Advanced Platform (x86)	Linux 5 (x86)	Linux 5
Red Hat Enterprise Linux 5.1 (x86)		
Red Hat Enterprise Linux 5.1 Advanced Platform (AMD/Intel 64)	Linux 5 (AMD/ Intel 64)	
Red Hat Enterprise Linux 5.1 (AMD/Intel 64)		
Red Hat Enterprise Linux 5.2 Advanced Platform (AMD/Intel 64)		
Red Hat Enterprise Linux 5.2 (AMD/Intel 64)		
Red Hat Enterprise Linux 5.3 Advanced Platform (AMD/Intel 64)		
Red Hat Enterprise Linux 5.3 (AMD/Intel 64)		
Red Hat Enterprise Linux 5.4 Advanced Platform (AMD/Intel 64)		
Red Hat Enterprise Linux 5.4 (AMD/Intel 64)		
turbolinux 7 Server for AP8000	Linux for AP8000	
Microsoft(R) Windows NT(R) Workstation Operating System Version 4.0	Windows NT	
Microsoft(R) Windows NT(R) Server Network Operating System Version 4.0		
Microsoft(R) Windows(R) 2000 Professional Operating System	Windows 2000	
Microsoft(R) Windows(R) 2000 Server Operating System		
Microsoft(R) Windows(R) 2000 Datacenter Server Operating System		
Microsoft(R) Windows(R) 2000 Advanced Server Operating System		
Microsoft(R) Windows(R) 2000 Advanced Server Operating System	Windows 2000 Advanced Server	

Full name or meaning	Abbreviation	
Microsoft(R) Windows Server(R) 2003, Standard Edition	Windows Server 2003 Standard Edition	Windows Server 2003
Microsoft(R) Windows Server(R) 2003, Enterprise Edition	Windows Server 2003 Enterprise Edition	
Microsoft(R) Windows Server(R) 2003, Standard x64 Edition	Windows Server 2003 Standard x64 Edition	
Microsoft(R) Windows Server(R) 2003, Enterprise x64 Edition	Windows Server 2003 Enterprise x64 Edition	
Microsoft(R) Windows Server(R) 2003 R2, Standard Edition	Windows Server 2003 R2	
Microsoft(R) Windows Server(R) 2003 R2, Enterprise Edition		
Microsoft(R) Windows Server(R) 2003 R2, Standard x64 Edition		
Microsoft(R) Windows Server(R) 2003 R2, Enterprise x64 Edition		
Microsoft(R) Windows Server(R) 2003 R2, Standard x64 Edition	Windows Server 2003 R2 x64 Editions	
Microsoft(R) Windows Server(R) 2003 R2, Enterprise x64 Edition		
Microsoft(R) Windows Server(R) 2003, Enterprise Edition (64-bit version)	Windows Server 2003 (IPF)	
Microsoft(R) Windows Server(R) 2008 Standard	Windows Server 2008 Standard	Windows Server 2008
Microsoft(R) Windows Server(R) 2008 Enterprise	Windows Server 2008 Enterprise	
Microsoft(R) Windows Server(R) 2008 R2 Standard (x64)	Windows Server 2008 R2	
Microsoft(R) Windows Server(R) 2008 R2 Enterprise (x64)		
Microsoft(R) Windows Server(R) 2008 R2 Datacenter (x64)		
Microsoft(R) Windows Server(R) 2008 Standard (x64)	Windows Server 2008 (x64)	
Microsoft(R) Windows Server(R) 2008 Enterprise (x64)		

Full name or meaning	Abbreviation	
Microsoft(R) Windows Server(R) 2003, Standard x64 Edition	Windows Server 2003 x64 Editions	Windows (x64)
Microsoft(R) Windows Server(R) 2003, Enterprise x64 Edition		
Microsoft(R) Windows Server(R) 2003 R2, Standard x64 Edition		
Microsoft(R) Windows Server(R) 2003 R2, Enterprise x64 Edition		
Microsoft(R) Windows(R) XP Professional x64 Edition	Windows XP x64 Edition	
Microsoft(R) Windows Server(R) 2003, Enterprise Edition (64-bit version)	Windows Server 2003 (IPF)	Windows(IPF)
Microsoft(R) Windows(R) XP Professional x64 Edition	Windows XP x64 Edition	Windows XP
Microsoft(R) Windows(R) XP Professional Operating System	Windows XP Professional	
Microsoft(R) Windows(R) XP Home Edition Operating System	Windows XP Home Edition	
Microsoft(R) Windows Vista(R) Home Basic	Windows Vista Home Basic	Windows Vista
Microsoft(R) Windows Vista(R) Home Premium	Windows Vista Home Premium	
Microsoft(R) Windows Vista(R) Ultimate	Windows Vista Ultimate	
Microsoft(R) Windows Vista(R) Business	Windows Vista Business	
Microsoft(R) Windows Vista(R) Enterprise	Windows Vista Enterprise	
Microsoft(R) Windows Vista(R) Home Basic (x64)	Windows Vista (x64)	
Microsoft(R) Windows Vista(R) Home Premium (x64)		
Microsoft(R) Windows Vista(R) Ultimate (x64)		
Microsoft(R) Windows Vista(R) Business (x64)		
Microsoft(R) Windows Vista(R) Enterprise (x64)		
Microsoft(R) Windows(R) 7 Home Premium	Windows 7	
Microsoft(R) Windows(R) 7 Professional		

Full name or meaning	Abbreviation
Microsoft(R) Windows(R) 7 Enterprise	
Microsoft(R) Windows(R) 7 Ultimate	
Microsoft(R) Windows(R) 7 Home Premium (x64)	Windows 7 (x64)
Microsoft(R) Windows(R) 7 Professional (x64)	
Microsoft(R) Windows(R) 7 Enterprise (x64)	
Microsoft(R) Windows(R) 7 Ultimate (x64)	
Single server	SDS
System manager	MGR
Front-end server	FES
Dictionary server	DS
Back-end server	BES

- Windows Server 2003 and Windows Server 2008 may be referred to collectively as *Windows Server*. Windows 2000, Windows XP, Windows Server, Windows Vista, and Windows 7 may be referred to collectively as *Windows*.
- The hosts file means the `hosts` file stipulated by TCP/IP (including the `/etc/hosts` file). As a rule, a reference to the hosts file means the `%windir%\system32\drivers\etc\hosts` file.

This manual also uses the following acronyms:

Acronym	Full name or meaning
ACK	Acknowledgement
ADM	Adaptable Data Manager
ADO	ActiveX Data Objects
ADT	Abstract Data Type
AP	Application Program
API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
BES	Back End Server
BLOB	Binary Large Object

Acronym	Full name or meaning
BMP	Basic Multilingual Plane
BOM	Byte Order Mark
CD-ROM	Compact Disc - Read Only Memory
CGI	Common Gateway Interface
CLOB	Character Large Object
CMT	Cassette Magnetic Tape
COBOL	Common Business Oriented Language
CORBA(R)	Common ORB Architecture
CPU	Central Processing Unit
CSV	Comma Separated Values
DAO	Data Access Object
DAT	Digital Audio Tape
DB	Database
DBM	Database Module
DBMS	Database Management System
DDL	Data Definition Language
DF for Windows NT	Distributing Facility for Windows NT
DF/UX	Distributing Facility/for UNIX
DIC	Dictionary Server
DLT	Digital Linear Tape
DML	Data Manipulate Language
DNS	Domain Name System
DOM	Document Object Model
DS	Dictionary Server
DTD	Document Type Definition
DTP	Distributed Transaction Processing
DWH	Data Warehouse

Acronym	Full name or meaning
EUC	Extended UNIX Code
EX	Exclusive
FAT	File Allocation Table
FD	Floppy Disk
FES	Front End Server
FQDN	Fully Qualified Domain Name
FTP	File Transfer Protocol
GUI	Graphical User Interface
HBA	Host Bus Adapter
HD	Hard Disk
HTML	Hyper Text Markup Language
ID	Identification number
IP	Internet Protocol
IPF	Itanium(R) Processor Family
JAR	Java Archive File
Java VM	Java Virtual Machine
JDBC	Java Database Connectivity
JDK	Java Developer's Kit
JFS	Journaled File System
JFS2	Enhanced Journaled File System
JIS	Japanese Industrial Standard code
JP1	Job Management Partner 1
JRE	Java Runtime Environment
JTA	Java Transaction API
JTS	Java Transaction Service
KEIS	Kanji processing Extended Information System
LAN	Local Area Network

Acronym	Full name or meaning
LDAP	Lightweight Directory Access Protocol
LIP	Loop Initialization Process
LOB	Large Object
LRU	Least Recently Used
LTO	Linear Tape-Open
LU	Logical Unit
LUN	Logical Unit Number
LVM	Logical Volume Manager
MGR	System Manager
MIB	Management Information Base
MRCF	Multiple RAID Coupling Feature
MSCS	Microsoft Cluster Server
MSFC	Microsoft Failover Cluster
NAFO	Network Adapter Fail Over
NAPT	Network Address Port Translation
NAT	Network Address Translation
NIC	Network Interface Card
NIS	Network Information Service
NTFS	New Technology File System
ODBC	Open Database Connectivity
OLAP	Online Analytical Processing
OLE	Object Linking and Embedding
OLTP	On-Line Transaction Processing
OOCOBOL	Object Oriented COBOL
ORB	Object Request Broker
OS	Operating System
OSI	Open Systems Interconnection

Acronym	Full name or meaning
OTS	Object Transaction Service
PC	Personal Computer
PDM II E2	Practical Data Manager II Extended Version 2
PIC	Plug-in Code
PNM	Public Network Management
POSIX	Portable Operating System Interface for UNIX
PP	Program Product
PR	Protected Retrieve
PU	Protected Update
RAID	Redundant Arrays of Inexpensive Disk
RD	Relational Database
RDB	Relational Database
RDB1	Relational Database Manager 1
RDB1 E2	Relational Database Manager 1 Extended Version 2
RDO	Remote Data Objects
RiSe	Real time SAN replication
RM	Resource Manager
RMM	Resource Manager Monitor
RPC	Remote Procedure Call
SAX	Simple API for XML
SDS	Single Database Server
SGML	Standard Generalized Markup Language
SJIS	Shift JIS
SNMP	Simple Network Management Protocol
SNTP	Simple Network Time Protocol
SQL	Structured Query Language
SQL/K	Structured Query Language / VOS K`

Acronym	Full name or meaning
SR	Shared Retrieve
SU	Shared Update
TCP/IP	Transmission Control Protocol / Internet Protocol
TM	Transaction Manager
TMS-4V/SP	Transaction Management System - 4V / System Product
UAP	User Application Program
UOC	User Own Coding
VOS K	Virtual-storage Operating System Kindness
VOS1	Virtual-storage Operating System 1
VOS3	Virtual-storage Operating System 3
WS	Workstation
WWW	World Wide Web
XDM/BASE E2	Extensible Data Manager / Base Extended Version 2
XDM/DF	Extensible Data Manager / Distributing Facility
XDM/DS	Extensible Data Manager / Data Spreader
XDM/RD E2	Extensible Data Manager / Relational Database Extended Version 2
XDM/SD E2	Extensible Data Manager / Structured Database Extended Version 2
XDM/XT	Extensible Data Manager / Data Extract
XDS	Extended Data Server
XFIT	Extended File Transmission program
XML	Extensible Markup Language

Path name representations

- The backslash (\) is used as the delimiter in path names. Readers who are using a UNIX edition of HiRDB must replace the backslash with a forward slash (/). When the path names in the Windows and UNIX editions differ, both path names are given.
- The HiRDB directory path is represented as `%PDDIR%`. However, when the path names in the Windows and UNIX editions differ, the directory path in the UNIX

edition is represented as `$PDDIR`, as shown in the following example:

Windows edition: `%PDDIR%\CLIENT\UTL\`

UNIX edition: `$PDDIR/client/lib/`

- `%windir%` refers to a Windows installation directory path.

Log representations

- Windows edition

  The application log that is displayed by Windows Event Viewer is referred to as the *event log*. The following procedure is used to view the event log.

To view the event log:

1. Choose **Start**, **Programs**, **Administrative Tools (Common)**, and then **Event Viewer**.

2. Choose **Log**, and then **Application**.

   The application log is displayed. Messages with **HiRDBSingleServer** or **HiRDBParallelServer** displayed in the **Source** column were issued by HiRDB.

   If you specified a setup identifier when you installed HiRDB, the specified setup identifier follows **HiRDBSingleServer** or **HiRDBParallelServer**.

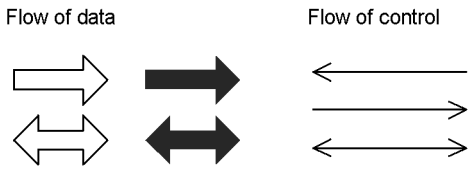
- UNIX edition

  The OS log is referred to generically as *syslogfile*. syslogfile is the log output destination specified in `/etc/syslog.conf`. Typically, the following files are specified as syslogfile.

OS	File
HP-UX	`/var/adm/syslog/syslog.log`
Solaris	`/var/adm/messages` or `/var/log/syslog`
AIX	`/var/adm/ras/syslog`
Linux	`/var/log/messages`

Symbols used in figures

The following symbols are used in the figures in this manual:



Symbols used in text

In addition to those used in figures, this manual uses the following symbols in text:

↑ formula ↑

The resulting value is to be rounded up.

↑ CI-length / 1000 ↑

↓ formula ↓

The resulting value is to be rounded off.

↓ CI-length / 1000 ↓

Organization of SQL syntax explanations

This manual explains the syntax of SQL expressions in the following general format:

Function

Explains the function of the SQL expression.

Privileges

Explains the privileges that are required in order to use the SQL expression.

Format

Shows the format of the operands.

Operands

Explains the operands that can be specified, when they should be specified, and their specification conventions.

Rules

Explains rules applicable to the SQL expression.

Notes

Provides helpful notes, such as the relationships between this SQL expression and

other SQL expressions.

Examples

Shows examples of the use of the SQL expression.

Conventions: Fonts and symbols

The following table explains the fonts used in this manual:

Font	Convention
Bold	**Bold** type indicates text on a window, other than the window title. Such text includes menus, menu options, buttons, radio box options, or explanatory labels. For example: • From the **File** menu, choose **Open**. • Click the **Cancel** button. • In the **Enter name** entry box, type your name.
Italics	*Italics* are used to indicate a placeholder for some actual text to be provided by the user or system. For example: • Write the command as follows: `copy` *source-file target-file* • The following message appears: `A file was not found. (file =` *file-name*`)` *Italics* are also used for emphasis. For example: • Do *not* delete the configuration file.
`Code font`	A `code font` indicates text that the user enters without change, or text (such as messages) output by the system. For example: • At the prompt, enter `dir`. • Use the `send` command to send mail. • The following message is displayed: `The password is incorrect.`
`SD`	Bold code-font characters indicate the abbreviation for a command.
`perm`	Underlined characters indicate the default value.

The following table explains the symbols used in this table:

Symbol	Convention
|	In syntax explanations, a vertical bar separates multiple items, and has the meaning of OR. For example: `A|B|C` means A, or B, or C.
{ }	In syntax explanations, curly brackets indicate that only one of the enclosed items is to be selected. For example: `A|B|C` means A, or B, or C.

Symbol	Convention
`[ ]`	In syntax explanations, square brackets indicate that the enclosed item or items are optional. For example: `[A]` means that you can specify `A` or nothing. `[B|C]` means that you can specify `B`, or `C`, or nothing.
. . .	In coding, an ellipsis (. . .) indicates that one or more lines of coding are not shown for purposes of brevity. In syntax explanations, an ellipsis indicates that the immediately preceding item can be repeated as many times as necessary. For example: `A, B, B, . . .` means that, after you specify `A, B,` you can specify B as many times as necessary.
`()`	Parentheses indicate the range of items to which the vertical bar (|) or ellipsis (. . .) is applicable.
Δ	A single delta symbol indicates one space. For example `*DC` Δ
ΔΔ	Two delta symbols indicate one or more spaces. For example: `WHERE` ΔΔ `IPNO=1`
`::=`	The item to the left of the `::=` notation is specified in terms of the items to the right of the `::=` notation. For example: *table-name* `::=` [*authorization-identifier* `.`]*table-identifier*

Syntax element conventions

The following table explains the syntactical element symbols used in this manual:

Syntax element	Meaning
<*alphabetics*>	The alphabetic characters (`A` to `Z` and `a` to `z`) and the underscore (_)
<*alphabetics and special characters*>	The alphabetic characters (`A` to `Z` and `a` to `z`) and the special characters #, @, and \
<*alphanumerics*>	Alphabetic characters and the numeric digits (`0` to `9`)
<*alphanumerics and special characters*>	Alphabetic characters, special characters, and numeric characters
<*unsigned-integer*>	Numeric values
<*unsigned-decimal*>[1]	Numeric values (`0` to `9`) and the period (.), numeric value (0-9)
<*identifier*>[2]	Alphanumeric character string beginning with an alphabetic character
<*character-string*>	String of any characters

Syntax element	Meaning
<*symbolic-name*>	Alphanumeric character string beginning with an alphabetic character or a special character In the UNIX edition, symbolic names cannot include a backslash (\).
<*path-name*>[3]	In the UNIX edition, path names can include forward slashes (/), alphanumeric characters, periods (.), hash marks (#), and at marks (@). In the Windows edition, path names can include backslashes (\), alphanumeric characters, periods (.), spaces, parentheses (()), hash marks (#), and at marks (@).

Notes on Windows path names

- In this manual, the Windows terms *directory* and *folder* are both referred to as *directory*.
- Include the drive name when you specify an absolute path name.

Example: `C:\win32app\hitachi\hirdb_s\spool\tmp`

- When you specify a path name in a command argument, in a control statement file, or in a HiRDB system definition file, and that path name includes a space or a parenthesis, you must enclose the entire path name in double quotation marks (").

Example:
```
pdinit -d "C:\Program
Files(x86)\hitachi\hirdb_s\conf\mkinit"
```

However, double quotation marks are not necessary when you use the `set` command in a batch file or at the command prompt to set an environment variable, or when you specify the installation directory. If you do use double quotation marks in such a case, the double quotation marks become part of the value assigned to the environment variable.

Example: `set PDCLTPATH=C:\Program Files\hitachi\hirdb_s\spool`

- HiRDB cannot use files on a networked drive, so you must install HiRDB and configure the HiRDB environment on a local drive. Files used by utilities, such as utility input and output files, must also be on the local drive.
- Do not use a short path name in place of the full path name (for example, do not use `C:\PROGRA~1`).

Conventions: KB, MB, GB, and TB

This manual uses the following conventions:

- 1 KB (kilobyte) is 1,024 bytes.
- 1 MB (megabyte) is $1{,}024^2$ bytes.

- 1 GB (gigabyte) is $1{,}024^3$ bytes.
- 1 TB (terabyte) is $1{,}024^4$ bytes.

Conventions: Version numbers

The version numbers of Hitachi program products are usually written as two sets of two digits each, separated by a hyphen. For example:

- Version 1.00 (or 1.0) is written as 01-00.
- Version 2.05 is written as 02-05.
- Version 2.50 (or 2.5) is written as 02-50.
- Version 12.25 is written as 12-25.

The version number might be shown on the spine of a manual as *Ver. 2.00,* but the same version number would be written in the program as *02-00*.

HiRDB database language acknowledgements

The interpretations and specifications developed by Hitachi, Ltd. for the HiRDB database language specifications described in this manual are based on the standards listed below. Along with citing the standards relevant to HiRDB database language specifications, we would like to take this opportunity to express our appreciation to the original developers of these standards.

- JIS X 3005 Information technology -- Database languages -- SQL
- ISO/IEC 9075 Information technology -- Database languages -- SQL

Note:

JIS: Japanese Industrial Standards

ISO: International Organization for Standardization

IEC: International Electrotechnical Commission

Important notes on this manual

The following facilities are explained, but they are not supported:

- Distributed database facility
- Server mode system switchover facility
- User server hot standby
- Rapid system switchover facility
- Standby-less system switchover (1:1) facility
- Standby-less system switchover (effects distributed) facility

- HiRDB External Data Access facility
- Inner replica facility
- Updatable online reorganization
- Sun Java System Directory Server linkage facility
- Simple setup tool
- Extended syslog facility
- Rapid batch facility
- Memory database facility
- Linkage with JP1/NETM/Audit

The following products and option program products are explained, but they are not supported:

- HiRDB CM
- HiRDB Disaster Recovery Light Edition
- uCosminexus Grid Processing Server
- HiRDB Text Search Plug-in
- HiRDB XML Extension
- TP1/Server Base
- JP1/PFM-Agent Option for HiRDB
- JP1/VERITAS NetBackup Agent for HiRDB License
- HiRDB Dataextractor
- HiRDB Datareplicator
- XDM/RD
- HiRDB SQL Tuning Advisor
- COBOL2002

Contents

Chapter 1. Basics

This chapter explains the basics of using SQL. It contains the following sections:

1.1 SQL coding format

1.1.1 Order of specifying operands

The operands of an SQL statement must be specified in the order in which they are shown in the format.

1.1.2 Keyword specification

A keyword is a fixed character string that specifies a function, such as the name of an SQL statement (e.g., `SELECT` or `UPDATE`). Because most keywords are registered as system-reserved words, they can be specified by users only at prescribed positions in a command.

However, keywords that are not registered as reserved words can be used as names.

Following is an example of keywords (for a list of the reserved words, see *A. Reserved Words*):

```
CREATE TABLE ID. T1 (C1 CHAR, C2 INTEGER)
Keyword Keyword          Keyword  Keyword
```

1.1.3 Specifying a numeric value

Numeric values (other than numeric literals) that are to be specified in an SQL statement must be specified in accordance with the notation rules and restrictions that are applicable to unsigned integers; these notation rules and restrictions are shown in Tables *1-11* and *1-12*.

SQL recognizes the following as numeric values (not as numeric literals):

- Sort item specification number (sort item specification number of the `ORDER BY` clause)
- Length and maximum length (the length and the maximum length of character data)
- Maximum number of resources (total number of tables and indexes used in a program)
- Percentage of unused space (the percentage of unused space specified in table and index definitions)
- Precision (the number of significant digits in decimal data)
- Decimal scaling position (the number of digits following the decimal point in decimal data)
- Data guarantee level (`ALTER PROCEDURE`, `ALTER ROUTINE`, `ALTER TRIGGER`, `CREATE PROCEDURE`, `CREATE TYPE`, and `CREATE TRIGGER`)

- Subscript (for repetition columns)
- Maximum number of elements (in a repetition column)
- Precision of fractional seconds (number of digits to the right of the decimal point in time data)

1.1.4 Insertion of delimiters

The following characters can be used as delimiters:

- Space (`X'20'`)
- TAB (`X'09'`)
- NL (`X'0a'`)
- CR (`X'0d'`)
- Two-byte space
- Comment

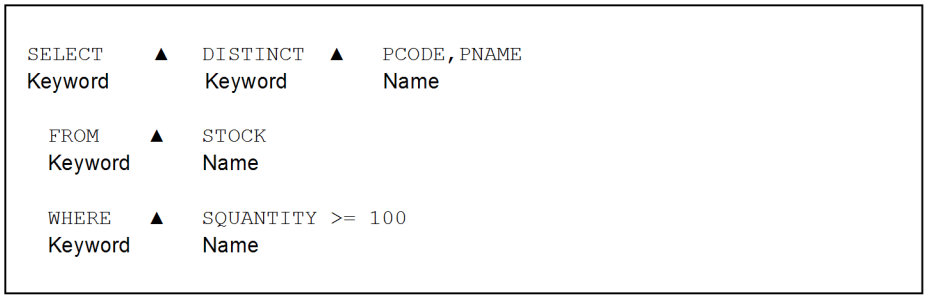
(1) Delimiter insertion locations

A delimiter can be inserted at the following locations:

- Between keywords
- Between a keyword and a name
- Between names
- Between a keyword and a numeric value
- Between a name and a numeric value

The following figure shows examples of inserting delimiters.

Figure 1-1: Examples of delimiter insertion



(2) Locations where a delimiter is not allowed

A delimiter cannot be inserted in any of the following locations:

- Within a keyword
- Within a name not enclosed in double quotation marks
- Immediately following the opening double quotation mark enclosing a name
- Immediately before the closing double quotation mark enclosing a name
- Within a numeric literal
- Within an operator
- Between the N and the first single quotation mark (') in N'...' that represents a national character string literal
- Between the M and the first single quotation mark (') in M'...' that represents a mixed character string literal
- Between the X and the first single quotation mark (') in X'...' that represents a hexadecimal character string literal
- Between periods in a component specification that is written as . .

The following figure shows examples of locations where delimiters are not allowed.

Figure 1-2: Examples of locations where delimiter is not allowed

S∆ELECT	678∆9	"STOCK∆"	"∆STOCK"	<∆=
Keyword	Numeric literal	Name		Operator

(3) Locations where a delimiter is allowed

Delimiters can be inserted in the following locations:

- Before and after any of the following special characters, except as indicated in (2) above:

  ,, ., -, +, *, ', ", (,), <, >, =, ^, !, /, ?, :, ;, |, [,], TAB, NL, CR, space, two-byte space

The following figure shows examples of locations where delimiters can be inserted.

Figure 1-3: Examples of locations where delimiter can be inserted

SCORE∆, PNAME	(Space is inserted before comma (,) special character)
SCORE = ∆1	(Space is inserted after equals sign (=) special character)

(4) Comment

In an SQL statement, any characters that appear after a `/*` and before the first `*/` that is encountered are treated as a comment. When inserting a comment, observe the following notes:

- `/*-*/` marks that are enclosed in double quotation marks (`"`) or single quotation marks (`'`) are not treated as a comment.
- Use of an SQL statement consisting only of a comment can cause an error.
- Comments cannot be nested.
- The `/*>>-<<*/` format is treated as an SQL optimization specification. See *2.24 SQL optimization specification*.

Examples of specifying a comment are given as follows:

Correct examples:

```
CREATE TABLE T1(C1 INT) /* COMMENT */

CREATE /* COMMENT */ TABLE T1(C1 INT)
```

Incorrect examples:

```
SELECT * FROM T1 /* COMMENT
```
*. . . An error occurs due to the absence of a */ termination symbol.*

```
CREATE TABLE T1 /* COMMENT1 /* COMMENT2 */ COMMENT3 */(C1
INT)
```
. . . Nested comments cause an error.

1.1.5 SQL character set

The following table lists the characters that can be used in SQL statements.

Table 1-1: SQL character set

Type	Permissible characters in SQL
Character string literal	One-byte character codes (not including `X'00'`)
National character string literal	All two-byte code characters
Mixed character string literal	One-byte character codes (not including `X'00'`) and all two-byte code characters

Type	Permissible characters in SQL
Other than above	• Following one-byte code characters: Upper-case alphabetic characters (A to Z, $, @, #) Lower-case alphabetic characters (a to z) Numeric characters (0 to 9) Space Underscore character (_) Kana characters • All two-byte code characters • Following special characters (one-byte character codes): Comma (,) Period (.) Hyphen or minus sign (-) Plus sign (+) Asterisk (*) Single quotation mark (') Double quotation mark (") Left parenthesis (( ) Right parenthesis ( )) Less than sign (<) Greater than sign (>) Equals sign (=) Circumflex (^) Exclamation mark (!) Forward slash (/)
Other than above	Question mark (?) Colon (:) Semicolon (;) Percent sign (%) Vertical bar (|) Left square bracket ([) Right square bracket (]) TAB (X'09') NL (X'0a') CR (X'0d')

Characters that can be used in SQL vary depending on the character code type specified in the pdsetup command. For details about the pdsetup command, see the manual *HiRDB Version 9 Command Reference*.

SQL allows the use of one-byte and two-byte characters. These two types of characters require different character codes (two-byte characters are not available among the single-byte character codes). The following table indicates the relationships between characters and the character code types:

Specified character code		Single-byte character	Double-byte character	Remarks
Multiple-byte character code	`sjis`[#3] (Shift JIS kanji)	JISX0201	JISX0208	Double-byte characters include gaiji characters.
	`ujis`[#2] (EUC Japanese kanji)	JISX0201	JISX0208	Double-byte characters do not include gaiji characters.[#1]
	`chinese`[#6] (EUC Chinese kanji)	ISO-8859-1 (other than `80` to `FF`)	GB2312-80	Double-byte characters do not include gaiji characters[#1]
	`utf-8`[#3, #4] (Unicode (UTF-8))	JISX0221	JISX0221	Double-byte characters include gaiji characters. For characters in the ASCII code range, these characters are treated the same as other characters, except that in some cases a single character is represented in six bytes.[#5]
		MS-Unicode	MS-Unicode	
	`chinese-gb18030`[#6] (Chinese kanji GB18030)	ISO-8859-1 (other than `80` to `FF`)	GB18030-2000	Double-byte characters include gaiji characters. For characters in the ASCII code range, these characters are treated the same as other characters, except that in some cases a single character is represented in four bytes.
Single-byte character code	`lang-c`[#2, #6] (8-bit code)	Same as the specified code	--	These codes can be used in US ASCII and 8-bit codes.

Legend:

--: Not applicable

#1: Gaiji codes assigned to EUC Code Set 3 (character codes that are represented in three bytes as $(8F)_{16}$ $(xxxx)_{16}$) cannot be used.

#2: Cannot be used in the Windows edition.

#3: Passing and receiving Japanese data through a `String` class or a class inheriting that class between a Java UAP and HiRDB or between HiRDB and a Java routine is performed according to the rules regarding the mapping of Java character codes (mapping between a given character code and Unicode). In this case, some gaiji codes may fail to be converted correctly.

#4: HiRDB is governed by the UTF-8 encoding rules only; the mapping of codes and characters is transparent to HiRDB. Therefore, you can use characters that comply with the UTF-8 encoding rules. However, when performing character code conversion, you must pay attention to the relationship between the character set and the encoding rules. Therefore, to specify `PDCLTCNVMODE` in the client environment definition when the character codes of the HiRDB client are SJIS and the character codes of the HiRDB server are UTF-8, you must determine whether JISX0221 or MS-Unicode is being used. For details about `PDCLTCNVMODE`, see the *HiRDB Version 9 UAP Development Guide*.

#5: To use characters of four bytes or longer, you may need to specify the `pd_substr_length` operand of the system definition and `PDSUBSTRLEN` in the client environment definition. For details about the `pd_substr_length` operand, see the manual *HiRDB Version 9 System Definition*; for details about `PDSUBSTRLEN`, see the *HiRDB Version 9 UAP Development Guide*.

In ISO/IEC 10646, characters are allocated to bytes 1 through 4. Bytes 5 and 6 are reserved for future specifications, and no characters are allocated. Therefore, if you use bytes 5 or 6, there is no assurance that a conflict will not occur in the future.

#6: Cannot be used if XDS is used.

1.1.6 Maximum length of an SQL statement

The maximum allowable length of an SQL statement is 2,000,000 bytes.

1.1.7 Specification of names

A name can be specified by either enclosing it in double quotation marks (`"`) or not enclosing it.

Reference note:

For specifying a name, Hitachi recommends enclosing it in double quotation marks ("). If a name containing alphabetic characters is enclosed in double quotation marks ("), the alphabetic characters will be case sensitive.

Reasons

A name must be distinct from any of the reserved words. However, a name enclosed in double quotation marks (") can be identical to a reserved word. Because, as the SQL is expanded, additional reserved words may be registered in the system, the potential problem of conflict with newly added reserved words can be avoided by enclosing names in double quotation marks (").

However, except for a cursor name specified within a process or function, if you use the following names in a UAP, specify them without enclosing them in double quotation marks (").

- Cursor name
- SQL statement identifier
- Embedded variable name, indicator variable name, or host identifier

(1) Common rules

- For restrictions on characters and length (in bytes) that can be used in a name, see *Table 1-1 SQL character set* and *Table 1-2 Characters in names and maximum number of characters*.
- Single-byte and double-byte characters can be used in a name on a mixed basis.
- If a name containing alphabetic characters is not enclosed in double quotation marks ("), the alphabetic characters are handled as upper-case characters.
- If a name containing alphabetic characters is enclosed in double quotation marks ("), the alphabetic characters will be case sensitive.
- A name must begin with a single-byte upper-case alphabetic character, a single-byte lower-case alphabetic character, a single-byte upper-case katakana (ｱ～ﾝ, ﾜ) or a double-byte character.
- A name cannot contain a double-byte space.
- Any of the following characters, if contained in a name, must be enclosed in Double quotation marks (")
  - Single-byte space
  - Single-byte hyphen

  Any cursor name containing a single-byte hyphen should be enclosed in double

quotation marks (") only if the cursor name is specified in a procedure or function.

Table 1-2: Characters in names and maximum numbers of characters

Type of name	Max number of bytes or chars	Single-byte characters						Double-byte chars
		UC, numeric	LC	Kana	Underscore (_)	Space	Hyphen (-)	
Index-type identifier	30 bytes	Y	Y	Y#2	Y	Y	Y	Y#2
Index identifier		Y	Y	Y#2	Y	Y	Y	Y#2
Embedded variable name	21 characters, 22 characters, 30 characters, 31 characters, 63 characters #1	Y#4	Y	Y#4	Y	N	Y#4	Y#4
Cursor name	30 bytes	Y	Y	Y#2	Y	N	Y	Y#2
External routine name	255 bytes	Y	Y	Y#2	Y	N	N	Y#2
Condition name	30 bytes	Y	Y	Y#2	Y	Y	Y	Y#2
Correlation name		Y	Y	Y#2	Y	Y	Y	Y#2
Attribute name		Y	Y	Y#2	Y	Y	Y	Y#2
Data type identifier		Y	Y	Y#2	Y	Y	Y	Y#2
Query name		Y	Y	Y#2	Y	Y	Y	Y#2
Trigger identifier		Y	Y	Y#2	Y	Y	Y	Y#2
Authorization identifier	8 bytes	Y	Y	N	N	N	N	N

Type of name	Max number of bytes or chars	Single-byte characters						Double-byte chars
		UC, numeric	LC	Kana	Underscore (_)	Space	Hyphen (-)	
Password[#3]	30 bytes	Y	Y	N	N	N	N	N
Table identifier		Y	Y	Y[#2]	Y	Y	Y	Y[#2]
Indicator variable name	30 characters, 31 characters, 63 characters [#1]	Y[#4]	Y	Y[#4]	Y	N	Y[#4]	Y[#4]
Statement label	30 bytes	Y	Y	Y[#2]	Y	Y	Y	Y[#2]
Loop variable name		Y	Y	Y[#2]	Y	Y	Y	Y[#2]
Host identifier	30 characters, 31 characters, 63 characters [#1]	Y[#4]	Y	Y[#4]	Y	N	Y[#4]	Y[#4]
List name	30 bytes	Y	Y	Y[#2]	Y	Y	Y	Y[#2]
Routine identifier		Y	Y	Y[#2]	Y	Y	Y	Y[#2]
Column name		Y	Y	Y[#2]	Y	Y	Y	Y[#2]
RDAREA name		Y	Y	N	Y	Y	Y	Y[#2]
SQL parameter name		Y	Y	Y[#2]	Y	Y	Y	Y[#2]
SQL statement identifier		Y	Y	Y[#2]	Y	N	Y	Y[#2]
SQL variable name		Y	Y	Y[#2]	Y	Y	Y	Y[#2]
Constraint name		Y	Y	Y[#2]	Y	Y	Y	Y[#2]

Type of name	Max number of bytes or chars	Single-byte characters						Double-byte chars
		UC, numeric	LC	Kana	Underscore (_)	Space	Hyphen (-)	
XQuery variable identifier		Y	Y	Y#2	Y	Y	Y	Y#2
Sequence generator identifier		Y	Y	Y#2	Y	Y	Y	Y#2
DBAREA name		Y	Y	N	Y	N	Y	N

UC: Upper-case characters

LC: Lower-case characters

Y: Can be used.

N: Cannot be used.

#1: The lengths of embedded variable names, indicator variable names, and host identifiers are as follows:

Type of name		Host language		
		COBOL85	COBOL2002	C language
Embedded variable name	BLOB type	21 characters maximum	22 characters maximum	63 characters maximum
	Non-BLOB type	30 characters maximum	31 characters maximum	
Indicator variable name or host identifier				

For COBOL85 and COBOL2002, double-byte characters count as one character. For the C language, universal character names count as one character.

#2: Restricted by the type of character code used. Do not use if you specified `utf-8` or `chinese-gb18030` as the character code classification with the `pdntenv` command (`pdsetup` command in the UNIX edition).

#3: The following rules apply to passwords:

- A password must begin with an alphabetic character.
- A password cannot consist of numeric characters only.
- It is possible to require that a password not consist of only upper-case letters or only lower-case letters.

#4

- COBOL85 and COBOL2002:

  Embedded variable names, indicator variable names, and host identifiers can include `a` to `z`, `A` to `Z`, `0` to `9`, underscores (_), hyphens (`-`), half-width katakana, and double-byte characters.

  You can mix single-byte and double-byte characters.

- C language:

  Embedded variable names, indicator variable names, and host identifiers can include `a` to `z`, `A` to `Z`, `0` to `9`, and underscores (_).

  You can specify universal character names (`\u`*xxxx* and `\U`*xxxxxxxx*) if the compiler supports them.

  Multi-byte characters are not allowed.

(2) What to do if a name conflicts with an SQL reserved word

The following text shows what to do when a specified name conflicts with an SQL reserved word. Hitachi recommends the method described in (a). Method (b) should be used when the SQL cannot be revised, including the situation in which an application cannot be modified.

(a) Revising the SQL

Revise the SQL so that the name that is in conflict with a reserved word is enclosed in double quotation marks (`"`). Note that if a name containing alphabetic characters is enclosed in double quotation marks (`"`), the alphabetic characters will be case sensitive.

(b) Using the SQL reserved word deletion facility

The SQL reserved word deletion facility provides a facility for deleting keywords registered as SQL reserved words from the list of reserved words. If the reserved word that is in conflict with a specified name can be deleted using the SQL reserved word deletion facility, by deleting it from the list of reserved words, you can use the specified name without enclosing it in double quotation marks (`"`). For a list of reserved words that can be deleted using the SQL reserved word deletion facility, see *A.3 Reserved words that can be deleted using the SQL reserved word deletion facility*. For reserved words that cannot be deleted by the SQL reserved word deletion facility, revise the SQL using method (a).

Although a deleted reserved word can be used as a name, it is no longer a keyword. Consequently, the SQL facility that uses the deleted reserved word may cease to function properly. For SQL facilities that lose their functionality in this manner, see *A.3 Reserved words that can be deleted using the SQL reserved word deletion facility*.

Usage method

SQL reserved word deletion files for which the reserved words to be deleted using the SQL reserved word deletion facility must be specified in advance in the system common definition `pd_delete_reserved_word_file` operand. For the system common definition `pd_delete_reserved_word_file` operand, see the manual *HiRDB Version 9 System Definition*.

When using the SQL reserved word deletion facility, you need to specify the SQL reserved word definition file to be used, by means of the client environment variable `PDDELRSVWDFILE`. For the client environment variable `PDDELRSVWDFILE`, see the *HiRDB Version 9 UAP Development Guide*.

Notes

1. When executing an SQL that recreates an SQL object, in the client environment variable `PDDELRSVWDFILE`, specify the reserved word deletion file that was specified at the time of the definition of the object.
2. When importing or exporting a table using the dictionary import/export utility (`pdexp`), in the client environment variable `PDDELRSVWDFILE`, specify the reserved word deletion file that was specified at the time of the definition of the table. Also, ensure that the following items match between the export source and import destination directories:
   - All files under `%PDDIR%\conf\pdrsvwd`
   - Values specified in the system common definition `pd_delete_reserved_word_file` operand

1.1.8 Qualifying a name

You can use a name qualifier to do things like explicitly specifying an authorization identifier or making a name unique. A name qualifier concatenates an authorization identifier to a table identifier, using a character such as a period (.) between the two strings.

(1) Table name, index name, index type name, user-defined type name, routine name, trigger name, and sequence generator name

Explanations of these names and their formats are given as follows:

Table name:

A table identifier qualified with an authorization identifier.

Index name:

An index identifier qualified with an authorization identifier.

Index type name:

An index type identifier qualified with an authorization identifier.

User-defined type name:

A data type identifier qualified with an authorization identifier.

Routine name:

A routine identifier qualified with an authorization identifier.

Trigger name:

A trigger identifier qualified with an authorization identifier.

Sequence generator name:

A sequence generator identifier qualified with an authorization identifier.

table-name : : = [*authorization-identifier* .] *table-identifier*
index-name : : = [*authorization-identifier* .] *index-identifier*
index-type-name : : = [*authorization-identifier* .] *index-type-identifier*
user-defined-type-name : : = [*authorization-identifier* .] *data-type-identifier*
routine-name : : = [*authorization-identifier* .] *routine-identifier*
trigger-name : : = [*authorization-identifier* .] *trigger-identifier*
sequence-generator-name : : = [*authorization-identifier* .] *sequence-generator-identifier*

Item	Specification	Rules
Authorization identifier	If you are specifying your own table identifier, index identifier, index type identifier, data type identifier, routine identifier, or trigger identifier, specify your own authorization identifier. If the relevant identifier is owned by another user, specify that user's authorization identifier. However, if you specify the name of a public view as the table identifier, or the name of a public function or public procedure as the routine identifier, specify PUBLIC.	If you omit the authorization identifier, it is determined as follows: • When a utility is executed: Authorization identifier of the user who starts the utility. • When a UAP is executed:#1 If you omit the authorization identifier in a UAP, it is determined in the following order. 1. The authorization identifier specified in pre-processing 2. The authorization identifier specified in an operand of the CONNECT statement 3. If nothing is specified in the CONNECT statement's operands, the authorization identifier specified in the client environment definition PDUSER 4. PUBLIC (when a table identifier or routine identifier is specified#2)
Table identifier	Specify the name of a base table or view table.	
Index identifier	Specify the name of an index.	
Index type identifier	Specify the name of an index type.	
Data type identifier	Specify the name of a user-defined type.	
Routine identifier	Specify the name of a procedure or function.	
Trigger identifier	Specify the name of a trigger.	
Sequence generator identifier	Specify the name of a sequence generator.	

#1

When the authorization identifier is omitted from an SQL character string specified in a definition SQL, PREPARE statement, or EXECUTE IMMEDIATE statement, the authorization identifier that is assumed is determined in the following priority:

1. Authorization identifier that was in effect during CONNECT
2. Authorization identifier that was specified in a client environment variable
3. UAP user
4. PUBLIC (for a table identifier or routine identifier defined outside of definition SQL)

#2

For details about using the authorization identifier MASTER for a routine identifier, see *1.1.9 Schema path*.

(2) Table specification

When more than one table is specified in the same SQL statement, *table-specification* indicates a qualifier that enables identification of a specific table to which a column, *, or row applies. The table name or a correlation name can be specified as the table specification.

When the same table is joined or the same table is specified in a subquery to reference columns in a table used in an outer query in that subquery, a correlation name can be used as an alias for those tables. A correlation name allows the user to use a table as two distinct tables.

table-specification : : = { [*authorization-identifier* .] *table-identifier*
| *correlation-name* | *query-name* }

If a query name specified in the WITH clause and the same table identifier as the query name are to be specified in a FROM clause in the query expression body of a query expression that uses a WITH clause, it is necessary to explicitly qualify the table identifier with an authorization identifier and to differentiate the name by specifying a correlation name for the query name and the table identifier.

If a name in a table specification for the query expression body of a query expression that uses a WITH clause is qualified by an authorization identifier, that name is treated as a table identifier; if the name is not qualified by an authorization identifier, that name is treated as a query name or a table identifier. Note that if the name is not qualified by an authorization identifier, the query name has a higher priority than the table identifier.

Examples of table specifications are shown below.

Example 1

The two tables STOCK and ORDERS both have a column named PCODE. To reference the correct PCODE column, its name is qualified with its table name (STOCK):

```
SELECT STOCK.PCODE,PNAME,CCODE
    FROM STOCK,ORDERS
    WHERE STOCK.PCODE=ORDERS.PCODE
```

Example 2

In a table created by joining two of the same table (e.g., STOCK tables), correlation names (X, Y) are used as qualifiers (products in the same color as product code 101M are searched):

```
SELECT X.* FROM STOCK X,STOCK Y
```

```
    WHERE X.COLOR=Y.COLOR AND Y.PCODE='101M'
```

Example 3

Correlation names simplify the description of lengthy table names (the name of the RDAREA storing a table (STOCK) belonging to the user is searched from a data dictionary table):

```
SELECT X.RDAREA_NAME
    FROM MASTER.SQL_RDAREAS X,
         MASTER.SQL_TABLES Y
    WHERE Y.TABLE_SCHEMA='U'
      AND Y.TABLE_NAME='STOCK'
      AND X.RDAREA_NAME=Y.RDAREA_NAME
```

Example 4

In a query expression using a WITH clause, specifying, in a FROM clause in the query expression itself, a table identifier (STOCK) which is identical to the query name used in the WITH clause (STOCK):

```
WITH STOCK(QC1,QC2)
  AS (SELECT PCODE,PRICE*SQUANTITY FROM STOCK)
    SELECT * FROM STOCK X,USER1.STOCK Y
```

(3) Column specification

In the following cases, a column name or repetition column name must be qualified with a specified table name; such a qualified column name or repetition column name is called a column specification:

- In a retrieval in which multiple tables are specified in one FROM clause (by joining two or more tables) and the multiple tables contain identically named columns (without a qualification, it would not be clear which table was intended)
- In a retrieval in which multiple query names or table names are specified in one FROM clause in the query expression body of a query expression using a WITH clause (by joining two or more tables) and the multiple tables to be searched or derived by the derived query expression in the WITH clause contain identically named columns (without a qualification, it would not be clear which table was intended)

(Incorrect)

Whether CLM1 specified in the selection expression is a column name of query name QRY1 or of query name QRY2 is not clear; the column name must be qualified:

```
WITH QRY1(CLM1) AS (SELECT PNAME FROM STOCK),
     QRY2(CLM1) AS (SELECT PCODE FROM STOCK)
     SELECT CLM1 FROM QRY1,QRY2
```

(Correct)

The column name of the specified column is qualified:

```
WITH QRY1(CLM1) AS (SELECT PNAME FROM STOCK),
     QRY2(CLM1) AS (SELECT PCODE FROM STOCK)
     SELECT QRY1.CLM1,QRY2.CLM1 FROM QRY1,QRY2
```

- In a `WHERE` clause or `HAVING` clause in a subquery, a table specified in a `FROM` clause that occurs outside the subquery is to be referenced, or a column in the table to be updated is to be referenced (without a qualification, columns in an outer table cannot be referenced).

Some column names can be qualified while others cannot, due to syntactic considerations. The description *column-specification* in a format specification indicates a column name that can be qualified. The description *column-name* indicates that the column name cannot be qualified.
Column-specification::= [table-specification.]
{column-name|repetition-column-name[[subscript]]}

Subscript::={`integer|ANY`}

The following rules apply when a column name is qualified with a specified table name.

1. A column name can be qualified with a specified correlation name or table name only within the scope of the correlation name or table name.

   For the scope of a correlation name or table name, see *2.6 Table reference*, *DELETE statement Format 1 (Delete rows)* and *UPDATE statement Format 1 (Update data)* in Chapter 4.

2. If a subquery specifies more than one valid table specification (correlation name or table name) with the same name, any reference to that name refers to the table specification (correlation name or table name) that was specified in the innermost query.

   The following is an example of specifying multiple, valid table names that have the same name:

   Example:

   In the subquery, both the `T1`[1] and `T1`[2] specifications are valid. Referencing `T1` specifies `T1`[2] in the innermost query.

```
SELECT * FROM T1[1]
   WHERE T1[1].C2 >=
      (SELECT AVG(C2) FROM T1[2]
       WHERE T1[2].C1 = 1      )
```

Valid range of T1[1]: SELECT*FROM T1[1] and subsequent lines

Valid range of T1[2]: (SELECT AVG (C2) FROM T1[2] and subsequent lines

Tables having the most local scope (identically named tables occur in the same FROM clause) must be qualified with correlation names. The following is an example of identically named tables occurring in the same FROM clause:

(Incorrect)

This subquery, in which T1[1] and T1[2] which have the most local scope occur, contains an invalid scope specification. To avoid this error, specify appropriate correlation names.

```
SELECT*FROM T2
   WHERE C3 IN
      (SELECT T1[1].C3 FROM T1[1],T1[2]
       WHERE T1[1].C1=T1[2].C3   )
```

Most local scope: (SELECT T1[1].C3 FROM T1[1], T1[2] and subsequent specifications

(Correct)

The specified column must exist in the table that is specified with a qualifier.

```
SELECT*FROM T2
   WHERE C3 IN
      (SELECT X.C3 FROM T1[1] X,T1[2] Y
       WHERE X.C1=Y.C2          )
```

3. Column names can be qualified with a table specification. If column names are unqualified, the subquery that has the most local scope must contain one table that contains the specified column.

1.1.9 Schema path

When a routine name, index type name, or user-defined type that is not qualified explicitly with an authorization identifier is to be searched, a schema path determines the order in which different schemas are searched.

In the case of an index type name or a user-defined type name, if the target schema does not contain an index type name or a user-defined type name, the next schema is searched.

In the case of a function name, if executable candidate functions are not found according to the rules for determining the function to be called, the next schema is searched. For the rules for determining the function to be called, see *2.20 Function calls*.

(a) Order of search

Schemas are searched in the following order:

1. Schemas for the user associated with the default authorization identifier (for the default authorization identifier, see *1.1.8 Qualifying a name*)
2. MASTER schema

(b) Scope of a schema path

In a specification that is not qualified with an authorization identifier, a schema path is applied to the referencing of defined routine names, index type names, and user-defined type names.

1.2 Data types

1.2.1 Data types

Data types can be divided into the following two classes:

- Predefined data types
- User-defined data types (Specifies in the [*authorization-identifier* .] format.)

Format

data-type : : = {*Predefined-data-type* | *user-defined-type*}

Explanation

- Predefined data type

  Specifies a data type provided by HiRDB.

- User-defined data types

  A user-defined data type is specified in the following format:

  [*authorization-identifier* .]*data-type-identifier*

  authorization-identifier

  Specifies a user-defined type of authorization identifier.

  data-type-identifier

  Specifies a user-defined type of data-type identifier.

(1) Predefined data type

The following table lists the predefined data types.

Table 1-3: Predefined data types

Classification	Data type#1	Data format	Description
Numeric data	INT[EGER]	Integer (4-byte binary)	Integer value in the range -2147483648 to 2147483647
	SMALLINT	Integer (2-byte binary)	Integer value in the range -32768 to 32767

Classification	Data type[#1]	Data format	Description
	[LARGE]DEC[IMAL] [(*m*[, *n*])] or NUMERIC [(*m*[, *n*])]	Fixed-point number (packed decimal with ↑(*m* + 1) ÷ 2↑ bytes)[#8]	Fixed-point number whose precision (total number of digits) is *m* and whose decimal scaling position (number of digits following the decimal point) is *n*, where *m* and *n* are positive integers such that $1 \leq m \leq 38$, $0 \leq n \leq 38$, $n \leq m$. The default for *m* is 15; the default for *n* is 0.
	FLOAT or DOUBLE PRECISION	Double-precision floating-point number (8 bytes)	Double-precision floating-point number with a value of approximately $\pm 4.9 \times 10^{-324}$ to $\pm 3.4 \times 10^{308}$ [#4]
	SMALLFLT or REAL	Single-precision floating-point number (4 bytes)	Single-precision floating-point number with a value of approximately $\pm 1.4 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$ [#4]
Character data[#10]	CHAR[ACTER] [(*n*)][CHARACTER SET *character-set-specification*]	Fixed-length character string (length: *n* bytes)	• If no character set is specified: Fixed-length (*n* bytes) string of one-byte characters. • If a character set is specified: Fixed-length (*n* bytes) string of characters from that character set. *n* is a positive integer such that $1 \leq n \leq 30{,}000$. The default for *n* is 1. Attributes for the character data are specified by *character-set-specification*. For details, see *1.3 Character sets*.

Classification	Data type[#1]	Data format	Description
	[LONG] VARCHAR(*n*) [CHARACTER SET *character-set-specification*] or CHAR[ACTER] VARYING(*n*) [CHARACTER SET *character-set-specification*]	Variable-length character string (maximum length: *n* bytes)	• If no character set is specified: Variable-length (maximum length of *n* bytes) character string of one-byte characters. • If a character set is specified: Variable-length (maximum length of *n* bytes) string of characters from that character set. *n* is a positive integer such that 1 ≤ *n* ≤ 32,000. The actual length is 0 or greater. Attributes for the character data are specified by *character-set-specification*. For details, see *1.3 Character sets*.
National character data[#6, #9]	NCHAR[(*n*)] or NATIONAL CHAR[ACTER] [(*n*)]	Fixed-length national character string (length: *n* characters)	Fixed-length national character string of two-byte characters with a length of *n* characters (2*n* bytes), where *n* is a positive integer such that 1 ≤ *n* ≤ 15,000. The default for *n* is 1.
	[LONG] NVARCHAR(*n*) or NATIONAL CHAR[ACTER] VARYING(*n*) or NCHAR VARYING(*n*)	Variable-length national character string (maximum length: *n* characters)	Variable-length national character string of two-byte characters with a maximum length of *n* character (2*n* bytes), where *n* is a positive integer such that 1 ≤ *n* ≤ 16,000. The real length is 0 or greater.
Mixed character data[#6]	MCHAR [(*n*)]	Fixed length, mixed character string (length: *n* bytes)	This is a fixed length, mixed character string with a length of *n* bytes containing both one-byte characters and two-byte characters, where *n* is a positive integer, 1 ≤ *n* ≤ 30,000. The default for *n* is 1.
	[LONG] MVARCHAR(*n*)	Variable length, mixed character string (maximum length: *n* bytes)	This is a variable length, mixed character string with a maximum length of *n* bytes containing both one-byte characters and two-byte characters, where *n* is a positive integer, 1 ≤ *n* ≤ 32,000. The real length is 0 or greater.

Classification	Data type[#1]	Data format	Description
Date data	`DATE`	Date (4-byte, unsigned, packed format, *YYYYMMDD*) *YYYY*: `0001` to `9999` (year) *MM*: `01` to `12` (month) *DD*: `01` to the last day of the specified month of the specified year (day)	Date represented by the year, month, and day.
Time data	`TIME`	Time (3-byte, unsigned, packed format, *hhmmss*) *hh*: `00` to `23` (hour) *mm*: `00` to `59` (minute) *ss*: `00` to `59` (second)[#11]	Time represented by the hour, minute, and second.
Time stamp data	`TIMESTAMP[(`*p*`)]`	Time stamp (unsigned packed format 7 to 10 bytes long, *YYYYMMDDhhmmss*`[`*nn*`...`*n*`]`) *YYYY*: `0001` to `9999` (year) *MM*: `01` to `12` (month) *DD*: `01` to the last day of the specified month of the specified year (day) *hh*: `00` to `23` (hour) *mm*: `00` to `59` (minute) *ss*: `00` to `59` (second)[#11] *nn...n*: fractional seconds in *p* digits (*n*: `0` to `9`)	This is the time stamp data type having six areas: year, month, day, hour, minute, and second. *p* is an integer such that *p*=0, 2, 4, or 6. The default is *p*=0.
Date interval data	`INTERVAL YEAR TO DAY`	Date interval (5-byte, packed format, 0*YYYYMMDDs*) *YYYY*: `0000` to `9999` (number of years) *MM*: `00` to `99` (number of months)[#2] *DD*: `00` to `99` (number of days) *s*: Sign (positive: `C`, `F`; negative: `D`)[#8]	Interval between two dates, in the range -9999 years, 11 months, 99 days to 9999 years, 11 months, 99 days.

Classification	Data type[#1]	Data format	Description
Time interval data	INTERVAL HOUR TO SECOND	Time interval (4-byte, packed format, 0*hhmmsst*) *hh*: 00 to 99 (hours) *mm*: 00 to 99 (minutes)[#3] *ss*: 00 to 99 (seconds)[#3] *t*: Sign (positive: C, F; negative: D)[#8]	Interval between two times, in the range -99 hours, 59 minutes, 59 seconds to 99 hours, 59 minutes, 59 seconds.
Large object data	BLOB [(*n*[{K|M|G}])] or BINARY LARGE OBJECT [(*n*[{K|M|G}])]	Binary data string (maximum length: *n* bytes) K: Kilobytes M: Megabytes G: Gigabytes	This is a binary data string with a maximum length of *n* bytes. The default for *n* is 2,147,483,647 bytes. The real length is 0 or greater. Units K, M, and G can be specified[#5]. If a unit (K, M, G) is omitted, the length is assumed to be in bytes by default.
Binary data	BINARY(*n*)	Binary data string (maximum length: *n* bytes)	This is a binary data string with a maximum length of *n* bytes, where *n* is required. The actual length is 0 or greater. *n* is a positive integer such that $1 \leq n \leq$ 2,147,483,647 bytes.
Logical data	BOOLEAN[#7]	Logical value (4 bytes)	Permissible logical values are TRUE, FALSE, and UNKNOWN.

Note 1

The following shows the single-byte characters that are used for any whitespace characters when comparing or converting character data and mixed character data:

Data type	Character set specification	Space character code
Character data	Default	X'20'
	EBCDIK	X'40'
	UTF16	X'0020'[#]
Mixed character data		X'20'
National character data		Depends on the character set being used

#

To use data encoded in UTF-16 in the ? parameter, specify the character set

name in the character set descriptor area.

Specifying UTF-16 data handling in the preprocessing options or embedded variable definitions allows data encoded in UTF-16 to also be used in embedded variables.

The possible values for the character set name are as follows:

- `UTF16`
- `UTF-16BE`
- `UTF-16LE`

The space character codes used for the specifiable character set names are as follows:

- `UTF16` or `UTF-16BE`

  `X'0020'`
- `UTF-16LE`

  `X'2000'`

For details about UTF-16LE and UTF-16BE (information on character sets that can be set in the character set descriptor area (SQLCSN)), see the *HiRDB Version 9 UAP Development Guide*.

Note 2

See the *Permissible characters in SQL* column in *Table 1-1* for the character codes with the available one-byte and two-byte characters.

#1: If there are multiple data types for a data format, the typical data type is used thereafter in this manual as a representative example.

#2: When the number 12 or greater is used, the year is incremented by 1.

#3: When the number 60 or greater is specified in mm or ss, the hour or minute, respectively, is incremented by 1.

#4: The allowable range of a floating-point number is limited by the data representation available on the hardware used to execute the SQL.

#5: The following table lists, for each unit of length, the allowable range of values and the maximum length.

Unit	Allowable range of n	Actual maximum length (bytes)
`K`	$1 \leq n \leq 2097152$	*n* x 1024
`M`	$1 \leq n \leq 2048$	*n* x 1048576

Unit	Allowable range of n	Actual maximum length (bytes)
G	$1 \leq n \leq 2$	*n* x 1073741824

If the calculated value of an actual maximum length is 2147483648, the value is reduced to 2147483647.

#6: When lang-c is specified in the pdsetup command as the character codes type, national character data and mixed character data cannot be defined (UNIX edition only).

#7: BOOLEAN can be used only as the data type for a function that provides a return value; it cannot be used as the data type for a column, SQL variable, or SQL parameter.

#8: For details about the sign part of the decimal, date interval, and time interval types, see *1.2.4 Notes on using the decimal type*.

#9: If you specify utf-8 or chinese-gb18030 as the character encoding type in the pdntenv command (pdsetup command in the UNIX edition), national character data cannot be defined.

#10

- If you specify a character encoding type other than sjis in the pdntenv command (pdsetup command in the UNIX edition), you cannot use character data for which EBCDIK is specified as the *character-set-specification*.
- If you specify a character encoding type other than utf-8 in the pdntenv command (pdsetup command in the UNIX edition), you cannot use character data for which UTF16 is specified as the *character-set-specification*.

  When *character-set-specification* is set to UTF16, *n* must be specified as a multiple of 2.

#11: The range of *ss* is 00 to 61 (seconds) if you set the pd_leap_second operand to allow leap seconds to be specified. For details about the pd_leap_second operand, see the manual *HiRDB Version 9 System Definition*.

(2) User-defined type

The following table describes the user-defined type.

Table 1-4: User-defined type

Data type	Data format	Explanation
Abstract data type	Not applicable	Data type defined by CREATE TYPE. Attribute definitions and routines can be defined in the data type.

1.2.2 Data types that can be converted (assigned or compared)

The table below indicates data types that can be converted (assigned or compared). For data types that can be converted by using a CAST specification or a scalar function, see *2.16 Scalar functions* and *2.25 CAST specification*, respectively.

Table 1-5: Data types that can be converted (assigned or compared) (1/2)

Data type before conversion		Data type after conversion								
		Numeric data	**Character data**			**National character data**	**Mixed character data**	**Date data**	**Time data**	**Time stamp data**
		INTEGER	**CHARACTER**			**NCHAR**	**MCHAR**	**DATE**	**TIME**	**TIMESTAMP**
		SMALLINT								
		DECIMAL	**VARCHAR**			**NVARCHAR**	**MVARCHAR**			
		FLOAT								
		SMALLFLT	**DF**	**EK**	**U16**					
Numeric data • INTEGER • SMALLINT • DECIMAL • FLOAT • SMALLFLT		Y, Y	Y, N[#1]			N, N	Y, N[#1]	N, N	N, N	N, N
Character data • CHARACTER • VARCHAR	DF	C, C[#2, #3]	Y, Y	C, C [#14]	C, C [#17]	C, C[#7]	Y, Y	C, C[#4]	C, C[#5]	C, C[#6]
	EK		C, N [#15]	Y, Y	N, N	N, N	N, N			
	U16		C, C[#16]	N, N	Y, Y	N, N	C, C [#16]			
National character data • NCHAR • NVARCHAR		N, N	N, N			Y, Y	N, N	N, N	N, N	N, N

Data type before conversion	Data type after conversion								
	Numeric data	Character data			National character data	Mixed character data	Date data	Time data	Time stamp data
	INTEGER	CHARACTER			NCHAR	MCHAR	DATE	TIME	TIMESTAMP
	SMALLINT								
	DECIMAL	VARCHAR			NVARCHAR	MVARCHAR			
	FLOAT								
	SMALLFLT	DF	EK	U16					
Mixed character data • MCHAR • MVARCHAR	C, C#2, #3	Y, Y	N, N	C, C #17	N, N	Y, Y	N, N	N, N	N, N
Date data • DATE	N, N	N, C#8			N, N	N, N	Y, Y	N, N	N, N
Time data • TIME	N, N	N, C#9			N, N	N, N	N, N	Y, Y	N, N
Time stamp data • TIMESTAMP	N, N	N, C#10			N, N	N, N	N, N	N, N	Y, Y#11
Date interval data • INTERVAL YEAR TO DAY	N, C#12	N, N			N, N	N, N	N, N	N, N	N, N
Time interval data • INTERVAL HOUR TO SECOND	N, C#13	N, N			N, N	N, N	N, N	N, N	N, N
Large object data • BLOB	N, N	N, N			N, N	N, N	N, N	N, N	N, N
Binary data • BINARY(*n*) $1 \leq n \leq 32{,}000$	N, N	N, N			N, N	N, N	N, N	N, N	N, N

Data type before conversion	Data type after conversion								
	Numeric data	Character data			National character data	Mixed character data	Date data	Time data	Time stamp data
	INTEGER	CHARACTER			NCHAR	MCHAR	DATE	TIME	TIMESTAMP
	SMALLINT								
	DECIMAL	VARCHAR			NVARCHAR	MVARCHAR			
	FLOAT								
	SMALLFLT	DF	EK	U16					
Binary data • BINARY(*n*) *n* > 32,000	N, N	N, N			N, N	N, N	N, N	N, N	N, N
Boolean data • BOOLEAN	N, N	N, N			N, N	N, N	N, N	N, N	N, N
Abstract data type	N, N	N, N			N, N	N, N	N, N	N, N	N, N

Legend:

Y: Can be converted.

C: Can be converted, subject to restrictions.

N: Cannot be converted.

DF: Default character set

EK: EBCDIK

U16: UTF-16, UTF-16LE, or UTF-16BE

You must specify the character set names `UTF-16LE` or `UTF-16BE` in the following cases:

- When handling data whose character encoding is set to UTF-16 using the `?` parameter
- When specifying UTF-16 data handling in the preprocessing options or embedded variable definitions in order to handle data encoding as UTF-16 in embedded variables

For details about the UTF-16LE and UTF-16BE character sets (information on character sets that can be set in the character set descriptor area (SQLCSN)), see the *HiRDB Version 9 UAP Development Guide*.

Note

In the table, whether a given data type is convertible, is indicated in the form "*assignable*, *comparable*". For example, the entry "Y, Y" means that the data type can be assigned and compared, whereas "Y, N" means that it can be assigned but cannot be compared.

#1: The correspondence between numeric and character data, when they are converted, is given below. For any converted data type that is mixed character data, `VARCHAR` reads `MVARCHAR`, and `CHAR` reads `MCHAR`.

- `INTEGER`-type numeric data is converted into `VARCHAR(11)` character data.
- `SMALLINT`-type numeric data is converted into `VARCHAR(6)` character data.
- `DECIMAL(`*p*`,0)`-type numeric data is converted into `VARCHAR(`*p*`+1)` character data.
- `DECIMAL(`*p*`,`*s*`)`-type numeric data is converted into `VARCHAR(`*p*`+2)` character data.
- `FLOAT`-type or `SMALLFLT`-type numeric data is converted into `CHAR(23)` character data.

For `INTEGER`-type, `SMALLINT`-type, or `DECIMAL`-type numeric data, the converted character data does not include a positive sign. Conversion of `FLOAT`-type or `SMALLFLT`-type numeric data into character data always appends a sign to the characteristic and the mantissa.

#2: Comparison of character data with numeric data involves a conversion of the character data into numeric data. Character data can be compared with numeric data only if the character data is present in a comparison predicate, quantified predicate, `IN` predicate, or `BETWEEN` predicate. The conditions under which a comparison can be made for each predicate are summarized in the following table:

Predicate	Comparable condition
Comparison predicate	A comparison can be made if either the right- or left-hand side of the comparison operator is numeric data.
`IN` predicate, quantified predicate	A comparison can be made if the left-hand side of the comparison operator is numeric data.
`BETWEEN` predicate	A comparison can be made if the row value constructor element in *row value constructor 1* is numeric data.

Character data can be compared with numeric data if any of the following items

is specified:

Character data

- Literal (including the case where the literal is specified in a selection expression in a subquery)
- `?` parameter
- Embedded variable
- SQL variable
- SQL parameter

However, if either character or numeric data is specified in any of the following positions, the character data specified in an SQL variable or SQL parameter cannot be converted to numeric data:

- A row value constructor element in a row value constructor having two or more row value constructor elements
- A selection expression in a row or table subquery
- A selection expression in a scalar subquery that is specified in a position other than the right-hand side of a comparison predicate, `IN` predicate, or quantified predicate

Numeric data

- Column specification (including the cases in which numeric data is specified in a selection expression in a subquery)

#3: The correspondence relationship for assignment or comparison of character data with numeric data is as follows:

- Integer representations using a character string are converted into the `INTEGER` type.
- Decimal representations using a character string are converted into the `DECIMAL` type.
- Floating-point numeric representations using a character string are converted into the `FLOAT` type.

A space occurring before or after a numeric character string representation is ignored during the conversion process.

#4: The following items can be assigned to and compared with date data:

- A literal in which a date is represented in a predefined character string
- An embedded variable corresponding to `CHAR(10)` (or an embedded variable corresponding to `CHAR(20)` if the embedded variable is encoded in

UTF-16, UTF-16LE, or UTF-16BE)

#5: The following items can be assigned to and compared with time data:

- A literal in which time is represented in a predefined character string
- An embedded variable corresponding to `CHAR(8)` (or an embedded variable corresponding to `CHAR(16)` if the embedded variable is coded in UTF-16, UTF-16LE, or UTF-16BE)

#6: The following items can be assigned to and compared with time stamp data:

- A literal in which a time stamp is represented in a predefined character string
- An embedded variable corresponding to a `CHAR` variable with a length of 19 to 26 bytes (or an embedded variable corresponding to a `CHAR` variable with a length that is of 38 to 52 bytes and is a multiple of 2 if the embedded variable is coded in UTF-16, UTF-16LE, or UTF-16BE)

#7: Character string literals coded in any of the following items, exclusive of definition SQL statements, are treated as national character string literals, for which only the length of the character data is checked; the character code is not checked:

- `INSERT` statement
- `UPDATE` statement
- Search condition statement

For details about the `INSERT` and `UPDATE` statements, see *4. Data Manipulation SQL*. For details about search conditions, see *2.7 Search conditions*.

#8: The following items can be compared with date data:

- A literal in which a date is represented in a predefined character string
- An embedded variable corresponding to `CHAR(10)` (or an embedded variable corresponding to `CHAR(20)` if the embedded variable is encoded in UTF-16, UTF-16LE, or UTF-16BE)

The following item can be converted into character data:

- Date data that is assigned to a `CHAR` or `VARCHAR` embedded variable with a minimum length of 10 bytes (or a `CHAR` or `VARCHAR` embedded variable with a length that is a minimum of 20 bytes and is a multiple of 2 if the embedded variable is coded in UTF-16, UTF-16LE, or UTF-16BE)

#9: The following items can be compared with time data:

- A literal in which time is represented in a predefined character string
- An embedded variable corresponding to `CHAR(8)` (or an embedded variable corresponding to `CHAR(16)` if the embedded variable is coded in UTF-16, UTF-16LE, or UTF-16BE)

The following item can be converted into character data:

- Time data that is assigned to a `CHAR` or `VARCHAR` embedded variable with a minimum length of 8 bytes (or a `CHAR` or `VARCHAR` embedded variable with a length that is a minimum of 16 bytes and is a multiple of 2 if the embedded variable is coded in UTF-16, UTF-16LE, or UTF-16BE)

#10: The following items can be compared with time stamp data:

- A literal in which a time stamp is represented in a predefined character string
- An embedded variable corresponding to a `CHAR` variable with a length of 19 to 26 bytes (or an embedded variable corresponding to a `CHAR` variable with a length that is of 38 to 52 bytes and is a multiple of 2 if the embedded variable is coded in UTF-16, UTF-16LE, or UTF-16BE)

In addition, the following item can be assigned to character data:

- Assignment to a `CHAR` or `VARCHAR` embedded variable with a minimum length of 19 bytes if $p = 0$, or with a minimum length of $20 + p$ bytes if $p > 0$ where p denotes the fractional second precision of the time stamp data before conversion (or, if the embedded variable is coded in UTF-16, UTF-16LE, or UTF-16BE, assignment to a `CHAR` or `VARCHAR` embedded variable with a length that is a minimum of 38 bytes if $p = 0$, or $40 + 2p$ bytes if $p > 0$, and that is a multiple of 2, where p denotes the fractional second precision of the time stamp data before conversion)

#11: If the source of assignment has a fractional second precision higher than that of the target of assignment, the fractional line part is truncated to match the precision of the target of assignment. If the source of assignment has a fractional second precision lower than that of the target of assignment, data is stored by zero-filling the expanded fractional second part to match the precision of the target of assignment.

#12: The following items can be compared with date interval data:

- A literal in which a date interval is represented in a decimal
- An embedded variable corresponding to `DECIMAL(8,0)`

The following item can be converted into decimal data:

- Date interval data that is assigned to a `DECIMAL(8,0)` embedded variable

#13: The following items can be compared with time interval data:

- A literal in which a time interval is represented in a decimal.
- An embedded variable corresponding to `DECIMAL(6,0)`

The following item can be assigned to decimal data:

- Time interval data that is assigned to a `DECIMAL(6,0)` embedded variable

#14

Before conversion, assignment or comparison can be performed on the following items in order to convert them to the target character set:

- Character string literal
- Embedded variable (default character set)

#15

After conversion, assignment can be performed on the following item in order to convert it to the target character set:

- Embedded variable (default character set)

#16

Before conversion, assignment or comparison can be performed on the following item in order to convert it to the target character set:

- Embedded variable (default character set)

Before conversion, assignment or comparison can be performed on the following item in order to convert it to the target default character set:

- Embedded variable (UTF-16, UTF-16LE, or UTF-16BE)

#17

Before conversion, assignment or comparison can be performed on the following items in order to convert them to the target character set:

- Character string literal
- Embedded variable (default character set)

After conversion, assignment can be performed on the following item in order to convert it to the target character set:

- Embedded variable (UTF-16, UTF-16LE, or UTF-16BE)

Table 1-6: Data types that can be converted (assigned or compared) (2/2)

Data type before conversion		Data type after conversion						
		Date interval data	Time interval data	Large object data	Binary data		Boolean data	Abstract data type
		INTERVAL YEAR TO DAY	INTERVAL HOUR TO SECOND	BLOB	BINARY(n) 1 ≦ n ≦ 32,000	BINARY(n) n > 32,000	BOOLEAN	
Numeric data • INTEGER • SMALLINT • DECIMAL • FLOAT • SMALLFLT		C, C[#1]	C, C[#2]	N, N	N, N	N, N	N, N	N, N
Character data • CHARACTER • VARCHAR	DF	N, N	N, N	N, N	C, C[#3]	N, N	N, N	N, N
	EK				N, N			
	U16							
National character data • NCHAR • NVARCHAR		N, N	N, N	N, N	N, N	N, N	N, N	N, N
Mixed character data • MCHAR • MVARCHAR		N, N	N, N	N, N	N, N	N, N	N, N	N, N
Date data • DATE		N, N	N, N	N, N	N, N	N, N	N, N	N, N
Time data • TIME		N, N	N, N	N, N	N, N	N, N	N, N	N, N
Time stamp data • TIMESTAMP		N, N	N, N	N, N	N, N	N, N	N, N	N, N
Date interval data • INTERVAL YEAR TO DAY		Y, Y[#4]	N, N	N, N	N, N	N, N	N, N	N, N

Data type before conversion	Data type after conversion						
	Date interval data	Time interval data	Large object data	Binary data		Boolean data	Abstract data type
	INTERVAL YEAR TO DAY	INTERVAL HOUR TO SECOND	BLOB	BINARY(n) 1 ≦ n ≦ 32,000	BINARY(n) n > 32,000	BOOLEAN	
Time interval data • INTERVAL HOUR TO SECOND	N, N	Y, Y[#5]	N, N	N, N	N, N	N, N	N, N
Large object data • BLOB	N, N	N, N	Y, N	Y, N	Y, N	N, N	N, N
Binary data • BINARY(*n*) 1 ≦ *n* ≦ 32,000	N, N	N, N	Y, N	Y, Y	Y, N	N, N	N, N
Binary data • BINARY(*n*) *n* > 32,000	N, N	N, N	Y, N	Y, N	Y, N	N, N	N, N
Boolean data • BOOLEAN	N, N	N, N	N, N	N, N	N, N	N, N	N, N
Abstract data type	N, N	N, N	N, N	N, N	N, N	N, N	C, N[#6]

Legend:

Y: Can be converted.

C: Can be converted, subject to restrictions.

N: Cannot be converted.

DF: Default character set

EK: EBCDIK

U16: UTF-16, UTF-16LE, or UTF-16BE

Note

In the table, whether a given data type is convertible is indicated in the form of "*assignable*, *comparable*." For example, the entry "Y, Y" means that the data type can be assigned and compared, whereas "Y, N" means that it can be assigned but cannot be compared.

#1: The following items can be assigned to and compared with date interval data:

- A literal in which a date interval is represented in a decimal
- An embedded variable corresponding to `DECIMAL(8,0)`

#2: The following items can be assigned to and compared with time interval data:

- A literal in which a time interval is represented in a decimal
- An embedded variable corresponding to `DECIMAL(6,0)`

#3: The following items can be assigned to and compared with the `BINARY` type:

- Hexadecimal character string literals

#4: Date interval data items are compared in the following order: year, month, and day.

#5: Time interval data items are compared in the following order: hour, minute, and second.

#6: The abstract data type `ADT1` can be assigned to a column or variable defined on an abstract data type `ADT2` if `ADT1` is a value of `ADT2` or a subtype value of the same.

1.2.3 Notes on using character data, national character data, and mixed character data

The following table provides notes on the use of character data, national character data, and mixed character data.

Table 1-7: Notes on use of character data, national character data and mixed character data

Item	Data type			
	CHAR, NCHAR or MCHAR		VARCHAR, NVARCHAR or MVARCHAR	
	Defined length			
	1-255 (1-127)	256-30000 (128-15000)	1-255 (1-127)	256-32000 (128-16000)
Index definition	Y	Y#	Y	Y#
Sorting	Y	Y	Y	Y
Grouping	Y	Y	Y	Y

Item	Data type			
	CHAR, NCHAR or MCHAR		VARCHAR, NVARCHAR or MVARCHAR	
	Defined length			
	1-255 (1-127)	256-30000 (128-15000)	1-255 (1-127)	256-32000 (128-16000)
Set functions	Y	Y	Y	Y
Search conditions	Y	Y	Y	Y
Data insertion/updating	Y	Y	Y	Y
Exclusion of duplicates	Y	Y	Y	Y
Set operations	Y	Y	Y	Y

Y: Can be used.

( ): In the *Defined length* row, the values in parentheses are for national character data.

#: The total length of a column comprising an index must satisfy the following formula:

Total column length ≤

MIN((*page-size-of-index-storage-RDAREA* ÷ 2) - 1242, 4036)

1.2.4 Notes on using the decimal type

The following signs can be used in the decimal, date interval, and time interval types:

Sign value	Meaning
`X'A'`	Not recognized as a sign (error)#
`X'B'`	Not recognized as a sign (error)#
`X'C'`	Recognized as the positive sign
`X'D'`	Recognized as the negative sign
`X'E'`	Not recognized as a sign (error)#
`X'F'`	Recognized as the positive sign

#: This is recognized as a sign when `Y` is specified in the `pd_dec_sign_normalize` operand of the system definition. For details about the `pd_dec_sign_normalize` operand, see the manual *HiRDB Version 9 System Definition*.

When N is specified in the pd_dec_sign_normalize operand or this operand is omitted, the signs of stored data are not normalized. In such a case, the signs of stored data items may contain X'C' and X'F', both of which are recognized as the positive sign. Signs may also change during the type conversion and operation processes at the time of SQL statement execution.

To standardize the signs in a database that contains mixed sign codes, it may be necessary to reload the table data.

For details about the facility for conversion to a decimal signed normalized number, see the explanation of the facility in the *HiRDB Version 9 System Operation Guide*.

If the data stored in a decimal-type column is positive but the signs of the data are not standardized to X'C', column retrieval operations may yield a discrepancy between the signs of the data and those of retrieval results. When you create a UAP with sign codes in mind, you need to take the following items into consideration:

(1) Signs in retrieval results for a decimal-type column defining an index

When a decimal-type column for which an index is defined is retrieved, signs in the retrieval results are handled as follows:

- When all signs in the stored data are X'C':

  All signs in the retrieval results will be X'C'.

- When all signs in the stored data are X'F':

  If the index defined for the decimal-type column is a single-column index, all signs in the retrieval results will be X'F'.

  If the index is a multi-column index containing the decimal-type columns, the retrieval results may include X'C' as a sign code.

- When the signs in the stored data are both X'C' and X'F':

  The signs in the stored data and the retrieval results may not agree.

(2) Signs in retrieval results when the retrieval (in which a GROUP BY clause is specified) is performed by specifying a decimal-type grouping column in a selection expression

When a decimal-type grouping column is specified in a selection expression for a retrieval for which a GROUP BY clause is specified, signs in the retrieval results are handled as follows:

- When all signs in the stored data in a selection expression are X'C':

  All signs in the retrieval results will be X'C'.

- When all signs in the stored data in a selection expression are X'F':

  If rapid grouping is selected in the pd_optimize_level operand of the system

common definition, the retrieval results may include `X'C'` as a sign code.

If rapid grouping is not selected, all signs in the retrieval results will be `X'F'`.

- When the signs in the stored data include both `X'C'` and `X'F'`:

  The signs in the retrieval results will be either `X'C'` or `X'F'`.

(3) Signs in retrieval results when an SQL extension optimizing option is specified

When the *hash join, hash execution of a subquery* is specified as an SQL extension optimizing option, a retrieval specifying either queries or subqueries in which multiple tables are joined by = will result in the following signs:

- When all signs in the stored data are `X'C'`:

  All signs in the retrieval results will be `X'C'`.

- When all signs in the stored data are `X'F'`:

  The retrieval results may include `X'C'` as a sign code.

- When the signs in the stored data include both `X'C'` and `X'F'`:

  Signs in the retrieval results may be `X'C'` even when the signs in the stored data are `X'F'`.

1.2.5 Notes on using large-object data

When using large-object data, be aware of the following restrictions:

(1) Items in which large-object data is not allowed

Large-object data cannot be used in any of the following items:

- Index definition
- Sort
- Grouping
- Set operation or arithmetic operation
- Set functions
- Duplicate elimination
- `CASE` expressions
- `CAST` specification (`CAST(NULL AS BLOB` type`)` can be specified)
- Reference column to the outside from within an argument for a user-defined function

Large object data specified in a definition as an attribute of an abstract data type can

be specified in an index definition by using a plug-in function. Concatenation operations on large-object data can be used only on updated values in the SET clause of an UPDATE statement.

(2) Scalar functions

The only scalar functions in which large-object data can be specified are the LENGTH, SUBSTR, and POSITION functions.

(3) Data insertion and updating

A literal cannot be used in the INSERT statement or UPDATE statement to insert or update large-object data. The following items can be used to insert large-object data: an embedded variable, a ? parameter, an SQL variable, an SQL parameter, the SUBSTR scalar function, a function call, and NULL. The following items are used to update large-object data: a column specification, a component specification, an embedded variable, a ? parameter, an SQL variable, an SQL parameter, a concatenation operation, the SUBSTR scalar function, a function call, a subquery, and NULL.

(4) Using a locator

When handling large object data in a UAP, the use of a locator allows you to process the SQL that handles large object data without storing the data itself on a client. For a description of locators, see *1.13 Locator*.

1.2.6 Notes on using the BINARY type

(1) Items in which the BINARY type cannot be used

The BINARY type cannot be used in any of the following items:

- An index definition
- An external reference column

In addition, the following table lists items for which your ability to specify the BINARY type changes depending on its defined length.

Table 1-8: Items in which specifiable/not specifiable changes in a BINARY type defined length

Item	Defined length		
	1 to 255	256 to 32,000	32,001 to 2,147,483,647
Sorting	Y	Y	N
Grouping	Y	Y	N
Set function	Y	Y	N
Search condition	Y	Y	N

Item	Defined length		
	1 to 255	256 to 32,000	32,001 to 2,147,483,647
Data insertion/updating#1	Y	Y	Y
Duplicate elimination	Y	Y	N
Set operations	Y	Y	N
Concatenation operation	Y	Y	Y#2
Scalar functions	Y	Y	Y#3
CASE expression	Y	Y	N
CAST specification	Y	Y	N

Legend:

Y: Can be used.

N: Cannot be used.

#1: When specifying a literal in an update or insertion value for a BINARY-type column, you can specify only a hexadecimal character string literal.

#2: BINARY-type concatenation operations with a maximum length of greater than or equal to 32,001 bytes can be used only on update values for the SET clause of the UPDATE statement.

#3: The only scalar functions for which a BINARY type with a defined length of 32,001 bytes or greater can be specified are the LENGTH, SUBSTR, and POSITION functions; on other scalar functions, the BINARY type cannot be used with a defined length of 32,001 bytes or greater.

(2) Using a locator

When handling the BINARY type in a UAP, you can use a locator to allow you to process the SQL that handles BINARY type data without storing the data itself on a client. For a description of locators, see *1.13 Locator*.

1.2.7 Notes on using logical data

Logical data can be used only as a function return value.

1.2.8 Notes on using an abstract data type

An abstract data type cannot be used in any of the following items:

- Definition of an index

- Sorting
- Grouping
- Set, arithmetic or concatenation operations
- Set functions
- Duplicates elimination
- `CASE` expressions
- `CAST` specification
- External reference column

An abstract data type can be used for definition of an index by using a plug-in function.

1.3 Character sets

(1) Description

A character set defines the properties of character data, based on the following three attributes:

- Usage format

  The rules by which characters are represented.

- Character repertoire

  The set of characters that can be represented.

- Default collation sequence

  The rules by which the data in two different strings are compared.

  Every character set comes with a default collation sequence.

(2) Format

character-set-specification ::= [MASTER.] *character-set-name*

character-set-name ::= {EBCDIK|UTF16}

(3) Rules

1. The following table lists the character sets that are available in HiRDB.

   Table 1-9: Character sets available in HiRDB

Character set name	Usage format	Character repertoire	Default collation sequence
EBCDIK	EBCDIK code. Characters are represented by 8-bit (single-byte) character codes.	All EBCDIK-encoded characters	Code ordering based on bit combinations
UTF16	Characters are represented in the character encoding format defined by JIS X 0221 (ISO/IEC 10646), in which each character is encoded as two or four bytes. Byte order is big-endian.	All Unicode characters	Code ordering based on bit combinations

2. EBCDIK can only be specified as the character set if sjis is specified as the

character code classification in the pdntenv command (pdsetup command in the UNIX edition).

3. UTF16 can only be specified as the character set if utf-8 is specified as the character code classification in the pdntenv command (pdsetup command in the UNIX edition).
4. A character set can be specified in any place where a character data type can be specified. A character set cannot be specified for the mixed character data type or national character data type.
5. If no character set is specified, the character set is determined by the character code classification specified in the pdntenv command (pdsetup command in the UNIX edition). The character set that is assumed when no character set is specified is called the default character set. The following table lists the default character set that is assumed based on the character code classification specified in the pdntenv command (pdsetup command in the UNIX edition).

Table 1-10: Default character sets for character codes specified in the pdntenv (pdsetup) command

Character code specified in command	Default character set
sjis	Shift JIS kanji code
chinese	EUC Chinese kanji code
ujis	EUC Japanese kanji code
utf-8	Unicode (UTF-8)
lang-c	Single-byte character code
chinese-gb18030	Chinese kanji code (GB18030)

(4) Notes

To use data encoded as UTF-16 in the ? parameter, specify the character set name in the character set descriptor area. Specifying UTF-16 data handling in the preprocessing options or embedded variable definitions allows data encoded as UTF-16 to also be used in embedded variables. In this case, the SQL preprocessor determines the character set name based on the specified preprocessing options and the embedded variable.

In addition to UTF16, either UTF-16LE or UTF-16BE can be specified as the character set name.

In the following descriptions, the UTF-16 character set name is assumed to include UTF-16LE and UTF-16BE.

For details about specifying the character set in preprocessing options, embedded

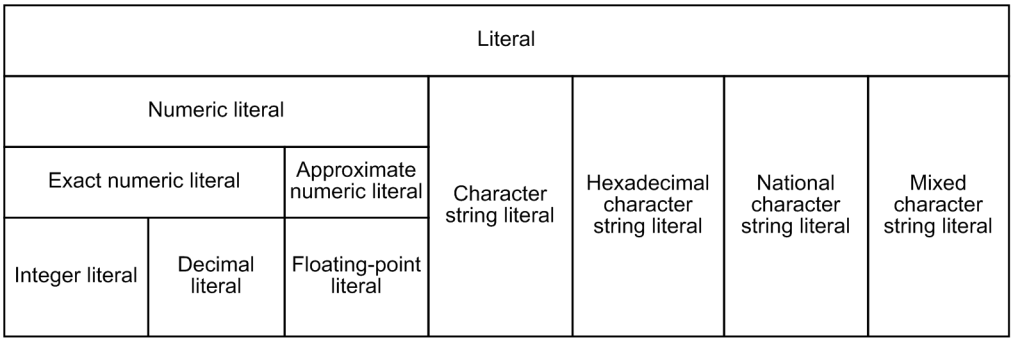
variable definitions, or the character set descriptor area, see the *HiRDB Version 9 UAP Development Guide*.

1.4 Literals

A literal is data whose value cannot be modified within the program. Literals can be numeric literals (which represent numbers) or character string literals, national character string literals, and mixed character string literals (which represent character strings).

The following figure shows the literals that can be specified in SQL.

Figure 1-4: Specifiable literals



The following table lists the notation used for literals.

Table 1-11: Notations for literals

Literal	Notation		Data type interpreted by HiRDB

Integer[#1]	[*sign*]*unsigned-integer*
	Examples:
	`-123`
	`45`
	`6789`
		Integer represented as a string of unsigned integers with the sign represented by `+` or `-`.	`INTEGER`

Decimal	[*sign*]*integer-part.fractional-part*
	Examples:
	`12.3`
	`-456.`
	`.789`
		An integer part and a fractional part represented as an unsigned integer. Either the integer part or the fractional part may be omitted; the decimal point cannot be omitted.	`DECIMAL`(*m*[,*n*]), where *m* and *n* denote numbers of digits.

Literal	Notation		Data type interpreted by HiRDB

Floating-point numeric	*mantissaExponent*
	Examples:
	`1.0E2`
	`.5E+67`
		Either an integer literal or a decimal literal as the mantissa, followed by a 1- to 3-digit integer literal as the exponent. The exponent represents a power of 10. The character `E` cannot be omitted.	`FLOAT`

Character string[#3]	`'`*character-string*`'`
	Examples:
	`'HITACHI'`
	`'88'`
	`'''95.7.30'`
		A character string is expressed by a string of 1-byte characters. To use an single quotation mark in a character string, two consecutive single quotation marks must be specified. The maximum length is 32,000 bytes.	`VARCHAR`(*n*), where *n* indicates the length of the character string.[#2]

Hexadecimal character string literal[#6]	`X'`*hexadecimal-character-string-literal*`'` Examples: `X'82A0'`, `X'82a0'`	A hexadecimal character string literal is represented with characters 0-9 and A-F (or a-f). The maximum allowable length of a hexadecimal character string literal is 64,000 characters, in a multiple of 2. Two hexadecimal characters make up one byte.	`VARCHAR(`*n*`)` where *n* is the indicated character string length ÷ 2.[#2]

National character string[#3, #4, #5]	`N'`*national-character-string*`'`
	Example:
	`N'SQL-syntax'`
		A character string is expressed by a string of 2-byte characters. The maximum length is 16,000 characters.	`NVARCHAR`(*n*), where *n* indicates the length of the character string.[#2]

Mixed character string[#3, #4]	`M'`character-string`'` Example: `M'1996'`	A character string is expressed by a string of 1- and 2-byte characters. The maximum length is 32,000 bytes.	`MVARCHAR`(*n*), where *n* indicates the length of the character string.[#2]

Note

Date and time values can be specified as a literal using a predefined character string representation. Similarly, date intervals and time intervals can be specified in SQL statements as decimal expression literals (decimal literals).

#1: If a literal that exceeds the allowable range of an integer literal is specified in the integer literal notation, HiRDB assumes the decimal point to be at the right side of the literal and interprets the literal to be a decimal literal.

#2: In the case of a character string literal (`' '`, `X' '`, `N' '`, or `M' '`) whose length is 0, *n* is 1.

#3: For character strings that can be specified in `COMMENT`, `EXECUTE IMMEDIATE`, and `PREPARE` statements, see the syntax section for each SQL statement.

#4: When `lang-c` is specified in the `pdsetup` command as the character codes type,

national character data and mixed character data cannot be defined (UNIX edition only).

#5: National character string literals cannot be used if you specified `utf-8` or `chinese-gb18030` as the character code classification in the `pdntenv` command (`pdsetup` command in the UNIX edition).

#6: Hexadecimal character string literals differ from character string literals only in the coding format. In this manual, any description on character string literals is also applicable to hexadecimal character string literals.

The following table lists restrictions on the use of numeric literals.

Table 1-12: Restrictions on use of numeric literals

Numeric literal type	Range	Maximum number of digits (including leading zeros)
Integer	-2147483648 to 2147483647	10
Decimal	$-(10^{39} - 1)$ to -10^{-38}, 0, 10^{-38} to $10^{39} - 1$	38
Floating-point numeric#	About -1.7×10^{308} to -4.9×10^{-324}, 0, About 4.9×10^{-324} to 1.7×10^{308}	Mantissa: 17 Exponent: 3

#: The allowable range of values is limited by the data representation available on the hardware used to execute the SQL.

1.4.1 Predefined character string representation of date data

Format: '*YYYY-MM-DD*'

A date can be made into a literal using a predefined character string representation by connecting the year (*YYYY*), month (*MM*), and day (*DD*) with hyphens (-), as '*YYYY-MM-DD*', where the year (*YYYY*), month (*MM*), and day (*DD*) fields must be zero-filled on the left as necessary.

Any date literal in a predefined character string representation is converted into the date data type when it is specified as an argument in the `DATE` scalar function or where date data is required.

An example of turning a date into a predefined character string representation is given as follows:

Example:

Date: July 30, 1995

Predefined character string representation of the date: `'1995-07-30'`

1.4.2 Predefined character string representation of time data

Format: '*hh*:*mm*:*ss*'

A time value can be made into a literal using a predefined character string representation by connecting the hour (*hh*), minute (*mm*), and second (*ss*) with colons (:), as '*hh*:*mm*:*ss*', where the hour (*hh*), minute (*mm*), and second (*ss*) fields must be zero-filled on the left as necessary.

Any time literal in a predefined character string representation is converted into the time data type when it is specified as an argument in the TIME scalar function or where time data is required.

An example of turning a time value into a predefined character string representation is given as follows:

Example:

Time: 11:3:58

Predefined character string representation of the time: '11:03:58'

1.4.3 Predefined character string representation of time stamp data

Format: '*YYYY*-*MM*-*DD hh*:*mm*:*ss*'

A time stamp can be made into a predefined character string representation by connecting the year (*YYYY*), month (*MM*), day (*DD*) with hyphens (-), space-filling any unused character positions, and then connecting the hour (*hh*), minute (*mm*), and second (*ss*) with colons (:), as '*YYYY*-*MM*-*DD hh*:*mm*:*ss*[*nn...n*]'. In this process, the year (*YYYY*), month (*MM*), and date (*DD*) fields must be zero-filled on the left as necessary. Similarly, the hour (*hh*), minute (*mm*), and second (*ss*) fields must be zero-filled on the left as necessary.

When representing a fractional second precision, use a period to connect the second (*ss*) to the fractional second (*nn...n*). If fractional second precision is omitted and only a period is specified, the fractional second precision is treated as zero data. A fractional second precision greater than 7 may cause an error.

Time stamp literals that are specified using a predefined character string representation can be specified as an argument in the TIMESTAMP scalar function or where time stamp data is required.

An example of turning a time stamp into a literal in a predefined character string representation is given as follows:

Example:

Time stamp: July 30, 1995, 11:3:58

Predefined character string representation of the time stamp: '1995-07-30 11:03:58'

1.4.4 Decimal representation of date interval data

Format: ± *YYYYMMDD*.

When making a date interval into a literal in decimal representation, the data is represented as ± *YYYYMMDD*. in terms of year (*YYYY*), month (*MM*), day (*DD*), and a sign (*s*), in fixed point with a precision of 8 and a scaling of 0.

Any date interval literal in decimal representation, when specified where date interval data is required, is converted into the date interval data type.

An example of turning a date interval into a literal in decimal representation is given as follows:

Example:

Representing an interval of 1 year, 1 month, and 1 day in decimal: `00010101`.

1.4.5 Decimal representation of time interval data

Format: ± *hhmmss*.

When making a time interval into a literal in decimal representation, the data is represented as ± *hhmmss*. in terms of hour (*hh*), minute (*mm*), second (*ss*), and a sign (*t*), in fixed point with a precision of 6 and a scaling of 0.

Any time interval literal in decimal representation, when specified where time interval data is required, is converted into the time interval data type.

An example of turning a time interval into a literal in decimal representation is given as follows:

Example:

Representing an interval of 1 hour, 1 minute, and 1 second in decimal: `010101`.

1.4.6 Decimal representation of datetime interval data

Format: ± *YYYYMMDDhhmmss*.

To represent a datetime interval in a decimal representation literal, represent the year (*YYYY*), month (*MM*), day (*DD*), hour (*hh*), minute (*mm*), second (*ss*), and a sign (*t*) as ± *YYYYMMDDhhmmss* in fixed point with a precision of 14 and a scaling of 0.

The following is an example of representing a datetime interval in a decimal representation literal:

Example:

Representing an interval of 1 year, 2 months, 3 days, 4 hours, 5 minutes, and 6 seconds in decimal: `00010203040506`.

1.5 USER, CURRENT_DATE value function, CURRENT_TIME value function, and CURRENT_TIMESTAMP value function

A USER, CURRENT_DATE value function, CURRENT_TIME value function, and CURRENT_TIMESTAMP value function can be used as a value specification in an SQL statement.

1.5.1 USER

(1) Function

This function indicates the authorization identifier of the execution user.

(2) Format

```
USER
```

(3) Rules

1. If USER is specified, HiRDB interprets it as specifying a VARCHAR(30) in the default character set.
2. USER cannot be used as a table name or the authorization identifier for an index name.

1.5.2 CURRENT_DATE value function

(1) Function

This function indicates the current date.

(2) Format

CURRENT_DATE *value-function* ::= {CURRENT_DATE|CURRENT DATE}

(3) Rules

1. If the CURRENT_DATE value function is specified, HiRDB interprets that the date data type (DATE) has been specified.
2. CURRENT_DATE represents the current date. The CURRENT_DATE function can be specified in the following items:
   - In a selection expression or a condition expression
   - As a value to be updated or inserted into a date data type
   - As a value to be updated or inserted into a CHAR(10) column (or into a CHAR(20) column when UTF16 is specified as the character set)

   When CURRENT_DATE is specified as an update or insertion into a

CHAR(10) column (or into a CHAR(20) column when UTF16 is specified as the character set), the update or insertion is performed after the current date is converted into the default string representation for dates.

3. Specifying CURRENT_DATE multiple times in an SQL statement produces the same value. Specifying CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP in combination in an SQL statement produces datetime values representing the same point in time.

1.5.3 CURRENT_TIME value function

(1) Function

Represents the current time converted from the localtime function of the OS. The time zone is determined based on the TZ operand of the system common definition.

(2) Format

CURRENT_TIME *value-function* ::= {CURRENT_TIME|CURRENT TIME}

(3) Rules

1. If the CURRENT_TIME value function is specified, HiRDB interprets that the time data type (TIME) has been specified.
2. CURRENT_TIME represents the current time. The CURRENT_TIME function can be specified in the following items:
   - In a selection expression or a condition expression
   - As a value to be updated or inserted into a time data type
   - As a values to be updated or inserted into a CHAR(8) column (or a CHAR(16) column when UTF16 is specified as the character set)

     When CURRENT_TIME is specified as an update or insertion into a CHAR(8) column (or into a CHAR(16) column when UTF16 is specified as the character set), the update or insertion is performed after the current time is converted into the default string representation for times.
3. Specifying CURRENT_TIME multiple times in an SQL statement produces the same value. Specifying CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP in combination in an SQL statement produces datetime values representing the same point in time.

1.5.4 CURRENT_TIMESTAMP value function

(1) Function

This function indicates the current time stamp.

(2) Format

```
CURRENT_TIMESTAMP value-function::= {CURRENT_TIMESTAMP
[(fractional-second-precision)]|CURRENT TIMESTAMP
[(fractional-second-precision)]}
```

(3) Rules

1. If the CURRENT_TIMESTAMP value function is specified, HiRDB interprets that the time stamp data type (TIMESTAMP) has been specified.
2. CURRENT_TIMESTAMP indicates the current time stamp. The CURRENT_TIMESTAMP function can be specified in the following items:
   - In a selection expression or a condition expression
   - As a value to be updated or inserted into a time stamp data type column
   - As a value to be updated or inserted into a CHAR string of length 19, 22, 24, or 26 bytes (or a CHAR string of length 38, 44, 48, or 52 bytes when UTF16 is specified as the character set)

   If a fractional second precision *p* (*p* = 0, 2, 4, or 6) is specified, HiRDB returns a time stamp including a fractional second with *p* effective digits to the right of the decimal point. The default is a fractional second precision of *p* = 0.

   If CURRENT_TIMESTAMP is specified as an update or insertion value for a CHAR column, the current time stamp is converted into a predefined character string representation, and the result is used for updating or insertion.

   The table below lists the defined lengths of the CHAR column that can be specified when specifying CURRENT_TIMESTAMP as an update or insertion value.

   Table 1-13: Correspondence between the value of fractional second precision p and the defined length of CHAR when specifying CURRENT_TIMESTAMP as an update or insertion value

Value of fractional second precision p	Defined length of CHAR	
	Default character set or a character set other than UTF-16	UTF-16 character set
0	19	38
2, 4, or 6	20 + *p*	40 + *p* x 2

3. Specifying CURRENT_TIMESTAMP multiple times in an SQL statement produces the same value. Specifying CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP in combination in an SQL statement produces datetime

values representing the same point in time.

1.6 Embedded variables, indicator variables, ? parameters, SQL parameters, and SQL variables

This section explains the variables and parameters that are used to pass and receive values in UAPs, SQL statements, external routines, and SQL routines.

- Embedded variables, indicator variables, ? parameters

  These variables and parameters can be specified in SQL statements in a UAP or external routine. When specified, these variables and parameters are used to pass and receive values between the SQL statement and the UAP.

- SQL variables

  These variables can be specified in SQL statements in an SQL routine. When specified, these variables are used to pass and receive values with the SQL routine. SQL variables cannot be specified in an external routine.

- SQL parameters

  SQL parameters can be specified when a routine to be called is defined. Specifying an SQL parameter allows the values to be passed between the CALL statement in the SQL (UAP or routine that calls a function) and the routine being called. In an SQL routine, the SQL parameter is specified directly; in an external routine, the specification is made to a variable in an external routine corresponding to that SQL parameter.

1.6.1 Embedded variables and indicator variables

(1) Format

:*embedded-variable* [:*indicator-variable*]

(2) Function, uses, and specifiable locations

The following table lists the functions, uses, and specifiable locations of embedded variables and indicator variables.

Table 1-14: Functions, uses, and specifiable locations of embedded variables and indicator variables

Function	Use		Specifiable locations
	Embedded variable	Indicator variable	
Receipt of a column value as a retrieval result	Receives a value (other than the null value).#1	Used in conjunction with an embedded variable to receive a column value, including the null value. In such a case, the indicator variable indicates whether the value read into the embedded variable is the null value. When character data, national character data, mixed character data, or large object data is received, the indicator variable indicates whether the data has been received correctly in the embedded variable.	`INTO` clause of `SELECT`, `FETCH`, `EXECUTE`, or `EXECUTE IMMEDIATE` statement
Altering a literal value	Specifies a value (other than the null value).#1	When used in conjunction with an embedded variable to specify the null value, indicates whether the value of the embedded variable to be passed to SQL is the null value.	In place of a literal when the value of the literal specified in the SQL statement is modified at the time of execution.
Specifying a value for a `?` parameter	Specifies a value (other than the null value).#1	When used in conjunction with an embedded variable to specify the null value, indicates whether the value of the embedded variable to be passed to SQL is the null value.	To specify a value in the `?` parameter specified in the SQL that is prepared in the `PREPARE` statement or that is prepared and executed in the `EXECUTE IMMEDIATE` statement, specify this item in the `USING` clause of the `EXECUTE`, `OPEN`, or `EXECUTE IMMEDIATE` statement.#2
Altering an embedded variable	Specifies another embedded variable in place of an embedded variable in a `SELECT` statement that is specified in a cursor declaration.	--	`USING` clause of `OPEN` statement

Function	Use		Specifiable locations
	Embedded variable	Indicator variable	
Specifying an SQL character string	Specifies an SQL character string to prepare and execute an SQL statement that was generated when the UAP was executed.	--	PREPARE or EXECUTE IMMEDIATE statement
Specifying an authorization identifier and password	Indicates to HiRDB the user executing the UAP.	--	Authorization identifier and password in CONNECT statement

--: Not applicable.

#1: To handle the null value, an indicator variable must also be specified; see *1.6.5 Setting a value for an indicator variable* below for the values of indicator variables.

#2: An indicator variable cannot be specified in the USING clause of the OPEN statement.

(3) Relationships between the data types of embedded and indicator variables and data descriptions in various languages

When you are coding embedded variables and indicator variables in a UAP, data can be exchanged between an SQL and the UAP without causing a data conversion if certain relationships hold between the data types and data descriptions involved. For those relationships, see the *HiRDB Version 9 UAP Development Guide*.

(4) Qualifying embedded variables and indicator variables

(a) COBOL language

COBOL allows the qualifying of embedded variables and indicator variables with a group item.

Group item qualification format:

: [*variable-name-1*.] *variable-name-2*

Any qualified embedded variable or indicator variable should be specified so that the result of qualification is unique. Names that need not be qualified can be qualified. If there are several possible combinations of qualifiers that result in a unique

qualification, any of such combinations can be used.

variable-name-1 is the group item to which *variable-name-2* is subordinate.

(b) C language

C allows the qualifying of a member of a structure with a structure or a pointer to a structure.

Structure qualification format:

```
: structure-name.member-name
```

Pointer qualification format:

```
: pointer-name->member-name
```

(5) Relationships between embedded variables and SQL data types

The following table lists the assignment relationships between UAP embedded variables and SQL data types.

Table 1-15: Assignment relationships between UAP embedded variables and SQL data types

Embedded variable data type	SQL data type		
	Character data	National character data	Mixed character data
Character data	Y	--	Y
National character data	--	Y	--
Mixed character data	Y	--	Y

Y: Can be assigned.

--: May be assigned; care must be exercised because assignment may occur regardless of the data type (see Note 2 and Note 3).

Note 1

Character data, national character data, and mixed character data include fixed-length and variable-length formats.

Note 2

If the data does not fill the field, the unused part of the field (on the right) is filled with spaces as follows.

- When a UAP embedded variable is assigned to an SQL data type:

  The field is filled with spaces from the SQL data type. If a character set is specified, the space from that character set is used. For the national character data type, a double-byte space is used.

- When an SQL data type is assigned to a UAP embedded variable:

  The field is filled with spaces from the UAP embedded variable. If a character set is specified, the space from that character set is used. For the national character data type, a double-byte space is used.

However, when data is assigned to the national character data type, the national character spaces in that data are also converted if a space conversion level is set in the `pd_space_level` operand of the system common definition or in `PDSPACELVL` in the client environment definition. As a result, the national character spaces may be converted into two single-byte spaces.

Note 3

If there are both regular spaces and one-byte spaces in national character data, operations such as a comparison that includes spaces may not execute as expected.

Note 4

One-byte characters should not be assigned to national character data; if an attempt is made to do so, the results cannot be predicted.

Note 5

When the `LIKE` predicate is used, special pattern characters must be in accordance with the SQL data type specification.

Note 6

If you specify `utf-8` or `chinese-gb18030` as the character code classification in the `pdntenv` command (`pdsetup` command in the UNIX edition), you cannot assign to an embedded variable that uses the national character data type. Similarly, an assignment cannot be made from the national character data type of an embedded variable to the character data type or the mixed character data type of SQL.

1.6.2 ? parameters

(1) Function

For execution of a UAP, the `?` parameter can be used to assemble SQL character strings in the program, prepare the SQL character strings using the `PREPARE` statement, execute them in the `EXECUTE`, `OPEN`, `FETCH`, or `CLOSE` statement, or prepare and execute them using the `EXECUTE IMMEDIATE` statement. In this case, specify a `?` in the

specific location in the SQL character string where a value is passed from the UAP, in the SQL character string that is prepared by the PREPARE statement, or prepared and executed by the EXECUTE IMMEDIATE statement. This facility is called the ? parameter.

(2) Specifying a value to be passed to a ? parameter

The value to be passed to a ? parameter is specified in terms of an embedded variable in the USING clause of the EXECUTE, OPEN, FETCH, CLOSE, or EXECUTE IMMEDIATE statement associated with the PREPARE statement. An indicator variable can be specified only in the EXECUTE or EXECUTE IMMEDIATE statement.

Following are examples of ? parameters that do not use the SQL descriptor area:

Example: Use of ? parameter (not using SQL descriptor area)

C language

```
       :
       :
       :
EXEC SQL BEGIN DECLARE SECTION;
    struct{
        long xcmnd_len;
        char xcmnd_txt[58];
   }xcmnd;
    char   XPCODE[5];
    char   XPNAME[21];
    char   XCOLOR[11];
    long   XPRICE;
    long   XSQUANTITY;
EXEC SQL END DECLARE SECTION;
       :
       :
       :
strcpy (xcmnd.xcmnd_txt, "INSERT INTO STOCK
VALUES(?,?,?,?,?)");
xcmnd.xcmnd_len = strlen(xcmnd.xcmnd_txt);
EXEC SQL PREPARE ST1 FROM :xcmnd;

strcpy(XPCODE,"595M");
strcpy(XPNAME,"SOCKS");
strcpy(XCOLOR,"RED");
XPRICE=3.00;
XSQUANTITY=200;
EXEC SQL
    EXECUTE ST1 USING :XPCODE, :XPNAME, :XCOLOR, :XPRICE,
:XSQUANTITY;
       :
       :
```

```
       :
```

COBOL

```
       :
       :
       :
DATA DIVISION.

WORKING-STORAGE SECTION.

    EXEC SQL
      BEGIN DECLARE SECTION
    END-EXEC.
01  XCMND.
  02  XCMND-LEN  PIC S9(9)  COMP VALUE 58.
  02  XCMND-TXT  PIC X(58)       VALUE SPACE.
77    XPCODE     PIC X(4).
77    XPNAME     PIC N(10).
77    XCOLOR     PIC N(5).
77    XPRICE     PIC S9(9) COMP.
77    XSQUANTITY PIC S9(9) COMP.

      EXEC SQL
        END DECLARE SECTION
      END-EXEC.
       :
       :
PROCEDURE DIVISION.
       :
       :
       :
      MOVE 'INSERT INTO STOCK VALUES(?,?,?,?,?)' TO XCMND-TXT.
      EXEC SQL
        PREPARE ST1 FROM :XCMND
      END-EXEC.

      MOVE '595M'   TO XPCODE.
      MOVE N'TSHIRTS' TO XPNAME.
      MOVE N'RED'    TO XCOLOR.
      MOVE 3.00     TO XPRICE.
      MOVE 300      TO XSQUANTITY.

      EXEC SQL
        EXECUTE ST1 USING :XPCODE, :XPNAME, :XCOLOR,
                          :XPRICE, :XSQUANTITY
      END-EXEC.
       :
       :
```

:

1.6.3 SQL parameters and SQL variables

(1) Format

- SQL parameter

[[*authorization-identifier*.] *routine-identifier*.] *SQL-parameter-name*

- SQL variable

[*statement-label*.] *SQL-variable-name*

(2) Function

(a) SQL parameters

SQL parameters are parameters for a routine that is declared in a procedure definition or a function definition.

When a routine is called by a `CALL` statement or a function call, an SQL parameter is a variable that permits values to be passed and received between the UAP or routine containing the `CALL` statement or function call and the routine that is called.

The functions that can be performed by SQL parameters depend on the parameter mode that is specified when the routine is declared:

- When the parameter mode is `IN` or `INOUT`:

  The value of the SQL parameter can be referenced from within the routine, which makes it possible to receive a value from the UAP or routine that calls the routine.

- When the parameter mode is `OUT` or `INOUT`:

  The value of the SQL parameter can be assigned within the routine in order to return a value to the UAP or routine that calls the routine.

How to specify an SQL parameter differs depending on whether it is specified in a routine coded in SQL or in an external routine coded using tools other than SQL, as follows:

- Specifying in an SQL-coded routine:

  Specify [[*authorization-identifier*.]*routine-identifier*.]*SQL-parameter-name*.

  However, if the authorization identifier is specified in the definition of a public procedure or public function, specify upper-case `PUBLIC` enclosed in double quotation marks (`"`) as the authorization identifier.

- Specifying in an external, non-SQL-coded routine:

  Specify the parameter name of the external routine associated with the SQL parameter. In this case, the external routine references/updates the specified parameter not as an SQL parameter, but as a parameter in the language in which

the external routine is implemented.

When values are passed or received between a UAP and a routine, the CALL statement in the UAP specifies the embedded variables, indicator variables, or ? parameters in the function call.

(b) SQL variables

SQL variables can be used to pass and receive data between SQL statements in an SQL routine or between a table and an SQL routine. SQL variables can be declared in a compound statement in an SQL routine and can be referenced in the compound statement in which they are declared.

The null value can be stored in SQL parameters and SQL variables; the null value cannot be stored in embedded variables. For this reason, it is not necessary to use an indicator variable or to place a colon (:) before the name of the variable.

(3) Data type

SQL parameters and SQL variables specify the data type of the SQL. For an explanation of SQL data types, see *1.2 Data types*.

1.6.4 Specifiable locations

The following table indicates the locations where embedded variables, indicator variables, ? parameters, SQL variables, and SQL parameters can be specified.

Table 1-16: Locations where variables and parameterscan be specified

SQL statement	Specifiable locations	Embedded variable	Indicator variable	? Parameter	SQL variable or parameter
WRITE specification	1st argument	N	N	N	N
	2nd argument, 3rd argument	Y	Y	Y	N
GET_JAVA_STORED_ ROUTINE_SOURCE specification	1st argument, 2nd argument	Y	Y	Y	Y
	3rd argument	N	N	N	N
DECLARE CURSOR	Anywhere in a search condition where a literal is allowed[#1]	Y	Y	N	Y
ALLOCATE CURSOR Format 1	Extended cursor name	Y	N	N	N
	Extended statement name	Y	N	N	N

SQL statement	Specifiable locations	Embedded variable	Indicator variable	? Parameter	SQL variable or parameter
ALLOCATE CURSOR Format 2	Extended cursor name	Y	N	N	N
SELECT	Anywhere in a search condition where a literal is allowed	Y	Y	Y	Y
	INTO clause	Y	Y	N	Y
INSERT	Anywhere in a VALUES clause where a literal is allowed	Y	Y	Y	Y
	Anywhere in a search condition where a literal is allowed	Y	Y	Y	Y
UPDATE	Anywhere in a SET clause where a literal is allowed	Y	Y	Y	Y
	Anywhere in a search condition where a literal is allowed	Y	Y	Y	Y
Preparable dynamic UPDATE statement: locating	Positions where a literal can be specified using a SET clause	N	N	Y	N
DELETE	Anywhere in a search condition where a literal is allowed	Y	Y	Y	Y
OPEN	USING clause	Y	N	N	N
FETCH	INTO clause	Y	Y	N	Y
PREPARE	Anywhere an SQL character string is allowed	Y	N	N	N
DEALLOCATE PREPARE	Extended statement name	Y	N	N	N
DESCRIBE	Extended statement name	Y	N	N	N
DESCRIBE CURSOR	Extended cursor name	Y	N	N	N

SQL statement	Specifiable locations	Embedded variable	Indicator variable	? Parameter	SQL variable or parameter
DESCRIBE TYPE	Extended statement name	Y	N	N	N
EXECUTE	INTO clause	Y	Y	N	N
	USING clause	Y	Y	N	N
	Extended statement name	Y	N	N	N
EXECUTE IMMEDIATE	Anywhere an SQL character string is allowed	Y	N	N	N
	INTO clause	Y	Y	N	N
	USING clause	Y	Y	N	N
CALL	Argument	Y	Y	Y	Y
Assignment statement	Assignment destination and assigned value	Y#2	Y	Y#2	Y
FREE LOCATOR	See *Locator*	Y	N	N	N
CONNECT	Authorization identifier and password	Y	N	N	N
SET SESSION AUTHORIZATION statement	Authorization identifier and password	Y	N	N	N
ALLOCATE CONNECTION HANDLE	PDCNCTHDL-type variable, return code-receiving variable; connection PDHOST variable, and connection PDNAMEPORT variable	Y	N	N	N
FREE CONNECTION HANDLE	PDCNCTHDL-type variable, and return code-receiving variable	Y	N	N	N
DECLARE CONNECTION HANDLE SET	PDCNCTHDL-type variable	Y	N	N	N

SQL statement	Specifiable locations	Embedded variable	Indicator variable	? Parameter	SQL variable or parameter
GET DIAGNOSTICS	Statement information item name, condition information item name	Y	N	N	N
WRITE LINE statement	value expression	N	N	N	Y

Y: Specifiable

N: Not specifiable

Note 1

Embedded variables and indicator variables are specified in a UAP. The ? parameter should be specified in an SQL character string that is preprocessed by the PREPARE statement. SQL parameters in an external routine are specified in a parameter variable specification in the external routine with which they are associated. When a parameter for an external routine is passed to an SQL or a routine, it is specified as an embedded variable or a ? parameter rather than as an SQL parameter. An SQL variable cannot be specified in an external routine.

Note 2

Embedded variables, indicator variables, and ? parameters cannot be specified in selection expressions.

Embedded variables, indicator variables, and ? parameters can be specified in the following cases:

- Specifying in a function call argument
- Specifying in an argument of the SUBSTR scalar function

For specification methods, see *2.3 Query specification*.

Note 3

Arithmetic or comparison operations cannot be specified between embedded variables, indicator variables, and ? parameters.

Note 4

Embedded variables, indicator variables, and ? parameters cannot be specified in an argument of a set function.

Note 5

Embedded variables, indicator variables, and ? parameters cannot be specified in an argument of the HEX scalar function.

Note 6

Embedded variables, indicator variables, or ? parameters cannot be specified singly (including specification in a unary operation expression) in an argument of a scalar function, with the exception of the second and third arguments of VALUE, BIT_AND_TEST, or SUBSTR, or in the third argument of POSITION. However, embedded variables, indicator variables, or ? parameters of the BLOB or BINARY type, and these types only, can be specified if the AS data-type is specified in the first argument of SUBSTR, in an argument of LENGTH, or in the first or second argument of POSITION.

Note 7

Embedded variables, indicator variables, and ? parameters cannot be specified in a date, time, or concatenation operation.

Note 8

Embedded variables, indicator variables, and ? parameters cannot be specified singly in the first value expression of the VALUE scalar function (including specification in monomial operational expressions).

Note 9

Embedded variables, indicator variables, and ? parameters cannot be specified singly in the CASE, THEN, or ELSE value expression of a simple CASE expression or searched CASE expression (including specification in monomial operational expressions).

Note 10

Embedded variables, indicator variables, and ? parameters cannot be specified singly in the first WHEN value expression during simple CASE expression specification, the first value expression of COALESCE, or both value expressions of NULLIF (including specification in monomial operational expressions).

Note 11

Embedded variables, indicator variables, or ? parameters cannot be specified singly (including specification in a unary operation expression) in the two value expressions of the BIT_AND_TEST scalar function.

#1: Excludes the search condition of a CASE expression in a selection expression.

When a cursor declaration is specified in a function call argument, the function call can be specified in a search condition in the CASE expression of the selection expression.

#2: When specifying an embedded variable, an indicator variable, or a ? parameter singly as an assignment value in an assignment statement (SET), you must always specify the AS data-type.

1.6.5 Setting a value for an indicator variable

(1) For receiving data (INTO clause of the FETCH, SELECT, EXECUTE, or EXECUTE IMMEDIATE statement)

When the FETCH, SELECT, EXECUTE, or EXECUTE IMMEDIATE statement is executed, the values listed in the following table are assigned to the indicator variable that is specified in the INTO clause of the statement. If the null value is returned to an embedded variable, the value of the embedded variable cannot be guaranteed; in this case, an error results unless an indicator variable also is specified.

Table 1-17: Indicator variable values returned by the FETCH, SELECT, EXECUTE, or EXECUTE IMMEDIATE statement (other than a repetition column, or in the case of the values of elements in a repetition column)

Indicator variable value	Value received by associated embedded variable
Negative	NULL value If the value of the indicator variable is -2, this is the null value that is assigned when the element specified by a subscript in a repetition column does not exist. If the value of the indicator variable is -4, this is the null value that is assigned by the overflow error suppression option in an arithmetic operation, set function operation, window function operation, or scalar function that is subject to overflow error suppression during SQL execution. Because the embedded variable that received the null value and the specific operation performed are associated with each other, the operation in which the overflow occurred can be determined. For the scalar functions that are subject to overflow error suppression, see *2.18 Operational results with overflow error suppression specified*.
0	NOT NULL value
Positive	NOT NULL value The received data is either character data or large object data and represents a value that has been truncated on the right because the embedded variable was too short. The indicator variable contains the length existing before this truncation.

Table 1-18: Indicator variable values returned by the FETCH, SELECT, EXECUTE, or EXECUTE IMMEDIATE statement (data on an entire repetition column)

Indicator variable value	Value received by associated embedded variable
Negative	NULL value (number of elements is 0)
0	NOT NULL value (number of elements is at least 1)

Indicator variable value	Value received by associated embedded variable
Positive	NOT NULL value (number of elements is at least 1) If the number of elements in the embedded variable area is insufficient, the value indicates that the remaining elements have been truncated. In this case, the number of elements before truncation is set in the indicator variable.

The following figures show the structures of indicator variables and embedded variables and show examples of receiving repetition column data.

Figure 1-5: Structures of indicator variables and embedded variables that receive repetition column data

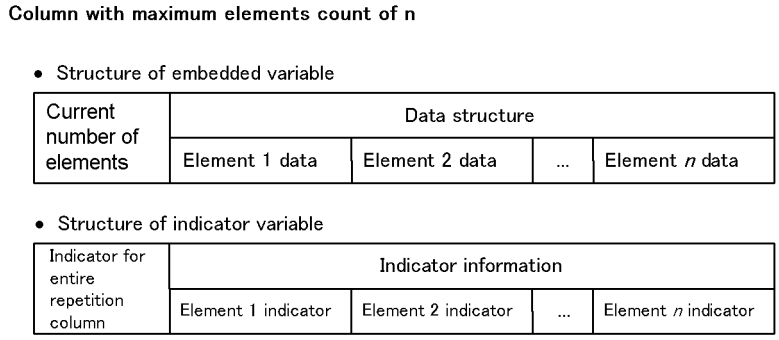


Figure 1-6: Examples of receiving repetition column data (1 of 2)

Example 1: Data column with 4 as the maximum number of elements and 3 as the current number of elements

Structure of embedded variable

Current number of elements	Element 1 data	Element 2 data	Element 3 data	Element 4 data
3	ABC		XYZ	
NOT NULL value	NOT NULL value	NULL value	NOT NULL value	

(3) (2) (3) (3)

Structure of indicator variable

Entire indicator	Element 1 indicator	Element 2 indicator	Element 3 indicator	Element 4 indicator
0	0	Negative	0	

(1)

Effective elements: 1–3

Explanation

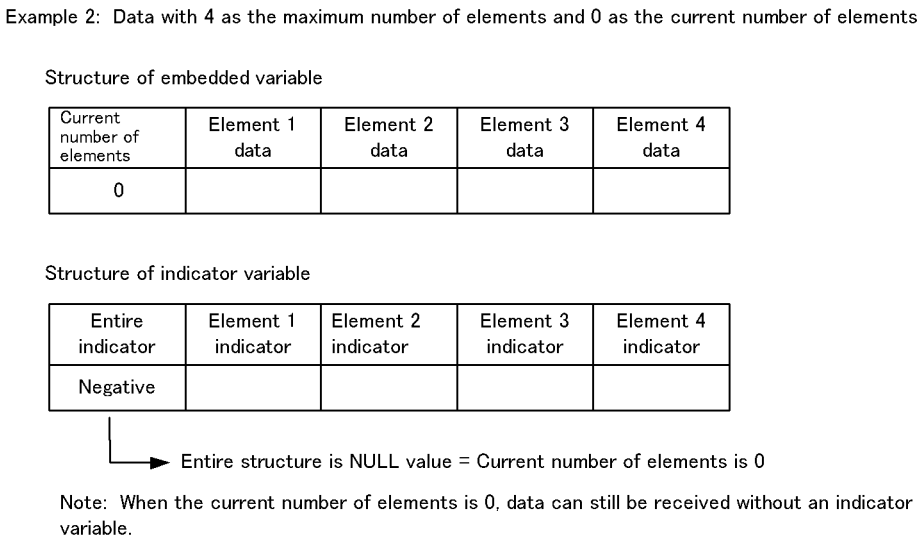
The repetition column values are referenced in the order (1)–(3):

(1) References a explicitly NULL test (non-negative or negative) for the entire repetition column for the indicator variable.

(2) If test (1) is not negative, references the current number of elements in the embedded variable.

(3) References the indicators for the elements in the indicator variable and the data in the elements of the embedded variable by matching element-by-element to the extent indicated by (2). If the indicator is negative, no data is referenced. If the indicator is non-negative, data is referenced.

Figure 1-7: Examples of receiving repetition column data (2 of 2)



(2) Passing data (not applicable to the INTO clause of the FETCH, SELECT, EXECUTE, or EXECUTE IMMEDIATE statement)

When executing an SQL statement other than the `FETCH`, `SELECT`, `EXECUTE`, or `EXECUTE IMMEDIATE` statement, in the UAP specify one of the values listed in the following table in the indicator variable before the SQL statement is executed. Depending on the value of the indicator variable, the value of the corresponding embedded variable must be used during execution of the SQL.

Table 1-19: Indicator variable value to be set before execution of SQL (other than a repetition column and element values of a repetition column)

Indicator variable value	Value passed to SQL by associated embedded variable
Negative	`NULL` value Any value contained in the embedded variable is ignored.
Non-negative	`NOT NULL` value. This is the value contained in the embedded variable.

Table 1-20: Indicator variable value to be set before execution of SQL (information on the entire repetition column)

Indicator variable value	Value passed by SQL to associated embedded variable
Negative	NULL value (number of elements is 0) Any value contained in the embedded variable is ignored.
Non-negative	Value of the element indicated by the elements count. 0 cannot be specified as the elements count.

The following figures show the structures of indicator variables and embedded variables and show examples of data passing from a repetition column.

Figure 1-8: Structures of indicator variables and embedded variables to which data from a repetition column is passed

Column with a maximum elements count of n

- Structure of embedded variable

Current number of elements	Data structure			
	Element 1 data	Element 2 data	...	Element n data

- Structure of indicator variable

Indicator for entire repetition column	Indicator information			
	Element 1 indicator	Element 2 indicator	...	Element n indicator

Figure 1-9: Example of data passed from repetition column (1 of 2)

Example 1: Data column with 4 as the maximum number of elements and 3 as the current number of elements

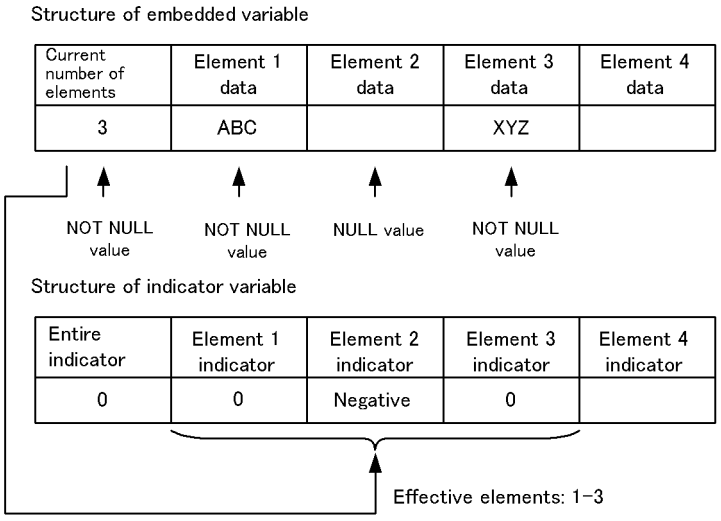
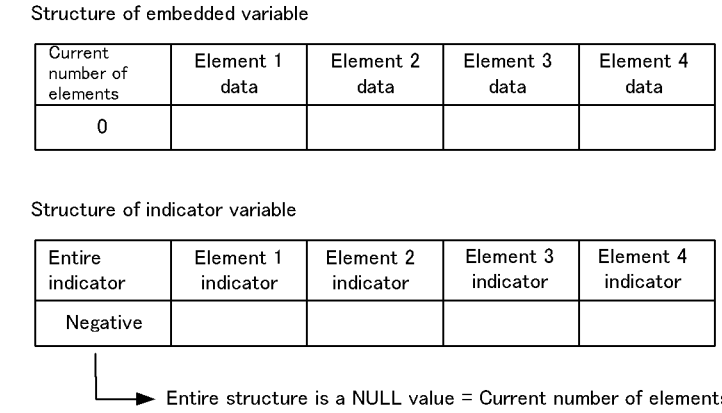


Figure 1-10: Example of data passed from repetition column (2 of 2)

Example 2: Data column with 4 as the maximum number of elements and 0 as the current number of elements



Note: When the current number of elements is 0, data can still be passed without an indicator variable.

1.6.6 Setting a null default value in an embedded variable

When data is retrieved into an embedded variable using the `FETCH`, `SELECT`,

EXECUTE, or EXECUTE IMMEDIATE statement, and if the retrieved value is the null value, a negative value is assigned to the indicator variable. When this occurs, the embedded variable assumes the value that existed before the SQL statement was issued. A default value can be set in an embedded variable by using the null-value default-setting function on the embedded variable. An indicator variable does not need to be specified for null-value data. The null-value default-setting function can be used only for fetching data.

When using the default-setting function, the user needs to have set PDDFLNVAL option in the client environment definition process at the time the UAP is executed. For details about the client environment definition process, refer to the *HiRDB Version 9 UAP Development Guide*.

The following table lists the default null values that can be set in an embedded variable.

Table 1-21: Default null values that can be set in an embedded variable

Category	Data type		Default value
Numeric data	INTEGER		0
	SMALLINT		
	DECIMAL		
	LARGE DECIMAL		
	FLOAT		
	SMALLFLT		
Character data	CHARACTER(*n*)	Default character set or a character set other than UTF-16	Space, *n* bytes
		UTF16 character set	Space, *n* ÷ 2 characters
	VARCHAR(*n*)	Default character set or a character set other than UTF-16	Space, 1 byte[#1]
		UTF-16 character set	Space, 1 character
National character data	NCHAR(*n*)		Space, *n* characters[#2]
	NVARCHAR(*n*)		Space, 1 byte[#2]
Mixed character data	MCHAR(*n*)		Space, *n* bytes
	MVARCHAR(*n*)		Space, 1 byte

Category	Data type	Default value
Date data	DATE	01/01/01
Time data	TIME	0:0:0
Time stamp data	TIMESTAMP	01/01/01 0:0:0#3
Date interval data	INTERVAL YEAR TO DAY	0 days, 0 months, 0 years
Time interval data	INTERVAL HOUR TO SECOND	0 seconds, 0 minutes, 0 hours
Large-object data#4	BLOB(*n*)	Data of length 0 bytes
Binary data#4	BINARY(*n*)	Data of length 0 bytes

Note: The data elements in simple structures and repetition structures are also defined according to this table. If all data in a repetition structure is NULL, the current number of elements will be 0.

#1: In the result of a WRITE specification, this will be an IP address.

#2: Depends on the character code used in HiRDB.

Example: X'8140' in shift JIS

If a space conversion level is specified in PDSPACELVL in the client environment definition or in the pd_space_level operand in the system common definition, any spaces in an embedded variable will also be subject to the conversion. Therefore, an NCHAR space can be converted into *n* x 2 bytes, an NVARCHAR space can be converted into 2 bytes.

#3: If a fractional second precision is specified, the specified digit positions are zero-filled.

#4: If a locator is used, a value that identifies data greater than 0 bytes in length on the server is assigned to the embedded variable of the locator.

1.6.7 Assignment rules

The following table lists the assignment types.

Table 1-22: Assignment types

Assignment type	Description	Source	Target
Retrieval assignment	Retrieves the value of a string.	String	Embedded variable, SQL variable, or SQL parameter
	Sets the output parameter value of a procedure into an argument of the `CALL` statement.	SQL parameter	Embedded variable, SQL variable, or SQL parameter
	Retrieves the return value from the value expression specified in the `RETURN` statement of a function.	Value expression	Return value of a function
Storage assignment	Assigns a value to a string.	Embedded variable, SQL variable, or SQL parameter	String
	Assigns a value from a `CALL` statement argument to an input parameter of a procedure.	Value expression	SQL parameter
	Assigns a value from a function call argument to the function parameter.	Value expression	SQL parameter
	Assigns a value using an assignment (`SET`) statement.	SQL variable, SQL parameter, or embedded variable	SQL variable, SQL parameter, or embedded variable

(1) Fixed-length target

If the target of an assignment is fixed-length character string data, mixed character string data, or national character string data, the applicable assignment rules vary with the length of the source data. The following table indicates the assignment rules for fixed-length target data.

Table 1-23: Assignment rules for fixed-length target data

Assignment type	Data length		
	Source#1 > target	Source#1 = target	Source#1 < target
Retrieval assignment	--#2	Y	--#3
Storage assignment	N	Y	--#3

Y: Assigned as is

--: Assigned left-justified according to length of target data

N: An error occurs.

#1: If the source is variable-length data, the length of the source data is its actual length.

#2: If an overflow occurs, the excess portion of the data is truncated on the right, and warning information is set in the SQLWARN1 parameter for the SQL communication area. If an indicator variable is specified, the length of data before truncation is set in the indicator variable.

#3: The trailing portion is space-filled on the right.

(2) If the target of assignment is variable-length data, large object data, or BINARY

If the target of an assignment is character string data, mixed character data, national character string data, large object data, or BINARY of variable length, the rules of assignment vary depending upon the length of the assignment source data. The following table indicates assignment rules that are applicable when the target of an assignment is variable-length data.

Table 1-24: Assignment rules for variable-length target data

Assignment type	Data length	
	Source#1 > target#2	Source#1 ≤ target#2
Retrieval assignment	--#3	--#4
Storage assignment	N	--#4

--: Assigned left-justified according to length of source data2

N: An error occurs.

#1: If the source is variable-length data, the length of the source data is its actual length.

If the source is the embedded variable for a locator, the data length of the source is the actual length of the data that is assigned to the locator.

#2: The length of target data is equal to the maximum allowable length of variable-length data.

If the source is the embedded variable for a locator, the target data length takes one of the following lengths:

BLOB locator: 2147483647

BINARY locator: 2147483647

#3: If an overflow occurs, the excess portion of the data is truncated on the right, and the actual length is equal to the maximum allowable length of the target string. When this occurs, warning information is set in the SQLWARN1 parameter for the SQL communication area. If an indicator variable is specified, the length of data before truncation is set in the indicator variable.

#4: If the source is fixed-length data, the length of the source data is its actual length.

(3) Rules on assignment between different character sets

In general, assignment between different character sets is not permitted. However, the following table indicates when such assignment is possible.

Table 1-25: When assignment between different character sets is possible.

Source character set	Target character set		
	Default character set	EBCDIK	UTF-16
Default character set	Y	Y#1	Y#2
EBCDIK	Y#3	Y	N
UTF-16	Y#4	N	Y

Legend:

Y: Can be assigned.

N: Cannot be assigned.

#1

For assignments from the SJIS default character set into EBCDIK, the permissible assignment combinations are as follows:

Source value expression	Target value expression
Character string literal or embedded variable	Any

#2

For assignments from the UTF-8 default character set into UTF-16, the permissible assignment combinations are as follows:

Source value expression	Target value expression
Character string literal or embedded variable	Any
Any other value expression	Embedded variable

#3

For assignments from EBCDIK into the SJIS default character set, the permissible assignment combinations are as follows:

Source value expression	Target value expression
Value expression	Embedded variable

#4

For assignments from UTF-16 into the UTF-8 default character set, the permissible assignment combinations are as follows:

Source value expression	Target value expression
Embedded variable	Any
Value expression	Embedded variable

When the source and target character sets are different, assignment proceeds as follows:

1. Retrieval assignments
   - Assignment is performed after conversion into the target character set. At this time, if the length of the converted character data is greater than the length of the target character field, the excess portion of the data is truncated on the right and warning information is assigned to the `SQLWARN1` parameter of the SQL communication area. If the target is an indicator variable, the length (number of bytes) of data before truncation is set to the indicator variable.
   - If the target is fixed-length character data and the length of the source character data is less than the length of the target field, the assignment is performed and the remaining portion of the field is filled with the space character from the target character set.
2. Storage assignments
   - Assignment is performed after conversion into the target character set. An error results if the length of the converted character data is greater than the length of the target character field.
   - If the target is fixed-length character data and the length of the source character data is less than the length of the target field, the assignment is performed and the remaining portion of the field is filled with the space character from the target character set.

(4) Rules on the structure of the source and target of an assignment

When the source and target have incompatible structures, an assignment may not work. The following table indicates the structure rules for the source and target of an assignment operation.

Table 1-26: Structure rules for the source and target of assignment operation

Source structure	Target structure	
	Simple Structure	Repetition Structure
Simple structure	Y	N
Repetition structure	N	Y

Y: Assignment allowed

N: Error may result.

Note: Subscripted repetition columns are treated as simple structures.

1.7 Null value

The null value is a special value that indicates either that no value exists or the value has not been set. The null value is set in an area that does not contain values or in which values have not been set. The null value in an abstract data type indicates whether a value has been generated by the constructor function.

The following explains how the null value is handled.

(1) Receiving a column value as a result of a retrieval

The value of an indicator variable indicates whether the null value was received. The null value cannot be received by an embedded variable. For details, see *1.6.5 Setting a value for an indicator variable*.

(2) Storing a value in a table

The value of an indicator variable indicates whether the null value was stored. The null value cannot be stored in a table by an embedded variable. For details, see *1.6.5 Setting a value for an indicator variable*.

(3) Comparison

If the value of a specified value expression, column, or embedded variable for a row in a predicate other than the `NULL` predicate is the null value, the predicate is undefined. An indicator variable is required to specify the null value using an embedded variable.

(4) Join

A row containing the null value in the joined column does not satisfy the join conditions.

(5) Sorting

In the case of an ascending-order sort, the null value is output at the end; in the case of a descending-order sort, the null value is output at the beginning.

(6) Grouping

If a row contains null values in grouping condition columns, SQL performs grouping by treating the null values as being the same value.

(7) Exclusion of duplicates

Multiple null values are treated as duplicates.

(8) Set functions

In general, set functions ignore the null value. The `COUNT(*)` function, however, calculates all eligible rows, regardless of null values that may be present in the rows.

(9) Window functions

The COUNT(*) and OVER() functions calculate all eligible rows, regardless of null values that may be present in the rows.

(10) Indexing

An index can be defined for a column that contains null values.

(11) Arithmetic, date, time, and concatenation operations

An arithmetic, date, time, or concatenation operation performed on the null value as a data value produces the null value.

(12) Scalar functions

Scalar functions other than VALUE and STRTONUM produce the null value as the result when any of the value expressions in an argument is the null value. The VALUE scalar function produces the null value as the result when all value expressions in an argument are the null value. The STRTONUM scalar function produces the null value as the result when the value expression for argument 1 is the null value.

(13) CASE expressions

In COALESCE of CASE abbreviation, if the value expressions of arguments are all null values, the results will also be null values.

(14) Abstract data type

How the null value in an abstract data type is handled is explained in terms of two cases: a value is generated by specifying a constructor function for the abstract data type, and a value is not generated by specifying a constructor function.

- Value not generated by specifying constructor function

  All values in the abstract data type will be the null value.

- Value generated by specifying constructor function

  Regardless of the values of the attributes that comprise the abstract data type, the entire abstract data type assumes NOT NULL values. Even if the value of an attribute comprising the abstract data type is NULL, the entire abstract data type will not be null.

(15) Boolean predicate

A Boolean predicate being undefined is equivalent to a Boolean value being the null value.

(16) Repetition column

Some elements may have a null value. If all column elements are 0, the entire column is treated as null.

(17) WRITE specification

In the case of a WRITE specification, the result will be the null value if any of the arguments is the null value.

(18) GET_JAVA_STORED_ROUTINE_SOURCE specification

The result of a GET_JAVA_STORED_ROUTINE_SOURCE specification will be the null value if any of the following conditions is satisfied:

- Any of the arguments is the null value
- The specified JAR file is not installed
- A source file associated with a class specified in the JAR file is not found

(19) CAST specification

If NULL is specified in the value expression or the result of the value expression is the null value, the value of the result is the null value.

(20) Referential constraint

If the null value is contained in a foreign key component column, that column is not subject to referential constraint operation.

1.8 Component specification

A component specification in an SQL statement specifies the attribute of a member of an abstract data type, defined in the abstract data type.
component-specification::=*item-specification*..*attribute-name*
[..*attribute-name*]...

A component specification is subject to the following rules:

- If a value is assigned to an attribute, the value of the abstract data type containing the attribute must be generated by the constructor function (the value of the abstract data type cannot be the null value).
- If an attribute is referenced by means of a component specification and the abstract data type containing the attribute is null (if that value was not generated by the constructor function), the referenced value will also be null.
- A subscripted column specification cannot be specified in an item specification.

1.9 Routines

Routines can be divided into the following two programming language categories:

- SQL routines

  Routines written in SQL

- External routines

  Routines written in languages other than SQL

Routines can be divided into two classes, depending on the definition SQL:

- Procedures
- Functions

1.9.1 Procedures

Procedures are defined in a `CREATE PROCEDURE` or `CREATE TYPE` procedure.

Procedures can be divided into the following categories depending on the language in which the routines are written:

- SQL procedures

  Procedures written in SQL

- External procedures

  Procedures written in languages other than SQL

Note that although trigger action procedures created in a trigger definition are also a part of SQL procedures, they are defined in `CREATE TRIGGER` as a schema element (a trigger) separate from a routine. Trigger action procedures, which are automatically executed upon a specified trigger event, cannot perform calls using a `CALL` statement or pass values using an SQL parameter.

1.9.2 Functions

(1) User-defined functions

This section describes functions that are defined using either `CREATE FUNCTION` or the `CREATE TYPE` function itself; it also shows functions that are provided by plug-ins. A function that is provided by a plug-in is called a *plug-in function*.

Functions can be classified into the following two categories:

- SQL functions

  Functions written in SQL

- External functions

  Functions written in languages other than SQL

(2) System-defined functions

Constructor functions are generated by the system using the CREATE TYPE function.

1.9.3 Results-set return facility

The use of the results-set return facility allows you to reference, at the source of the call, the results of a search using a cursor in a procedure.

This section explains how to return a results set in a procedure, and how to receive, into a UAP, the results set that is returned by the procedure.

(1) In an SQL procedure definition

In the DYNAMIC RESULT SETS clause of CREATE PROCEDURE, specify the maximum number of result sets (the maximum number of cursors to be returned to the source of the call). In addition, in the cursor declaration for the cursors that are returned as a results set from a procedure, specify WITH RETURN.

Closing the procedure with the cursors that were declared open by specifying WITH RETURN allows you to return the cursor results set to the source of the call. If there are two or more result sets to be returned, the system returns them in the order in which the cursors were opened.

An example of an SQL procedure definition is given below:

```
CREATE PROCEDURE ORDERED_EMPS(IN REGION INTEGER)
    DYNAMIC RESULT SETS 2
    BEGIN
        DECLARE CUR1 CURSOR WITH RETURN
            FOR SELECT id_no, name FROM emps_1
                WHERE id_no < REGION ORDER BY id_no;
        DECLARE CUR2 CURSOR WITH RETURN
            FOR SELECT id_no, name FROM emps_2
                WHERE id_no < REGION ORDER BY id_no;
        OPEN CUR1;
        OPEN CUR2;
    END;
```

(2) In an external Java procedure definition

In the DYNAMIC RESULT SETS clause of CREATE PROCEDURE, specify the maximum number of result sets (the maximum number of cursors to be returned to the source of the call).

The following shows an example of an external Java procedure definition:

```
CREATE PROCEDURE ORDERED_EMPS(IN REGION INTEGER)
    DYNAMIC RESULT SETS 2 LANGUAGE JAVA
    EXTERNAL NAME
      'jfile.jar:Routines3.orderedEmps
       (int,java.sql.ResultSet[],java.sql.ResultSet[])
       returns void'
    PARAMETER STYLE JAVA;
```

(3) By creating a Java method that is an external Java procedure entity

For the last argument of the Java method that is an external Java procedure entity, specify a `java.sql.ResultSet[]` type parameter. The Java method executes the SQL statement, receives result sets, and sets a variable that is a parameter of the `java.sql.ResultSet[]` type.

An example of how to create a Java method is given below:

```
import java.sql.*;

public class Routines3 {
    public static void orderedEmps(int region,
        java.sql.ResultSet[] rs1, java.sql.ResultSet[] rs2)
    throws SQLException {
        java.sql.Connection conn=DriverManager.getConnection(
            "jdbc:hitachi:PrdbDrive","USER1","USER1");
        java.sql.PreparedStatement stmt1=
          conn.prepareStatement(
              "SELECT id_no,name FROM emps_1 WHERE id_no < ?
ORDER BY id_no");
        stmt1.setInt(1, region);
        rs1[0]=stmt1.executeQuery();
        java.sql.PreparedStatement stmt2=
          conn.prepareStatement(
              "SELECT id_no,name FROM emps_2 WHERE id_no < ?
ORDER BY id_no");
        stmt2.setInt(1, region);
        rs2[0]=stmt2.executeQuery();
        return;
   }
}
```

(4) By creating an embedded type UAP

When an embedded type UAP is created, a procedure is executed from the embedded type UAP, and a group of the result sets returned by the `ALLOCATE CURSOR` statement is assigned to a cursor. The cursor is associated with the first result set, and data can be fetched from the result set using the `FETCH` statement.

The second and subsequent result sets are allocated to the group of result sets.

Executing the CLOSE statement on the cursor associated with the previous result set allows you to associate the new result set with the cursor and fetch data from the new result set using the FETCH statement.

An example of how to create an embedded type UAP in the C language is given below:

```
EXEC SQL WHENEVER SQLERROR GOTO error_end;
EXEC SQL CALL ORDERED_EMP(1000);
if (SQLCODE==120) {     /* A group of result sets was returned */
    EXEC SQL ALLOCATE GLOBAL :cur1 FOR PROCEDURE ORDERED_EMP;
                      /* A cursor is allocated */
                      /* Specify a cursor name in cur1 */
    while (1) {
        while (1) {
            EXEC SQL WHENEVER NOT FOUND DO break;
           EXEC SQL FETCH GLOBAL :cur1 INTO :emp_id, :emp_name;
            printf("ID No.=%s\n", emp_id);
            printf("Name=%s\n", emp_name);
       }
        EXEC SQL WHENEVER NOT FOUND DO break;
        EXEC SQL CLOSE GLOBAL :cur1;
   }
}
error_end:
```

(5) By creating a UAP using Java

A procedure is executed from a UAP coded in Java, and the result sets sent from the procedure are received.

An example of how to create a UAP using Java is given below, in which the UAP receives result sets by using java.sql.PreparedStatement.execute().

```
java.sql.CallableStatement stmt=conn.preparecall(
    "{call ordered_emps(?)}");
stmt.setInt(1,3);
stmt.execute();
java.sql.ResultSet rs=stmt.getResultSet();
                                    //Receives result sets
while(rs.next()) {
    int id_no=rs.getInt(1);
    java.lang.String name=rs.getString(2);
    System.out.println("ID No.="+id_no);
    System.out.println("Name="+name);
    System.out.println();
}
rs.close();
while (stmt.getMoreResults()){
    rs = stmt.getResultSet();
```

```
        while(rs.next()) {
            int id_no=rs.getInt(1);
            java.lang.String name=rs.getString(2);
            System.out.println("ID No.="+id_no);
            System.out.println("Name="+name);
            System.out.println();
        }
        rs.close();
    }
```

(6) Notes

The result set (`ResultSet`) that the method returns, which is in the class `DatabaseMetaData` in the external Java procedure, cannot be returned as a dynamic result set. To obtain the information, use the connection metadata from the call to the external Java procedure.

1.10 External routines

This section describes external routines. An external routine is a routine written in a language other than SQL. In HiRDB, the following external routines can be used:

- External Java routine

  An external routine written in the Java language. External Java routines can be classified as follows:

  - External Java procedures

    A procedure written in the Java language.

  - External Java functions

    A function written in the Java language.

- External C stored routines

  An external routine written in the C language. External C stored routines can be classified as follows:

  - External C procedures

    A procedure written in the C language.

  - External C functions

    A function written in the C language.

1.10.1 External Java routines

This section describes the following topics relating to external Java routines:

- External Java routine names
- Type mapping
- Results-set return facility

In addition, note that in HP-UX, Solaris, and AIX, external Java routines cannot be used if you have not set up the POSIX library version of HiRDB (using the `pdsetup` command), or if you have set up a non-POSIX library version of HiRDB over a POSIX library version of HiRDB. For details about the `pdsetup` command, see the manual *HiRDB Version 9 Command Reference*.

(1) External Java routine names

Since external routines consist of external Java procedures and external Java functions, external Java routine names are specified using `CREATE PROCEDURE` or `CREATE FUNCTION`.

(a) Format

external-Java-routine-name ::= *'JAR-file-name*:*Java-method-name* [*Java-signature*]'

Java-method-name ::= *Java-class-name*.*method-identifier*
Java-class-name ::= [*package-name*.]*class-identifier*
Java-signature ::= ([*Java-parameters*])[returns *type-name*]
Java-parameter ::= *type-name*[,*type-name*]...

(b) Explanation

JAR-file-name

Specifies the name of an archive file, which is a set of classes or packages defined in Java. The JAR file name should not be specified as a path name. The length of a JAR file name cannot exceed 255 bytes.

method-identifier

Specifies the identifier of the Java method in which the actual processing is coded. The length of a method identifier cannot exceed 255 bytes.

package-name

Specifies the name of a package, which is a set of classes defined in Java.

class-identifier

Specifies the identifier of the class in which the Java method is defined. The length of the package name and class identifier together cannot exceed 255 bytes.

returns *type-name*

Specifies the Java type name associated with the arguments and return values for the Java method.

When defining an external Java procedure, either make the return value type name void (no return value) or omit returns *type-name*.

When defining an external Java function, returns *type-name* is required, unless the Java signature is omitted, in which case there is no need to specify returns *type-name*. For an external Java function, you cannot specify void (no return value) as the return value type name.

type-name[,*type-name*]...

Specifies the Java type names associated with the arguments and the return values of the Java method. Java type names must conform to the following rules:

1. The type names of Java method parameters coded in Java parameters should be coded in the order of the SQL parameter names that are specified in CREATE PROCEDURE or CREATE FUNCTION.

2. All Java method parameter type names associated with the SQL parameter names must be coded.
3. If the maximum number of the results sets in the DYNAMIC RESULTS SET clause of the CREATE PROCEDURE command is equal to or greater than 1, specify at the end of the Java parameters the type name java.sql.ResultSet[] and a value that is no more than the number of specified results sets.
4. When a function is defined with the CREATE FUNCTION command, specify Java data types that can be mapped to HiRDB data types.

   In a procedure defined in CREATE PROCEDURE, if the input/output mode for the SQL parameters is IN, specify Java data types that can be mapped to HiRDB data types.

   In a procedure defined in CREATE PROCEDURE, if the input/output mode for the SQL parameters is OUT or INOUT, specify a one-dimensional array for Java data types that can be mapped to HiRDB data types.

   For example, with respect to the INTEGER or BLOB output parameters, specify either java.lang.Interger[] or byte[][]. For details about Java data types that can be mapped to HiRDB data types, see *1.10.1(2) Type mapping*.

(c) Rules

1. If the Java signature is omitted, the Java data type is determined based on the HiRDB data type specified in the external Java procedure or function and the input/output mode of the SQL parameters, according to the following rules:
   - If a function is defined using CREATE FUNCTION, the data type of the Java method is determined according to the mapping rules.
   - If the input/output mode for the SQL parameter of a procedure defined in CREATE PROCEDURE is IN, the data type of the Java method is determined according to the mapping rules.
   - If the input/output mode for the SQL parameter of a procedure defined in CREATE PROCEDURE is OUT or INOUT, the data type of the Java method is determined according to the one-dimensional array type of the mapping rules.
   - If the Java signature is omitted by specifying the maximum number of results sets in the DYNAMIC RESULTS SET clause of CREATE PROCEDURE as being equal to or greater than 1, java.sql.ResultSet[] type parameters equal to the maximum number of results sets are added as Java method parameters to the data type determined according to the mapping rules.

   For example, either java.lang.Integer[] or byte[][] is determined with

respect to the INTEGER or BLOB output parameters. For the mapping rules, see *1.10.1(2) Type mapping*.

2. External routine names must be enclosed in single quotation marks (').
3. The only Java method that can be specified in an external routine name is the class method that is declared as static in a class definition.
4. The following character sets can be specified in the various items:
   - JAR file names

     Upper- and lower-case alphabetic characters

     Numeric characters, underscores (_), dollar signs ($), periods (.), and hyphens (-)
   - Class identifiers, method identifiers, and type names

     Upper- and lower-case alphabetic characters

     Numeric characters, underscores (_), and dollar signs ($)
   - Package names

     Upper- and lower-case alphabetic characters

     Numeric characters, underscores (_), dollar signs ($), and periods (.)
5. A numeric cannot be specified as the first character of a package name, class identifier, method identifier, or type name.
6. If a type name with a package name omitted is coded, HiRDB interprets the type name as containing the following package name:

Default coding format	Type name containing a package name, as interpreted by HiRDB
Integer	java.lang.Integer
Short	java.lang.Short
BigDecimal	java.math.BigDecimal
Double	java.lang.Double
Float	java.lang.Float
String	java.lang.String
Date	java.sql.Date
Time	java.sql.Time
Timestamp	java.sql.Timestamp

Default coding format	Type name containing a package name, as interpreted by HiRDB
ResultSet	java.sql.ResultSet

(2) Type mapping

This section explains the mapping between the data types that are recognized in the Java language and the data types that are recognized by HiRDB.

The following table indicates the implicit mapping that occurs when a Java signature specifying an external routine is omitted.

Table 1-27: Implicit mapping that occurs when a Java signature specifying an external routine is omitted

HiRDB data type	Java data type (null value allowed)
INT [EGER]	java.lang.Integer
SMALLINT	java.lang.Short
[LARGE] DEC [IMAL]	java.math.BigDecimal
FLOAT, DOUBLE PRECISION	java.lang.Double
SMALLFLT, REAL	java.lang.Float
CHAR [ACTER]	java.lang.String
VARCHAR(*n*), CHAR [ACTER] VARYING	
NCHAR, NATIONAL CHAR [ACTER]	
NVARCHAR, NATIONAL CHAR [ACTER], NCHAR VARYING	
MCHAR	
MVARCHAR	
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
INTERVAL YEAR TO DAY	java.math.BigDecimal
INTERVAL HOUR TO SECOND	
BLOB, BINARY LARGE OBJECT	byte[]
BINARY	

The following table indicates the mapping between HiRDB data types and Java data types specifiable in a Java signature of an external routine specification.

Table 1-28: Mapping between HiRDB data types and Java data types specifiable in a Java signature of an external routine specification.

HiRDB data type	Java data type (null value allowed)	Java data type (null value not allowed)#1
INT [EGER]	java.lang.Integer	int
SMALLINT	java.lang.Short	short
[LARGE] DEC [IMAL]	java.math.BigDecimal	--
FLOAT, or DOUBLE PRECISION	java.lang.Double	double
SMALLFLT, or REAL	java.lang.Float	float
CHAR [ACTER]	java.lang.String, or byte[]#2	--#3
VARCHAR(*n*), or CHAR [ACTER] VARYING		
NCHAR, or NATIONAL CHAR [ACTER]		
NVARCHAR, NATIONAL CHAR [ACTER], or NCHAR VARYING		
MCHAR		
MVARCHAR		
DATE	java.sql.Date	--
TIME	java.sql.Time	
TIMESTAMP	java.sql.Timestamp	
INTERVAL YEAR TO DAY	java.math.BigDecimal	
INTERVAL HOUR TO SECOND		
BLOB, or BINARY LARGE OBJECT	byte[]	
BINARY		
Abstract data type	--	

--: No corresponding data type exists in the Java language.

#1: Assigning the null value may cause an error at runtime.

#2: Either Java.lang.String or byte[] can be specified as the Java data type corresponding to the HiRDB character string data type. Specifying byte[] suppresses

character code conversion.

#3: Passing and receiving Japanese data through a `String` class (or a class inheriting that class) between HiRDB and an external Java routine is performed according to the rules regarding the mapping of Java character codes (mapping between a given character code and Unicode). This means that some gaiji codes might not convert correctly.

(3) SQL executability using a non-POSIX library version of HiRDB(UNIX edition only)

The table below indicates the executability of SQL statements in non-POSIX library versions of HiRDB. For the HP-UX, Solaris, and AIX editions, the non-POSIX library version of HiRDB refers to a HiRDB in which the load module in the POSIX library version has not been set up with the `pdsetup` command.

Table 1-29: Executability of SQL statements in non-POSIX library versions of HiRDB

Classification	SQL statement	Executable?	Condition in which SQL statement cannot be executed
Definition SQL	`ALTER PROCEDURE`	C	Specification of an external Java procedure
	`ALTER ROUTINE`	C	Specification of an external Java procedure or function
	`CREATE FUNCTION`	C	Use of an external Java routine
	`CREATE PROCEDURE`	C	Use of an external Java routine
	`CREATE TYPE`	C	Use of an external Java routine
	`CREATE VIEW`	C	Specification of an external Java function
	`DROP FUNCTION`	Y	N/A
	`DROP PROCEDURE`	Y	N/A
	`DROP SCHEMA`	Y	N/A

Classification	SQL statement	Executable?	Condition in which SQL statement cannot be executed
Data manipulation SQL	ASSIGN LIST statement	Y	N/A
	CALL statement	C	• External Java procedure • Procedure that uses an external Java routine
	DELETE statement	Y	N/A
	DROP LIST statement	Y	N/A
	FREE LOCATOR statement	Y	N/A
	INSERT statement	C	Use of an external Java function
	PURGE TABLE statement	Y	N/A
	Single-row SELECT statement	C	Use of external Java function
	Dynamic SELECT statement	C	Use of an external Java function
	UPDATE statement	C	Use of an external Java function
	Assignment statement	C	Use of an external Java function
Control SQL	LOCK statement	Y	N/A
Embedded SQL language	INSTALL JAR	N	N/A
	REPLACE JAR	N	N/A
	REMOVE JAR	Y	N/A
	INSTALL CLIB	Y	N/A
	REPLACE CLIB	Y	N/A
	REMOVE CLIB	Y	N/A
Other	Function call	C	• Calling an external Java function • Function that uses an external Java function

Y: Can be executed.

C: Might result in an error depending on the condition.

N: Cannot be executed.

N/A: Not applicable

1.10.2 External C stored routines

This section describes external C stored routine names. Note that external C stored routines that contain SQL statements cannot be executed.

(1) External C stored routine names

External C stored routine names can be specified using `CREATE PROCEDURE` or `CREATE FUNCTION`. The external C procedure or function that you specify here serves as the external routine definition.

(a) Format

external-C-stored-routine-name : : = ' *C-library-file-name* ! *external-function-identifier* '

(b) Explanation

C-library-file-name

Specifies the name of the library file containing a function written in the C language.

Specify the C library file name as a file name only, not a path name.

external-function-identifier

Specifies the identifier of a C function in which the external C stored routine is actually coded.

(c) Rules

1. The following characters are allowed in the specifications for each item:
   - *C-library-file-name*

     The following single-byte characters can be used:

     Upper-case and lower-case letters, numeric characters, underscores (_), dollar signs ($), periods (.), and hyphens (-)
   - *external-function-identifier*

     The following single-byte characters can be used:

     Upper-case and lower-case letters, numeric characters, and underscores (_)
2. The first character of the external function identifier cannot be a numeric character.
3. The total combined length of the C library file name and external function identifier cannot exceed 254 bytes.

1.11 Specifying a datetime format

(1) Overview

The following operations require the specification of a datetime format:

- Converting date data, time data, or time stamp data into a non-predefined character string representation using the VARCHAR_FORMAT scalar function
- Converting a non-predefined character string representation of a date, time, or a time stamp into date data, time data, or time stamp data using the DATE, TIME, or TIMESTAMP_FORMAT scalar function

(2) Rules for the datetime format

1. The following items can be specified as a datetime format:
   - Character string literals and mixed character string literals
   - Column specifications
   - Component specifications
   - SQL variables or SQL parameters
   - Concatenation operation
   - Set functions (MAX and MIN)
   - Scalar functions
   - CASE expressions
   - CAST specifications
   - Function calls
   - Scalar subquery
2. The data type of the datetime format should be either the character data type (CHAR, VARCHAR) or the mixed character data type (MCHAR, MVARCHAR).
3. The maximum allowable length of the datetime format is 240 bytes. However, if the value expression being converted is in the UTF-16 character set, the maximum is 480 bytes.

(3) Elements of the datetime format

1. The following table lists the datetime format elements and their meanings.

Table 1-30: Datetime format elements and their meanings

Datetime item	Format item#1	Meaning
Year	YYYY	A 4-digit year (0001 to 9999)
	YY	A 2-digit year (00 to 99)#4
Month	MM	Month (01 to 12)
	MON	Month, abbreviated#2, #3
	MONTH	Name of the month#2, #3
Day	DD	Day (01 to the last day of the month)
Hour	HH	Hour (00 to 23)
Minute	MI	Minute (00 to 59)
Second	SS	Second (00 to 59)#8
Fractional second	FF	Fractional second#4, #5
	NN...N	Fractional second in *p* digits (*p* = *N*, where *N* is 1 to 6)#6
Other	Space ( )	Elements that can be used as delimiter characters
	Hyphen (-)	
	Forward slash (/)	
	Comma (,)	
	Period (.)	
	Semicolon (;)	
	Colon (:)	
	"*character-string*"	A character string enclosed in double quotation marks that denotes the character string itself#7

#1: All elements of the datetime format, with the exception of a character string enclosed in double quotation marks (") must be specified in single-byte characters. All characters with the exception of the first and second characters of MON and MONTH, and characters other than those in a character string enclosed in double quotation marks, are not case-sensitive.

#2: In MON and MONTH, you can specify an abbreviated name of the month and also whether the name of a month is spelled in upper or lower case characters. The

determination of upper case versus lower case is based on the first and second characters of a specified datetime element.

Examples

```
MONTH → JUNE
Month → June
month → june
```

The following table indicates the relationship between the first and second characters of the datetime format element MON or MONTH, the name of a month, and the format of an abbreviated month.

Table 1-31: Relationship between the first and second characters of the datetime format element MON or MONTH, the name of a month, and the format of an abbreviated month

Second character	First character	
	Upper case	Lower case
Upper case	All upper case	All lower case
Lower case	Upper case in the first character only	All lower case

#3

The following table indicates the abbreviated and full names of each month (when a datetime format element is specified in MON or MONTH).

Table 1-32: Abbreviated and full names of each month (when a datetime format element is specified in MON or MONTH)

Month	Abbreviated name	Name
1	JAN	JANUARY
2	FEB	FEBRUARY
3	MAR	MARCH
4	APR	APRIL
5	MAY	MAY
6	JUN	JUNE
7	JUL	JULY

Month	Abbreviated name	Name
8	AUG	AUGUST
9	SEP	SEPTEMBER
10	OCT	OCTOBER
11	NOV	NOVEMBER
12	DEC	DECEMBER

#4: The items YY and FF can be used only in the VARCHAR_FORMAT scalar function; when specified in other scalar functions, they can cause an error.

#5: With the item FF, the number of digits in the fractional second part of the resulting character string representation is governed by the type of the time stamp data that is specified in an argument in the VARCHAR_FORMAT scalar function. If the precision of the fractional second is 0, a character string of length zero is produced.

#6: The item NN...N is converted in the following format:

- Converting a datetime value into a character string representation:

  If *p* is smaller than the fractional second precision of time stamp data, the data is truncated; if *p* is larger, the expanded fractional second part is zero-filled.

- Converting a character string representation into a datetime value:

  The number of digits in the fractional second part of the character string representation must agree with *p*.

#7: Any double quotation mark specified in a character string enclosed in double quotation marks must be expressed as two successive double quotation marks ("").

#8: If you set the system common definition operand pd_leap_second to allow leap seconds to be specified, the range for seconds is as shown below. For details about the system common definition operand pd_leap_second, see the manual *HiRDB Version 9 System Definition*.

Datetime item	Datetime format element	Meaning
Second	SS	Second (00 to 61)

(4) Rules for datetime format elements

1. In a character string in a datetime format, elements of the datetime format of a datetime item, with the exception of delimiter characters and character strings enclosed in double quotation marks, can be specified only once.

2. When converting from datetime to a character string representation, if you specify datetime format elements for which there are no related datetime data to be converted, those unrelated parts are filled with the compatible character strings listed in the table below.

Example

```
VARCHAR_FORMAT(DATE('2002-01-01'),'YYYY-MM-DD HH:MI')
-> '2002-01-01 00:00'
```

1. The following table indicates the character strings compatible with datetime format elements.

Table 1-33: Character strings compatible with datetime format elements

Datetime format element	Compatible character string
YYYY	Current year (e.g., '2002')
YY	Last two digits of the current year (e.g., '02')
MM	Current month (e.g., '08')
MON	Current month, abbreviated name (e.g., 'AUG')
MONTH	Name of the current month (e.g., 'AUGUST')
DD	Current day (e.g., '05')
HH	'00'
MI	'00'
SS	'00'
FF	'00'
NN...N	'00...0' (a string of *p* zeros, where the number of zeros is equal to the value of *p* = *N*)

2. The table below lists the datetime items that are required by the scalar function that converts a given character string representation into a datetime value. An error occurs if a required datetime item is missing.

Table 1-34: Datetime items that are required by the scalar function that converts a given character string representation into a datetime value

Scalar function	Required datetime items
DATE	Year, month, day

Scalar function	Required datetime items
TIME	Hour, minute, second
TIMESTAMP_FORMAT	Year, month, day, hour, minute, second

3. During the process of converting a character string representation into a datetime value, datetime format elements that are not relevant to converted data do not appear in the results.
4. During the process of converting a character string representation into a datetime value, any space that is not in the specified datetime format but that occurs between format elements in the character string is ignored.
5. The following table indicates the relationship between scalar functions in which a datetime format can be specified and datetime format elements.

Table 1-35: Relationship between scalar functions in which a datetime format can be specified and datetime format elements

Datetime format element	VARCHAR_FORMAT			DATE	TIME	TIMESTAMP_FORMAT
	DATE type[#1]	TIME type[#1]	TIMESTAMP type[#1]			
YYYY	Y[#2]	Y[#2] (Current year)	Y[#2]	R	Y[#5]	R
YY	Y[#2]	Y[#2] (Last 2 digits of current year)	Y[#2]	N	N	N
MM	Y[#3]	Y[#3] (Current month)	Y[#3]	R[#3]	Y[#3, #5]	R[#3]
MON	Y[#3]	Y[#3] (Abbr. name of current month)	Y[#3]	R[#3]	Y[#3, #5]	R[#3]
MONTH	Y[#3]	Y[#3] (Name of current month)	Y[#3]	R[#3]	Y[#3, #5]	R[#3]
DD	Y	Y (Current day)	Y	R	Y[#5]	R
HH	Y ('00')	Y	Y	Y[#5]	R	R

Datetime format element	VARCHAR_FORMAT			DATE	TIME	TIMESTAMP_FORMAT
	DATE type#1	TIME type#1	TIMESTAMP type#1			
MI	Y ('00')	Y	Y	Y#5	R	R
SS	Y ('00')	Y	Y	Y#5	R	R
FF	Y#4 ('00')	Y#4 ('00')	Y#4	N	N	N
NN...N	Y#4 ('00...0')	Y#4 ('00...0')	Y#4	Y#5	Y#5	Y

Legend:

R: Required and can be specified only once. It causes an error if not specified.

Y: Can be specified only once.

N: Cannot be specified; it causes an error if specified.

( ): Parentheses indicate the character string to be converted.

#1: Indicates the conversion of data of the type specified in the VARCHAR_FORMAT scalar function into a character string representation.

#2: Either YYYY or YY, but not both, can be specified only once.

#3: Only one of MM, MON, and MONTH can be specified only once.

#4: Either FF or NN...N, but not both, can be specified only once.

#5: This item, not relevant to the data type of the result, does not appear in the result.

1.12 Restrictions on the use of the inner replica facility

The use of the inner replica facility through the installation of HiRDB Staticizer Option and the use of updatable online reorganization are subject to the following restrictions:

- Using the inner replica facility

  SQL statements that can be executed on RDAREAs using the inner replica facility are subject to certain restrictions.

- Using updatable online reorganization

  SQL statements that can be executed on online reorganization-hold RDAREAs are subject to certain restrictions.

The following table lists the SQL statements that can be executed in conjunction with the inner replica facility.

Table 1-36: SQL statements that can be executed in conjunction with the inner replica facility

SQL		RDAREA status	
		Not on online reorganization hold	On online reorganization hold
Definition SQL	ALTER TABLE	C[#1]	N
	CREATE INDEX	C[#1]	N
	CREATE TABLE	C[#1, #2]	N
	DROP INDEX	C[#1]	N
	DROP SCHEMA	C[#1]	N
	DROP TABLE	C[#1]	N
	GRANT	C[#1]	C[#1]

SQL		RDAREA status	
		Not on online reorganization hold	On online reorganization hold
Data manipulation SQL	ASSIGN LIST statement	Y	C#1
	CALL statement	Y	C#1
	DELETE statement	Y	C#1
	FETCH statement	Y	C#1
	INSERT statement	Y	C#1
	PURGE TABLE statement	Y	N
	Single-row SELECT statement	Y	C#1
	Dynamic SELECT statement	Y	C#1
	UPDATE statement	Y	C#1
	Assignment statement	Y	C#1
Control SQL	LOCK statement	Y	C#1

Legend:

Y: Can be executed.

C: Can cause an error depending on certain conditions.

N: Cannot be executed.

#1: For details about the conditions, see the manual *HiRDB Version 9 Staticizer Option*.

#2: Falsification prevented tables cannot be created in an RDAREA for which the inner replica facility is used.

1.13 Locator

(1) Overview

A locator is data containing a 4-byte value that identifies a specific data value on the server. The use of a locator allows you to process an SQL statement that handles data without storing the entity for the data on a client.

(2) Rules

1. The following table lists types of locators that are available.

   Table 1-37: Types of locators

Type	Description
BLOB locator	Has a value that identifies BLOB type data values on the server.
BINARY locator	Has a value that identifies BINARY type data values on the server.

2. The following table lists how to specify a locator and locations where a locator can be specified.

   Table 1-38: Locator specification method and locations where one can be specified

Type	Specification method	Specifiable locations
BLOB locator	Embedded variable	Anywhere a BLOB type embedded variable can be specified.
BINARY locator	Embedded variable	Anywhere a BINARY type embedded variable can be specified. However, if the data type of the allocated data on the server is an embedded variable of a locator of the BINARY type with a maximum length of 32,001 bytes or greater, a BINARY locator can be specified only in those locations where BINARY-type embedded variables with a maximum length of 32,001 bytes or greater can be specified.

3. Locators to which data values on the server are not allocated are invalid.
4. When a locator is specified in one of the following locations, a data value is allocated to the locator, and the locator is enabled. In a given transaction, enabled locators can be specified in SQL statements, in the same way as data of the corresponding data type.
   - `INTO` clause of the single-row `SELECT` statement

- INTO clause of the FETCH statement
- Target of an assignment statement
- An argument in the CALL statement with respect to the OUT or INOUT parameter for the procedure
- INTO clause of the EXECUTE statement

5. A locator is disabled in the following cases:
   - When specified in the FREE LOCATOR statement
   - When the COMMIT statement is executed
   - When the ROLLBACK statement is executed
   - When the DISCONNECT statement is executed
   - Automatic COMMIT when the PURGE TABLE statement is executed
   - Automatic COMMIT during execution of a definition SQL statement when YES is specified in the PDCMMTBFDDL client environment variable
   - Transaction termination through implicit rollback
6. An error results if a disabled locator is specified in an SQL statement that handles allocated data or in the FREE LOCATOR statement.
7. If multiple locators that identify the same data on the server are created by specifying embedded variables for the locators in the target of assignment statement format 2 or in the assignment value, and if any of those locators is disabled, the other locators remain enabled.
8. If another set of data is allocated to an enabled locator to which data is already allocated, the original value of the locator remains enabled.
9. Overwriting the value of a locator by a UAP coding language disables the locator. In some cases, any data that is different from the data before the overwriting is identified.
10. Assigning the value of an enabled locator to a disabled locator by means of a UAP coding language enables the disabled locator, and the data that is the same as the assignment source locator is identified. In this case, disabling either locator disables both locators.

1.14 XML type

The XML type, which is the abstract data type for storing XML documents, can be used when HiRDB is used together with HiRDB XML Extension.

An XML type value is an XQuery sequence. For details about XQuery sequences, see *1.15 XQuery*.

Using SQL (SQL/XML) to process XML type values facilitates tasks such as filtering XML documents that match certain criteria, or retrieving the value of a particular part of the structure within an XML document.

1.14.1 XML type description format

(1) Format

In places where a data type is called for, the XML type is specified as follows:

```
XML
```

(2) Data format

XML documents are processed in a parsed format using the XML conversion command (phdxmlcnv) or an XML conversion library. This format is known as *ESIS-B* format.

(3) Maximum length

An XML type value in ESIS-B format cannot exceed 2,147,483,647 bytes. In ESIS-B format, the length depends not only on the length of the XML document, but also on its structure, such as the number of XML elements.

(4) Generating values

XML type values can be generated using the XML constructor function or the XMLPARSE function.

For details about the XML constructor function, see *1.14.2 XML constructor function*. For details about the XMLPARSE function, see *1.14.3(3) XMLPARSE*.

(5) Retrieving values

XML type values cannot be retrieved directly by a UAP. XML type values can be retrieved as VARCHAR or BINARY type values using the following method.

- Converting the XML type value to a VARCHAR or BINARY type value by means of the XMLSERIALIZE function

(6) Value expression specification

The following can be specified as XML type value expressions:

- Column specification
- XML constructor function
- XMLQUERY function
- XMLAGG set function
- XMLPARSE function

(7) Data types that can be converted (assigned or compared)

XML type values can be assigned only to columns, SQL variables, and SQL parameters defined as the XML type. Values of other data types cannot be assigned to the XML type. Furthermore, comparisons between other data types and the XML type are not permitted.

(8) Null value

If no value is generated by the XML constructor function, or if a null value is specified as an argument to the XML constructor function, its XML type value is the null value. If the null value is specified as an argument to the XMLPARSE function, the XML type value is also the null value.

(9) Notes on usage

1. The owner must access HiRDB XML Extension as MASTER in order to be able to use the XML type, as well as the following functions:
   - XML constructor function
   - XMLQUERY function
   - XMLSERIALIZE function
   - XMLEXISTS predicate
   - XMLAGG set function
   - XMLPARSE function
   - CREATE INDEX Format 3 (define substructure index)
2. The XML type cannot be used with the following facilities:
   - Indexes defined using CREATE INDEX Format 1
   - Sorting
   - Grouping
   - Set operations, arithmetic operations, and concatenation operations

- Set functions other than XMLAGG
- Exclusion of duplicates
- CASE expressions
- CAST specifications
- External reference columns

3. The only XML type values that can be stored in a column using an INSERT or UPDATE statement are values generated by an XML constructor function that were specified as an insertion value or update value.

1.14.2 XML constructor function

(1) Function

Generates an XML type value.

(2) Format

```
XML ( value-expression [AS data-type] )
```

(3) Rules

1. The following can be specified as the value expression:
   - SQL variable or SQL parameter
   - Embedded variable or ? parameter
2. For an XML document, use the XML conversion command or XML conversion library, and specify the resulting ESIS-B format output data type as BINARY.
3. If the value expression is an embedded variable or a ? parameter, the data type of the value expression must be specified with AS *data-type*. If AS *data-type* is specified, no value is permitted other than an embedded variable or a ? parameter.
4. The data type of the result is the XML type.
5. If the value expression is the null value, the result is also the null value.
6. The XML constructor function can appear by itself in the following places:
   - The insertion value of an INSERT statement
   - The update value of an UPDATE statement

(4) Example

The XML document stored in the embedded variable :bookinfo is inserted into BOOK_MANAGEMENT_TABLE.

```
INSERT INTO BOOK_MANAGEMENT_TABLE
     VALUES (310494321, XML(:bookinfo AS BINARY(102400)))
```

1.14.3 SQL/XML scalar functions

(1) XMLQUERY

(a) Function

Evaluates XQuery expressions and generates an XML type value as the result.

(b) Format

```
XMLQUERY ( XQuery-query
    PASSING BY VALUE XML-query-argument [, XML-query-argument]...
    [RETURNING SEQUENCE [BY VALUE]]
       EMPTY ON EMPTY )

XQuery-query ::= character-string-literal
XML-query-argument ::= {XML-query-context-items | XML-query-variable}
XML-query-context-items ::= value-expression [BY VALUE]
XML-query-variable ::= value-expression AS XQuery-variable-identifier [BY VALUE]
```

(c) Operands

- *XQuery-query* ::= *character-string-literal*

Specifies the XQuery query to evaluate as a character string literal. For details about how to specify queries in XQuery, see *1.15 XQuery*.

- *XML-query-argument* ::= {*XML-query-context-items*|*XML-query-variable*}

Specifies the argument to pass to the XQuery query.

The parent properties of the XQuery sequence items that are passed to the XQuery query as the XML query argument are empty. If you code nodes representing the same part of the same XML type value in different XML query arguments, they are treated as different nodes in the XQuery evaluation.

- *XML-query-context-items* ::= *value-expression* [BY VALUE]

Specifies the context items to be evaluated for the XQuery query.

If not specified, the XQuery query specified in *XQuery-query* is evaluated once, and no context items are set.

Note that only one such item can be specified in the PASSING clause.

value-expression

Specifies a value expression whose result evaluates to an XML type value.

Each XQuery item in the sequence that makes up the XML type value resulting from this value expression will be a context item in the XQuery evaluation.

If the result of the value expression is the null value, the result of the XMLQUERY function is also the null value.

The following can be specified:

- Column specification
- Column from a named derived table that was derived from a column specification

BY VALUE

This format is supported only for compatibility with ISO standards. It has no effect on how a value resulting from a value expression specified in an XML query context item is returned.

- *XML-query-variable* : := *value-expression* AS *XQuery-variable-identifier* [BY VALUE]

Specified in order to pass a value to the XQuery variable specified in the XQuery query

value-expression

Specifies the value expression to pass to the XQuery variable specified in the XQuery query.

If the result of the value expression is not the XML type, it is converted to an XQuery sequence whose value is the XML type before being passed.

The following table lists the data types that can be specified and their formats after conversion.

Table 1-39: Data types of values passed to XQuery variables and their XML type value formats after conversion

Data type	Format after conversion
INTEGER	Value of type xs:int#
SMALLINT	Value of type xs:int#
DECIMAL	Value of type xs:decimal#
FLOAT	Value of type xs:double#
SMALLFLT	Value of type xs:double#

Data type	Format after conversion
CHAR	Value of type xs:string#
VARCHAR	Value of type xs:string#
MCHAR	Value of type xs:string#
MVARCHAR	Value of type xs:string#
DATE	Value of type xs:date#
TIME	Value of type xs:time#
TIMESTAMP	Value of type xs:dateTime#
XML	Not converted (same format as the XML type value specified in the value expression)

#

Becomes an XQuery sequence consisting of a single atomic value.

An error results if the XQuery sequence cannot be converted.

If the result of the value expression is the null value, the value passed to the XQuery variable is an empty XQuery sequence.

The following can be specified in the value expression:

- Literal
- USER, CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP
- Column specification
- SQL variable or SQL parameter
- Arithmetic operation, date operation, time operation, or concatenation operation
- Set function
- Scalar function
- Function invocation
- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

- Column from a named derived table that was derived from any of the above value expressions

AS *XQuery-variable-identifier*

Specifies the identifier of the XQuery variable to which the value is passed.

The XQuery variable identifier specifies an XQuery variable name that is specified in the XQuery query. For details about specifying names, see *1.1.7 Specification of names*.

The XML namespace of the XQuery variable identifier specified here is the default XML namespace of the XQuery query.

The effective scope of the XQuery variable corresponding to the XQuery variable identifier specified here is the entire XQuery query.

You cannot specify XQuery variable identifiers that are specified in other XML query variables in the same PASSING clause.

BY VALUE

This format is supported only for compatibility with ISO standards. It has no effect on how a value resulting from a value expression specified in an XML query variable is returned.

Specify this only if the result of the value expression is the XML type. An error results if the result of the value expression is not the XML type.

RETURNING SEQUENCE [BY VALUE]

This format is supported only for compatibility with ISO standards. It has no effect on either the format or the return method of a value resulting from the XQuery evaluation.

(d) Rules

1. The result is an XQuery sequence returned in an XML type value. The parent properties of the XQuery items comprising the XQuery sequence that is the resulting value is empty.
2. This function can only be specified in places where function invocation is permitted.
3. In an INSERT or UPDATE statement, the results of this function, or XML type values based on the results, cannot be stored in a column.

(e) Example

Retrieve book titles for the category specified in the embedded variable :category from the book_info column of BOOK_MANAGEMENT_TABLE.

```
SELECT XMLSERIALIZE(
```

```
    XMLQUERY('/book_info[category=$CATEGORY]/title'
             PASSING BY VALUE book_info,
             :category AS CATEGORY
             RETURNING SEQUENCE BY VALUE
             EMPTY ON EMPTY)
    AS VARCHAR(32000))
FROM BOOK_MANAGEMENT_TABLE
```

(2) XMLSERIALIZE

(a) Function

Generates a VARCHAR or BINARY type value out of an XML type value that has been serialized (converted to a string).

(b) Format

```
XMLSERIALIZE ( [CONTENT]
    value-expression AS data-type
    [VERSION '1.0']
    [{INCLUDING XMLDECLARATION
     |EXCLUDING XMLDECLARATION}])
```

(c) Operands

- CONTENT

This format is supported only for compatibility with ISO standards. It has no effect on processing.

The result does not have to be in the format of a well-formed XML document.

- *value-expression*

Specify an XML type value from which to generate a VARCHAR or BINARY type value.

The following can be specified:

- Column specification
- XMLQUERY function
- XMLAGG set function
- Column from a named derived table that was derived from a column specification

- AS *data-type*

Specify the data type of the result.

The following data types can be specified:

- VARCHAR type (cannot include a character set specification)
- BINARY type

An error results if the result cannot fit into the maximum length of the specified data type.

- [VERSION '1.0']

Specifies the XML version of the result. If omitted, '1.0' is assumed.

- [{INCLUDING XMLDECLARATION | EXCLUDING XMLDECLARATION}]

Specifies whether to include an XML declaration (example: <?xml version="1.0" encoding="UTF-8"?>) in the resulting XML code. If omitted, EXCLUDING XMLDECLARATION is assumed.

INCLUDING XMLDECLARATION

The resulting XML code includes the XML declaration.

EXCLUDING XMLDECLARATION

The resulting XML code does not include the XML declaration.

(d) Rules

1. If the value expression is the null value, the result is also the null value.
2. The character encoding of the result will be the character encoding specified in the pdntenv command (pdsetup command in the UNIX edition).
3. The serialized result is a character string containing the concatenation of the results of every item in the XQuery sequence listed in *value-expression* being converted according to the rules listed below. The concatenation includes a single space character inserted between adjacent atomic values.
   - The result of serializing a document node is the character string concatenation of the results of serializing the child nodes.
   - The result of serializing an element node is the character string representation of that element node as described below. If the element node has child nodes, the result will be the serialized character string starting from the start tag. If the element node has no child node, the result will be the serialized empty element tag.

```
element-node-character-string-representation ::= {start-tag child-node-character-string-representation
end-tag | empty-element-tag}
child-node-character-string-representation ::= {element-node-character-string-representation |
processing-instruction-node-character-string-representation |
        comment-node-character-string-representation | text-node-character-string-representation}

start-tag ::= < qualified-name [space XML-namespace-character-string-representation] . . .
              [space attribute-node-character-string-representation] . . . >
end-tag ::= </ qualified-name >
empty-element-tag ::= < qualified-name [space XML-namespace-character-string-representation] . . .
                      [space attribute-node-character-string-representation] . . . />
```

XML-namespace-character-string-representation

A character string expressed in as an attribute node character string representation of an XML attribute that indicates an XML namespace, not specified in any ancestor element node, from among the valid XML namespaces in, or below, the current element node.

- The result of serializing an attribute node will be a character string representation of the attribute node, as follows:

```
attribute-node-character-string-representation ::= qualified-name equals-sign-operator "attribute-value"
```

attribute-value

In the string value of the attribute node, the characters listed below will be replaced with their corresponding character strings.

Character to replace	Corresponding character string
& (ampersand)	&amp;
< (less-than sign)	&lt;
> (greater-than sign)	&gt;
" (double quotation mark)	&quot;
' (single quotation mark)	&apos;

- The result of serializing a processing instruction node will be a character string representation of the processing instruction node, as shown below:

processing-instruction-node-character-string-representation ::= <? *processing-instruction-target space* [*processing-instruction-node-contents*] ?>

- The result of serializing a comment node will be a character string representation of the comment node, as shown below:

comment-node-character-string-representation ::= <!-- *comment-node-contents* -->

- The result of serializing a text node will be a character string representation of the text node, as shown below:

text-node-character-string-representation ::= *text-value*

text-value

In the string value of the text node, the characters listed below will be replaced with their corresponding character strings.

Character to replace	Corresponding character string
& (ampersand)	&amp;
< (less-than sign)	&lt;
> (greater-than sign)	&gt;

- The result of serializing an atomic values will be a character string of the atomic value that has been converted to type `xs:string`, in which the characters listed below are replaced with their corresponding character strings.

Character to replace	Corresponding character string
& (ampersand)	&amp;
< (less-than sign)	&lt;
> (greater-than sign)	&gt;

4. This function can only be specified in places where function invocation is permitted.

(e) Example

Retrieve the value of the book_info column from BOOK_MANAGEMENT_TABLE as a value of type VARCHAR.

```
  SELECT XMLSERIALIZE(book_info AS VARCHAR(32000)
                      INCLUDING XMLDECLARATION) FROM
BOOK_MANAGEMENT_TABLE
```

(3) XMLPARSE

(a) Function

Generates an XML type value from an XML document.

(b) Format

```
XMLPARSE(  DOCUMENT value-expression
     [AS data-type]
     [WHITESPACE-specification] )

WHITESPACE-specification ::={PRESERVE | STRIP} WHITESPACE
```

(c) Operands

- DOCUMENT

Indicates that the XML document specified in the value expression is a well-formed XML document. This operand cannot be omitted.

- *value-expression*

Specifies a well-formed XML document from which to generate an XML type value. The following data types can be specified:

- CHAR type (cannot include a character set specification)
- VARCHAR type (cannot include a character set specification)
- MCHAR type
- MVARCHAR type
- BINARY type

- AS *data-type*

Specifies the data type of the value expression. If the value expression is an embedded variable or a ? parameter, AS *data-type* is required. If AS *data-type* is specified, the value expression must be an embedded variable or a ? parameter.

■ *WHITESPACE-specification* ::={PRESERVE | STRIP} WHITESPACE

Specifies how to handle whitespace in the XML document. Whitespace consists of space (X'20'), horizontal tab (X'09'), newline (X'0A'), and carriage return (X'0D'). If omitted, STRIP WHITESPACE is assumed.

Regardless of the WHITESPACE specification, all carriage return (X'0D') and combined carriage return-newline characters (X'0D0A') are replaced with a newline (X'0A') before processing *WHITESPACE-specification*.

PRESERVE WHITESPACE

Retains all whitespace.

STRIP WHITESPACE

The following normalization process is performed on whitespace contained in text nodes, except for the text node descendants of elements with the xml:space="preserve" attribute.

- Remove leading and trailing whitespace
- Replace contiguous whitespace with a single space (X'20')

(d) Rules

1. If the value expression is the null value, the result is also the null value.
2. This function can only be specified in places where function invocation is permitted.
3. DTDs, XML schemas, comments, and processing instructions are ignored rather than processed as described above. The validity of the XML document specified in the value expression is not checked.
4. The only entities that can be used are the predefined entities (< > & ' "). No internal or external entities other than these are permitted; if used, they will be treated as characters.
5. Character code classifications that can be specified in the encoding attribute in the XML declaration are listed in the following table. By default, UTF-8 is assumed.

Encoding attribute of XML document	HiRDB character encoding		
	sjis (Shift JIS kanji)	ujis (EUC Japanese kanji)	utf-8 (Unicode (UTF-8))
Shift_JIS	Y	N	N
EUC-JP	N	Y	N
UTF-8	N	N	Y

Encoding attribute of XML document	HiRDB character encoding		
	sjis (Shift JIS kanji)	ujis (EUC Japanese kanji)	utf-8 (Unicode (UTF-8))
US-ASCII	Y	Y	Y
Other than the above	N	N	N

Legend:

Y: Can be specified.

N: Cannot be specified.

6. XML documents up to 5 megabytes in size can be specified in the value expression.
7. Element names and attribute names up to 4,096 bytes in length can be defined.
8. Up to 100 nested elements can be defined.
9. When defining prefixes, the rules differ depending on the version of HiRDB XML Extension you use.
   - HiRDB XML Extension versions earlier than 08-05:

     Even if prefixes are defined, processing is performed as if no prefix were specified, and all elements and attributes are considered to be in the default XML namespace.
   - HiRDB XML Extension version 08-05 or later:

     Follows the environment assignments (`XMLPARSE` namespace processing specifications) for the XML Extension's XML data type plug-in. For details about environment assignments for the XML data type plug-in, see the manual *HiRDB Version 9 XML Extension*.

(e) Example

Insert the XML document stored in the `BINARY` type embedded variable `:bookxml` into `BOOK_MANAGEMENT_TABLE`.

```
  INSERT INTO BOOK_MANAGEMENT_TABLE
    VALUES(310494321, XMLPARSE(DOCUMENT :bookxml AS
BINARY(32000)))
```

(f) Notes

1. This function requires HiRDB XML Extension version 08-04 or later.

2. If the character encodings of the HiRDB server and client are different, set the data type of the value expression to BINARY.

1.14.4 SQL/XML predicates

(1) XMLEXISTS predicate

(a) Function

Determines whether an XML type value, resulting from the evaluation of one or more XQuery arguments, is an XQuery sequence that is composed of one or more XQuery items.

(b) Format

```
XMLEXISTS (XQuery-query
             PASSING BY VALUE XML-query-argument [, XML-query-argument]... )

XQuery-query ::= character-string-literal
XML-query-argument ::= {XML-query-context-items|XML-query-variable}
XML-query-context-items ::= value-expression [BY VALUE]
XML-query-variable ::= value-expression AS XQuery-variable-identifier [BY VALUE]
```

(c) Operands

- *XQuery-query* ::= *character-string-literal*

Specifies the XQuery query to evaluate as a character string literal. For details about how to specify XQuery queries, see *1.15 XQuery*.

- *XML-query-argument* ::= {*XML-query-context-items*|*XML-query-variable*}

Specifies the argument to pass to the XQuery query.

The parent properties of the XQuery sequence items that are passed to the XQuery query as the XML query argument are empty. If you code nodes representing the same part of the same XML type value in XQuery sequences specified in different XML query arguments, they are treated as different nodes in the XQuery evaluation.

- *XML-query-context-items* ::= *value-expression* [BY VALUE]

Specifies the context items to be evaluated for the XQuery query.

If not specified, the XQuery query specified in *XQuery-query* is evaluated once, and no context items are set.

Note that only one such item can be specified in the PASSING clause.

value-expression

Specifies a value expression whose result evaluates to an XML type value.

Each XQuery item in the sequence that makes up the XML type value that results from this value expression will be a context item in the XQuery evaluation.

If the result of the value expression is the null value, the XMLEXISTS predicate is undefined.

The following can be specified:

- Column specification
- Column from a named derived table that was derived from a column specification

BY VALUE

This format is supported only for compatibility with ISO standards. It has no effect on how a value resulting from a value expression specified in an XML query context item is returned.

■ *XML-query-variable* ::= *value-expression* AS *XQuery-variable-identifier* [BY VALUE]

Specified in order to pass a value to an XQuery variable specified in the XQuery query.

value-expression

Specifies the value expression to pass to the XQuery variable specified in the XQuery query.

If the result of the value expression is not the XML type, it is converted to an XQuery sequence whose value is the XML type before being passed.

The following table lists the data types that can be specified and their formats after conversion.

Table 1-40: Data types of values passed to XQuery variables and their XML type value formats after conversion

Data type	Format after conversion
INTEGER	Value of type xs:int#
SMALLINT	Value of type xs:int#
DECIMAL	Value of type xs:decimal#
FLOAT	Value of type xs:double#
SMALLFLT	Value of type xs:double#
CHAR	Value of type xs:string#

Data type	Format after conversion
VARCHAR	Value of type xs:string#
MCHAR	Value of type xs:string#
MVARCHAR	Value of type xs:string#
DATE	Value of type xs:date#
TIME	Value of type xs:time#
TIMESTAMP	Value of type xs:dateTime#
XML	Not converted (same format as the XML type value specified in the value expression)

#

Becomes an XQuery sequence consisting of a single atomic value.

An error results if the XQuery sequence cannot be converted.

If the result of the value expression is the null value, the value passed to the XQuery variable is an empty XQuery sequence.

The following can be specified in the value expression:

- Literal
- USER, CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP
- Column specification
- SQL variable or SQL parameter
- Arithmetic operation, date operation, time operation, or concatenation operation
- Scalar function
- Function invocation
- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery
- Column from a named derived table that was derived from any of the above value expressions

AS *XQuery-variable-identifier*

Specifies the identifier of the XQuery variable to which the value is passed.

The XQuery variable identifier specifies an XQuery variable name that is specified in the XQuery query. For details about specifying names, see *1.1.7 Specification of names*.

The XML namespace of the XQuery variable identifier specified here is the default XML namespace of the XQuery query.

The effective scope of the XQuery variable corresponding to the XQuery variable identifier specified here is the entire XQuery query.

You cannot specify XQuery variable identifiers that are specified in other XML query variables in the same PASSING clause.

BY VALUE

This format is supported only for compatibility with ISO standards. It has no effect on how a value resulting from a value expression specified in an XML query variable is returned.

Specify this only if the result of the value expression is the XML type. An error results if the result of the value expression is not the XML type.

(d) Conditions under which predicate is TRUE

The XMLEXISTS predicate is TRUE if the resulting XML type value of the XQuery evaluation is an XQuery sequence consisting of one or more XQuery items.

(e) Rules

1. This predicate can be specified only in a WHERE clause.
2. It cannot be specified in a subquery.

(f) Example

Retrieve from BOOK_MANAGEMENT_TABLE the values of the book_id column in the rows stored in bookinfo where the price is greater than or equal to the value specified in the embedded variable :price.

```
SELECT book_id FROM BOOK_MANAGEMENT_TABLE
  WHERE XMLEXISTS('/bookinfo[price>=$PRICE]'
                  PASSING BY VALUE bookinfo,
                  :price AS PRICE)
```

1.14.5 SQL/XML set functions

(1) XMLAGG

(a) Function

Generates an XML type value from the concatenation of values from a collection of rows values where each row of the argument specifies the XML type.

(b) Format

```
XMLAGG ( value-expression
         [RETURNING SEQUENCE])
```

(c) Operands

- *value-expression*

Specifies the concatenated XML type value expression.

The following can be specified:

- Column specification
- XMLQUERY function
- Column from a named derived table that was derived from a column specification

- RETURNING SEQUENCE

This format is supported only for compatibility with ISO standards. It has no effect on the format of the value of the concatenation result.

(d) Rules

1. The result is an XQuery sequence whose value is the XML type.
2. The order of concatenation is not guaranteed.
3. Those rows for which the result of the value expression is the null value are ignored.
4. The result is the null value if the target is empty or consists solely of null values.
5. For INSERT and UPDATE statements, the result of this function (or XML type values based on the result) cannot be stored in a column.

For details about other rules, see the rules in *2.14 Set functions*.

(e) Example

From the bookinfo column of every row of BOOK_MANAGEMENT_TABLE, retrieve book information of the same category as the book whose title is SQL Explained.

```
  SELECT XMLSERIALIZE(
    XMLQUERY(
      '$BOOKS/bookinfo[category=$BOOKS/bookinfo[title="SQL
Explained"]/category]'
      PASSING BY VALUE XMLAGG(bookinfo) AS BOOKS
      RETURNING SEQUENCE BY VALUE EMPTY ON EMPTY)
  AS VARCHAR(32000))
    FROM BOOK_MANAGEMENT_TABLE
```

1.14.6 Definition SQL for the XML type

(1) CREATE INDEX Format 3 (Define substructure index)

(a) Function

Defines an index for a column of type XML, with a specified substructure as the key.

(b) Usage privileges

Owner of a table

A user can define an index for a table they own in the public user RDAREA.

Owner of a table with private user RDAREA usage privileges

This user can define an index for a table they own in the private user RDAREA for which they have usage privileges.

(c) Format (Define substructure index)

```
CREATE [UNIQUE] INDEX [authorization-identifier.]index-identifier
   ON [authorization-identifier.]table-identifier (column-name [{ASC|DESC}])
   [IN{RDAREA-name
     |(RDAREA-name)
     |((RDAREA-name)[, (RDAREA-name)]...)
     |RDAREA-specification-for-the-matrix-partitioning-index}]
   KEY [USING UNIQUE TAG]FROM substructure-specification AS data-type
   [index-option]...

RDAREA-specification-for-the-matrix-partitioning-index
::=RDAREA-specification-for-two-dimensional-storage
RDAREA-specification-for-two-dimensional-storage::=
        (matrix-partitioning-RDAREAs-list [, matrix-partitioning-RDAREAs-list]...)
matrix-partitioning-RDAREAs-list ::= (RDAREA-name [, RDAREA-name]...)
substructure-specification ::= character-string-literal
index-option ::={PCTFREE=percentage-of-unused-space
              |UNBALANCED SPLIT
              |EMPTY}
```

(d) Operands

For rules and other details about operands other than the KEY clause, see *CREATE INDEX Format 1 (Define index)* in Chapter 3.

- KEY [USING UNIQUE TAG] FROM *substructure-specification* AS *data-type*

USING UNIQUE TAG

Specifies to uniquely determine the substructure in *substructure-specification* from among the values in a single XML type column. However, if the substructure is an XML attribute, XML elements that have that XML attribute must be uniquely determined.

If the UNIQUE operand immediately follows CREATE, this operand is assumed.

substructure-specification ::= *character-string-literal*

Specifies that the substructure to be used as the key is expressed as a character string literal in the format of a substructure path.

The substructure path format is shown below. Note that all keywords in the substructure path are specified in lower case.

```
substructure-path ::=[XML-namespace-declaration]... substructure-path-expression
XML-namespace-declaration ::={declare namespace prefix = XML-namespace-URI;
            |declare default element namespace XML-namespace-URI;}

substructure-path-expression ::= /step-expression [/step-expression]...
step-expression ::= [@] qualified-name
qualified-name ::= [prefix:]local-name
```

- declare namespace *prefix* = *XML-namespace-URI*;

  If the substructure path expression contains qualified names with a prefix, this operand declares the XML namespace of the prefix. The same prefix cannot have multiple XML namespace declarations.

  prefix

  Specifies the prefix of the qualified name in the substructure path expression.

  This operand is case-sensitive.

  XML-namespace-URI

  The URI of the XML namespace associated with the prefix is specified, enclosed in double quotation marks (") or single quotation marks (').

  This operand is case-sensitive.

- declare default element namespace *XML-namespace-URI*;

Declares the default XML namespace for the substructure path expression. Any unprefixed qualified names in the substructure path expression will be searched for in the XML namespace declared here.

You cannot declare multiple default XML namespaces in a substructure path.

If omitted, the default XML namespace URI `http://www.w3.org/XML/1998/namespace` is assumed.

XML-namespace-URI

Specifies the URI of the XML namespace for any unprefixed qualified names, enclosed in double quotation marks (`"`) or single quotation marks (`'`).

This operand is case-sensitive.

- *substructure-path-expression*

  Specifies the path of the substructure that is issued as the key.

  The following table describes the meaning of each element in the specification:

Element	Meaning
Leading forward slash (/)	Indicates that the element in the step expression after the forward slash is the topmost XML element of the `XML` type value.
Middle forward slash (/)	Indicates that the XML element or attribute in the step expression after the forward slash is a child element or an attribute of the XML expression in the step expression specified before the forward slash.
step-expression	Indicates an XML element or attribute. However, a step expression referring to an XML attribute can be specified only at the end of the expression.
@	Indicates that the qualified name after the at mark (@) is the name of an XML attribute. A qualified name without @ refers to an XML element.
qualified-name	Indicates the name of an XML element or XML attribute.
prefix：	Indicates that the immediately following *local-name* is a local name within the XML namespace indicated by the prefix. The prefix that is bound to the XML namespace is specified in the XML namespace declaration. If *prefix*： is omitted, *local-name* is the name of the default XML namespace.
local-name	Indicates the name of an XML element or attribute in the XML namespace.

The following table provides examples of substructure specifications:

No.	Example	Meaning
1	`'/bookinfo/category'`	The key is the XML element `category`, which is a child of `bookinfo`, the topmost XML element.
2	`'/bookinfo/@book_id'`	The key is the attribute `book_id`, which is a child of `bookinfo`, the topmost XML element.
3	'declare namespace b="http://www.foo.co.jp/bookinfo"; `/b:bookinfo/b:category'`	The key is the XML element `category`, which is a child of `bookinfo`, the topmost XML element. However, the names of the XML elements `bookinfo` and `category` are located in the XML namespace whose URI is `http://www.foo.co.jp/bookinfo`.
4	'declare default element namespace "http://www.foo.co.jp/bookinfo"; `/bookinfo/category'`	Same as No. 3.

data-type

Specifies the data type of the key value.

The following data types can be specified:

- INTEGER
- DECIMAL
- FLOAT
- VARCHAR (cannot include a character set specification)

The value of the key is determined after the values of the `typed-value` properties of the nodes in the XQuery data model substructure that is used as the key are converted to the SQL data type specified here.

When an index is defined, if the values stored in the column cannot be converted to the SQL data type specified for the substructure that is used as the key, the index cannot be defined. Furthermore, when inserting or updating values in the column that defines the index, if the values to be inserted or updated cannot be converted to the SQL data type specified for the substructure that is used as the key, the insert or update operation cannot be performed.

For details about the value of the `typed-value` property for each kind of node, see *1.15.1 XQuery data model*. The following table indicates the relationship between the values of the `typed-value` properties of the XQuery data types and their corresponding SQL data types.

Table 1-41: Correspondence between XQuery data types and SQL data types

XQuery data type of typed-value properties of nodes in an XQuery data model substructure	SQL data type			
	INTEGER	DECIMAL	FLOAT	VARCHAR
xs:untypedAtomic	C	C	C	C
xs:int	E	NC	NC	NC
xs:decimal	NC	E	NC	NC
xs:double	NC	NC	E	NC
xs:string	NC	NC	NC	E
Other	NC	NC	NC	NC

Legend:

E: Equivalent types, so the key value is used without conversion.

C: When the value of a `typed-value` property is recognized as a character string, if the character string representation corresponds to an SQL data type, the value is converted and becomes the key value. In this case, it may produce different results than if no index was defined. Otherwise it cannot be converted.

NC: Cannot be converted.

(e) Common rules

1. A maximum of 255 indexes can be created for a table.
2. An index can be defined for columns that contain the null value, columns that do not have any rows, and columns that include `XML` type values that are not part of the specified substructure.
3. A substructure index can be specified only for `XML` type columns.
4. The length of the key value must satisfy the following formula:

   Length of key value ≤

   MIN((*page size of index storage RDAREAs* ÷ 2) - 1242, 4036)

   The following table indicates the length of each key value.

   Table 1-42: Lengths of key values

Data type	Length of key value
INTEGER	4

Data type		Length of key value
DECIMAL[(*m*[, *n*])]		↓*m* ÷ 2↓ + 1
FLOAT		8
VARCHAR	Actual data length is 255 bytes or less	*n1* + 1
	Actual data length is 256 bytes or more	*n1* + 2

Legend:

m, *n*: Positive integers

n1: Actual data length

5. Only one index can be defined with the same substructure of the same column being used as the key.
6. When the substructure in which the index is defined is evaluated in XQuery, it is evaluated as a value typed in the XQuery data type that corresponds to the SQL data type specified in `AS` *data-type*. The following table indicates the XQuery data type evaluated from each of these SQL data types.

Table 1-43: XQuery data type when an SQL data type is evaluated in XQuery

SQL data type	XQuery data type
INTEGER	xs:int
DECIMAL	xs:decimal
FLOAT	xs:double
VARCHAR	xs:string

If the XQuery data type of the `typed-value` property of the node in the XQuery data model node substructure in which the index is defined is the `xs:untypedAtomic` type, the XQuery data type evaluated in XQuery will differ depending on whether an index corresponding to that substructure is defined. Therefore, the results may differ if the index is deleted.

7. If a procedure and a trigger are already defined for the table in which you are defining an index, the index information in the SQL object is lost (invalidated) and the trigger cannot be executed. Because the affected procedure or trigger cannot be executed from another procedure, you must re-create the SQL object.
8. The same index option cannot be specified more than once.
9. `CREATE INDEX` cannot be executed from a Java procedure if the execution

invalidates the index information of the SQL object being executed.

10. You cannot mix RDAREAs that use the inner replica facility with those that do not using the facility in the index storage RDAREAs. When specifying an RDAREA that use the inner replica facility, specify the name of the original RDAREA.
11. For details about execution conditions affecting a `CREATE INDEX` statement that uses the inner replica facility, see the *HiRDB Version 9 Staticizer Option Description and User's Guide*.
12. A maximum of 500 indexes can be stored per RDAREA.

(f) Notes

1. When a value in an indexed column is updated, the associated index is also updated.
2. The `CREATE INDEX` statement cannot be specified from an X/Open-compliant UAP running under OLTP.
3. When an index that specifies the `EMPTY` option is defined, it must be re-created with the database reorganization utility; for details, see the manual *HiRDB Version 9 Command Reference*.
4. Using the `CREATE INDEX` statement to create an index for a table containing a large quantity of data can require substantial processing time. Before executing the `CREATE INDEX` statement, set a timer for monitoring purposes, as follows:
   - System definition `pd_watch_time`

     Either specify `0` or omit.
   - Client environment definition `PDCWAITTIME`

     If you can estimate how long `CREATE INDEX` will take to execute based on the amount of data and past experience, specify a non-zero value with some extra leeway.

     If you cannot estimate how long it will take to execute, specify `0` or omit.
5. When defining an index on a table with rows partitioned across servers on a HiRDB/Parallel Server, if the RDAREA for tables or the RDAREA for the specified index are in a blocked state, the RDAREA state changes to lock-release pending. At this time, even if the lock-release waiting time reaches the value specified in the `pd_lck_wait_timeout` operand of the system definition and times out, it may not immediately return an error. To avoid this situation, cancel the blocked state of the RDAREA for tables or the RDAREA for the specified index before executing the `CREATE INDEX` statement.

(g) Example

1. For the bookinfo column in BOOK_MANAGEMENT_TABLE, define the index INDX1 whose key is converted into VARCHAR type from the XML element category, which is a child of the XML element bookinfo.

```
CREATE INDEX INDX1 ON BOOK_MANAGEMENT_TABLE(bookinfo)
  KEY FROM '/bookinfo/category' AS VARCHAR (100)
```

2. For the bookinfo column in BOOK_MANAGEMENT_TABLE, define the index INDX1 whose key is converted into INTEGER type from the XML element price, which is a child of the XML element bookinfo. The XML element price, which is a child of the XML element bookinfo, must be uniquely determined in the values of each row.

```
CREATE INDEX INDX2 ON BOOK_MANAGEMENT_TABLE(bookinfo)
  KEY USING UNIQUE TAG FROM '/bookinfo/price' AS INTEGER
```

3. For the bookinfo column in BOOK_MANAGEMENT_TABLE, define the index INDX3 whose key is converted into INTEGER type from the value of the XML attribute book_id, which is a child of the XML element bookinfo. The XML element bookinfo is uniquely determined among the values of each row, and the key value must be different in every row.

```
  CREATE UNIQUE INDEX INDX3 ON
BOOK_MANAGEMENT_TABLE(bookinfo)
    KEY FROM '/bookinfo/@book_id' AS INTEGER
```

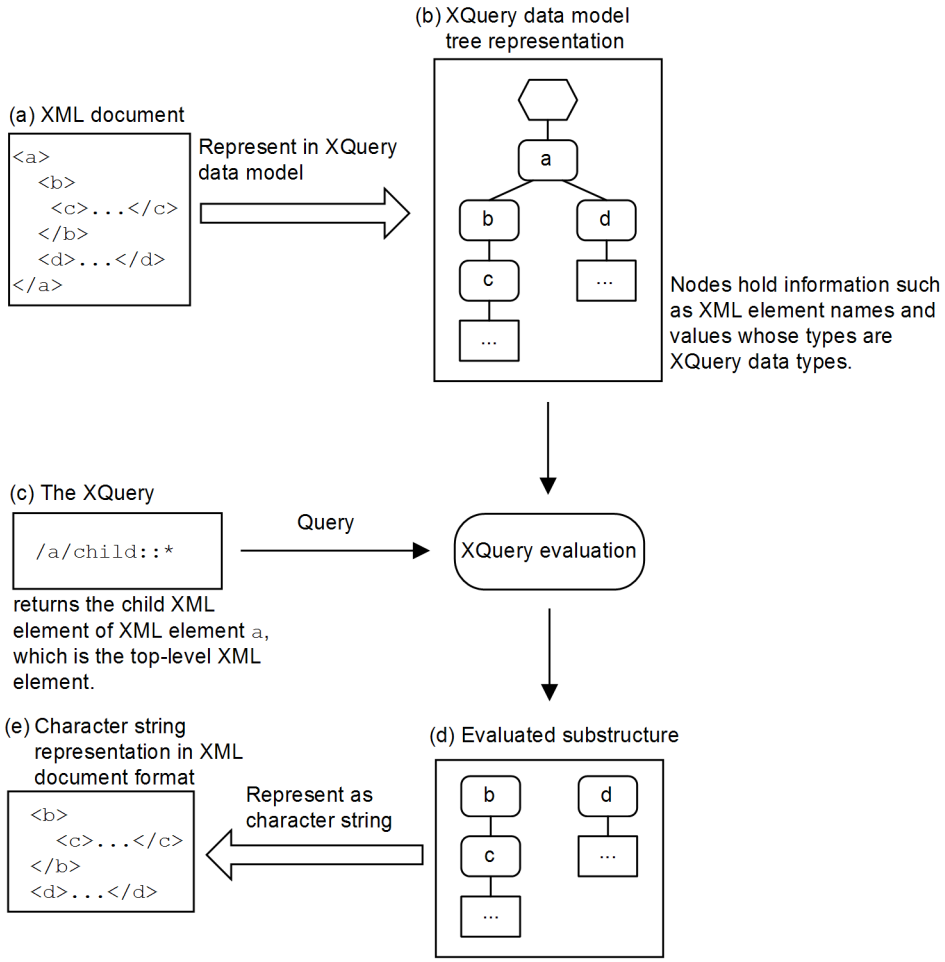
1.15 XQuery

XQuery is a language for querying the contents of XML documents represented as values in the XQuery data model. In HiRDB, the contents of XML documents are stored in `XML` type values, and XQuery expressions are specified in the `XMLQUERY` function and the `XMLEXISTS` predicate in order to extract specific substructures from the XML document.

The XQuery data model represents XML document information as a tree structure. The XQuery data model consists of six kinds of nodes. These nodes hold information such as the names used to identify XML elements and attributes (qualified names), and the atomic values whose types are XQuery data types. XQuery is used to execute queries on values in the XQuery data model that are represented in the nodes and atomic values that hold such information.

The figure below shows a conceptual diagram of a query in XQuery. An XML document (*(a)* in the figure) is represented as a tree (*(b)* in the figure) in the XQuery data model. For the XQuery data model tree, the XQuery query that was entered (*(c)* in the figure) is evaluated, and a specified substructure (*(d)* in the figure) is extracted from the tree. The substructure extracted by the XQuery query also becomes a value represented in the model. The extracted substructure can be represented as a character string in the XML document format (*(e)* in the figure).

Figure 1-11: Conceptual diagram of a query in XQuery



1.15.1 XQuery data model

The XQuery data model is a model for representing XML documents for processing by XQuery. The XQuery data model is a tree structure that holds nodes.

Nodes are of one of the kinds listed below. Each node also holds information in the form of properties.

- Document node
- Element node
- Attribute node

- Processing instruction node
- Comment node
- Text node

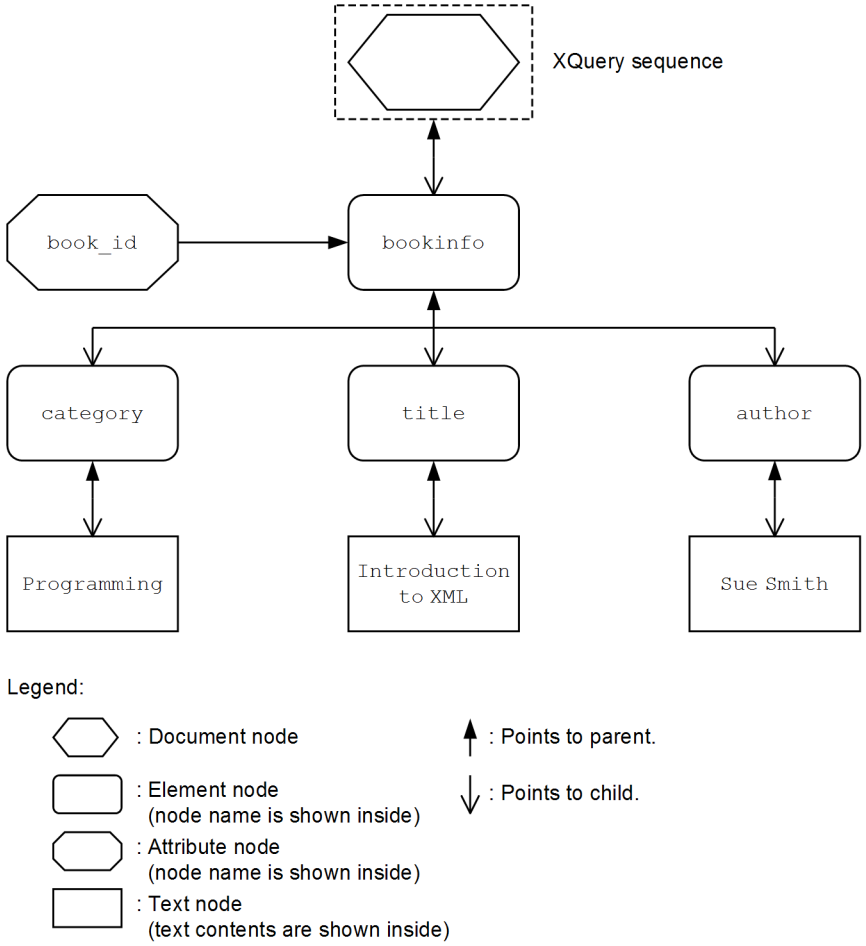
A node or an atomic value is called an XQuery item. An ordered collection of zero or more XQuery items is called an XQuery sequence. Values are represented in the XQuery data model as XQuery sequences.

As an example, the figure below shows a tree representation of an XML document in the XQuery data model. The XQuery sequence corresponding to the example XML document includes only the document node inside the dotted lines in the figure. Because the document node designates the descendant nodes holding the contents of the XML document, the document node itself can be treated as representing the entire XML document.

XML document

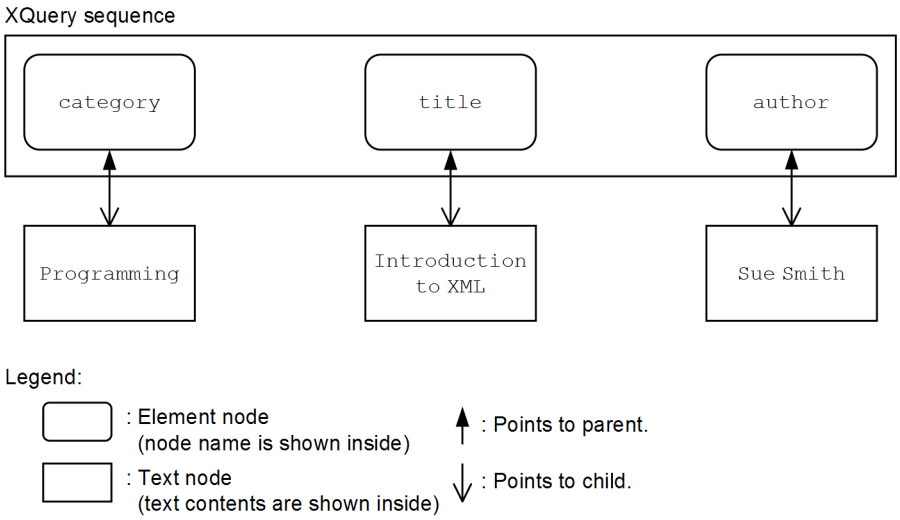
```
<bookinfo book_id="310494321">
  <category>programming</category>
  <title>Introduction to XML</title>
  <author>Sue Smith</author>
</bookinfo>
```

Figure 1-12: XQuery data model example



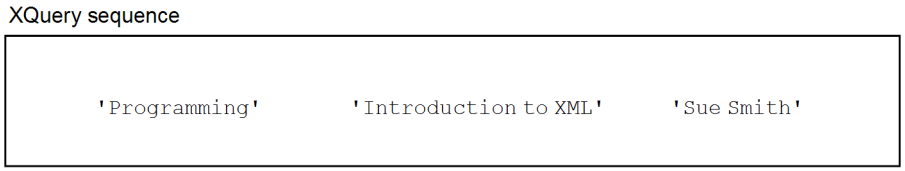
For the XML document shown in *Figure 1-12*, the result of the XQuery query `/bookinfo/child::element()` (extract the element node that is the child of the `bookinfo` element node), shown within the dotted lines in the figure, is an XQuery sequence consisting of three nodes.

Figure 1-13: Example of XQuery sequence consisting of nodes



For the XML document shown in *Figure 1-12*, the result of the XQuery query `fn:data(/bookinfo/child::element()/text())` (extract the atomic value of the text node of the element node that is the child of the bookinfo element node), shown within the dotted lines in the figure below, is the XQuery sequence consisting of the three atomic values `'Programming'`, `'Introduction to XML'`, and `'Sue Smith'`.

Figure 1-14: Example of an XQuery sequence consisting of atomic values



(1) Order of nodes

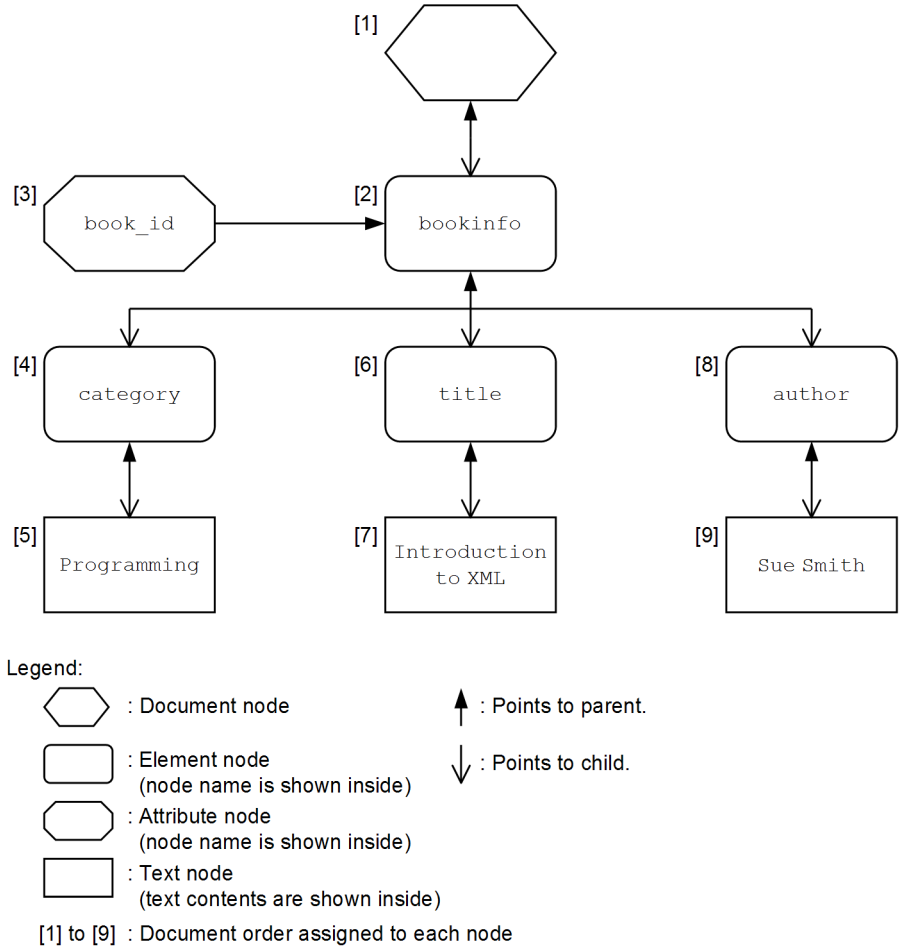
Nodes are given an ordering, called *document order*. In document order, the XML elements and attributes of each node are considered to be in the order in which they appear in the XML document. The document order rules are as follows:

1. Every node appears before any of its descendant nodes.
2. An attribute node appears immediately following its associated element node.
3. The order of siblings is the order in which they appear in the `children` property of the parent node.

4. Descendant nodes appear before sibling nodes.

The following figure shows the document order given to the nodes of the tree in *Figure 1-12*:

Figure 1-15: Example of document order



(2) Property types

The following table lists the properties that nodes can have.

Table 1-44: Property types and their contents

No.	Property name	Explanation
1	node-name	Qualified name of an XML element or XML attribute

No.	Property name	Explanation
2	`namespace`	XML namespace to which the node belongs
3	`parent`	Parent node
4	`children`	XQuery sequence of child nodes
5	`attribute`	XQuery sequence of attribute nodes
6	`type-name`	Qualified name of the XQuery data type of an XML element or XML attribute
7	`string-value`	Value of the `xs:string` type character string representing the text that is specified as the node contents. See `fn:string` function.
8	`typed-value`	Value of the XQuery data type obtained from the `string-value` property. See `fn:data` function.
9	nilled	Property that indicates whether the contents of an XML element are empty. If the contents are empty, the `children` property does not contain any element nodes or text nodes.
10	`processing-instruction-target`	The application that is target of the XML processing instructions
11	`content`	Holds the contents of an XML comment, XML element, or XML processing instruction

The property types that a node has depends on what kind of node it is. The table below lists the properties that each node has. For details about the properties of each node, see *1.15.1(3) Node details*.

Table 1-45: Properties of each node

No.	Property name	Document node	Element node	Attribute node	Processing instruction node	Comment node	Text node
1	`node-name`	N	Y	Y	N	N	N
2	`namespace`	N	Y	N	N	N	N
3	`parent`	N	Y	Y	Y	Y	Y
4	`children`	Y	Y	N	N	N	N
5	`attribute`	N	Y	N	N	N	N
6	`type-name`	N	Y	Y	N	N	N
7	`string-value`	Y	Y	Y	N	N	N

No.	Property name	Document node	Element node	Attribute node	Processing instruction node	Comment node	Text node
8	`typed-value`	Y	Y	Y	N	N	N
9	`nilled`	N	Y	N	N	N	N
10	`processing-instruction-target`	N	N	N	Y	N	N
11	`content`	N	N	N	Y	Y	Y

Legend:

Y: Has the property.

N: Does not have the property.

(3) Node details

This subsection describes the six kinds of nodes that are in the XQuery data model.

(a) Document node

A document node holds the information for an XML document.

- Properties

  A document node has the following properties:

Property name	Explanation
`children`	XQuery sequence of child nodes of the document node
`string-value`	Value of the result of concatenating, in document order, the values of the `content` properties of the descendant text nodes, or a character string of length zero if there are no text nodes descendants
`typed-value`	Value of the `string-value` property as an `xs:untypedAtomic` type

- Constraints

  Document nodes must obey the following constraints:

  1. The only possible child nodes are element nodes, processing instruction nodes, comment nodes, and text nodes. The set of child nodes may be empty. Attribute nodes and document nodes cannot be child nodes.
  2. If node N is included in the `children` property of document node D, the `parent` property of node N is node D.

3. If node N has document node D as its `parent` property, node N is set to the `children` property of node D.

(b) Element node

An element node holds the information for an XML element.

- Properties

  An element node has the following properties:

Property name	Explanation
`node-name`	Indicates the qualified name of the XML element.
`parent`	Indicates the parent node of the element node. The parent node can be either a document node or an element node.
`type-name`	Indicates the XQuery data type of the element node. If XML schema validation is not performed, or if XML schema validation results in an element type name that does not correspond to an XQuery data type provided by HiRDB, the value of the `type-name` property is undefined.
`children`	Indicates the XQuery sequence of child nodes of the element node.
`attribute`	Indicates the XQuery sequence of attribute nodes for the element node.
`namespace`	Indicates the XML namespace to which the element node belongs.
`nilled`	Indicates whether the contents of the element node are empty. If the contents are empty, the `children` property does not contain any element nodes or text nodes.
`string-value`	Indicates the result of concatenating, in document order, the values of the `content` properties of the descendant text nodes. If the contents of the element are empty, the value of this property is a character string of length zero.
`typed-value`	If the value of the `type-name` property is undefined, this is the actual value of the `string-value` property as an `xs:untypedAtomic` type. If the XML element is empty, the value of this property is an empty XQuery sequence. Otherwise, it is an XQuery sequence of atomic values obtained from the value of the `string-value` property based on the XQuery data type of the `type-name` property.

- Constraints

  Element nodes must obey the following constraints:

  1. The only possible child nodes are element nodes, processing instruction nodes, comment nodes, and text nodes. The set of child nodes may be empty. Attribute nodes and document nodes cannot be child nodes.
  2. The qualified name of the XML attribute for each attribute node whose `parent` property contains the same element node must be distinct.
  3. If node N is included in the `children` property of element node E, the

parent property of node N is set to node E.

4. If a node N that is not an attribute node has an element node E as its parent property, node N is included in the children property of node E.
5. If attribute node A is included in the attribute property of element node E, the parent property of node A is node E.
6. If attribute node A has element node E as its parent property, node A is included in the attribute property of node E.
7. If the type-name property of an element node is undefined, the type-name property of all its descendant element nodes is also undefined, and the type-name property of all of its attribute nodes is xs:untypedAtomic.
8. If the type-name property of an element node is undefined, the nilled property is FALSE.
9. If the nilled property is TRUE, the children property does not contain element nodes or text nodes.
10. For any qualified name in the node-name property of element node E or in the node-name property of any attribute node A of node E, there must be an XML namespace prefix associated with the XML namespace URI for node E.

(c) Attribute node

An attribute node holds the information for an XML attribute.

- Properties

  An attribute node has the following properties:

Property name	Explanation
node-name	Indicates the qualified name of the XML attribute.
parent	Indicates the parent node of the attribute node. The parent node is the node of the elements to which the XML attributes belong.
type-name	Indicates the name of the XQuery data type of the attribute node.
string-value	Indicates the values held by the attribute node. This property cannot be empty.
typed-value	If type-name is xs:untypedAtomic, this is the actual value of the string-value property as an xs:untypedAtomic type. Otherwise, it is an XQuery sequence of atomic values obtained from the string-value property based on the XQuery data type resulting from validation against the XML schema.

- Constraints

  Attribute nodes must obey the following constraints:

1. If attribute node A is included in the `attribute` property of element node E, the `parent` property of node A is node E.
2. If attribute node A has element node E as its `parent` property, node A is included in the `attribute` property of node E.

(d) Processing instruction node

A processing instruction node holds information for XML processing instructions.

- Properties

  A processing instruction node has the following properties:

Property name	Explanation
`processing-instruction -target`	Indicates the target to which the processing instructions are applied.
`content`	Indicates the contents of the processing instructions.
`parent`	Indicates the parent node of the processing instruction node.

- Constraints

  A processing instruction node must obey the following constraint:

  1. The processing instruction target is a local name.

(e) Comment node

A comment node holds information for an XML comment.

- Properties

  A comment node has the following properties:

Property name	Explanation
`content`	Indicates the contents of the XML comment.
`parent`	Indicates the parent node of the comment node.

(f) Text node

A text node holds the information for the contents of an XML element.

- Properties

  A text node has the following properties:

Property name	Explanation
`content`	Indicates the character string contents of the XML element.

Property name	Explanation
`parent`	Indicates the parent node of the text node.

■ Constraints

Text nodes must obey the following constraints:

1. If a text node has a parent node, the value of the `content` property cannot be a character string of length zero. If the text node does not have a parent node, the value will be a character string of length zero.

(4) Parent-child relationships between nodes

The following tables indicate which kinds of nodes can act as the parent node or child nodes of which other kinds of nodes.

Table 1-46: Possible parent nodes for each kind of node

Node kind	Parent node kind					
	Document node	Element node	Attribute node	Processing instruction node	Comment node	Text node
Document node	N	N	N	N	N	N
Element node	Y	Y	N	N	N	N
Attribute node	N	Y	N	N	N	N
Processing instruction node	Y	Y	N	N	N	N
Comment node	Y	Y	N	N	N	N
Text node	Y	Y	N	N	N	N

Legend:

Y: Can be parent node.

N: Cannot be parent node.

Table 1-47: Possible child nodes for each node kind

Node kind	Child node kind					
	Document node	Element node	Attribute node	Processing instruction node	Comment node	Text node
Document node	N	Y	N	Y	Y	Y
Element node	N	Y	N	Y	Y	Y
Attribute node	N	N	N	N	N	N
Processing instruction node	N	N	N	N	N	N
Comment node	N	N	N	N	N	N
Text node	N	N	N	N	N	N

Legend:

Y: Can be a child node.

N: Cannot be a child node.

1.15.2 Basic items

This section describes the XQuery basic items listed in the following table.

Table 1-48: XQuery basic items

No.	Basic item	Summary
1	Focus	The XQuery item information that is to be evaluated when the XQuery expression is evaluated.
2	Qualified name	Names used in XQuery in order to identify XML elements, XML attributes, XQuery functions, and XQuery variables.
3	XQuery data type	Types in which nodes and atomic values are entered in the XQuery data model.
4	Atomization	The process of converting an XQuery sequence into an XQuery sequence consisting of only atomic values.
5	Implicit type conversion to Boolean values	The process of automatically converting to Boolean values any values other than Boolean values that are the target of evaluation in the XQuery expression.

(1) Focus

The focus is the XQuery item information that is the target of evaluation at the time the XQuery expression is evaluated.

The following table describes the information item making up the focus.

Table 1-49: Focus

No.	Information item	Summary
1	Context item	XQuery item that is the target of evaluation
2	Context position	Position of the context item in the XQuery sequence composed of context items
3	Context size	Number of context items

The following sections describe each information item in detail.

(a) Context item

Holds the XQuery item that is the target of evaluation when the XQuery expression is evaluated.

(b) Context position

Holds the positions of the context item, in a XQuery sequence composed of context items, at the time the XQuery expression is evaluated. Context position is an integer value starting at 1.

(c) Context size

Holds the number of context items at the time the XQuery expression is evaluated.

The focus shifts as the XQuery expression is sequentially evaluated. This is illustrated in the following example.

Example:

XML document to be evaluated:

```
<bookinfo book_id="452469630">
  <title>Relational Databases Explained</title>
  <author>Jeff Jones</author>
  <author>Bob Adams</author>
</bookinfo>
```

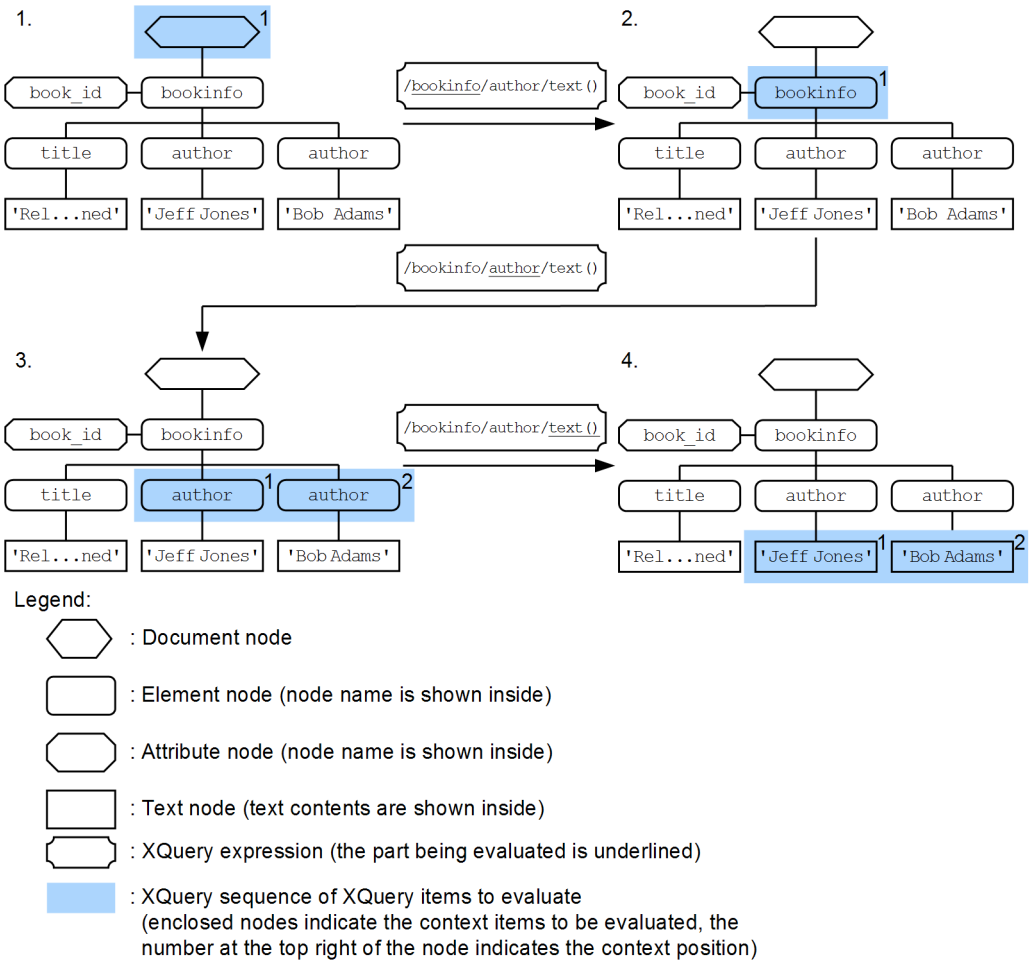
XQuery expression:

```
/bookinfo/author/text()
```

In this example XQuery expression, `bookinfo`, `author`, and `text()` will be

evaluated in the given order. The shifting of the focus as this happens is shown in the following figure.

Figure 1-16: Flow of the shifting of the focus



Explanation

1. Before the evaluation of the XQuery expression begins, the context items are initialized in the document node. The context position in the document node is set to `1`, and the context size is set to `1`.
2. When the `bookinfo` portion of the XQuery expression is evaluated, the context item moves from the document node down to the `bookinfo` element node. Because there is one context item, the context position in the `bookinfo` element node is set to `1`, and the context size is set to `1`.

3. When the `author` portion of the XQuery expression is evaluated, the context item moves from the bookinfo element node down to the `author` element node. The children of the `bookinfo` element node include two `author` element nodes. So, according to the document order, the context position of the first `author` element node is set to `1`, the context position of the second `author` element node is set to `2`, and the context size is set to `2`.
4. When the `text()` portion of the XQuery expression is evaluated, the context item moves from the `author` element node down to the text node. Because the children of the `author` element node include two text nodes, the context position of the text node of the child of the first `author` element node is set to `1`, the context position of the text node of the child of the second `author` element node is set to `2`, and the context size is set to `2`.

(2) Qualified names

The names used in XQuery to identify XML elements, XML attributes, XQuery functions, and XQuery variables are called qualified names. A qualified name is composed of a prefix, the colon character, and a local name. Using an XML namespace declaration (for details about XML namespace declarations, see *1.15.5 XQuery declaration*), the user can associate an XML namespace URI with a prefix. An XML namespace URI is a URI that identifies an XML namespace. A local name is a name used within an individual XML namespace.

The following table lists the prefixes that are already defined in HiRDB.

Table 1-50: Prefixes defined in HiRDB

No.	Prefix	XML namespace URI
1	xml	http://www.w3.org/XML/1998/namespace
2	xs	http://www.w3.org/2001/XMLSchema
3	xsi	http://www.w3.org/2001/XMLSchema-instance
4	fn	http://www.w3.org/2006/xpath-functions
5	err	http://www.w3.org/2005/xqt-errors
6	hi-fn	http://www.hitachi.co.jp/Prod/comp/soft1/hirdb/xquery-functions

You may specify a qualified name by omitting the prefix and the colon character and specifying only a local name. A qualified name where the prefix and colon are omitted will be searched for in the default XML namespace. For details about the default XML namespace, see *1.15.5 XQuery declaration*. The following table provides examples of qualified names.

Table 1-51: Examples of qualified names

No.	Qualified name	Prefix	Local name
1	xs:string	xs	string
2	fn:data	fn	data
3	element_name	(omitted)	element_name

A name consisting of the XML namespace URI corresponding to a prefix and a local name is called an expand qualified name. Two qualified names are equivalent if their local names and the XML namespace URLs of their expand qualified names are both the same. In this case, even if the prefixes of the two qualified names are different, the two qualified names are considered to be equivalent. For example, the two qualified name listed in the table below have different prefixes. Nevertheless, the two qualified names are considered to be equivalent because their local names and the expand qualified names of their XML namespace URIs are both the same.

Table 1-52: Even with different prefixes, the two qualified names are considered equivalent

No.	Qualified name	Prefix	Local name	Associated XML namespace URI
1	xxx:element_name	xxx	element_name	http://www.hitachi.com
2	yyy:element_name	yyy	element_name	http://www.hitachi.com

The following table lists the characters that can be used to specify the prefix and local name.

Table 1-53: Characters that can be used in the prefix and local name

No.	Classification	Characters that can be used in the prefix and local name	Beginning character?
1	One-byte character code characters	Lower-case letters (A to Z)	Y
2		Lower-case letters (a to z)	Y
3		Numeric characters (0 to 9)	N
4		Katakana characters	Y
5	Two-byte character code	All two-byte character code characters	Y

No.	Classification	Characters that can be used in the prefix and local name	Beginning character?
6	Special characters (one-byte character code)	Period (.)	N
7		Hyphen or minus sign (-)	N
8		Underscore (_)	Y

Legend:

Y: Can be specified at the beginning of a prefix or local name.

N: Cannot be specified at the beginning of a prefix or local name.

The characters that can be used in XQuery prefixes and local names depend on the character code classification specified in the `pdsetup` command. For details about the `pdsetup` command, see the manual *HiRDB Version 9 Command Reference*. For details about the relationships between specific character code classifications and the available characters, see *1.1.5 SQL character set*.

(3) XQuery data types

(a) XQuery data types defined in HiRDB

XQuery items must be typed with an XQuery data type in order to be used in XQuery. The table below lists the XQuery data types provided in HiRDB. XQuery data types with a single value are called atomic types. These include the character string data type, numeric data types, time data types, certain other types, and the `xs:untypedAtomic` type. The value of an atomic type is called an atomic value.

Table 1-54: XQuery data types defined in HiRDB

No.	Classification	XQuery data type	Explanation
1	Character string data type	xs:string	Type representing character strings.
2	Numeric data type	xs:decimal	Type representing fixed-point numbers of up to 38 digits.
3		xs:int	Type representing integer values in the range `-2147483648` to `2147483647`.
4		xs:double	Type representing double-precision floating-point numbers with a value of approximately $\pm$ 4.9 x 10^{-324} to $\pm$ 1.7 x 10^{308} #.

No.	Classification	XQuery data type	Explanation
5	Time data type	xs:dateTime	Type representing a particular time on a particular date as a combined date and time value. Expressed in the format *YYYY*-*MM*-*DD*T*hh*:*mm*:*ss*[.*nn...n*]. *YYYY* represents the year, *MM* the month, *DD* the day, *hh* the hour, *mm* the minute, *ss* the seconds, and *nn...n* the fractional seconds (one to six digits). T is the notation used to separate the date and time. The range of possible values for each element in the format is *YYYY*: 0001 to 9999 (year), *MM*: 01 to 12 (month), *DD*: 01 to the last day of that month (day), *hh*: 00 to 23 (hour), *mm*: 00 to 59 (minute), *ss*: 00 to 59 (second), *nnnnnn*: 000000 to 999999 (fractional seconds).
6		xs:date	Type representing a date as the value. Expressed in the format *YYYY*-*MM*-*DD*, where *YYYY* represents the year, *MM* the month, and *DD* the day. The range of possible values for each element in the format is *YYYY*: 0001 to 9999 (year), *MM*: 01 to 12 (month), *DD*: 01 to the last day of that month (day).
7		xs:time	Type representing a time as the value. Expressed in the format *hh*:*mm*:*ss*, where *hh* represents the hour, *mm* the minute, and *ss* the seconds. The range of possible values for each element in the format is *hh*: 00 to 23 (hour), *mm*: 00 to 59 (minute), *ss*: 00 to 59 (seconds).
8	Other data types	xs:hexBinary	Type representing binary data as a sequence of hexadecimal characters (sequence of characters 0 to 9, a to f, and A to F).
9		xs:boolean	Type representing a logical value of TRUE or FALSE.
10	Undefined data type	xs:untypedAtomic	Type representing an atomic value whose type is undefined.

#

The range of values of floating point numbers depends on the hardware representation.

(b) Constructor function

For each atomic type shown in *(a)*, there is an implicitly defined constructor function that generates that type of atomic value from another atomic value. The constructor function takes the following format:

- Format

XQuery-data-type-name (*argument*)

- Rules

  1. Specify the qualified name of the XQuery data type for the atomic value to be generate in *XQuery-data-type-name*.
  2. Specify an atomic value or an empty XQuery sequence as the argument. If an empty XQuery sequence is specified, an empty XQuery sequence is returned.
  3. An error results if the specified atomic value cannot be converted to an atomic value of the requested XQuery data type. The combinations of allowable XQuery data type conversions are listed below in *(c) Convertible XQuery data types*.

(c) Convertible XQuery data types

The following table lists all possible XQuery data type conversions.

Table 1-55: Convertible XQuery data types

XQuery data type before conversion	XQuery data type after conversion									
	Numeric data types			Character string data type	Time data types			Other data types		
	xs: double	xs: decimal	xs: int	xs: string	xs: dateTime	xs: date	xs: time	xs: hexBinary	xs: boolean	xs: untyped Atomic
xs:double	Y	C	C	Y	N	N	N	N	Y	Y
xs:decimal	Y	Y	C	Y	N	N	N	N	Y	Y
xs:int	Y	Y	Y	Y	N	N	N	N	Y	Y
xs:string	C	C	C	Y	C	C	C	C	C	Y
xs:dateTime	N	N	N	Y	Y	Y	Y	N	N	Y
xs:date	N	N	N	Y	Y	Y	N	N	N	Y
xs:time	N	N	N	Y	N	N	Y	N	N	Y
xs:hexBinary	N	N	N	Y	N	N	N	Y	N	Y

XQuery data type before conversion	XQuery data type after conversion									
	Numeric data types			Character string data type	Time data types			Other data types		
	xs: double	xs: decimal	xs: int	xs: string	xs: dateTime	xs: date	xs: time	xs: hexBinary	xs: boolean	xs: untyped Atomic
xs:boolean	Y	Y	Y	Y	N	N	N	N	Y	Y
xs:untyped Atomic	C	C	C	Y	C	C	C	C	C	Y

Legend:

Y: Can be converted.

C: Can be converted depending on the value.

N: Cannot be converted.

- Conversion to `xs:string` type and `xs:untypedAtomic` type

  Conversions obey the rules described in the following table.

  Table 1-56: Rules for conversion to xs:string type and xs:untypedAtomic type

No.	Before conversion		Conversion result
	XQuery data type	Conditions on value	
1	xs:string xs:untypedAtomic	All	Not converted.
2	xs:int	All	Converted to XQuery integer literal format. Leading zeros are removed, and a minus sign (-) is added for values less than zero.
3	xs:decimal	Fractional part is 0	
4		Fractional part is not 0	Converted to XQuery decimal literal format. Leading zeros (except in the ones column) and trailing zeros are removed, and a minus sign (-) is added for values less than zero.

No.	Before conversion		Conversion result
	XQuery data type	Conditions on value	
5	xs:double	Absolute value is at least 0.000001 but less than 1000000	Follows `xs:decimal` type conversion rules (No. 3 and 4) above.
6		Positive 0	Converted to `0`
7		Negative 0	Converted to `-0`.
8		Positive maximum value	Converted to `INF`.
9		Negative maximum value	Converted to `-INF`.
10		`NaN` (not a number)	Converted to `NaN`.
11		Other than the above	Converted to a string in XQuery decimal literal format into which the mantissa is converted, followed by `E`, followed by a string in XQuery integer literal format into which the exponent part is converted. Leading zeros are removed. The integer part of the mantissa is one non-zero digit. Trailing zeros are removed from the fractional mantissa (except from the tenths column). A minus sign (`-`) is added for values less than zero.
12	xs:boolean	`TRUE`	Converted to `true`.
13		`FALSE`	Converted to `false`.
14	xs:dateTime	All	Converted to the format *YYYY*`-`*MM*`-`*DD*`T`*hh*`:`*mm*`:`*ss*`.`*s...s*, where *YYYY* is the year, *MM* the month, *DD* the day, *hh* the hour, *mm* the minute, *ss* the seconds, and *s...s* the fractional seconds. If the fractional seconds part is not `0`, any trailing zeros are removed from the fractional seconds. If the fractional seconds part is `0`, it is not displayed.
15	xs:date	All	Converted to the format *YYYY*`-`*MM*`-`*DD*, where *YYYY* is the year, *MM* the month, and *DD* the day.
16	xs:time	All	Converted to the format *hh*`:`*mm*`:`*ss*, where *hh* is the hour, *mm* the minutes, and *ss* the seconds.
17	xs:hexBinary	All	A hexadecimal character sequence is returned unchanged. However, `A` to `F` is converted to upper case.

- Conversion to `xs:double` type

  Conversions obey the rules described in the following table.

  Table 1-57: Rules for conversion to xs:double type

No.	Before conversion		Conversion result
	XQuery data type	Conditions on value	
1	xs:double	All	Not converted.
2	xs:int xs:decimal	All	Converted to the value of the corresponding `xs:double` type.
3	xs:boolean	`TRUE`	Converted to `1.0E0`.
4		`FALSE`	Converted to `0.0E0`.
5	xs:string	`INF`	Converted to positive infinity.
6		`-INF`	Converted to negative infinity.
7		`NaN`	Converted to `NaN` (not a number).
8		XQuery numeric literal format character string#	First converted to the corresponding numeric data type, then converted to `xs:double` type.
9		Other than the above	Cannot be converted.
10	xs:untypedAtomic	All	First converted to the `xs:string` type, then converted to `xs:double` type.

#

Applies if removing leading and trailing whitespace (one-byte space (`X'20'`), tab (`X'09'`), NL (`X'0A'`), and CR (`X'0D'`)) results in the XQuery numeric literal format.

- Conversion to `xs:decimal` type

  Conversions obey the rules described in the following table.

  Table 1-58: Rules for conversion to xs:decimal type

No.	Before conversion		Conversion result
	XQuery data type	Conditions on value	
1	xs:decimal	All	Not converted.

No.	Before conversion		Conversion result
	XQuery data type	Conditions on value	
2	xs:double	Positive 0	Converted to `0`.
3		Negative 0	Converted to `0`.
4		Positive infinity	Cannot be converted.
		Negative infinity	
		`NaN` (not a number)	
5		Values where the integer part is not more than 38 digits long.	Converted to the `xs:decimal` type value, not exceeding 38 digits in length, that is numerically closest to the pre-conversion value. If there are two possible values, it is converted to the value closer to zero.
6		Other than the above	Cannot be converted.
7	xs:int	All	Converted to the corresponding `xs: decimal` type value.
8	xs:boolean	`TRUE`	Converted to `1`.
9		`FALSE`	Converted to `0`.
10	xs:string	XQuery numeric literal format character string#	First converted to the corresponding numeric data type, then converted to `xs:decimal` type.
11		Other than the above	Cannot be converted.
12	xs:untypedAtomic	All	First converted to `xs:string` type, then converted to `xs:decimal` type.

#

Applies if removing leading and trailing whitespace (one-byte space (`X'20'`), tab (`X'09'`), NL (`X'0A'`), and CR (`X'0D'`)) results in the XQuery numeric literal format.

- Conversion to `xs:int` type

  Conversions obey the rules described in the following table.

Table 1-59: Rules for conversion to xs:int type

No.	Before conversion		Conversion result
	XQuery data type	Conditions on value	
1	xs:int	All	Not converted.
2	xs:decimal	Values that fall between the minimum and maximum values of the `xs:int` type once the digits after the decimal point are truncated.	Converted to a value in which the digits after the decimal point are truncated.
3		Other than the above	Cannot be converted.
4	xs:double	Positive 0	Converted to `0`.
5		Negative 0	Converted to `0`.
6		positive infinity	Cannot be converted.
		negative infinity	
		`NaN` (not a number)	
7		Values that fall between the minimum and maximum values of the `xs:int` type once the digits after the decimal point are truncated.	Converted to a value in which the digits after the decimal point are truncated.
8		Other than the above	Cannot be converted.
9	xs:boolean	`TRUE`	Converted to `1`.
10		`FALSE`	Converted to `0`.
11	xs:string	XQuery numeric literal format character string#	First converted to the corresponding numeric data type, then converted to the `xs:int` type.
12		Other than the above	Cannot be converted.

No.	Before conversion		Conversion result
	XQuery data type	Conditions on value	
13	xs:untypedAtomic	All	First converted to the `xs:string` type, then converted to `xs:int` type.

#

Applies if removing leading and trailing whitespace (one-byte space (`X'20'`), tab (`X'09'`), NL (`X'0A'`), and CR (`X'0D'`)) results in the XQuery numeric literal format.

- Conversion to `xs:dateTime` type

  Conversions obey the rules described in the following table.

  Table 1-60: Rules for conversion to xs:dateTime type

No.	Before conversion		Conversion result
	XQuery data type	Conditions on value	
1	xs:dateTime	All	Not converted.
2	xs:date	All	The date part is equal to the value before conversion, and the time part is converted to the value `00:00:00`.
3	xs:string	`xs:dateTime` type character string representation format value	Converted to the value of the corresponding character string representation.
4		Other than the above	Cannot be converted.
5	xs:untypedAtomic	All	First converted to the `xs:string` type, then converted to the `xs:dateTime` type.

■ Conversion to `xs:date` type

Conversions obey the rules described in the following table.

Table 1-61: Rules for conversion to xs:date type

No.	Before conversion		Conversion result
	XQuery data type	Conditions on value	
1	xs:date	All	Not converted.
2	xs:dateTime	All	Converted to a value that is equal to the value of the date part before conversion.
3	xs:string	`xs:date` type character string representation format value	Converted to the value of the corresponding character string representation.
4		Other than the above	Cannot be converted.
5	xs:untypedAtomic	All	First converted to the `xs:string` type, then converted to the `xs:date` type.

■ Conversion to `xs:time` type

Conversions obey the rules described in the following table.

Table 1-62: Rules for conversion to xs:time type

No.	Before conversion		Conversion result
	XQuery data type	Conditions on value	
1	xs:time	All	Not converted.
2	xs:dateTime	All	Converted to a value that is equal to the value of the time part before conversion (excluding any fractional seconds).
3	xs:string	`xs:time` type character string representation format value	Converted to the value of the corresponding character string representation.
4		Other than the above	Cannot be converted.
5	xs:untypedAtomic	All	First converted to the `xs:string` type, then converted to the `xs:time` type.

■ Conversion to `xs:boolean` type

Conversions obey the rules described in the following table.

Table 1-63: Rules for conversion to xs:boolean type

No.	Before conversion		Conversion result
	XQuery data type	Conditions on value	
1	xs:boolean	All	Not converted.
2	Numeric data type	0	Converted to `FALSE`.
		Positive 0	
		Negative 0	
		0.0	
		0.0E0	
		`NaN` (not a number)	
3		Other than the above	Converted to `TRUE`.
4	xs:string	`true`#	Converted to `TRUE`.
5		`false`#	Converted to `FALSE`.
6		Other than the above	Cannot be converted.
7	xs:untypedAtomic	All	First converted to `xs:string` type, then converted to `xs:boolean` type.

#

Applies if removing leading and trailing whitespace (one-byte space (`X'20'`), tab (`X'09'`), NL (`X'0A'`), and CR (`X'0D'`)) results in the XQuery numeric literal format. Not case-sensitive.

- Conversion to `xs:hexBinary` type

  Conversions obey the rules described in the following table.

  Table 1-64: Rules for conversion to xs:hexBinary type

No.	Before conversion		Conversion result
	XQuery data type	Conditions on value	
1	xs:hexBinary	All	Not converted.
2	xs:string	Hexadecimal character sequence	Converted to the value of the corresponding `xs:hexBinary` type.
3		Other than the above	Cannot be converted.
4	xs:untypedAtomic	All	First being converted to `xs:string` type, then converted to `xs:hexBinary` type.

(4) Atomization

Atomization is the process of converting an XQuery sequence to an XQuery sequence consisting of only atomic values. In XQuery, atomization is performed when evaluating the following expressions, for which an XQuery sequence that consists of only atomic values is required:

- XQuery arithmetic expression
- XQuery comparison expression
- Arguments and return values for XQuery function invocation

The atomization result is the result of calling the XQuery `fn:data` function on the XQuery sequence. For details about the `fn:data` function, see *1.15.8 XQuery functions*.

(5) Implicit type conversion to Boolean values

In XQuery, when a value other than a Boolean value is evaluated in a place where a Boolean value (`xs:boolean` type value) is required, it is automatically converted to a Boolean value. The result of converting to a Boolean value will be the result of calling the XQuery `fn:boolean` function on the XQuery sequence. For details about the `fn:boolean` function, see *1.15.8 XQuery functions*.

1.15.3 Specifying an XQuery

(1) Function

Specifies a query for an `XML` type value.

An XQuery query can be specified as a character string literal in the following places:

- `XMLQUERY` function
- `XMLEXISTS` predicate

(2) Format

```
XQuery ::= XQuery-declaration  XQuery-query-body
```

(3) Example

Declare the default XML namespace in *XQuery-declaration*. Define the result of the XQuery query inside *XQuery-query-body*.

```
declare default element namespace 'http://www.aaa.com' ;  ... XQuery declaration
/bookinfo/book/price/child::text()                  ... XQuery query body
```

1.15.4 XQuery description format

(1) XQuery keywords

Words that are specified for use in XQuery functions (`for`, `if`, etc.) are called keywords. Keywords in XQuery are specified in all lower case. XQuery distinguishes between upper-case and lower-case letters.

(2) XQuery separators

The following characters or elements can serve as separators in XQuery:

- Space (`X'20'`)
- TAB (`X'09'`)
- NL (`X'0a'`)
- CR (`X'0d'`)
- XQuery comment

(a) Where separators must be inserted

Separators must be inserted in the following places:

- Between two keywords
- Between a keyword and a qualified name
- Between a keyword and a numeric literal

- After a qualified name and before the subtraction operator (-)

(b) Where separators cannot be inserted

Separators cannot be inserted in the following places:

- Inside a keyword
- Inside a qualified name
- Inside a numeric literal
- Inside a character string literal
- Inside an operator
- Inside an XQuery path expression specified within //
- Inside an XQuery path expression specified within ..
- Inside an XQuery path expression specified within ::
- In an XQuery variable reference between the $ and the XQuery variable name
- In an XQuery comment between (: and :
- In an XQuery comment between : and)
- Inside a FLWOR expression specified within :=

(c) Where separators may be inserted

Separators may be inserted in the following places:

- In places not prohibited under *Where separators cannot be inserted* above, as well as before and after the following special characters:

  , . - + * ' " ( ) < > = ! / : ; | [ ] space (X'20'), TAB (X'09'), NL (X'0a'), and CR (X'0d')

1.15.5 XQuery declaration

(1) Function

Declares the XML namespaces for the qualified names used in an XQuery query.

(2) Format

XQuery-declaration ::= [{*XML-namespace-declaration*|*default-XML-namespace-declaration*};]...

XML-namespace-declaration ::= declare namespace *prefix* = *XML-namespace-URI*
default-XML-namespace-declaration ::= declare default {element|function}
namespace *XML-namespace-URI*

(3) Operands

- *XML-namespace-declaration* ::= `declare namespace` *prefix* = *XML-namespace-URI*

If the body of an XQuery query contains prefixed qualified names, declares the XML namespace corresponding to the prefix for those qualified names.

The same prefix cannot be declared for multiple XML namespaces.

prefix

Specifies the prefix used to qualify names in the body of an XQuery query.

XML-namespace-URI

Specifies the URI of the XML namespace corresponding to the prefix in the format of an XQuery character string literal.

A character string of length 0 can be specified in order to unmap the XML namespace from the XML namespace prefix defined in Table *1-50 Prefixes defined in HiRDB*.

- *default-XML-namespace-declaration* ::= `declare default {element|function} namespace` *XML-namespace-URI*

Declares the default XML namespace to be used in the body of the Xquery query. The XML namespace for any qualified name in the XQuery query that does not have a prefix will be searched for in the XML namespace declared here.

In XQuery, both a default XML namespace declaration for specifying `elements` and a default XML namespace declaration for specifying `functions` can be declared, but only one of each.

If no declaration is made, the XML namespace URI for the default XML namespace to be used for both XML element names and XQuery data type names is assumed to be `http://www.w3.org/XML/1998/namespaces`, and the XML namespace URI for the default XML namespace to be used for XQuery functions is assumed to be `http://www.w3.org/2006/xpath-functions`.

element

Specified in order to declare the default XML namespace for XML element names and XQuery data type names.

function

Specified in order to declare the default XML namespace for XQuery function names.

XML-namespace-URI

Specifies the XML namespace URI for unprefixed qualified names in the format

of an XQuery character string literal.

If a character string of length 0 is specified, any unprefixed qualified name will not belong to any XML namespace.

1.15.6 XQuery query body

(1) Function

The body of an XQuery query specifies an XQuery expression that determines the results of the XQuery query.

(2) Format

XQuery-query-body `::=` *XQuery-expression*

XQuery-expression `::=` *XQuery-sequence-concatenation-expression*

(a) Expressions that can be included in an XQuery expression

XQuery expressions are constructed as combinations of the expressions listed in the following table.

Table 1-65: Expressions that can be included in an XQuery expression

No.	Expression	Explanation
1	XQuery sequence concatenation expression	Expression that concatenates XQuery sequences.
2	FLWOR expression	• Evaluates an XQuery expression by examining each separate XQuery item in the specified XQuery sequence, and returns the resulting XQuery sequence. • Binds an XQuery variable to the specified XQuery sequence, evaluates an XQuery expression in which that XQuery variable is specified, and returns the resulting XQuery sequence.
3	XQuery quantified expression	Expression that determines whether any (`some`) or all (`every`) of the XQuery items making up the specified XQuery sequence satisfy the specified criteria.
4	XQuery conditional expression	Expression that selects an XQuery expression depending on the result of the specified criteria and then evaluates the selected XQuery expression.
5	XQuery logical expression	Expression that performs logical operations (OR, AND).
6	XQuery comparison expression	Expression that performs a comparison. Comparisons include value comparisons, general comparisons, and node comparisons.
7	XQuery range expression	Expression that generates an XQuery sequence of consecutive integer (`xs:int` type) atomic values.

No.	Expression	Explanation
8	XQuery arithmetic expression	Expression that performs addition, subtraction, multiplication, division, or modulo calculation.
9	XQuery sequence operation expression	Expression for obtaining the union, intersection or difference for an XQuery sequence consisting of nodes.
10	XQuery unary expression	Expression that performs a unary operation.
11	XQuery value expression	Expression that specifies a value.
12	XQuery path expression	Expression that traverses an XQuery data model tree and selects specific nodes.
13	XQuery primary expression	The basic components, listed below, that constitute an XQuery path expression: • XQuery literal • XQuery variable reference • Parenthesized XQuery expression • Context item expression • XQuery function invocation

(3) XQuery sequence concatenation expressions

(a) Function

Concatenates XQuery sequences.

When concatenating multiple XQuery sequences, the XQuery sequences to be concatenated are evaluated from left to right and each XQuery item in the XQuery sequences is added to the end of the resulting XQuery sequence. The resulting concatenated XQuery sequence includes all the XQuery items in the XQuery sequences being concatenated. The number of XQuery items contained in the resulting concatenated XQuery sequence will be equal to the number of XQuery items in the XQuery sequence being concatenated.

(b) Format

XQuery-sequence-concatenation-expression ::= *XQuery-single-expression* [, *XQuery-single-expression*]...

XQuery-single-expression ::= {*FLWOR-expression* | *XQuery-quantified-expression* | *XQuery-conditional-expression* | *XQuery-logical-expression*}

(c) Example

The following is an example of an XQuery sequence concatenation expression.

No.	Example	Result	Explanation
1	1, 2, 3, 4, 5	(1, 2, 3, 4, 5)	Returns an XQuery sequence that is the concatenation of five XQuery sequences consisting of single atomic values of type `xs:int`, resulting from the evaluation of the XQuery integer literals from `1` to `5`.
2	(1, 2, 3), (), (4, 5)	(1, 2, 3, 4, 5)	Returns an XQuery sequence that is the concatenation of the XQuery sequence consisting of the atomic values `1`, `2`, and `3` of type `xs:int`, an empty XQuery sequence, and an XQuery sequence consisting of the atomic values `4` and `5` of type `xs:int`. The evaluation results are the same as for the XQuery expression `1, 2, 3, 4, 5`.

(4) FLWOR expressions

(a) Function

Binds an XQuery variable to an XQuery sequence, evaluates an XQuery expression containing that XQuery variable, and then returns the resulting XQuery sequence.

The methods for binding an XQuery variable and evaluating the XQuery expression are as follows.

- For each XQuery item in the specified XQuery sequence, bind an XQuery variable to a single XQuery sequence, and then evaluate the XQuery expression for each XQuery item (`FOR` clause).
- Bind an XQuery variable to the specified XQuery sequence and evaluate the XQuery expression (`LET` clause).

In the W3C standard, the FLWOR expressions include the `FOR` clause, `LET` clause, `WHERE` clause, `ORDER BY` clause, and `RETURN` clause, but HiRDB supports only the `FOR` clause, `LET` clause, and `RETURN` clause.

(b) Format

```
FLWOR-expression ::= {FOR-clause|LET-clause }[{FOR-clause|LET-clause }]...
        RETURN-clause
 FOR-clause ::= for $ XQuery-variable-name in XQuery-single-expression
        [, $ XQuery-variable-name in XQuery-single-expression]...
 LET-clause ::= let $ XQuery-variable-name : equals-sign XQuery-single-expression
        [, $ XQuery-variable-name : equals-sign XQuery-single-expression]...
 RETURN-clause ::= return XQuery-single-expression
```

(c) Operands

- {*FOR-clause*|*LET-clause* }[{*FOR-clause*|*LET-clause* }]...

Any combination of one or more FOR or LET clauses can be specified.

If multiple FOR and LET clauses are specified, each clause is treated as nested.

Example:

XQuery expression 1 and XQuery expression 2 below have the same contents:

- XQuery expression 1

```
for $XQuery-variable-name-1 in XQuery-single- expression-1
let $XQuery-variable-name-2 := XQuery-single- expression-2
for $XQuery-variable-name-3 in XQuery-single- expression-3
let $XQuery-variable-name-4 := XQuery-single- expression-4
return XQuery-single-expression-5
```

- XQuery expression 2

```
for $XQuery-variable-name-1 in XQuery-single- expression-1
return
  let $XQuery-variable-name-2 := XQuery-single- expression-2
  return
    for $XQuery-variable-name-3 in XQuery-single- expression-3
    return
      let $XQuery-variable-name-4 := XQuery-single- expression-4
        return XQuery-single-expression-5
```

- *FOR-clause* ::=for $ *XQuery-variable-name* in *XQuery-single-expression*

  [, $ *XQuery-variable-name* in *XQuery-single-expression*] . . .

The evaluation of the RETURN clause is repeated as described below:

1. If a single XQuery variable is used, the RETURN clause is executed repeatedly to evaluate each XQuery item that makes up the XQuery sequence that is the result of evaluating the XQuery single expression.
2. If multiple XQuery variables are used, the FOR clauses are treated as nested.

   Example:

   XQuery expression 1 and XQuery expression 2 below have the same contents:

   - XQuery expression 1

```
for $XQuery-variable-name-1 in XQuery-single- expression-1 ,
    $XQuery-variable-name-2 in XQuery-single- expression-2
return XQuery-single- expression-3
```

   - XQuery expression 2

```
for $XQuery-variable-name-1 in XQuery-single- expression-1
```

```
return for $XQuery-variable-name-2 in XQuery-single- expression-2
        return XQuery-single- expression-3
```

XQuery-variable-name

Specifies the name of the XQuery variable to be associated with an XQuery item in the XQuery sequence resulting from the evaluation of the XQuery single expression.

The XQuery variable specified here in the relevant FLWOR expression will be bound within any XQuery single expression specified as input to any XQuery variable name specified after this XQuery variable name, and within XQuery single expressions in any `FOR` or `LET` clauses that are specified after this `FOR` clause, and then within the XQuery single expression specified in the `RETURN` clause.

XQuery-single-expression

Specifies the XQuery single expression that will be used to return the XQuery sequence that becomes the input.

- *LET-clause* `::=` `let $` *XQuery-variable-name* `:` *equals-sign XQuery-single-expression*

  `[, $` *XQuery-variable-name* `:` *equals-sign XQuery-single-expression*`] . . .`

Associates the result of the evaluation of *XQuery-single-expression* with an XQuery variable and then evaluates the `RETURN` clause.

XQuery-variable-name

Specifies the name of the XQuery variable to be associated with the XQuery sequence resulting from the evaluation of the XQuery single expression.

The XQuery variable specified here in the relevant FLWOR expression will be bound within any XQuery single expression that is specified as input to any XQuery variable name specified after this XQuery variable name, and within the XQuery single expression in any `FOR` or `LET` clauses that are specified after this `LET` clause, and then within the XQuery single expression specified in the `RETURN` clause.

XQuery-single-expression

Specifies the XQuery single expression that will return the XQuery sequence to be the input.

- *RETURN-clause* `::=` `return` *XQuery-single-expression*

XQuery-single-expression

Specifies an XQuery single expression that represents the XQuery sequence to be

returned.

In a FOR clause, it is evaluated once for each iteration of XQuery items.

If a LET clause is specified without a FOR clause, it is evaluated only once.

The evaluation result of a FLWOR expression is the XQuery sequence that is the concatenation of the results of repeatedly evaluating the *XQuery-single-expression* specified here.

(d) Example

The following is an example of a FLWOR expression.

Element represented by the XQuery variable book:

```
<bookinfo book_id="452469630">
  <title>Relational Databases Explained</title>
  <category>database </category>
  <author>Jeff Jones</author>
  <author>Bob Adams</author>
 <price>30</price>
</bookinfo>
```

FLWOR expression examples and results:

No.	Example	Result	Explanation
1	for $a in $book/author return fn:string($a)	("Jeff Jones","Bob Adams")	Applies the fn:string function to every author element node that is a child of the bookinfo element node and returns the results expressed as character strings.
2	for $i in (10, 20), $j in (1,2) return ($i + $j)	(11, 12, 21, 22)	Returns the same result as for $I in (10, 20) return for $j in (1, 2) return ($i + $j)
3	let $a := $book/author return $a/fn:string()	("Jeff Jones","Bob Adams")	Applies the fn:string function to every author element node that is a child of the bookinfo element node and returns the results expressed as character strings.
4	for $i in (1,2) let $a := $book/author[$i] return fn:string($a)	("Jeff Jones","Bob Adams")	Applies the fn:string function to the first and second element nodes of the author element nodes that are children of the bookinfo element node and returns the results expressed as character strings.

No.	Example	Result	Explanation
5	let $a := $book/author for $i in (1,2) return fn:string($a[$i])	("Jeff Jones","Bob Adams")	Applies the fn:string function to the first and second element nodes of the author element nodes that are children of the bookinfo element node and returns the results expressed as character strings.
6	for $a in $book/author return fn:count($a)	(1,1)	Returns the XQuery sequence that is the concatenation of the results of applying the fn:count function to every author element node that is a child of the bookinfo element node.
7	let $a := $book/author return fn:count($a)	(2)	Returns the result of applying the fn:count function to the XQuery sequence consisting of the author element nodes that are the children of the bookinfo element node.

(5) XQuery quantified expressions

(a) Function

Determines whether any (some) or all (every) of the XQuery items making up the specified XQuery sequence satisfy the specified criteria, and returns the result.

(b) Format

XQuery-quantified-expression::= {some|every} $ *XQuery-variable-name* in *XQuery-single-expression*
[, $ variable name in *XQuery-single-expression*]... satisfies *XQuery-single-expression*

(c) Operands

- {some|every} $ *XQuery-variable-name* in *XQuery-single-expression*

  [, $ variable name in *XQuery-single-expression*]... satisfies *XQuery-single-expression*

some

If some is specified, and if the condition specified in the satisfies clause is TRUE for at least one of the XQuery items in the XQuery sequence generated by the in clause, the result will be TRUE. Otherwise, the result will be FALSE. If the XQuery sequence generated by the in clause is an empty XQuery sequence, the result of the XQuery quantified expression will be FALSE.

every

If `every` is specified, and if the condition specified in the `satisfies` clause is `TRUE` for all of the XQuery items in the XQuery sequence generated by the `in` clause, the result will be `TRUE`. Otherwise, the result will be `FALSE`. If the XQuery sequence generated by the `in` clause is an empty XQuery sequence, the result of the XQuery quantified expression will be `TRUE`.

XQuery-variable-name

Specifies the name of an XQuery variable to be associated with the XQuery item in the XQuery sequence resulting from the evaluation of *XQuery-single-expression*.

The XQuery variable specified here in the relevant XQuery quantified expression will be bound within any XQuery single expression specified as input to any XQuery variable name specified after this XQuery variable name, and within the XQuery single expression specified in the `satisfies` clause.

`in` *XQuery-single-expression*

Specifies the XQuery single expression specifying the XQuery sequence that will be used as the input.

`satisfies` *XQuery-single-expression*

Specifies the XQuery single expression specifying the condition to be evaluated for the XQuery variable. If multiple `in` clauses are specified, *XQuery-single-expression* is used to evaluate multiple XQuery variables.

(d) Example

The following is an example of an XQuery quantified expression.

Element represented by the XQuery variable `book`:

```
<bookinfo book_id="452469630">
  <title>Relational Databases Explained</title>
  <category>database </category>
  <author>Jeff Jones</author>
  <author>Bob Adams</author>
 <price>30</price>
</bookinfo>
```

XQuery quantified expression examples and results:

No.	Example	Result	Explanation
1	`some $text in $book /author/ text() satisfies ( $text eq "Bob Adams" )`	`TRUE`	`TRUE`, because a text node of an `author` element node which is a child of the `bookinfo` element node has contents equal to `Bob Adams`.

No.	Example	Result	Explanation
2	some $i in (1, 2, 3), $j in (4, 5, 6) satisfies $i + $j >= 6	TRUE	Result of this XQuery quantified expression is TRUE because some was specified and there are combinations of XQuery items (example: $i = 1, $j = 5) such that $i + $j >= 6.
3	every $i in (1, 2, 3), $j in (4, 5, 6) satisfies $i + $j >= 6	FALSE	Result of this XQuery quantified expression is FALSE because every was specified and there are combinations of XQuery items (example: $i = 1, $j = 4) that do not satisfy $i + $j >= 6.

(6) XQuery conditional expressions

(a) Function

Selects an XQuery expression depending on the result of the specified criteria and then returns the result of evaluating the selected XQuery expression.

(b) Format

XQuery-conditional-expression ::= if (*XQuery-expression*) then *XQuery-single-expression* else *XQuery-single-expression*

(c) Operands

- if (*XQuery-expression*)

XQuery-expression

Specifies an XQuery expression to evaluate as a condition. If the result of evaluating the XQuery expression specified here is TRUE, the XQuery single expression specified immediately after the then clause is evaluated. If it is FALSE, the XQuery single expression specified immediately after the else clause is evaluated. However, if the result of evaluating the XQuery expression specified here is not a single XQuery sequence containing an atomic value of the xs:boolean type, it is evaluated after converting it to a single XQuery sequence containing an atomic value of the xs:boolean type.

- then *XQuery-single-expression*

XQuery-single-expression

Specifies the XQuery single expression to evaluate if the condition specified in the if clause is TRUE.

- else *XQuery-single-expression*

XQuery-single-expression

Specifies the XQuery single expression to evaluate if the condition specified in the `if` clause is `FALSE`.

(d) Example

The following is an example of an XQuery conditional expression.

XML element represented by the XQuery variable `book1`:

```
<bookinfo book_id="452469630">
  <title>Relational Databases Explained</title>
  <category>database </category>
  <author>Jeff Jones</author>
  <author>Bob Adams</author>
  <price>30</price>
</bookinfo>
```

XML element represented by the XQuery variable `book2`:

```
<bookinfo book_id="452469631">
  <title> Introduction to XML</title>
  <author>Sue Smith</author>
  <price>25</price>
</bookinfo>
```

XQuery conditional expression examples and results:

No.	Example	Result	Explanation
1	`if ($book1/price > $book2/price)` then $book1 else $book2	The element node indicated in the XQuery variable `book1` specified in the `then` clause	Because the typed-value (`=30.00`) of the price element node that is the child of the element node indicated in `$book1` is greater than the typed-value (`=25.00`) of the price element node that is the child of the element node indicated in `$book2`, the element node indicated in `$book1` is returned.

(7) XQuery logical expressions

(a) Function

Returns the result of performing a logical operation (OR, AND) on XQuery comparison expressions.

Under certain conditions, the evaluation result from the XQuery comparison expression is returned unchanged.

(b) Format

XQuery-logical-expression ::= *XQuery-OR-expression*
XQuery-OR-expression ::= *XQuery-AND-expression* [or *XQuery-AND-expression*]...
XQuery-AND-expression ::= *XQuery-comparison-expression* [and *XQuery-comparison-expression*]...

(c) Operands

- *XQuery-OR-expression* ::= *XQuery-AND-expression* [or *XQuery-AND-expression*]...

If the or operator is specified, the OR logical operation is performed on the results of evaluating XQuery AND expressions.

If you specify the XQuery AND expression by itself, the evaluation result from that XQuery AND expression is returned unchanged.

When the OR logical operation is performed, if the result of evaluating each XQuery AND expression is not a single XQuery sequence with a Boolean value (atomic value of the xs:boolean type), the OR logical operation is performed after implicitly converting the expression to a single XQuery sequence with a Boolean value (atomic value of the xs:boolean type).

The following table indicates the results of the OR operation based on the Boolean values of the first operand and second operand.

Table 1-66: Result of the OR logical operation in an XQuery logical expression

No.	First operand	Second operand	
		TRUE	FALSE
1	TRUE	TRUE	TRUE
2	FALSE	TRUE	FALSE

- *XQuery-AND-expression* ::= *XQuery-comparison-expression* [and *XQuery-comparison-expression*]...

If the and operator is specified, the AND logical operation is performed on the results of evaluating XQuery comparison expressions.

If you specify the XQuery comparison expression by itself, the evaluation result from that XQuery comparison expression is returned unchanged.

When the AND logical operation is performed, if the result of evaluating each XQuery comparison expression is not a single XQuery sequence with a Boolean value (atomic value of the xs:boolean type), the AND logical operation is performed after implicitly converting the expression to a single XQuery sequence with a Boolean value

(atomic value of the `xs:boolean` type).

The following table indicates the results of the AND operation based on the Boolean values of the first operand and second operand.

Table 1-67: Result of the AND logical operation in an XQuery logical expression

No.	First operand	Second operand	
		TRUE	FALSE
1	TRUE	TRUE	FALSE
2	FALSE	FALSE	FALSE

(8) XQuery comparison expression

(a) Function

Returns the result of a comparison to an XQuery range expression.

Under certain conditions, the evaluation result from the XQuery range expression is returned unchanged.

(b) Format

XQuery-comparison-expression `::=` *XQuery-range-expression* `[{`*value-comparison*`|` *general-comparison*`|` *node-comparison*`}` *XQuery-range-expression*`]`

value-comparison `::= {eq| ne| lt| le| gt| ge}`
general-comparison `::= {=| !=| <| <=| >| >=| <>}`
node-comparison `::= {is| <<| >>}`

(c) Operands

- *XQuery-range-expression* `[{`*value-comparison*`|` *general-comparison*`|` *node-comparison*`}` *XQuery-range-expression*`]`

If you specify a value comparison, general comparison or node comparison, the results of evaluating the XQuery range expressions are returned.

If you specify the XQuery range expression by itself, the evaluation result from that XQuery range expression is returned unchanged.

- *value-comparison* `::= {eq| ne| lt| le| gt| ge}`

Compares single atomic values to each other.

The following table describes each of the value comparison functions.

Table 1-68: Value comparison methods

No.	Value comparison operator	Conditions which evaluate to TRUE
1	eq	The values of the first comparison term and the second comparison term are equal.
2	ne	The values of the first comparison term and the second comparison term are not equal.
3	lt	The value of the first comparison term is less than the value of the second comparison term.
4	le	The value of the first comparison term is less than or equal to the value of the second comparison term.
5	gt	The value of the first comparison term is greater than the value of the second comparison term.
6	ge	The value of the first comparison term is greater than or equal to the value of the second comparison term.

Comparisons are performed in the following order:

1. Each comparison term is atomized.
2. If the result of atomizing either comparison term is an empty XQuery sequence, the result of the comparison will be an empty XQuery sequence.
3. If the result of atomizing either comparison term is an XQuery sequence of two or more XQuery items, an error results.
4. If the result of atomizing a term is an `xs:untypedAtomic` type, it is converted to `xs:string` type before being compared.
5. When comparing `xs:string` type values, if the lengths of the data to compare are different, the length of the shorter item determines how many characters will be compared. The characters are compared, starting from the left. If the results of comparing characters are equal, the lengths of the character strings are compared. The comparison results in the longer `xs:string` type value being considered to have the greater value.
6. When comparing numeric data, if the XQuery data types to be evaluated are different, the comparison is performed using the XQuery data type with the greater range. The relative ranges of the numeric data types are as follows:

   xs:double > xs:decimal > xs:int
7. When comparing `xs:hexBinary` types, if the lengths of the data to compare are different, the length of the shorter item determines how many characters will be compared. The characters are compared, starting from the left. If the results are

equal, the lengths of the character strings are compared. The comparison results in the longer `xs:hexBinary` type value being considered to have the greater value.

An error results if the XQuery data types of the comparison terms cannot be compared. The following table indicates which combinations of XQuery data types can be compared.

Table 1-69: Combinations of XQuery data types that can be compared

XQuery data type of the first operand	XQuery data type of the second operand						
	xs:string	Numeric data type	xs:dateTime	xs:date	xs:time	xs:hexBinary	s:boolean
xs:string	Y	N	N	N	N	N	N
Numeric data type	N	Y	N	N	N	N	N
xs:dateTime	N	N	Y	N	N	N	N
xs:date	N	N	N	Y	N	N	N
xs:time	N	N	N	N	Y	N	N
xs:hexBinary	N	N	N	N	N	Y#	N
xs:boolean	N	N	N	N	N	N	Y#

Legend:

Y: Can compare.

N: Cannot compare.

#

Can only be compared using the comparison operators `eq` or `ne`, and not the other comparison operators.

The following tables list the results of comparing `xs:double` types with each operator.

Table 1-70: Results of comparing xs:double types (eq operator)

No.	First operand (value converted to xs:double type)	Second operand (value converted to xs:double type)						
		+0	-0	+INF	-INF	NaN	Other negative numbers	All others
1	+0	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
2	-0	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
3	+INF	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
4	-INF	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE
5	NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
6	Other negative numbers	FALSE	FALSE	FALSE	FALSE	FALSE	Comparison result	Comparison result
7	All others	FALSE	FALSE	FALSE	FALSE	FALSE	Comparison result	Comparison result

Legend:

+0: Positive 0

-0: Negative 0

`+INF`: Positive infinity

`-INF`: Negative infinity

`NaN`: Not a number

Comparison result: Result of the comparison is determined according to *Table 1-68*.

Table 1-71: Results of comparing xs:double types (ne operator)

No.	First operand (value converted to xs:double type)	Second operand (value converted to xs:double type)						
		+0	-0	+INF	-INF	NaN	Other negative numbers	All others
1	+0	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE

No.	First operand (value converted to xs:double type)	Second operand (value converted to xs:double type)						
		+0	-0	+INF	-INF	NaN	Other negative numbers	All others
2	-0	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE
3	+INF	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE
4	-INF	TRUE	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE
5	NaN	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
6	Other negative numbers	TRUE	TRUE	TRUE	TRUE	TRUE	Comparison result	Comparison result
7	All others	TRUE	TRUE	TRUE	TRUE	TRUE	Comparison result	Comparison result

Legend:

+0: Positive 0

-0: Negative 0

`+INF`: Positive infinity

`-INF`: Negative infinity

`NaN`: Not a number

Comparison result: Result of the comparison is determined according to *Table 1-68*.

Table 1-72: Results of comparing xs:double types (lt operator)

No.	First operand (value converted to xs:double type)	Second operand (value converted to xs:double type)						
		+0	-0	+INF	-INF	NaN	Other negative numbers	All others
1	+0	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	TRUE
2	-0	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	TRUE
3	+INF	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

No.	First operand (value converted to xs:double type)	Second operand (value converted to xs:double type)						
		+0	-0	+INF	-INF	NaN	Other negative numbers	All others
4	-INF	TRUE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE
5	NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
6	Other negative numbers	TRUE	TRUE	TRUE	FALSE	FALSE	Comparison result	Comparison result
7	All others	FALSE	FALSE	TRUE	FALSE	FALSE	Comparison result	Comparison result

Legend:

+0: Positive 0

-0: Negative 0

`+INF`: Positive infinity

`-INF`: Negative infinity

`NaN`: Not a number

Comparison result: Result of the comparison is determined according to *Table 1-68*.

Table 1-73: Results of comparing xs:double types (le operator)

No.	First operand (value converted to xs:double type)	Second operand (value converted to xs:double type)						
		+0	-0	+INF	-INF	NaN	Other negative numbers	All others
1	+0	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	TRUE
2	-0	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	TRUE
3	+INF	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
4	-INF	TRUE	TRUE	TRUE	TRUE	FALSE	TRUE	TRUE
5	NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

No.	First operand (value converted to xs:double type)	Second operand (value converted to xs:double type)						
		+0	-0	+INF	-INF	NaN	Other negative numbers	All others
6	Other negative numbers	TRUE	TRUE	TRUE	FALSE	FALSE	Comparison result	Comparison result
7	All others	FALSE	FALSE	TRUE	FALSE	FALSE	Comparison result	Comparison result

Legend:

+0: Positive 0

-0: Negative 0

+INF: Positive infinity

-INF: Negative infinity

NaN: Not a number

Comparison result: Result of the comparison is determined according to *Table 1-68*.

Table 1-74: Results of comparing xs:double types (gt operator)

No.	First operand (value converted to xs:double type)	Second operand (value converted to xs:double type)						
		+0	-0	+INF	-INF	NaN	Other negative numbers	All others
1	+0	FALSE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE
2	-0	FALSE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE
3	+INF	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE
4	-INF	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
5	NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
6	Other negative numbers	FALSE	FALSE	FALSE	TRUE	FALSE	Comparison result	Comparison result

No.	First operand (value converted to xs:double type)	Second operand (value converted to xs:double type)						
		+0	-0	+INF	-INF	NaN	Other negative numbers	All others
7	All others	TRUE	TRUE	FALSE	TRUE	FALSE	Comparison result	Comparison result

Legend:

+0: Positive 0

-0: Negative 0

+INF: Positive infinity

-INF: Negative infinity

NaN: Not a number

Comparison result: Result of the comparison is determined according to *Table 1-68*.

Table 1-75: Results of comparing xs:double types (ge operator)

No.	First operand (value converted to xs:double type)	Second operand (value converted to xs:double type)						
		+0	-0	+INF	-INF	NaN	Other negative numbers	All others
1	+0	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE
2	-0	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE
3	+INF	TRUE	TRUE	TRUE	TRUE	FALSE	TRUE	TRUE
4	-INF	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE
5	NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
6	Other negative numbers	FALSE	FALSE	FALSE	TRUE	FALSE	Comparison result	Comparison result
7	All others	TRUE	TRUE	FALSE	TRUE	FALSE	Comparison result	Comparison result

Legend:

+0: Positive 0

-0: Negative 0

`+INF`: Positive infinity

`-INF`: Negative infinity

`NaN`: Not a number

Comparison result: Result of the comparison is determined according to *Table 1-68*.

- *general-comparison* `::=` {`=`| `!=`| `<`| `<=`| `>`| `>=`| `<>`}

Compares XQuery sequences of any length to each other.

Comparisons are performed in the following sequence:

1. Each comparison term is atomized.
2. In the combination of atomic values for each atomized comparison term from among the XQuery sequences, if there is any set of atomic values for which the value comparison performed as shown below results in a `TRUE` evaluation, the result will be `TRUE`. If there is no such set, the result will be `FALSE`.
   - For two atomic values, the value comparison corresponding to the specified general comparison is performed as shown in *Table 1-76*. However, if the atomic value is `xs:untypedAtomic` type, the value comparison is performed after the type conversion shown in *Table 1-77* is performed (determined by the XQuery data type of the other atomic value).

Table 1-76: General comparisons and their corresponding value comparisons

No.	General comparison operator	Corresponding value comparison
1	=	eq
2	!=, <>	ne
3	<	lt
4	<=	le
5	>	gt
6	>=	ge

Table 1-77: Type conversion of atomic values of the xs:untypedAtomic type

No.	XQuery data type of the other atomic value	XQuery data type after type conversion
1	Numeric data type	xs:double
2	xs:untypedAtomic	xs:string
3	xs:string	xs:string
4	Other than the above	Same XQuery data type as the other atomic value

- *node-comparison* ::= {is| <<| >>}

Compares nodes to each other based on node ID and document order.

The node ID, which is bound to all nodes in the tree representation of the XQuery data model, acts as a unique identifier within the tree.

The following table describes the ways nodes can be compared.

Table 1-78: Node comparisons

No.	Node comparison operator	Conditions under which TRUE
1	is	The node IDs of the two nodes being compared are the same.
2	<<	The first node being compared appears before the second node being compared in the document order.
3	>>	The first node being compared appears after the second node being compared in the document order.

Comparisons are performed according to the following procedures:

1. Each comparison term is either a single node XQuery sequence or an empty XQuery sequence.
2. If either is an empty XQuery sequence, the result will be an empty XQuery sequence.
3. If both are single node XQuery sequences, compare the nodes that make up each comparison term XQuery sequence.
4. If the nodes specified for each comparison term belong to different trees in the XQuery data model, the node specified as the first comparison term will be first in the ordering.

(d) Example

The following is an example of an XQuery comparison expression.

Element represented by the XQuery variable `book`:

```
<bookinfo book_id="452469630">
  <title>Relational Databases Explained</title>
  <category>database </category>
  <author>Jeff Jones</author>
  <author>Bob Adams</author>
  <price>3000</price>
</bookinfo>
```

XQuery comparison expression examples and results:

No.	Example	Result	Explanation
1	`$book[@book_id` eq "452469630"]	`bookinfo` element node	Because the `typed-value` of the `book_id` attribute node of the `bookinfo` element node in the XQuery variable `book` equals `452469630`, the result is the `bookinfo` element node from the XQuery variable `book`.
2	(1, 2) = (2, 3)	`TRUE`	The result is `TRUE` because the `eq` relation holds between the second XQuery item in the first comparison term and the first XQuery item in the second comparison term.
3	(1, 2) != (2, 3)	`TRUE`	The result is `TRUE` because the `ne` relation holds between the first XQuery item in the first comparison term and the first XQuery item in the second comparison term.

(9) XQuery range expression

(a) Function

Returns an XQuery sequence consisting of atomic values that are consecutive integer values (`xs:int` type).

Under certain conditions, the evaluation result from the XQuery arithmetic expression is returned unchanged.

(b) Format

XQuery-range-expression `::=` *XQuery-arithmetic-expression* `[to` *XQuery-arithmetic-expression*`]`

(c) Operands

- *XQuery-arithmetic-expression* [to *XQuery-arithmetic-expression*]

If to is specified, compares the results of evaluating the XQuery arithmetic expressions.

If you specify the XQuery arithmetic expression by itself, the evaluation result from that XQuery arithmetic expression is returned unchanged.

If to is specified, the XQuery sequence generated according to the following rules is returned.

1. Atomize the results of evaluating each XQuery arithmetic expression.
2. The atomization result must be an empty XQuery sequence or a single XQuery sequence consisting of a single atomic value that can be converted into xs:int type.
3. If any of the results of evaluating the XQuery arithmetic expression is an empty XQuery sequence, the result will be an empty XQuery sequence.
4. If the integer value resulting from converting the atomic value of the result of evaluating the first XQuery arithmetic expression into xs:int type is greater than the integer value resulting from converting the atomic value of the result of evaluating the second XQuery arithmetic expression into xs:int type, the result will be an empty XQuery sequence.
5. If neither rule 2 nor 3 above is applicable, the result is the XQuery sequence consisting of all integer values between (greater than or equal to the first integer, less than or equal to the second integer) the two integer values resulting from converting the atomic value of the results of evaluating the XQuery arithmetic expressions into xs:int type.

(d) Example

The following example shows an XQuery range expression.

No.	Example	Result	Explanation
1	(1 to 10)	(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)	Generates the XQuery sequence consisting of integer values from 1 to 10.

(10) XQuery arithmetic expression

(a) Function

Returns the result of addition, subtraction, multiplication, division, or modulo calculations.

Under certain conditions, the evaluation result from the XQuery sequence operation expression is returned unchanged.

(b) Format

```
XQuery-arithmetic-expression ::= XQuery-addition-or-subtraction-expression
XQuery-addition-or-subtraction-expression ::= XQuery-multiplication-or-division-expression [{+| -}
XQuery-multiplication-or-division-expression]...
XQuery-multiplication-or-division-expression ::= XQuery-sequence-operation-expression
    [{*| div| idiv| mod} XQuery-sequence-operation-expression]...
```

(c) Operands

- *XQuery-addition-or-subtraction-expression* ::=
  XQuery-multiplication-or-division-expression [{+|-}
  XQuery-multiplication-or-division-expression]...

If an operator (+, -) is specified, the results of evaluating the XQuery multiplication or division expression are added or subtracted.

If you specify the XQuery multiplication or division expression by itself, the evaluation result from that XQuery multiplication or division expression is returned unchanged.

The following table describes the meaning and function of each operator.

Table 1-79: Functions of operators in XQuery addition or subtraction expressions

No.	Operator	Meaning	Function
1	+	Addition	Add the second operand to the first operand.
2	-	Subtraction	Subtract the second operand from the first operand.

- *XQuery-multiplication-or-division-expression* ::=
  XQuery-sequence-operation-expression

  [{*|div|idiv|mod} *XQuery-sequence-operation-expression*]...

If an operator (*, div, idiv, mod) is specified, multiplication, division, or modulo calculations are performed on the results of evaluating the XQuery sequence operation expressions.

If you specify the XQuery sequence operation expression by itself, the evaluation result from that XQuery sequence operation expression is returned unchanged.

The following table describes the meaning and function of each operator.

Table 1-80: Functions of operators in XQuery multiplication or division expressions

No.	Operator	Meaning	Function
1	*	Multiplication	Multiply the first operand by the second operand.
2	div	Division	Divide the first operand by the second operand.
3	idiv	Integer division	Truncate the result of dividing the first operand by the second operand to an integer value.
4	mod	Modulo	Calculate the remainder when the first operand *a* is divided by the second operand *b*. The result of this operation is *a* - (*a* idiv *b*) * *b*.

(d) Evaluation of XQuery arithmetic expressions

Addition, subtraction, multiplication, division, and modulo calculations are performed on XQuery arithmetic expressions in the following sequence:

1. Each operand is atomized.
2. If any atomized operand is an empty XQuery sequence, the result will be an empty XQuery sequence.
3. An error results if any atomized operand is an XQuery sequence with two or more atomic values.
4. If the XQuery data type of the atomized operand is the `xs:untypedAtomic` type, it is converted to `xs:double` type. An error results if this conversion is not possible.
5. Operations are performed on the two XQuery sequences obtained according to the steps above. The following tables list the XQuery data type of the result based on the XQuery data type of each operator.

   Table 1-81: Relationship between the XQuery data types of the operands and the XQuery data type of the result (+ operator)

No.	First operand	Second operand		
		xs:int	xs:decimal	xs:double
1	xs:int	xs:int	xs:decimal	xs:double
2	xs:decimal	xs:decimal	xs:decimal	xs:double
3	xs:double	xs:double	xs:double	xs:double

Table 1-82: Relationship between the XQuery data types of the operands and the XQuery data type of the result (- operator)

No.	First operand	Second operand		
		xs:int	xs:decimal	xs:double
1	xs:int	xs:int	xs:decimal	xs:double
2	xs:decimal	xs:decimal	xs:decimal	xs:double
3	xs:double	xs:double	xs:double	xs:double

Table 1-83: Relationship between the XQuery data types of the operands and the XQuery data type of the result (* operator)

No.	First operand	Second operand		
		xs:int	xs:decimal	xs:double
1	xs:int	xs:int	xs:decimal	xs:double
2	xs:decimal	xs:decimal	xs:decimal	xs:double
3	xs:double	xs:double	xs:double	xs:double

Table 1-84: Relationship between the XQuery data types of the operands and the XQuery data type of the result (div operator)

No.	First operand	Second operand		
		xs:int	xs:decimal	xs:double
1	xs:int	xs:decimal	xs:decimal	xs:double
2	xs:decimal	xs:decimal	xs:decimal	xs:double
3	xs:double	xs:double	xs:double	xs:double

Table 1-85: Relationship between the XQuery data types of the operands and the XQuery data type of the result (idiv operator)

No.	First operand	Second operand		
		xs:int	xs:decimal	xs:double
1	xs:int	xs:int	xs:int	xs:int
2	xs:decimal	xs:int	xs:int	xs:int
3	xs:double	xs:int	xs:int	xs:int

Table 1-86: Relationship between the XQuery data types of the operands and the XQuery data type of the result (mod operator)

No.	First operand	Second operand		
		xs:int	xs:decimal	xs:double
1	xs:int	xs:int	xs:decimal	xs:double
2	xs:decimal	xs:decimal	xs:decimal	xs:double
3	xs:double	xs:double	xs:double	xs:double

6. An error results if an operand meets either of the following conditions:
   - If the operands are not both `xs:double` type, and `0` is specified as the second operand of the `div` or `mod` operations.
   - If 0 is specified as the second operand of the `idiv` operation.
7. The following tables list the results for each operator when at least one of the operands is `xs:double` type.

   Table 1-87: Result when at least one of the operands is xs:double type (+ operator)

No.	First operand (value converted to xs:double type)	Second operand (value converted to xs:double type)						
		+0	-0	+INF	-INF	NaN	Other negative numbers	All others
1	+0	+0	+0	+INF	-INF	NaN	Result of addition	Result of addition
2	-0	+0	-0	+INF	-INF	NaN	Result of addition	Result of addition
3	+INF	+INF	+INF	+INF	NaN	NaN	+INF	+INF
4	-INF	-INF	-INF	NaN	-INF	NaN	-INF	-INF
5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6	Other negative numbers	Result of addition	Result of addition	+INF	-INF	NaN	Result of addition	Result of addition
7	All others	Result of addition	Result of addition	+INF	-INF	NaN	Result of addition	Result of addition

Legend:

+0: Positive 0

-0: Negative 0

`+INF`: Positive infinity

`-INF`: Negative infinity

`NaN`: Not a number

Result of addition: The result of the operation is determined according to the functions shown in *Table 1-79*.

Table 1-88: Result when at least one of the operands is xs:double type (- operator)

No.	First operand (value converted to xs:double type)	Second operand (value converted to xs:double type)						
		+0	-0	+INF	-INF	NaN	Other negative numbers	All others
1	+0	+0	+0	-INF	+INF	NaN	Result of subtraction	Result of subtraction
2	-0	-0	+0	-INF	+INF	NaN	Result of subtraction	Result of subtraction
3	+INF	+INF	+INF	NaN	+INF	NaN	+INF	+INF
4	-INF	-INF	-INF	-INF	NaN	NaN	-INF	-INF
5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6	Other negative numbers	Result of subtraction	Result of subtraction	-INF	+INF	NaN	Result of subtraction	Result of subtraction
7	All others	Result of subtraction	Result of subtraction	-INF	+INF	NaN	Result of subtraction	Result of subtraction

Legend:

+0: Positive 0

-0: Negative 0

+INF: Positive infinity

-INF: Negative infinity

NaN: Not a number

Result of subtraction: Result of the operation is determined according to the functions shown in *Table 1-79*.

Table 1-89: Result when at least one of the operands is xs:double type (* operator)

No.	First operand (value converted to xs:double type)	Second operand (value converted to xs:double type)						
		+0	-0	+INF	-INF	NaN	Other negative numbers	All others
1	+0	+0	-0	NaN	NaN	NaN	Result of multiplication	Result of multiplication
2	-0	-0	+0	NaN	NaN	NaN	Result of multiplication	Result of multiplication
3	+INF	NaN	NaN	+INF	-INF	NaN	-INF	+INF
4	-INF	NaN	NaN	-INF	+INF	NaN	+INF	-INF
5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6	Other negative numbers	Result of multiplication	Result of multiplication	-INF	+INF	NaN	Result of multiplication	Result of multiplication
7	All others	Result of multiplication	Result of multiplication	+INF	-INF	NaN	Result of multiplication	Result of multiplication

Legend:

+0: Positive 0

-0: Negative 0

+INF: Positive infinity

-INF: Negative infinity

NaN: Not a number

Result of multiplication: Result of the operation is determined according to the functions shown in *Table 1-80*.

Table 1-90: Result when at least one of the operands is xs:double type (div operator)

No.	First operand (value converted to xs:double type)	Second operand (value converted to xs:double type)						
		+0	-0	+INF	-INF	NaN	Other negative numbers	All others
1	+0	NaN	NaN	+0	-0	NaN	Result of division	Result of division
2	-0	NaN	NaN	-0	+0	NaN	Result of division	Result of division
3	+INF	+INF	-INF	NaN	NaN	NaN	-INF	+INF
4	-INF	-INF	+INF	NaN	NaN	NaN	+INF	-INF
5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6	Other negative numbers	-INF	INF	-0	+0	NaN	Result of division	Result of division
7	All others	INF	-INF	+0	-0	NaN	Result of division	Result of division

Legend:

+0: Positive 0

-0: Negative 0

`+INF`: Positive infinity

`-INF`: Negative infinity

`NaN`: Not a number

Result of division: Result of the operation is determined according to the functions shown in *Table 1-80*.

Table 1-91: Result when at least one of the operands is xs:double type (idiv operator)

No.	First operand (value converted to xs:double type)	Second operand (value converted to xs:double type)						
		+0	-0	+INF	-INF	NaN	Other negative numbers	All others
1	+0	Divide by zero error	Divide by zero error	0	0	Type conversion error	Result of integer division	Result of integer division
2	-0	Divide by zero error	Divide by zero error	0	0	Type conversion error	Result of integer division	Result of integer division
3	+INF	Divide by zero error	Divide by zero error	Type conversion error	Type conversion error	Type conversion error	Type conversion error	Type conversion error
4	-INF	Divide by zero error	Divide by zero error	Type conversion error	Type conversion error	Type conversion error	Type conversion error	Type conversion error
5	NaN	Divide by zero error	Divide by zero error	Type conversion error	Type conversion error	Type conversion error	Type conversion error	Type conversion error
6	Other negative numbers	Divide by zero error	Divide by zero error	0	0	Type conversion error	Result of integer division	Result of integer division
7	All others	Divide by zero error	Divide by zero error	0	0	Type conversion error	Result of integer division	Result of integer division

Legend:

0: `xs:int` type `0`

Result of integer division: Result of the operation is determined according to the functions shown in *Table 1-80*.

Divide by zero error: Error because the divisor is `0`.

Type conversion error: Error because the result of the `div` operation cannot be represented as `xs:int` type.

Table 1-92: Result when at least one of the operands is xs:double type (mod operator)

No.	First operand (value converted to xs:double type)	Second operand (value converted to xs:double type)						
		+0	-0	+INF	-INF	NaN	Other negative numbers	All others
1	+0	NaN	NaN	+0	+0	NaN	Result of modulo operation	Result of modulo operation
2	-0	NaN	NaN	-0	-0	NaN	Result of modulo operation	Result of modulo operation
3	+INF	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	-INF	NaN	NaN	NaN	NaN	NaN	NaN	NaN
5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
6	Other negative numbers	NaN	NaN	First operand (value converted to `xs:double` type)	First operand (value converted to `xs:double` type)	NaN	Result of modulo operation	Result of modulo operation
7	All others	NaN	NaN	First operand (value converted to `xs:double` type)	First operand (value converted to `xs:double` type)	NaN	Result of modulo operation	Result of modulo operation

Legend:

+0: Positive 0

-0: Negative 0

`+INF`: Positive infinity

`-INF`: Negative infinity

`NaN`: Not a number

First operand: Value of first operand

Result of modulo operation: Result of the operation is determined according to the functions shown in *Table 1-80*.

(e) Notes

The subtraction operator (-) must be preceded by either a space (X'20'), TAB (X'09'), NL (X'0a') or CR (X'0d'). Without one of these characters, it is considered part of the name specified before the subtraction operator. For example, a-b is interpreted as a name, whereas a - b and a -b are interpreted as XQuery arithmetic expressions.

(f) Example

The following are examples of XQuery arithmetic expressions.

No.	Example	Result (XQuery data type)
1	`4 - 2`	2 (xs:int)
2	`5 div 2`	2.5 (xs:decimal)
3	`5 idiv 2`	2 (xs:int)
4	`5 mod 2`	1 (xs:int)

(11) XQuery sequence operation expression

(a) Function

Treats XQuery sequences consisting of nodes as ordered sets and returns the XQuery sequence that results from performing the union, intersection, or difference operations.

Under certain conditions, the evaluation result from the XQuery unary expression is returned unchanged.

(b) Format

XQuery-sequence-operation-expression ::= *XQuery-union-expression*
XQuery-union-expression ::= {*XQuery-intersection-expression*| *XQuery-difference-expression*}
[{union| *vertical-bar*} {*XQuery-intersection-expression*| *XQuery-difference-expression*}]...
XQuery-intersection-expression ::= *XQuery-unary-expression* [intersect *XQuery-unary-expression*]...
XQuery-difference-expression ::= *XQuery-unary-expression* [except *XQuery-unary-expression*]...

(c) Operands

- *XQuery-union-expression* ::= {*XQuery-intersection-expression*| *XQuery-difference-expression*}

  [{union| *vertical-bar*} {*XQuery-intersection-expression*| *XQuery-difference-expression*}]...

If `union` or a vertical bar (|) is specified, this derives an XQuery sequence consisting of all the different nodes that are contained in the two XQuery sequences that are the results of evaluating the XQuery intersection expressions or XQuery difference expressions.

If you specify the XQuery intersection expression or the XQuery difference expression by itself, the evaluation result from that XQuery intersection expression or XQuery difference expression is returned unchanged.

- *XQuery-intersection-expression* `::=` *XQuery-unary-expression* `[intersect` *XQuery-unary-expression*`] . . .`

If `intersect` is specified, this derives an XQuery sequence consisting of all the different nodes that are contained in both of the two XQuery sequences that are the results of evaluating the XQuery unary expressions.

If you specify the XQuery unary expression by itself, the evaluation result from that XQuery unary expression is returned unchanged.

- *XQuery-difference-expression* `::=` *XQuery-unary-expression* `[except` *XQuery-unary-expression*`] . . .`

If `except` is specified, this derives an XQuery sequence consisting of all the different nodes that are contained in the XQuery sequence that is the result of evaluating the first XQuery unary expression, but that are not contained in the XQuery sequence that is the result of evaluating the second XQuery unary expression.

If you specify the XQuery unary expression by itself, the evaluation result from that XQuery unary expression is returned unchanged.

(d) Rules

1. Duplicate nodes are removed from the result XQuery sequence. Also, the order of the nodes in the result XQuery sequence will be their order in the document.
2. In the result XQuery sequence, if there are nodes that belongs to different trees, the order of the tree is determined according to the rules below. Within any given tree, the order of nodes in the result is determined by their order in the document. Here, let ($T_{1\text{-}1}$, $T_{1\text{-}2}$, ..., $T_{1\text{-}n}$) be the trees to which each node in the first operand's XQuery sequence belong, and let ($T_{2\text{-}1}$, $T_{2\text{-}2}$, ..., $T_{2\text{-}m}$) be the trees to which each node in the second operand's XQuery sequence belong.
   - The order of trees $T_{1\text{-}1}$ to $T_{1\text{-}n}$ will be order in which they appeared in the first operand's XQuery sequence.
   - Of the trees $T_{2\text{-}1}$ to $T_{2\text{-}m}$, that are different than any trees $T_{1\text{-}1}$ to $T_{1\text{-}n}$, they will be in the order in which the trees first appeared in the second operand's XQuery sequence. Also, they will follow any trees $T_{1\text{-}1}$ to $T_{1\text{-}n}$ in the final order.

(e) Note

Note that the order of evaluation is different for the XQuery sequence operation expressions (`union`, `intersect`, and `except`) and the set operations (`UNION` and `EXCEPT`) in the body of an SQL query.

(f) Example

The following are examples of XQuery sequence operation expressions.

XQuery sequence operation expression examples and results:

No.	Example	Result
1	`(A,B) union (A,B)`	(A, B)
2	`(A,B) union (B,C)`	(A, B, C)
3	`(A,B) intersect (A,B)`	(A, B)
4	`(A,B) intersect (B,C)`	(B)
5	`(A,B) except (A,B)`	()
6	`(A,B) except (B,C)`	(A)

Note

`A`, `B`, and `C` represent element nodes in the same XML document.

Document order is bound to the order `A`, `B`, `C`.

(12) XQuery unary expression

(a) Function

Returns the result of performing a unary operation on an XQuery value expression.

Under certain conditions, the evaluation result from the XQuery value expression is returned unchanged.

(b) Format

XQuery-unary-expression `::= [{-| +}]...` *XQuery-value-expression*

(c) Operands

- `[{-| +}]...` *XQuery-value-expression*

If a unary operator (`-`, `+`) is specified, the result of performing the unary operation on the XQuery value expression is returned.

If you specify the XQuery value expression by itself, the evaluation result from that XQuery value expression is returned unchanged.

The following table describes the meaning and function of each operator.

Table 1-93: Functions of operators in XQuery unary expressions

No.	Operator	Meaning	Function
1	-	Minus sign	Reverses the sign.
2	+	Plus sign	Does not reverse the sign.

(d) Evaluation of XQuery unary expressions

In XQuery unary expressions, unary operations are performed the following sequence:

1. The operand is atomized.
2. If the atomized operand is an empty XQuery sequence, the result will be an empty XQuery sequence.
3. An error results if the atomized operand is an XQuery sequence of two or more atomic values.
4. If the XQuery data type of the atomized operand is `xs:untypedAtomic` type, it is converted to `xs:double` type. An error results if this conversion is not possible.
5. The operation is performed on the XQuery sequence obtained according to the steps above. The XQuery data type of the result will be the same as the XQuery data type at the end of step 4.

(13) XQuery value expression

(a) Function

Returns the result of evaluating an XQuery path expression.

(b) Format

XQuery-value-expression `::=` *XQuery-path-expression*

(c) Operands

- *XQuery-path-expression*

Specify the value to be evaluated in an XQuery path expression.

(14) XQuery path expression

(a) Function

Returns the result of evaluating the selected nodes in the XQuery data model.

(b) Format

```
XQuery-path-expression ::= {/[relative-path-expression]|//
relative-path-expression|relative-path-expression}
relative-path-expression ::= step-expression [{/|//}step-expression]...
step-expression ::= {axis-step-expression| filter-expression}
axis-step-expression ::= axis-step [XQuery-predicate]...
axis-step ::= {axis {name-test |kind-test}| optional-axis-step}
name-test ::= {qualified-name|*|prefix:*|*:local-name}
optional-axis-step ::= {[@]{name-test|kind-test}| ..}
filter-expression ::= XQuery-primary-expression [XQuery-predicate]...
XQuery-predicate ::= left-bracket XQuery-expression right-bracket
```

(c) Operands

- XQuery path expression, leading forward slash (/)

Selects a node at the root of the tree in which the context item belongs.

An error results if the context item is not a node.

- XQuery path expression, leading double forward slash (//)

Selects a node at the root of the tree node in which the context item belongs, and all of the descendant nodes of that node.

An error results if the context item is not a node.

- XQuery path expression, forward slash (/) located other than at the beginning

A forward slash (/) specified other than at the beginning of an XQuery path expression is treated as a separator for a step expression. The XQuery sequence obtained by evaluating the step expressions is evaluated from left to right.

For example, the relative path expression `E1/E2` is evaluated as follows:

1. Step expression `E1` is evaluated.
2. Assuming each XQuery item in the XQuery sequence obtained by evaluating step expression `E1` is a context item in the evaluation of step expression `E2`, step expression `E2` is evaluated for each of those context items. If the XQuery sequence obtained by evaluating step expression `E1` is empty, `E2` will not be evaluated, and the result of evaluating `E1/E2` will be an empty XQuery sequence.
3. All of the XQuery sequences obtained by evaluating step expression `E2` for each context item in step 2 are joined sequentially. The resulting concatenated XQuery

sequence becomes the result of evaluating `E1/E2`.

The XQuery sequence resulting from evaluating the step expression on the left side of the forward slash cannot contain atomic values.

- XQuery path expression, double forward slash (`//`) located other than at the beginning

This is a short form of the expression `/descendant-or-self::node()/`.

For each context item from the results of evaluating the step expression specified on the left side, evaluates the step expression (the context item itself and its descendant nodes) specified on the right side of the XQuery sequence.

The XQuery sequence resulting from evaluating the step expression on the left side of the double forward slash cannot contain atomic values.

- *axis-step-expression* `::=` *axis-step* `[`*XQuery-predicate*`]...`

The axis step expression returns an XQuery sequence consisting of result nodes that have been narrowed down according to the name, kind of node, and other criteria you specify. They are obtained by traversing the specified tree starting from the context item. The nodes in this XQuery sequence become the next context items.

An error results if the context item is not a node.

- *axis-step* `::=` {*axis* {*name-test* | *kind-test*} | *optional-axis-step*}

The axis step returns an XQuery sequence consisting of those nodes that meet the name test or kind test out of all of the nodes that are reachable along the specified axis starting from the context item. The nodes in this XQuery sequence become the next context items.

axis

Specifies how the tree is to be traversed from the context item.

The following table and figures shows the items that can be specified as the axis and how the position of the context items being returned is assigned.

Table 1-94: List of axes

No.	Axis	Explanation	Assigning context positions
1	child::	Indicates child nodes.	Document order
2	descendant::	Indicates descendant nodes.	
3	attribute::	Indicates the attribute nodes belonging to the context item node.	
4	self::	Indicates the context item node itself.	

No.	Axis	Explanation	Assigning context positions
5	descendant-or-self	Indicates the context item node itself and all of its descendant nodes.	
6	following-sibling::	Indicates the sibling nodes that follow the context item node, in document order. If the context item node is an attribute node, the result is an empty XQuery sequence.	
7	following::	Indicates nodes that are not descendants of the context item node and that follow it in document order. Attribute nodes are not selected.	
8	parent::	Indicates the parent node. If the context item node has no parent node, the result is an empty XQuery sequence.	
9	ancestor::	Indicates the ancestor nodes.	Reverse document order
10	preceding-sibling::	Indicates the sibling nodes that precede the context item node, in document order. If the context item node is an attribute node, the result is an empty XQuery sequence.	
11	preceding::	Indicates nodes that are not ancestors of the context item node and that precede it in document order. Attribute nodes are not selected.	
12	ancestor-or-self::	Indicates the context item node itself and all of its ancestor nodes.	

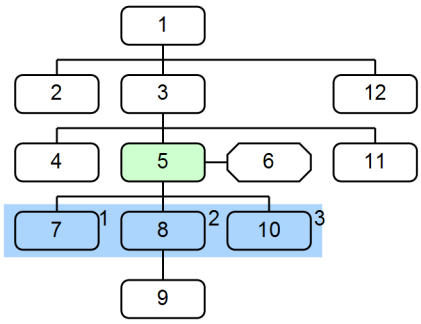
Legend:

Document order: Context position is assigned according to document order.

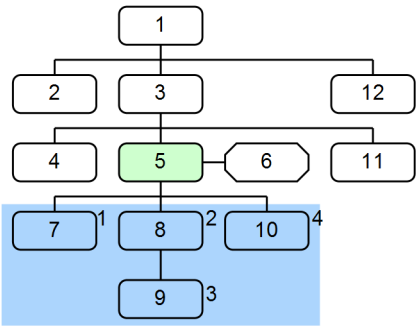
Reverse document order: Context position is assigned according to reverse document order.

Figure 1-17: Axes

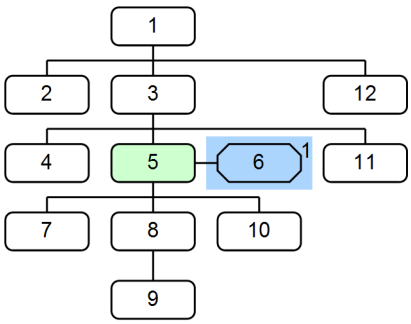
- `child::`



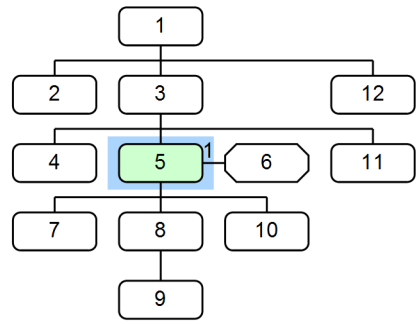
- `descendant::`



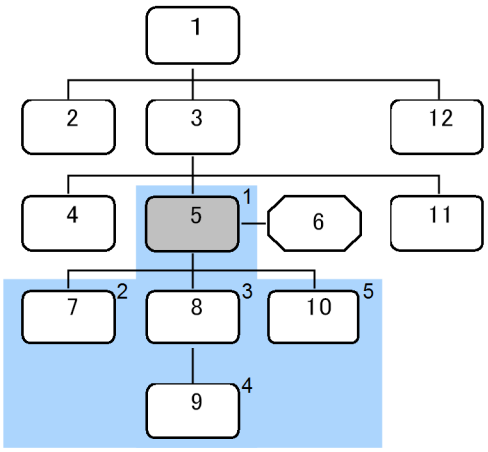
- `attribute::`



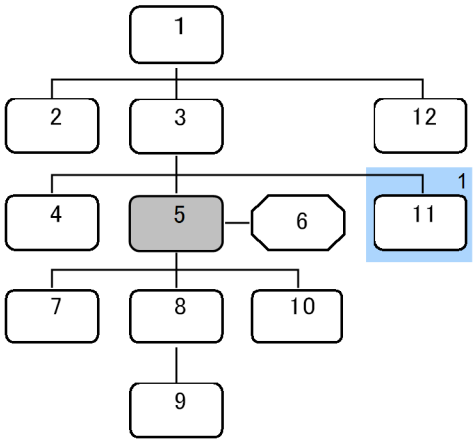
- `self::`



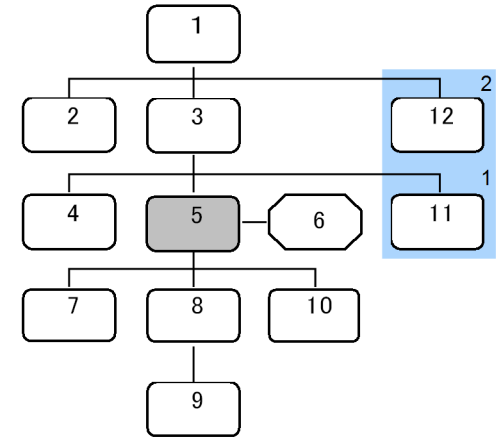
- descendant-or-self::



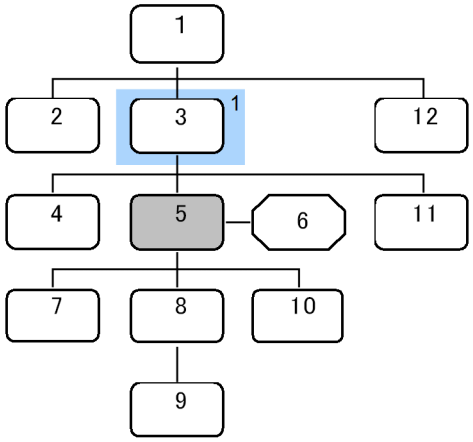
- following-sibling::



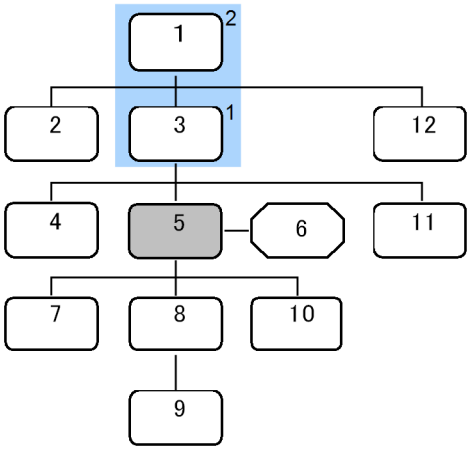
- following::



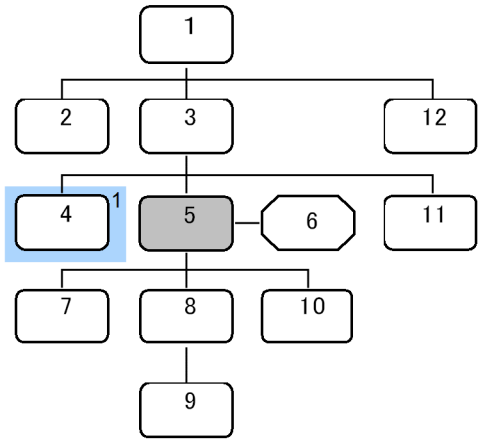
- parent::



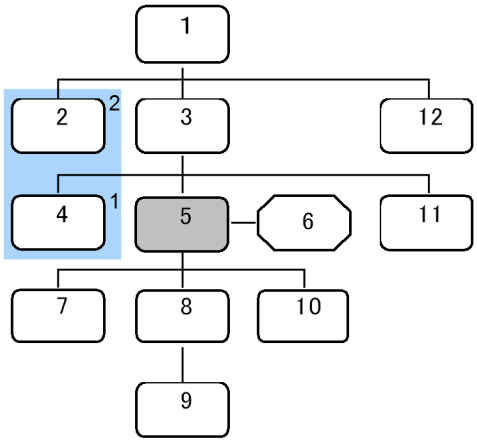
- `ancestor::`



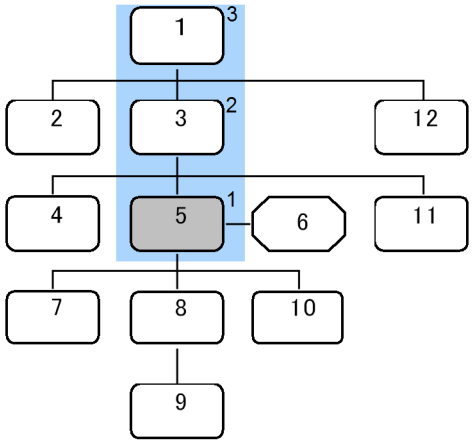
- `preceding-sibling::`



- `preceding::`



- `ancestor-or-self::`



Legend:

5 : Context item

: Scope of axis
(the number at the top right of the node indicates the context position)

■ *name-test* ::= {*qualified-name* | * | *prefix*:* | *:*local-name*}

Specify that only nodes that match the specified name are to be selected from all of the nodes that can be reached by traversing the specified axis.

qualified-name

From among the nodes that are the target of the name test, selects only those nodes for which the expanded qualified name is equivalent to the specified qualified name.

*

Specified when not using the name test to narrow down the nodes being targeted.

prefix : *

From among the nodes that are the targets of the name test, selects only those nodes for which the URI of the prefix specified here is equal to the XML namespace URI for the expanded qualified name. Does not depend on the local name.

* : *local-name*

From among the nodes that are the targets of the name test, selects only those nodes that are equal to the local name specified here as *local-name*. Does not depend on the XML namespace URI.

■ *kind-test*

Specify that only nodes that match the specified kind are selected from among all of the nodes that can be reached by traversing the specified axis.

The following table lists the kind tests.

Table 1-95: List of kind tests

No.	Category	How specified	Explanation
1	Document test	document-node()	Matches any document node.
2		`document-node(`*element-test*`)`	Among document nodes that have only one element node as a child node (all their other child nodes being comment nodes or processing instruction nodes only), matches any document nodes in which the child element node matches the element test.
3	Element test	element()	Matches any element node.
4		element(*)	
5		`element(`*qualified-name*`)`	Among XML elements, matches element nodes whose expanded qualified name is equivalent to the post-expand qualified name of the specified qualified name.
6	Attribute test	attribute()	Matches any attribute node.
7		attribute(*)	

No.	Category	How specified	Explanation
8		`attribute(`*qualified-name*`)`	Among XML attributes, matches element nodes whose expanded qualified name is equivalent to the specified qualified name.
9	Processing instruction test	processing-instruction()	Matches any processing instruction node.
10		`processing-instruction(`*processing-instruction-target*`)`	Among processing instruction nodes, matches processing instruction nodes with values for the processing instruction target property that match the specified processing instruction target.
11	Comment test	comment()	Matches any comment node.
12	Text test	text()	Matches any text node.
13	Any kind test	node()	Matches any kind of node.

■ *optional-axis-step* `::=` { `[@]` {*name-test*|*kind-test*} | `..` }

Describes an axis step in a simplified manner.

`[@]` {*name-test*|*kind-test*}

`@` is a short form of the expression `attribute::`.

If `@` is omitted, `attribute::` is assumed if the kind test is an attribute test, otherwise, `child::` is assumed to precede the name test or kind test.

`..`

`..` is a short form of the expression `parent::node()`.

The following table provides examples of specifying an optional axis step.

Table 1-96: Optional axis step examples

No.	Optional axis step	Full notation with the same meaning	Meaning
1	/elem/`ElemName`	/elem/`child::ElemName`	Denotes the child node `ElemName` from among the child nodes of the context item after the evaluation of `elem`.
2	/elem/@AttrName	/elem/`attribute::AttrName`	Denotes the attribute node AttrName from among the attribute nodes of the context item after the evaluation of `elem`.

No.	Optional axis step	Full notation with the same meaning	Meaning
3	/elem/attribute()	/elem/attribute::attribute()	Denotes all the attribute nodes of the context item after the evaluation of elem.
4	/elem/..	/elem/parent::node()	Denotes the parent node of the context item after the evaluation of elem.

- *filter-expression* ::= *XQuery-primary-expression* [*XQuery-predicate*] . . .

Evaluates *XQuery-primary-expression*. If *XQuery-predicate* is specified, the condition in *XQuery-predicate* is used to narrow down the XQuery items in the XQuery sequence that is the result of evaluating *XQuery-primary-expression*.

- *XQuery-predicate* ::= *left-bracket XQuery-expression right-bracket*

The condition used to narrow down the context items is specified in *XQuery expression*.

The specified XQuery expression is evaluated for each context item in the XQuery predicate.

The results of evaluating the XQuery predicate are as follows:

- If the result of the XQuery expression specified in the XQuery predicate is an XQuery sequence of a single atomic value of numeric data type, the context item whose context position is equal to the that value will be the result of evaluating the XQuery predicate.
- If the result of the XQuery expression specified in the XQuery predicate is not an XQuery sequence of a single atomic value of numeric data type, the context items that are TRUE when the XQuery expression result is converted to Boolean values will be the result of evaluating the XQuery predicate.

(d) Rules

1. When evaluating an XQuery path expression that contains more than one forward slash (/), the resulting XQuery sequence is as follows:
   - If the result of the step expression at the end of the XQuery path expression is an XQuery sequence that contains only nodes, the result is an XQuery sequence whose nodes are sorted in document order and with duplicate nodes excluded.
   - If the result of the step expression at the end of the XQuery path expression is an XQuery sequence that contains only atomic values, that XQuery sequence is returned unchanged.

- An error results if the result of the step expression at the end of the XQuery path expression is an XQuery sequence that contains nodes and atomic values.

(e) Example

The following is an example of an XQuery path expression.

No.	Example	Explanation
1	`child::bookinfo`	Selects the `bookinfo` element node that is the child node of the context item.
2	`bookinfo`	Short for `child::bookinfo`.
3	`attribute::book_id`	Selects the `book_id` attribute node of the context item.
4	`@book_id`	Short for `attribute::book_id`.
5	attribute::*	Selects all the attribute nodes belonging to the context item.
6	parent::element()	Selects the element node whose parent node is the context item.
7	`bookinfo[category="programming"]`	Selects the `bookinfo` element node that has a child node with a `category` element node for which the value of the typed-value property is `programming`.
8	`bookinfo[@book_id = "12345"]`	Selects the `bookinfo` element node whose child node is the context item for which the value of its `book_id` attribute is `12345`.
9	`author[2]`	Select the second `author` element node whose child node is the context item.

(15) XQuery primary expressions

(a) Function

The basic components that constitute XQuery path expressions.

XQuery primary expressions include the following:

- XQuery literal
- XQuery variable reference
- Parenthesized XQuery expression
- Context item expression
- XQuery function invocation

(b) Format

XQuery-primary-expression ::= {*XQuery-literal*| *XQuery-variable-reference*|
parenthesized-XQuery-expression| *context-item-expression*| *XQuery-function-invocation*}

Each XQuery primary expression is described below.

(c) XQuery literal

- Function

  An XQuery literal is an expression that directly represents an atomic value. The value is determined during parsing.

  The following table lists the XQuery literals that can be specified in XQuery.

  Table 1-97: XQuery literals

No.	XQuery literal		Result		XQuery data type to interpret
			Format	Explanation	
1	XQuery numeric literal	XQuery integer literal#	[*sign*] *unsigned-integer* Examples: -123 45 6789	An unsigned integer is represented as a sequence of numeric characters. The sign is expressed as + or -.	xs:int
2		XQuery decimal literal	[*sign*] [*integer-part*] . [*fractional-part*] Examples: 12.3 -456. .789	The integer and fractional parts are represented as unsigned integers. At least one part, either the integer part or fractional part, must be specified.	xs:decimal
3		XQuery floating-point literal	*mantissa*{E|e}*exponent* Examples: 1.0E2 .5E+67	Either an integer literal or a decimal literal as the mantissa, followed by a one- to three-digit integer literal as the exponent. The exponent represents a power of 10.	xs:double

No.	XQuery literal	Result		XQuery data type to interpret
		Format	Explanation	

4	XQuery character string literal	{"*character-string*" | '*character-string*'}
		Examples:
		`"HiRDB"`
		`'HITACHI'`
		`'''06.9.13'`
			A character string is represented as a sequence of characters.
			To use a double quotation mark in a string that is itself enclosed in double quotation marks, specify two consecutive double quotation marks.
			To use a single quotation mark in a character string, specify two consecutive single quotation marks.
			The following character strings are used in place of the corresponding special characters.
			• `&`: `&` (ampersand)
			• `<`: `<` (less than operator)
			• `>`: `>` (greater than operator)
			• `&quot;`: `"` (double quotation mark)
			• `&apos;`: `'` (single quotation mark)
			`&` cannot appear by itself.
				xs:string

#

If you use integer literal notation to specify a literal that is beyond the range of XQuery integer literal values, HiRDB assumes a decimal point to the right of the literal and interprets it as a decimal literal.

(d) XQuery variable reference

■ Function

An XQuery variable reference returns the value of an XQuery variable inside an XQuery expression.

■ Format

XQuery-variable-reference `::=` `$` *XQuery-variable-name*

■ Operands

• XQuery variable name

Specifies the qualified name of the XQuery variable to reference.

Returns the value of a valid XQuery variable with the specified qualified name.

If more than one XQuery variable matches the specification, the value of the XQuery variable specified in the innermost FLWOR expression or XQuery quantified expression is returned.

- Example

The XML element `bookinfo` that has a child XML element `price` whose `typed-value` property is greater than the value of the XQuery variable whose variable name is `PRICE`.

```
/bookinfo[price> $PRICE]
```

(e) Parenthesized XQuery expression

- Function

Returns the result of evaluating an XQuery expression. Used when first evaluating an XQuery expression enclosed in parentheses.

If there is no XQuery expression in the parentheses, an empty XQuery sequence is returned.

- Format

parenthesized-XQuery-expression `::= ( [`*XQuery-expression*`] )`

- Example

The following is an example of a parenthesized XQuery expression.

No.	Example	Result	Explanation
1	`(2 + 4) * 5`	(30)	The `2 + 4` in parentheses is evaluated first. If the `2 + 4` were not in parentheses, the XQuery multiplication expression `4 * 5` would be evaluated first, and the result would be `22`.

(f) Context item expression

- Function

A context item expression returns the XQuery sequence consisting of the context item just after evaluation of the context item expression.

- Format

```
context-item-expression ::= period
period ::= .
```

■ Example

The following is an example of a context item expression.

No.	Example	Result	Explanation
1	`(1 to 20)[. mod 5 eq 0]`	(5, 10, 15, 20)	First, the XQuery sequence consisting of the integer values from `1` to `20` is generated by the XQuery range expression. The XQuery predicate following the range expression is then evaluated, with each XQuery item in the generated XQuery sequence as a context item. The result is the XQuery sequence consisting of the XQuery items for which this evaluation result is `TRUE`.

(g) XQuery function invocation

■ Function

An XQuery function invocation calls an XQuery function and returns the evaluation result.

■ Format

```
function-invocation ::= XQuery-function-name ( [argument [, argument]...] )
argument ::= XQuery-single-expression
```

■ Operands

- *XQuery-function-name*

  Specifies the qualified name of the XQuery function to call.

- *argument* ::= *XQuery-single-expression*

  XQuery-single-expression

  Specifies an XQuery single expression that represents the parameter values for the XQuery function to be called.

  If *argument* includes the comma operator, the highest-level XQuery expression that includes the comma operator must be enclosed in

parentheses.

- Rules

  1. The arguments must correspond to the order in which the parameters were specified.
  2. If the XQuery data types of the arguments for the XQuery function are atomic types, the arguments are evaluated and passed to the function in the following steps:

     (a) Evaluate the XQuery unary expression specified in the argument for the XQuery function and find the XQuery sequence that is the result of the evaluation.

     (b) Atomize the results of evaluating the argument to convert it to an XQuery sequence consisting of atomic values.

     (c) Convert each XQuery item contained in the XQuery sequence as follows:

     • If the XQuery data type of the XQuery item is `xs:untypedAtomic` type or `xs:string` type, and the XQuery data type of the function parameter is a numeric data type:

     Convert to `xs:double` type.

     • Otherwise:

     Convert to the XQuery data type of the function parameter.
  3. If the XQuery data type of the argument for the XQuery function is a non-atomic type, the XQuery unary expression specified in the argument for the XQuery function is evaluated, and the resulting XQuery sequence is passed to the function.
  4. At each step of the conversion of the XQuery data types in rules 2 and 3, an error results if the conversion cannot be performed due to incompatible data types. For details about compatible data types, see *1.15.2(3) XQuery data types*.

- Function invocation rules

  1. If the qualified name and number of arguments match those of the XQuery function, that XQuery function is called. Otherwise an error results.

1.15.7 XQuery comment

An XQuery comment is an explanatory note that has no effect on the results of evaluating XQuery expressions.

Comments are specified in the following format:

XQuery-comment ::= (:[{*XQuery-comment-contents*| *XQuery-comment*}]... :)
XQuery-comment-contents ::= *characters*

1.15.8 XQuery functions

This section describes the XQuery functions provided by HiRDB.

Arguments are described in the format of each XQuery function. They are passed as parameters to that XQuery function according to the XQuery function invocation rules. For details about the rules of XQuery function invocation, see *1.15.6(15)(g) XQuery function invocation*.

XQuery functions include functions defined in XQuery and functions defined in HiRDB.

Functions defined in XQuery use the XML namespace that is bound to the prefix `xs` (`http://www.w3.org/2001/XMLSchema`) or the XML namespace that is bound to the prefix `fn` (`http://www.w3.org/2006/xpath-functions`).

Functions defined in HiRDB use the XML namespace that is bound to the prefix `hi-fn` (`http://www.hitachi.co.jp/Prod/comp/soft1/hirdb/xquery-functions`). When functions defined in HiRDB are called, unless the default XML namespace for XQuery function names is declared as `http://www.hitachi.co.jp/Prod/comp/soft1/hirdb/xquery-functions`, the prefix must be specified.

For details about the constructor function, see *1.15.2(3)(b) Constructor function*.

The following table lists the XQuery functions available in HiRDB.

Table 1-98: XQuery functions provided in HiRDB

Classification	Function	Purpose	XQuery function category
Constructor function	xs:string	Generate a value of type `xs:string`.	XQuery
	xs:decimal	Generate a value of type `xs:decimal`.	XQuery
	xs:int	Generate a value of type `xs:int`.	XQuery
	xs:double	Generate a value of type `xs:double`.	XQuery
	xs:dateTime	Generate a value of type `xs:dateTime`.	XQuery

Classification	Function	Purpose	XQuery function category
	xs:date	Generate a value of type `xs:date`.	XQuery
	xs:time	Generate a value of type `xs:time`.	XQuery
	xs:hexBinary	Generate a value of type `xs:hexBinary`.	XQuery
	xs:boolean	Generate a value of type `xs:boolean`.	XQuery
	xs:untypedAtomic	Generate a value of type `xs:untypedAtomic`.	XQuery
Conversion functions	fn:boolean	Return the result of evaluating the XQuery sequence as a value of the `xs:boolean` type.	XQuery
	fn:data	Atomize an XQuery sequence and return an XQuery sequence consisting of atomic values.	XQuery
	fn:number	Return the result of converting the value specified in the argument to `xs:double` type.	XQuery
	fn:string	Return the character string representation of the value of the XQuery item as an `xs:string` type value.	XQuery
Character string operation functions	fn:compare	Compare two character strings.	XQuery
	fn:concat	Return the result of concatenating two or more atomic values that have been converted to an `xs:string` type value.	XQuery
	fn:contains(function defined in XQuery)	Return whether or not the character string specified in *argument-2* is part of the character string specified in *argument-1*.	XQuery

Classification	Function	Purpose	XQuery function category
	hi-fn:contains(function defined in HiRDB)	Return whether or not the character string value of the node specified in *argument-1* satisfies the full-text search conditions specified in *argument-2*.	HiRDB
	fn:ends-with	Return whether or not the character string specified in *argument-1* ends with the character string specified in *argument-2*.	XQuery
	fn:normalize-space	Return the result of normalizing whitespace in a character string.	XQuery
	fn:starts-with	Return whether or not the character string specified in *argument-1* starts with the character string specified in *argument-2*.	XQuery
	fn:string-length	Return the length (in number of characters) of the character string.	XQuery
	fn:substring	Return a substring of a character string.	XQuery
	fn:substring-after	Return a substring of the character string specified in *argument-1* that begins immediately after the position of the first occurrence of the character string specified in *argument-2*.	XQuery
	fn:substring-before	Return a substring of the character string specified in *argument-1* that ends immediately before the position of the first occurrence of the character string specified in *argument-2*.	XQuery
	fn:translate	Return the result of replacing the characters specified in *argument-2* with the characters specified in *argument-3* in the character string specified in *argument-1*.	XQuery
Math functions	fn:abs	Return the absolute value of a numeric data type value.	XQuery

Classification	Function	Purpose	XQuery function category
	fn:ceiling	Return the smallest integer value that is greater than or equal to an argument.	XQuery
	fn:floor	Return the greatest integer value that is less than or equal to an argument.	XQuery
	fn:round	Return the integer value closest to the value of an argument.	XQuery
Date extraction functions	fn:year-from-dateTime	Return only the year part extracted from the time stamp.	XQuery
	fn:month-from-dateTime	Return only the month part extracted from the time stamp.	XQuery
	fn:day-from-dateTime	Return only the day part extracted from the time stamp.	XQuery
	fn:hours-from-dateTime	Return only the hours part extracted from the time stamp.	XQuery
	fn:minutes-from-dateTime	Return only the minutes part extracted from the time stamp.	XQuery
	fn:seconds-from-dateTime	Return only the seconds part extracted from the time stamp.	XQuery
	fn:year-from-date	Return only the year part extracted from the date.	XQuery
	fn:month-from-date	Return only the month part extracted from the date.	XQuery
	fn:day-from-date	Return only the day part extracted from the date.	XQuery
	fn:hours-from-time	Return only the hours part extracted from the time.	XQuery
	fn:minutes-from-time	Return only the minutes part extracted from the time.	XQuery
	fn:seconds-from-time	Return only the seconds part extracted from the time.	XQuery
Boolean functions	fn:false	Return `FALSE`.	XQuery

Classification	Function	Purpose	XQuery function category
	fn:not	Convert an argument to the `xs:boolean` type and return the negation of its value.	XQuery
	fn:true	Return `TRUE`.	XQuery
XQuery sequence aggregation functions	fn:count	Return the number of XQuery items in an XQuery sequence.	XQuery
	fn:distinct-values	Return an XQuery sequence consisting of atomic values with duplicates removed.	XQuery
	fn:max	Return the largest value of the atomic values that make up an XQuery sequence.	XQuery
	fn:min	Return the smallest value of the atomic values that make up an XQuery sequence.	XQuery
	fn:sum	Return the sum of the atomic values that make up an XQuery sequence.	XQuery
XQuery sequence operation functions	fn:deep-equal	Determine whether two XQuery sequences are deep-equal.	XQuery
	fn:index-of	Return the positions in the specified XQuery sequence of all occurrences of the XQuery items that match the specified atomic value.	XQuery
	fn:insert-before	Return the XQuery sequence that results from inserting an XQuery item before the specified position in an XQuery sequence.	XQuery
	fn:remove	Return the XQuery sequence that results from removing the XQuery item at the specified position in the XQuery sequence.	XQuery
	fn:reverse	Return the XQuery sequence that results from sorting the XQuery items in an XQuery sequence in reverse order.	XQuery

Classification	Function	Purpose	XQuery function category
	fn:subsequence	Return an XQuery subsequence of an XQuery sequence.	XQuery
Context item functions	fn:last	Return the context size.	XQuery
	fn:position	Return the context position.	XQuery
Node functions	fn:local-name	Return the local name of a node.	XQuery
	fn:name	Return the qualified name of a node.	XQuery
	fn:namespace-uri	Return the XML namespace URI of the qualified name of a node.	XQuery

Legend:

XQuery: Function defined in XQuery

HiRDB: Function defined in HiRDB

(1) abs

(a) Function

Returns the absolute value of a numeric data type value.

(b) Format

```
fn:abs ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-99: fn:abs function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	num	Numeric data type (including empty XQuery sequences)	Input numeric value

2. The result is an empty XQuery sequence or a value of one of the following

numeric data types:

Table 1-100: XQuery data type of the result of the fn:abs function

No.	XQuery data type of the value of the parameter num	XQuery data type of the result
1	xs:double	xs:double
2	xs:decimal	xs:decimal
3	xs:int	xs:int

An error results if the absolute value falls outside the range of the resulting XQuery data type. For example, if `fn:abs(-2147483648)` is specified, an error results because the absolute value is `2147483648`, which is outside the range for `xs:int`.

3. If the value of the parameter `num` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. If the value of the parameter `num` is a numeric data type value, the absolute value of the value of the parameter `num` is returned.
5. If the value of the parameter `num` is the `xs:double` type `NaN` (not a number), `NaN` is returned.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (XQuery data type in parentheses)	Explanation
1	`fn:abs(10.5)`	10.5 (xs:decimal)	Returns `10.5`, the absolute value of `10.5`.
2	`fn:abs(-10.5)`	10.5 (xs:decimal)	Returns `10.5`, the absolute value of `-10.5`.

(2) boolean

(a) Function

Returns the result of evaluating an XQuery sequence as an `xs:boolean` type value.

(b) Format

```
fn:boolean ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-101: fn:boolean function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	seq	XQuery sequence of zero or more arbitrary XQuery items	Input XQuery sequence

2. The result is an `xs:boolean` type value.
3. Returns the result indicated in the following table depending on the XQuery sequence that is specified by the parameter `seq`.

Table 1-102: Results of the fn:boolean function

No.	Value of the parameter seq	Result (xs:boolean type)
1	Empty XQuery sequence	`FALSE`
2	XQuery sequence in which the first XQuery item is a node	`TRUE`
3	XQuery sequence consisting of one `xs:boolean` type atomic value	Atomic value of the XQuery sequence *argument*
4	XQuery sequence consisting of one `xs:string` type or `xs:untypedAtomic` type atomic value	`FALSE` if the atomic value of the XQuery sequence *argument* has length 0, otherwise `TRUE`.
5	XQuery sequence consisting of one numeric data type atomic value	`FALSE` if the atomic value of the XQuery sequence *argument* is `0` or `NaN` (not a number), otherwise `TRUE`.
6	Other than the above	Error

(d) Example

Below are examples of function invocations and their results when the XQuery variable `book` indicates the node representing the elements shown below.

Element represented by the XQuery variable `book`:

```
<bookinfo book_id="452469630">
  <title>Relational Databases Explained</title>
  <author>Jeff Jones</author>
  <author>Bob Adams</author>
</bookinfo>
```

Examples of function invocations and their results:

No.	Function invocation	Result (xs:boolean type)	Explanation
1	`$book/fn:boolean(author[text() = "Mark Davis"])`	`FALSE`	Returns `FALSE` because the value of the parameter `seq` is an empty XQuery sequence.
2	`fn:boolean($book)`	`TRUE`	Returns `TRUE` because the value of the parameter `seq` is an XQuery sequence consisting of one element node `bookinfo`.
3	`$book/fn:boolean(@book_id = 452469630)`	`TRUE`	Returns `TRUE` because the value of the parameter `seq` is an XQuery sequence consisting of one `xs:boolean` type value.
4	`$book/fn:boolean(title/ fn:string(text()))`	`TRUE`	Returns `TRUE` because the value of the parameter `seq` is an XQuery sequence consisting of one `xs:string` type value of length 1 or greater.
5	`$book/fn:boolean(@book_id)`	`TRUE`	Returns `TRUE` because the value of the parameter `seq` is an XQuery sequence consisting of one numeric data value.
6	`$book/fn:boolean(author/ fn:string(text()))`	Error	Error because the value of the parameter `seq` is an XQuery sequence consisting of two `xs:string` type values.

(3) ceiling

(a) Function

Returns the smallest integer value that is greater than or equal to an argument.

(b) Format

```
fn:ceiling ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-103: fn:ceiling function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	num	Numeric data type (including empty XQuery sequences)	Input numeric value

2. The result is an empty XQuery sequence, or a value of one of the numeric data types below.

Table 1-104: XQuery data type of the result of the fn:ceiling function

No.	XQuery data type of the value of the parameter num	XQuery data type of the result
1	xs:double	xs:double
2	xs:decimal	xs:decimal
3	xs:int	xs:int

3. If the value of the parameter `num` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. The following table describes what is returned when the parameter `num` is `xs:double` type and it's value is one of the following special values.

Table 1-105: fn:ceiling function special values result

No.	Value of the parameter num	Result value
1	Positive 0	Positive 0
2	Negative 0	Negative 0
3	Greater than -1 and less than 0	Negative 0
4	Positive infinity	Positive infinity
5	Negative infinity	Negative infinity
6	`NaN` (not a number)	`NaN` (not a number)

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (XQuery data type in parentheses)	Explanation
1	`fn:ceiling(10.5)`	11. (xs:decimal)	Returns `11`, which is the integer value that is greater than or equal to `10.5`.
2	`fn:ceiling(-10.5)`	-10. (xs:decimal)	Returns `-10`, which is the integer value that is greater than or equal to `-10.5`.

(4) compare

(a) Function

Returns the result of comparing two character strings.

(b) Format

```
fn:compare ( argument-1, argument-2 )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-106: fn:compare function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument-1*	str1	`xs:string` type (including empty XQuery sequences)	Character string to compare
2	*argument-2*	str2	`xs:string` type (including empty XQuery sequences)	Character string to compare

2. The result is an empty XQuery sequence or an `xs:int` type value.
3. If the value of either parameter `str1` or `str2` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. If parameters `str1` and `str2` are `xs:string` type values, the result shown below based on the relationship between the values of parameters `str1` and `str2` is returned.

Table 1-107: Results of the fn:compare function

No.	Relationship between the values of parameters str1 and str2	Result (xs:int type)
1	Value of parameter `str1` is less than the value of parameter `str2` (`str1 lt str2` is `TRUE`)	-1
2	Value of parameter `str1` is equal to the value of parameter `str2` (`str1 eq str2` is `TRUE`)	0
3	Value of parameter `str1` is greater than the value of parameter `str2` (`str1 gt str2` is `TRUE`)	1

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (xs:int type)	Explanation
1	`fn:compare("abcde", "abcdef")`	-1	Returns `-1` because `"abcde"` is less than `"abcdef"`.
2	`fn:compare("abcde", "abcde")`	0	Returns `0` because the values of parameters `str1` and `str2` are both `"abcde"`.
3	`fn:compare("abcde", "abade")`	1	Returns `1` because `"abcde"` is greater than `"abade"`.

(5) concat

(a) Function

Returns the result of concatenating two or more atomic values that have been converted to an `xs:string` type value.

(b) Format

```
fn:concat ( argument, argument [, argument]... )
```

(c) Rules

1. This function takes any number (two or more) of the parameter shown below.

Table 1-108: fn:concat function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	val	Any atomic type (including empty XQuery sequences)	Atomic value to concatenate

2. The result is an empty XQuery sequence or an `xs:string` type value.
3. If the value of every parameter is an empty XQuery sequence, an empty XQuery sequence is returned.
4. If at least one parameter with atomic value is included, those values are converted to `xs:string` type values, and an `xs:string` type value that is the result of concatenating them in the specified order is returned. In this case, the value of any parameter that is an empty XQuery sequence is treated as a character string of length zero.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (xs:string type)	Explanation
1	`fn:concat("abc","defg")`	"abcdefg"	Returns the character string `"abcdefg"`, which is the concatenation of `"abc"` and `"defg"`.
2	`fn:concat("hijk", (), "lmn")`	"hijklmn"	Returns the character string `"hijklmn"`, which is the concatenation of `"hijk"`, `""`, and `"lmn"`, because an empty XQuery sequence is treated as a character string of length zero.
3	`fn:concat((),(),())`	Empty XQuery sequence	Returns an empty XQuery sequence because the value of every parameter is an empty XQuery sequence.

(6) contains (Function defined in XQuery)

(a) Function

Returns a value indicating whether the character string specified in *argument-2* is part of the character string specified in *argument-1*.

(b) Format

```
fn:contains ( argument-1, argument-2 )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-109: fn:contains function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument-1*	str1	`xs:string` type (including empty XQuery sequences)	Character string in which to find the substring
2	*argument-2*	str2	`xs:string` type (including empty XQuery sequences)	Substring to find within the character string in parameter `str1`

2. The result is an `xs:boolean` type value.
3. Returns `FALSE` if the value of parameter `str1` is an empty XQuery sequence or a character string of length zero, and the value of parameter `str2` is a character string of length 1 or greater.
4. If the value of parameter `str2` is an empty XQuery sequence or a character string of length zero, `TRUE` is returned regardless of the value of parameter `str1`.
5. When the values of parameters `str1` and `str2` are character strings of length 1 or greater, `TRUE` is returned if the character string in parameter `str2` is found in the character string in parameter `str1`, and returns `FALSE` if it is not found.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (xs:boolean type)	Explanation
1	`fn:contains("abcdefg", "cde")`	`TRUE`	Returns `TRUE` because `"cde"` is part of `"abcdefg"`.
2	`fn:contains("abcdefg", "cdf")`	`FALSE`	Returns `FALSE` because `"cdf"` is not part of `"abcdefg"`.

No.	Function invocation	Result (xs:boolean type)	Explanation
3	`fn:contains((), "")`	`TRUE`	Returns `TRUE` because the value of parameter `str2` is a character string of length zero.
4	`fn:contains((), "a")`	`FALSE`	Returns `FALSE` because the value of parameter `str1` is an empty XQuery sequence, and the value of parameter `str2` is a character string of length 1 or greater.
5	`fn:contains("abcdefg", ())`	`TRUE`	Returns `TRUE` because the value of parameter `str2` is an empty XQuery sequence.

(7) contains (function defined in HiRDB)

(a) Function

Returns a value indicating whether the character string value of the node specified in *argument-1* satisfies the full-text search conditions specified in *argument-2*.

(b) Format

```
hi-fn:contains ( argument-1, argument-2 )
```

(c) Rules

1. The following table lists the parameters of this function.

   Table 1-110: hi-fn:contains function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument-1*	seq	XQuery sequence with zero or more nodes	Nodes to search
2	*argument-2*	str	`xs:string` type	Full-text search conditions character string

2. The result is an `xs:boolean` type value.
3. This function can be used only in the `XMLEXISTS` predicate.

4. When this function is used, an XML type full-text search index must be defined on the column specified in the XML query context item. In addition, version 08-04 or later of the HiRDB XML Extension must be used.
5. A substructure path that conforms to the usage conditions of the XML type full-text search index must be specified in parameter seq. An error results if they do not conform, or if the index alone cannot be used to evaluate the criteria. For details about the usage conditions for the XML type full-text search index, see the *HiRDB Version 9 Installation and Design Guide*.
6. The full-text search conditions are specified as an XQuery character string literal in parameter str. The search conditions shown below can be specified as text search conditions. An error results if the value of the parameter str is a character string of length zero, or if it does not follow the format of the text search conditions. For details about how to specify text search conditions, see the manual *HiRDB Version 9 XML Extension*.

Table 1-111: Search conditions that can be specified as text search conditions

No.	Search conditions	Summary	Example

1	Simple character string conditions	Search for character strings that match a search string that contains special characters, or any of the following wildcards:
		• `*`
		Equivalent to any character string of zero or more characters.
		• `?`
		Equivalent to any one character.
		• `|`
		Matches the beginning or end of the construction.
		• `\`
		Turn off the wildcard or special character meaning.
			• Character strings that contain any character string of zero or more characters between `network` and `computer`
			`"network*computer"`
			• Character strings that contain any one character between `network` and `computer`
			`"network?computer"`
			• Character strings that start with `network`
			`"|network"`

No.	Search conditions	Summary	Example
		The character string to search for is enclosed in double quotation marks (`"`). However, if an XQuery character string literal is to be enclosed in double quotation marks, either enclose it in `&quot;` or code two consecutive double quotation marks.	

2	Excluded character search conditions	Search while excluding a particular character in the middle or at the ends of the search string from simple character string conditions. A ^ is specified before the character to exclude.	• From character strings that contain `plan` followed by any one character, exclude any character strings that contain `plant`
			`"plan^t"`
			• From character strings that contain any one character between `do` and `key`, exclude any character strings that contain `donkey`
			`"do^nkey"`
			• From character strings that contain `round` preceded by any one character, exclude any character strings that contain `around`
			`"^around"`

3	NOT conditions	Search for character strings that do not contain the search string.	• Character strings that do not contain `network` `NOT ("network")`

4	AND/OR conditions	Specify multiple search conditions linked with AND or OR.	• Character strings that contain both `network` and `computer`
			`"network" AND "computer"`
			Character strings that contain either `network` or `computer`
			`"network" OR "computer"`

No.	Search conditions	Summary	Example

5	Proximity conditions	Specify the number of characters (distance) between two character strings matching two search conditions.	• Character strings with no more than 20 characters between `new` and `technology` (no particular order)
			`PROXIMITY("new", <=20`[#1]`,`
			`CHARACTERS`[#2]`,ANY_ORDER`[#3]`,"technology`[#4]`")`

6	Synonym expansion conditions	Automatically expand the search string to include synonyms based on the definitions in a synonym dictionary. For details about how to create and register a synonym dictionary, see the manual *HiRDB Version 9 XML Extension*.	Character strings that contain a synonym of `COMPUTER` (including different alphabetical and one- and two-byte representations) SYNONYM(USR01[#4], "COMPUTER", "AE[#5]")

7	Representation expansion conditions	Automatically expand the search string to include different representations based on rules.	• Character strings that include the character string `COMPUTER` represented in various alphabetical and one- and two-byte representations
			`SOUNDEX_EXP("COMPUTER", "AE"`[#5]`)`
			• Character strings that include any character string representing a derivation of the English word `sing`
			`SOUNDEX_EXP("sing", "S"`[#6]`)`

#1

Means the search is performed with the distance less than or equal to 20.

#2

Means that the unit of distance is characters.

#3

Means no particular order.

#4

Refers to the name of the synonym dictionary.

#5

`A` means different alphabetic representations, `E` means different one- and two-byte representations.

#6

Refers to representations of the derivations of English words.

7. Returns `FALSE` if the value of the parameter `seq` is an empty XQuery sequence.
8. Returns `TRUE` if at least one character string value among the character string values of the nodes in the XQuery sequence in parameter `seq` includes the character string represented by the text search conditions in parameter `str`. Returns `FALSE` if none of the character string values contains the character string represented by the text search conditions.

(d) Example

In the examples of function invocations and their results given below, the XML document shown below is represented by values in columns (on which an `XML` type full-text search index is defined) specified in XML query context items.

XML document represented by the values stored in the columns specified in XML query context items:

```
<bookinfo book_id="452469630">
  <category>database</category>
  <title>Relational Databases Explained</title>
  <author>Jeff Jones</author>
  <author>Bob Adams</author>
  <description>Explains the structure of the RDBMS (Relational Database Management
System) based on the concept of the relational model.</description>
</bookinfo>
```

Examples of function invocations and their results:

No.	Function invocation	Result (xs:boolean type)	Explanation
1	`/bookinfo/title[ hi-fn:contains(text(), "SYNONYM(USR01, &quot;DB&quot;)")]`	`TRUE`	Suppose `"database"` and `"DB"` are registered as synonyms in synonym dictionary `USR01`. Returns `TRUE` because in the value of the parameter `seq` there is a node containing `"database"`, which is a synonym of `"DB"`.

No.	Function invocation	Result (xs:boolean type)	Explanation
2	`/bookinfo/description[ hi-fn:contains(text(), "SOUNDEX_EXP(&quot;DATABASE &quot;, &quot;AE&quot;) AND &quot;Explain&quot;")]`	`TRUE`	Returns `TRUE` because the value of the parameter `seq` includes a node that contains both `"Explain"` and `"Database"`, which is an alphabetic variant of `"DATABASE"`.
3	`/bookinfo/author[ hi-fn:contains(text(), "&quot;Smith&quot;")]`	`FALSE`	Returns `FALSE` because there is no node containing `"Smith"` in the value of the parameter `seq`.
4	`/bookinfo/contents[ hi-fn:contains(text(), "SYNONYM(USR01, &quot;DB&quot;)")]`	`FALSE`	Returns `FALSE` because the parameter `seq` is an empty XQuery sequence.

(8) count

(a) Function

Returns the number of XQuery items in an XQuery sequence.

(b) Format

```
fn:count ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-112: fn:count function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	seq	XQuery sequence of zero or more arbitrary XQuery items	Input XQuery sequence

2. The result is an `xs:int` type value.

(d) Example

In the examples of function invocations and their results given below, the XQuery

variable book indicates the node representing the elements shown below.

Element represented by the XQuery variable book:

```
<bookinfo book_id="452469630">
  <title>Relational Databases Explained</title>
  <author>Jeff Jones</author>
  <author>Bob Adams</author>
</bookinfo>
```

Examples of function invocations and their results:

No.	Function invocation	Result (xs:int type)	Explanation
1	fn:count($book)	1	Returns 1 because the value of the parameter seq is an XQuery sequence consisting of one bookinfo element node.
2	fn:count($book/author)	2	Returns 2 because the value of the parameter seq is an XQuery sequence consisting of two author element nodes that are children of the bookinfo element node.
3	fn:count($book/author[.="Mark Davis"])	0	Returns 0 because the value of the parameter seq is an empty XQuery sequence.

(9) data

(a) Function

Atomizes an XQuery sequence and returns an XQuery sequence consisting of atomic values.

(b) Format

```
fn:data ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-113: fn:data function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	seq	XQuery sequence of zero or more arbitrary XQuery items	Input XQuery sequence

2. The result is an empty XQuery sequence or an XQuery sequence consisting of atomic values.
3. If the value of the parameter `seq` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. If the value of the parameter `seq` is not an empty XQuery sequence, the XQuery sequence consisting of the results of applying the following rules to each XQuery item in that XQuery sequence is returned.
   - If the XQuery item is an atomic value, that value will be the result.
   - If the XQuery item is a node, the result will be the value listed in the following table, depending on the kind of node.

Table 1-114: Results when the XQuery item is a node

No.	Kind of node	Result
1	Document node	Value of the `typed-value` property
2	Element node	Value of the `typed-value` property
3	Attribute node	Value of the `typed-value` property
4	Processing instruction node	Empty XQuery sequence
5	Comment node	Empty XQuery sequence
6	Text node	Value of converting the value of the `content` property to `xs:untypedAtomic` type

(d) Example

In the examples of function invocations and their results given below, the XQuery variable `book` indicates the node representing the elements shown below.

Element represented by the XQuery variable `book`:

```
<bookinfo book_id="452469630">
  <title>Relational Databases Explained</title>
  <author>Jeff Jones</author>
  <author>Bob Adams</author>
</bookinfo>
```

Examples of function invocations and their results:

No.	Function invocation	Result	Explanation
1	`fn:data($book/price)`	Empty XQuery sequence	Returns an empty XQuery sequence because the value of the parameter `seq` is an empty XQuery sequence.
2	`fn:data($book/author)`	`("Jeff Jones","Bob Adams")`	Returns the XQuery sequence consisting of the two `typed-value` property values because the parameter `seq` is an XQuery sequence consisting of two author element nodes.

(10) day-from-date

(a) Function

Returns only the day part extracted from the date.

(b) Format

```
fn:day-from-date ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-115: fn:day-from-date function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	dat	`xs:date` type (including empty XQuery sequences)	Date to extract from

2. The result is an empty XQuery sequence or an `xs:int` type value.
3. If the value of the parameter `dat` is an empty XQuery sequence, an empty XQuery sequence is returned.

4. If the value of the parameter `dat` is an `xs:date` type value, the value of the day part as an `xs:int` type is returned.

(d) Example

The following table provides an example of a function invocation and its result:

No.	Function invocation	Result (xs:int type)	Explanation
1	`fn:day-from-date (xs:date("2006-09-26"))`	26	Returns `26`, which is the day part of the value of the parameter `dat`.

(11) day-from-dateTime

(a) Function

Returns only the day part extracted from the time stamp.

(b) Format

```
fn:day-from-dateTime ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-116: fn:day-from-dateTime function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	dtm	`xs:dateTime` type (including empty XQuery sequences)	Time stamp to extract from

2. The result is an empty XQuery sequence or an `xs:int` type value.
3. If the value of the parameter `dtm` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. If the value of the parameter `dtm` is an `xs:dateTime` type value, the value of the day part as an `xs:int` type is returned.

(d) Example

The following table provides an example of a function invocation and its result:

No.	Function invocation	Result (xs:int type)	Explanation
1	`fn:day-from-dateTime(xs:dateTime("2006-09-26T18:44:58.153"))`	26	Returns `26`, which is the day part of the value of the parameter `dtm`.

(12) deep-equal

(a) Function

Determines whether two XQuery sequences are deep-equal (have the same structure and the same values).

(b) Format

```
fn:deep-equal ( argument-1, argument-2 )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-117: fn:deep-equal function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument-1*	seq1	XQuery sequence of zero or more arbitrary XQuery items	XQuery sequence to compare
2	*argument-2*	seq2	XQuery sequence of zero or more arbitrary XQuery items	XQuery sequence to compare

2. The result is an `xs:boolean` type value.
3. If parameters `seq1` and `seq2` are both empty XQuery sequences, the result is `TRUE`.
4. If the number of XQuery items in parameter `seq1` and the number of XQuery items in parameter `seq2` are different, the result is `FALSE`.
5. If all of the XQuery items that make up parameter `seq1` are in the same position and all are deep-equal to the XQuery items that make up parameter `seq2`, the result is `TRUE`, but if there is even one XQuery item for which this not true, it is `FALSE`.

   Whether two XQuery items are deep-equal is determined by the rules shown below.

Let item1 and item2 be the two XQuery items.

Table 1-118: XQuery item deep-equal rules

No.	Category of item1	Category of item2	Rules
1	Atomic value	Atomic value	TRUE only if all of the following conditions are met: 1. The XQuery data types of item1 and item1 can be compared with the eq operator. 2. (item1 eq item2) is TRUE, or item1 and item2 are both xs:double type with the value NaN (not a number).
2		Other than the above	FALSE
3	Document node	Document node	TRUE only if the fn:deep-equal function is TRUE for the XQuery sequence of element and text nodes that are children of item1 and the XQuery sequence of element and text nodes that are children of item2.
4		Other than the above	FALSE
5	Element node	Element node	TRUE only if all of the following conditions are met: 1. The namespaces and node names of the two element nodes match. 2. item1 and item2 have the same number of attribute nodes, and each attribute node of item1 is deep-equal to the corresponding attribute node of item2. In this case, the order of attribute nodes for item1 and the order of attribute nodes for item1 need not be the same. 3. The fn:deep-equal function is TRUE for the XQuery sequence of element node and text node children of item1 and the XQuery sequence of element node and text node children of item2.
6		Other than the above	FALSE
7	Attribute node	Attribute node	TRUE only if all of the following conditions are met: 1. The namespaces and node names of the two attribute nodes match. 2. The typed-value of item1 and the typed-value of item2 are deep-equal.
8		Other than the above	FALSE

No.	Category of item1	Category of item2	Rules
9	Processing instruction node	Processing instruction node	TRUE only if all of the following conditions are met: 1. The processing instruction targets of the two processing instruction nodes match. 2. The contents of item1 and the contents of item2 are deep-equal.
10		Other than the above	FALSE
11	Text node	Text node	TRUE only if the character string values of the two text nodes are deep-equal.
12		Other than the above	FALSE.
13	Comment node	Comment node	TRUE only if the character string values of the two comment nodes are deep-equal.
14		Other than the above	FALSE

(d) Example

In the examples of function invocations and their results given below, the XQuery variable books indicates the node representing the elements shown below.

Element represented by the XQuery variable books:

```
<booklist>
  <bookinfo book_id="452469630">
    <title>Relational Databases Explained</title>
    <author>Bob Adams</author>
    <author>Jeff Jones</author>
  </bookinfo>
  <bookinfo book_id="45241350">
    <title>Relational Databases Explained Vol. 2</title>
    <author>Bob Adams</author>
    <author>Jeff Jones</author>
  </bookinfo>
</booklist>
```

Examples of function invocations and their results:

No.	Function invocation	Result (xs:boolean type)	Explanation
1	`fn:deep-equal( $books/bookinfo[1]/title, "Relational Databases Explained")`	`FALSE`	`FALSE` because the parameter `seq1` is an XQuery sequence of one element node, and parameter `seq2` is an XQuery sequence of a single atomic value.
2	`fn:deep-equal($books/bookinfo[1], $books/bookinfo[2])`	`FALSE`	`FALSE` because although parameters `seq1` and `seq2` are both XQuery sequences of one `bookinfo` element node, their `book_id` attribute nodes are not deep-equal and their child `title` elements are not deep-equal.
3	`fn:deep-equal($books/bookinfo[1]/author, $books/bookinfo[2]/author)`	`TRUE`	`TRUE` because parameters `seq1` and `seq2` are both XQuery sequences with two author element nodes, and those author element nodes are deep-equal.

(13) distinct-values

(a) Function

Returns an XQuery sequence consisting of atomic values with duplicates removed.

(b) Format

```
fn:distinct-values ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-119: fn:distinct-values function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	seq	XQuery sequence of zero or more values of any atomic type	Input XQuery sequence

2. The result is an empty XQuery sequence or an XQuery sequence consisting of atomic values only.

3. If the value of the parameter `seq` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. To decide whether atomic values are duplicates, the values are compared using the `eq` operator. If values cannot be compared to each other with the `eq` operator or their comparison result is `FALSE`, they are not considered duplicates. If the comparison result is `TRUE`, they are considered duplicates, and one of them is removed.
5. If the XQuery sequence that is the value of the parameter `seq` contains more than one `xs:double` type `NaN` (not a number), all but one of them will be removed.
6. The order of the XQuery items in the resulting XQuery sequence is not guaranteed to match the order of the XQuery items in the XQuery sequence that is the value of the parameter `seq`.

(d) Example

The following table provides an example of a function invocation and its result:

No.	Function invocation	Result	Explanation
1	`fn:distinct-values(("a", 1, "A", 0, 1))`	("a", 1, "A", 0) Order may vary.	The duplicate 1 is removed from the XQuery sequence that is the value of the parameter `seq`.

(14) ends-with

(a) Function

Returns a value indicating whether the character string specified in *argument-1* ends with the character string specified in *argument-2*.

(b) Format

```
fn:ends-with ( argument-1, argument-2 )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-120: fn:ends-with function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument-1*	str1	`xs:string` type (including empty XQuery sequences)	Character string to be checked to see whether it ends with the substring
2	*argument-2*	str2	`xs:string` type (including empty XQuery sequences)	Substring to check against the end of the character string in parameter `str1`

2. The result is an `xs:boolean` type value.
3. Returns `FALSE` if the value of parameter `str1` is an empty XQuery sequence or a character string of length zero and the value of parameter `str2` is a character string of length 1 or greater.
4. If the value of parameter `str2` is an empty XQuery sequence or a character string of length zero, `TRUE` is returned regardless of the value of parameter `str1`.
5. Returns `TRUE` if the values of parameters `str1` and `str2` are character strings of length 1 or greater and the character string in parameter `str1` ends with the character string in parameter `str2`. Otherwise returns `FALSE`.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (xs:boolean type)	Explanation
1	`fn:ends-with("abcdefg", "efg")`	`TRUE`	Returns `TRUE` because `"abcdefg"` ends with `"efg"`.
2	`fn:ends-with("abcdefg", "abc")`	`FALSE`	Returns `FALSE` because `"abcdefg"` does not end with `"abc"`.
3	`fn:ends-with((), "")`	`TRUE`	Returns `TRUE` because the value of parameter `str2` is a character string of length zero.
4	`fn:ends-with((), "a")`	`FALSE`	Returns `FALSE` because the value of parameter `str1` is an empty XQuery sequence, and the value of parameter `str2` is a character string of length 1 or greater.

No.	Function invocation	Result (xs:boolean type)	Explanation
5	`fn:ends-with("abcdefg", ())`	`TRUE`	Returns `TRUE` because the value of parameter `str2` is an empty XQuery sequence.

(15) false

(a) Function

Returns `FALSE`.

(b) Format

```
fn:false ( )
```

(c) Rules

1. This function has no parameters.
2. The result is the `xs:boolean` type value `FALSE`.

(16) floor

(a) Function

Returns the largest integer value that is less than or equal to an argument.

(b) Format

```
fn:floor ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-121: fn:floor function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	num	Numeric data type (including empty XQuery sequences)	Input numeric value

2. The result is an empty XQuery sequence, or a value of one of the numeric data types listed below.

Table 1-122: XQuery data type of the result of the fn:floor function

No.	XQuery data type of the value of the parameter num	XQuery data type of the result
1	xs:double	xs:double
2	xs:decimal	xs:decimal
3	xs:int	xs:int

3. If the value of the parameter `num` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. The following table describes what happens when the data type of the parameter `num` is `xs:double` type and the value is one of the special values shown below:

Table 1-123: fn:floor function special value results

No.	Value of the parameter num	Result value
1	Positive 0	Positive 0
2	Negative 0	Negative 0
3	Positive infinity	Positive infinity
4	Negative infinity	Negative infinity
5	`NaN` (not a number)	`NaN` (not a number)

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (XQuery data type in parentheses)	Explanation
1	`fn:floor(10.5)`	10. (xs:decimal)	Returns `10`, which is the largest integer value that is less than or equal to `10.5`.
2	`fn:floor(-10.5)`	-11. (xs:decimal)	Returns `-11`, which is the largest integer value that is less than or equal to `-10.5`.

(17) hours-from-dateTime

(a) Function

Returns only the hours part extracted from the time stamp.

(b) Format

```
fn:hours-from-dateTime ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-124: fn:hours-from-dateTime function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	dtm	`xs:dateTime` type (including empty XQuery sequences)	Time stamp to extract from

2. The result is an empty XQuery sequence or an `xs:int` type value.
3. If the value of the parameter `dtm` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. If the value of parameter `dtm` is an `xs:dateTime` type value, the hours part is returned as an `xs:int` type value.

(d) Example

The following table provides an example of a function invocation and its result:

No.	Function invocation	Result (xs:int type)	Explanation
1	`fn:hours-from-dateTime(xs:dateTime("2006-09-26T18:44:58.153"))`	18	Returns `18`, which is the value of the hours part of parameter `dtm`.

(18) hours-from-time

(a) Function

Returns only the hours part extracted from the time.

(b) Format

```
fn:hours-from-time ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-125: fn:hours-from-time function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	tim	`xs:time` type (including empty XQuery sequences)	Time to extract from

2. The result is an empty XQuery sequence or an `xs:int` type value.
3. If the value of the parameter `tim` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. If the value of the parameter `tim` is an `xs:time` type value, the hours part is returned as an `xs:int` type value.

(d) Example

The following table provides an example of a function invocation and its result:

No.	Function invocation	Result (xs:int type)	Explanation
1	`fn:hours-from-time(xs:time("18:44:58"))`	18	Returns `18`, which is the hours part of the value of parameter `tim`.

(19) index-of

(a) Function

Returns the positions in the specified XQuery sequence of all occurrences of the XQuery items that match the specified atomic value.

(b) Format

```
fn:index-of ( argument-1, argument-2 )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-126: fn:index-of function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument-1*	seq	XQuery sequence of zero or more values of any atomic type	XQuery sequence to be searched
2	*argument-2*	val	Any atomic type	Atomic value to search for

2. The result is an XQuery sequence consisting of zero or more `xs:int` type values.
3. If the parameter `seq` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. If an XQuery item matching the value of parameter *val* is present in parameter `seq`, its position is returned as an `xs:int` type value. The first position is 1.
5. To decide whether atomic values match, the values are compared using the `eq` operator. If values cannot be compared to each other with the `eq` operator or their comparison result is `FALSE`, they are not considered matches.
6. If multiple matching XQuery items are present in the XQuery sequence in parameter `seq`, an XQuery sequence is returned, consisting of the `xs:int` type values indicating their positions, arranged in ascending order.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result	Explanation
1	`fn:index-of((10, 20, 30, 40), 35)`	Empty XQuery sequence	Returns an empty XQuery sequence because there is no matching XQuery item in the XQuery sequence in parameter `seq`.
2	`fn:index-of((10, 20, 30, 40), 30)`	(3)	Returns the XQuery sequence consisting of the `xs:int` type value `3`, because the third XQuery item in the XQuery sequence in parameter `seq` is a match.
3	`fn:index-of((10, 20, 30, 40,30,50), 30)`	(3,5)	Returns the XQuery sequence consisting of the `xs:int` type values `3` and `5`, because the third and fifth XQuery items in the XQuery sequence in parameter `seq` are matches.

(20) insert-before

(a) Function

Returns the XQuery sequence that results from inserting an XQuery item before the specified position in an XQuery sequence.

(b) Format

```
fn:insert-before ( argument-1, argument-2, argument-3 )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-127: fn:insert-before function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument-1*	seq1	XQuery sequence of zero or more arbitrary XQuery items	XQuery sequence into which you are inserting
2	*argument-2*	pos	`xs:int`	Insertion position
3	*argument-3*	seq2	XQuery sequence of zero or more arbitrary XQuery items	XQuery sequence containing the XQuery items to be insert

2. The result is an XQuery sequence consisting of zero or more XQuery items.
3. If the value of parameter `seq1` is an empty XQuery sequence, and if the value of parameter `seq2` is also an empty XQuery sequence, an empty XQuery sequence is returned; otherwise, the XQuery sequence that is the value of parameter `seq2` is returned.
4. If the value of parameter `seq2` is an empty XQuery sequence, the XQuery sequence that is the value of parameter `seq1` is returned.
5. The insertion position is specified in parameter `pos`. Specify `1` to insert at the beginning. If the value of parameter `pos` is less than `1`, the insertion position is treated as `1`. If the value of parameter `pos` is greater than the number of XQuery items in the XQuery sequence that is the value of parameter `seq1`, the insertion position is treated as (the number of XQuery items in the XQuery sequence that is the value of parameter `seq1` + 1).
6. The resulting XQuery sequence will consist of, in order, the XQuery items up to

one before the insertion position in the XQuery sequence that is the value of parameter `seq1`, the XQuery items in the XQuery sequence that is the value of parameter `seq3`, and the XQuery items after the insertion position in the XQuery sequence that is the value of parameter `seq1`.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result	Explanation
1	`fn:insert-before(("a", "b", "c"), 0, "z")`	("z", "a", "b", "c")	Inserts `"z"` at the beginning because the value of parameter `pos` is `0`.
2	`fn:insert-before(("a", "b", "c"), 1, "z")`	("z", "a", "b", "c")	Inserts `"z"` at the beginning because the value of parameter `pos` is `1`.
3	`fn:insert-before(("a", "b", "c"), 3, "z")`	("a", "b", "z", "c")	Inserts `"z"` before the third XQuery item `"c"` because the value of parameter `pos` is `3`.
4	`fn:insert-before(("a", "b", "c"), 4, "z")`	("a", "b", "c", "z")	Inserts `"z"` at the end because the value of parameter `pos` is `4`.

(21) last

(a) Function

Returns the context size.

(b) Format

```
fn:last ( )
```

(c) Rules

1. This function has no parameters.
2. The result is an `xs:int` type value.
3. Returns the number of context items as evaluated at the time this function was invoked.

(d) Example

The following table provides an example of a function invocation and its result:

No.	Function invocation	Result	Explanation
1	`("a", "b", "c")[fn:last() ]`	"c" (xs:string) Result of `fn:last` function is `3`	`fn:last` function returns `3` because the number of context items is `3`. Therefore, the third context item `"c"` will be the result.

(22) local-name

(a) Function

Returns the local name of a node.

(b) Format

```
fn:local-name ( [argument] )
```

(c) Rules

1. This function takes zero or one of the parameter listed in the following table.

   Table 1-128: fn:local-name function parameter

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	node	Any node (including empty XQuery sequences)	Node whose local name is to be obtained

2. The result is an `xs:string` type value.
3. If no argument is specified, the local name of the context item node as evaluated at the time this function was invoked is returned. In other words, the result is the same as specifying `fn:local-name(.)`. An error results if the context item is not a node.
4. If the value of the parameter `node` is an empty XQuery sequence, a character string of length zero is returned.
5. If the value of the parameter `node` is a node that does not have a name (document node, comment node, text node), a character string of length zero is returned.
6. If the value of the parameter `node` is a node that has a name (element node, attribute node, processing instruction node), the local name of that node is returned.

(d) Example

In the examples of function invocations and their results given below, the XQuery variable `book` indicates the node representing the elements shown below.

Element represented by the XQuery variable `book`:

```
<bookinfo:bookinfo book_id="452469630">
  <title>Relational Databases Explained</title>
  <author>Jeff Jones</author>
  <author>Bob Adams</author>
</bookinfo:bookinfo>
```

Examples of function invocations and their results:

No.	Function invocation	Result (xs:string type)	Explanation
1	`$book/fn:local-name()`	`"bookinfo"`	Returns the local name of the context item, which is the `bookinfo:bookinfo` element node, because no argument is specified.
2	`$book/fn:local-name(./title)`	`"title"`	Returns the local name `title` because the value of the parameter `node` is the `title` element node.
3	`$book/title/fn:local-name(./text())`	`""` (a character string of length zero)	Returns a character string of length zero because the value of the parameter `node` is a text node.

(23) max

(a) Function

Returns the largest value of the atomic values that make up an XQuery sequence.

(b) Format

```
fn:max ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-129: fn:max function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	seq	XQuery sequence of zero or more values of any atomic type	Input XQuery sequence

2. The result is an empty XQuery sequence or an atomic type value.
3. If the value of the parameter `seq` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. To compare the magnitudes of atomic values, the values are compared with the `ge` operator. An error results if the XQuery sequence that is the value of the parameter `seq` contains a mix of XQuery data types that cannot be compared with the `ge` operator.
5. To compare numeric data types, all the values are converted to a common numeric data type and then compared. For this reason, if the XQuery sequence in parameter `seq` consists of only numeric data type values, the result will be a value of the XQuery data type shown below.

Table 1-130: XQuery data type of the result of the fn:max function

No.	XQuery data type of the values in the XQuery sequence that is the value of the parameter seq			XQuery data type of the result
	xs:double	xs:decimal	xs:int	
1	Y	--	--	xs:double
2	N	Y	--	xs:decimal
3	N	N	--	xs:int

Legend:

Y: Includes.

N: Does not include.

--: Does not affect the data type of the result.

6. If the XQuery sequence that is the value of the parameter `seq` contains an `xs:double` type value that is `NaN` (not a number), the result will be `NaN`.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (XQuery data type in parentheses)	Explanation
1	`fn:max((3,4.0,5))`	5. (xs:decimal)	Returns `5`, which is the largest value among `3`, `4.0`, `5`, as an `xs:decimal` type value.
2	`fn:max(())`	Empty XQuery sequence	Returns an empty XQuery sequence because the value of the parameter `seq` is an empty XQuery sequence.
3	`fn:max((1.0E2, 2.0E-3, 100))`	1.0E2 (xs:double)	Returns `1.0E2`, which is the largest value among `1.0E2`, `2.0E-3`, `100`, as an `xs:double` type value.

(24) min

(a) Function

Returns the smallest value of the atomic values that make up an XQuery sequence.

(b) Format

```
fn:min ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-131: fn:min function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	seq	XQuery sequence of zero or more values of any atomic type	Input XQuery sequence

2. The result is an empty XQuery sequence or an atomic type value.
3. If the value of the parameter `seq` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. To compare the magnitude of atomic values, the values are compared with the `le` operator. An error results if the XQuery sequence that is the value of the parameter `seq` contains a mix of XQuery data types that cannot be compared with the `le` operator.
5. To compare numeric data types, all the values are converted to a common numeric

data type and then compared. For this reason, if the XQuery sequence in parameter `seq` consists of only numeric data type values, the result will be a value of the XQuery data type shown below.

Table 1-132: XQuery data type of the result of the fn:min function

No.	XQuery data type of the values in the XQuery sequence that is the value of the parameter seq			XQuery data type of the result
	xs:double	xs:decimal	xs:int	
1	Y	--	--	xs:double
2	N	Y	--	xs:decimal
3	N	N	--	xs:int

Legend:

Y: Includes.

N: Does not include.

--: Does not affect the data type of the result.

6. If the XQuery sequence that is the value of the parameter `seq` contains an `xs:double` type value that is `NaN` (not a number), the result will be `NaN`.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (XQuery data type in parentheses)	Explanation
1	`fn:min((3,4.0,5))`	3. (xs:decimal)	Returns `3`, which is the smallest value among `3`, `4.0`, `5`, as an `xs:decimal` type value.
2	`fn:min(())`	Empty XQuery sequence	Returns an empty XQuery sequence because the value of the parameter `seq` is an empty XQuery sequence.
3	`fn:min((1.0E2, 2.0E3, 100))`	1.0E2 (xs:double)	Returns `1.0E2`, which is the smallest value among `1.0E2`, `2.0E3`, `100`, as an `xs:double` type value.

(25) minutes-from-dateTime

(a) Function

Returns only the minutes part extracted from the time stamp.

(b) Format

```
fn:minutes-from-dateTime ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-133: fn:minutes-from-dateTime function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	dtm	`xs:dateTime` type (including empty XQuery sequences)	Time stamp to extract from

2. The result is an empty XQuery sequence or an `xs:int` type value.
3. If the value of the parameter `dtm` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. If the value of parameter `dtm` is an `xs:dateTime` type value, the value of the minutes part is returned as an `xs:int` type.

(d) Example

The following table provides an example of a function invocation and its result:

No.	Function invocation	Result (xs:int type)	Explanation
1	`fn:minutes-from-dateTime(xs:dateTime("2006-09-26T18:44:58.153"))`	44	Returns `44`, which is the minutes part of the value of the parameter `dtm`.

(26) minutes-from-time

(a) Function

Returns only the minutes part extracted from the time.

(b) Format

```
fn:minutes-from-time ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-134: fn:minutes-from-time function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	tim	`xs:time` type (including empty XQuery sequences)	Time to extract from

2. The result is an empty XQuery sequence or an `xs:int` type value.
3. If the value of the parameter `tim` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. If the value of the parameter `tim` is an `xs:time` type value, the value of the minutes part is returned as an `xs:int` type.

(d) Example

The following table provides an example of a function invocation and its result:

No.	Function invocation	Result (xs:int type)	Explanation
1	`fn:minutes-from-time(xs:time("18:44:58"))`	44	Returns `44`, which is the minutes part of the value of the parameter `tim`.

(27) month-from-date

(a) Function

Returns only the month part extracted from the date.

(b) Format

```
fn:month-from-date ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-135: fn:month-from-date function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	dat	`xs:date` type (including empty XQuery sequences)	Date to extract from

2. The result is an empty XQuery sequence or an `xs:int` type value.
3. If the value of the parameter `dat` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. If the value of parameter `dat` is an `xs:date` type value, return the value of the month part as an `xs:int` type.

(d) Example

The following table provides an example of a function invocation and its result:

No.	Function invocation	Result (xs:int type)	Explanation
1	`fn:month-from-date(xs:date("2006-09-26"))`	9	Returns `9`, which is the month part of the value of the parameter `dat`.

(28) month-from-dateTime

(a) Function

Returns only the month part extracted from the time stamp.

(b) Format

```
fn:month-from-dateTime ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-136: fn:month-from-dateTime function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	dtm	`xs:dateTime` type (including empty XQuery sequences)	Time stamp to extract from

2. The result is an empty XQuery sequence or an `xs:int` type value.
3. If the value of the parameter `dtm` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. If the value of parameter `dtm` is an `xs:dateTime` type value, return the value of the month part as an `xs:int` type.

(d) Example

The following table provides an example of a function invocation and its result:

No.	Function invocation	Result (xs:int type)	Explanation
1	`fn:month-from-dateTime(xs:dateTime("2006-09-26T18:44:58.153"))`	9	Returns `9`, which is the month part of the value of the parameter `dtm`.

(29) name

(a) Function

Returns the qualified name of a node.

(b) Format

```
fn:name ( [argument] )
```

(c) Rules

1. This function takes zero or one of the parameter listed in the following table.

Table 1-137: fn:name function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	node	Any node (including empty XQuery sequences)	Node whose qualified name is to be obtained.

2. The result is an `xs:string` type value.
3. If no argument is specified, the qualified name of the context item node as evaluated at the time this function was invoked is returned. In other words, the result is the same as specifying `fn:name(.)`. An error results if the context item is not a node.
4. If the value of the parameter `node` is an empty XQuery sequence, a character string of length zero is returned.
5. If the value of the parameter `node` is a node that does not have a name (document node, comment node, text node), a character string of length zero is returned.
6. If the value of the parameter `node` is a node that has a name (element node, attribute node, processing instruction node), the qualified name of that node is returned.

(d) Example

In the examples of function invocations and their results given below, the XQuery variable `book` indicates the node representing the elements shown below.

Element represented by the XQuery variable `book`:

```
<bookinfo:bookinfo book_id="452469630">
  <title>Relational Databases Explained</title>
  <author>Jeff Jones</author>
  <author>Bob Adams</author>
</bookinfo:bookinfo>
```

Examples of function invocations and their results:

No.	Function invocation	Result (xs:string type)	Explanation
1	`$book/fn:name()`	`"bookinfo:bookinfo"`	Returns the qualified name of the context item, which is the `bookinfo:bookinfo` element node, because no argument is specified.

No.	Function invocation	Result (xs:string type)	Explanation
2	`$book/fn:name(./title)`	`"title"`	Returns the qualified name `title` because the value of the parameter `node` is the title element node.
3	`$book/title /fn:name(./text())`	`""` (a character string of length zero)	Returns a character string of length zero because the value of the parameter `node` is a text node.

(30) namespace-uri

(a) Function

Returns the XML namespace URI of the qualified name of a node.

(b) Format

```
fn:namespace-uri ( [argument] )
```

(c) Rules

1. This function takes zero or one of the parameter listed in the following table.

Table 1-138: fn:namespace-uri function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	node	Any node (including empty XQuery sequences)	Node for which the XML namespace URI is to be obtained from the qualified name

2. The result is an `xs:string` type value.
3. If no argument is specified, the XML namespace URI obtained from evaluating the context item node's qualified name at the time this function was invoked is returned. In other words, the result is the same as specifying `fn:namespace-uri(.)`. An error results if the context item is not a node.
4. If the value of the parameter `node` is an empty XQuery sequence, a character string of length zero is returned.
5. If the value of the parameter `node` is a node other than an element node or an attribute node, a character string of length zero is returned.
6. If the value of the parameter `node` is an element node or attribute node whose

expand qualified name does not have an XML namespace URI, a character string of length zero is returned.

7. If the value of the parameter `node` is an element node or attribute node whose expand qualified name has an XML namespace URI, the XML namespace URI of the node's qualified name is returned.

(d) Example

In the examples of function invocations and their results given below, the XQuery variable `book` indicates the node representing the elements shown below. The example assumes that the XML namespace URI corresponding to the prefix `bookinfo` is `http://www.hirdb-example.com/bookinfo`, and the default XML namespace URI corresponding to the element and attribute names is `http://www.hirdb-example.com/default`.

Element represented by the XQuery variable `book`:

```
<bookinfo:bookinfo book_id="452469630">
  <title>Relational Databases Explained</title>
  <author>Jeff Jones</author>
  <author>Bob Adams</author>
</bookinfo:bookinfo>
```

Examples of function invocations and their results:

No.	Function invocation	Result (xs:string type)	Explanation
1	`$book/fn:namespace-uri()`	" http://www.hirdb-example.com/bookinfo"	Returns the XML namespace URI of the context item's qualified name, which is the `bookinfo:bookinfo` element node because no argument is specified.
2	`$book/fn:namespace-uri(./title)`	" http://www. hirdb-example.com/default"	Returns the XML namespace URI of that qualified name because the value of the parameter `node` is the `title` element node.

No.	Function invocation	Result (xs:string type)	Explanation
3	`$book/title /fn:namespace-uri(./ text())`	`""` (a character string of length zero)	Returns a character string of length zero because the value of the parameter `node` is a text node.

(31) normalize-space

(a) Function

Returns the result of normalizing whitespace in a character string.

Normalizing whitespace in a character string means removing leading and trailing whitespace (one-byte space (`X'20'`), TAB (`X'09'`), NL (`X'0A'`), and CR (`X'0D'`)), and replacing contiguous whitespace with a single one-byte space.

(b) Format

```
fn:normalize-space ( [argument] )
```

(c) Rules

1. This function takes zero or one of the parameter listed in the following table.

   Table 1-139: fn:normalize-space function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	str	`xs:string` type (including empty XQuery sequences)	Input character string

2. The result is an `xs:string` type value.
3. If no argument is specified, the result of normalizing whitespace in the character string representation of the value of the context item as evaluated at the time this function was invoked is returned. In other words, the result is the same as specifying `fn:normalize-space(fn:string(.))`.
4. If the value of the parameter `str` is an empty XQuery sequence, a character string of length zero is returned.

5. If the value of the parameter `str` is an `xs:string` type value, a result of normalizing the whitespace of that value is returned.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (xs:string type)	Explanation
1	`fn:normalize-space(())`	`""` (a character string of length zero)	Returns a character string of length zero because the value of the parameter `str` is an empty XQuery sequence.
2	`fn:normalize-space(" a b c d ")`	"a b c d"	Returns the result of normalizing whitespace of `"a b c d "`.

(32) not

(a) Function

Converts the value specified in an argument to the `xs:boolean` type and returns the negation of its value.

(b) Format

```
fn:not ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

   Table 1-140: fn:not function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	seq	XQuery sequence with zero or more XQuery items	Input XQuery sequence

2. The result is an `xs:boolean` type value.
3. Returns a result using the following procedure:
   - Applies the `fn:boolean` function to the value of the parameter `seq` to calculate its Boolean value.
   - If the calculated Boolean value is `TRUE`, return `FALSE`. If the calculated

Boolean value is FALSE, return TRUE.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (xs:boolean type)	Explanation
1	fn:not(fn:true())	FALSE	Returns FALSE because the value of the parameter seq is TRUE.
2	fn:not(0)	TRUE	Returns TRUE because 0 evaluates to the Boolean value FALSE.
3	fn:not("ABC")	FALSE	Returns FALSE because "ABC" evaluates to the Boolean value TRUE.

(33) number

(a) Function

Returns the result of converting the value specified in the argument to the xs:double type.

(b) Format

```
fn:number ( [argument] )
```

(c) Rules

1. This function takes zero or one of the parameter listed in the following table.

   Table 1-141: fn:number function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	item	Any XQuery item (including empty XQuery sequences)	Input XQuery item

2. The result is an xs:double type value.
3. If no argument is specified, the context item is atomized at the time this function is invoked, the value is converted to xs:double type and returned. In other words, the result is the same as specifying fn:number(.).

4. If the parameter `item` is an empty XQuery sequence, `NaN` (not a number) is returned.
5. If the value of the parameter `item` is an XQuery item, the atomized value of that XQuery item converted to `xs:double` type is returned. If conversion is not possible, `NaN` (not a number) is returned. For details about when conversion is possible, see *1.15.2(3)(c) Convertible XQuery data types*.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (xs:double type)	Explanation
1	`fn:number(())`	`NaN` (not a number)	Returns `NaN` because the value of the parameter `item` is an empty XQuery sequence.
2	`fn:number("abcde")`	`NaN` (not a number)	Returns `NaN` because `"abcde"` cannot be converted to `xs:double` type.
3	`fn:number(fn:false())`	0.0E0	Returns `0.0E0` because the `xs:boolean` type `TRUE` converts to the `xs:double` type value `0.0E0`.
4	`fn:number(" 15000000 ")`	1.5E7	Returns `1.5E7` because `" 15000000 "` converts to the `xs:double` type value `1.5E7`.
5	`fn:number("INF")`	Positive infinity	Returns positive infinity because `"INF"` converts to the `xs:double` type value positive infinity.

(34) position

(a) Function

Returns the context position.

(b) Format

```
fn:position ( )
```

(c) Rules

1. This function has no parameters.
2. The result is an `xs:int` type value.
3. Returns the context position of a context item as evaluated at the time this

function was invoked.

(d) Example

The following table provides an example of a function invocation and its result:

No.	Function invocation	Result	Explanation
1	`("a", "b", "c")[fn:position() ge 2]`	`("b", "c")` The results of the `fn:position` function on the context items `"a"`, `"b"`, `"c"` are `1`, `2`, `3` respectively.	The result is the XQuery sequence consisting of `"b"`, `"c"`, which are the context items whose context position is greater than or equal to `2`.

(35) remove

(a) Function

Returns the result of removing the XQuery item at the specified position in the XQuery sequence.

(b) Format

```
fn:remove ( argument-1, argument-2 )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-142: fn:remove function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument-1*	seq	XQuery sequence containing zero or more XQuery items	Input XQuery sequence
2	*argument-2*	pos	xs:int	Position of the context item to be removed

2. The result is an XQuery sequence consisting of zero or more XQuery items.
3. If the value of the parameter `seq` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. The position of the XQuery item to be remove is specified in parameter `pos`. If the value of the parameter `pos` is less than `1`, or greater than the number of

XQuery items in the XQuery sequence that is the value of the parameter `seq`, the XQuery sequence that is the value of the parameter `seq` is returned unchanged.

5. The resulting XQuery sequence consists of the XQuery items up to one before the XQuery item to be delete from the XQuery sequence that is the value of the parameter `seq`, followed by the XQuery items after the XQuery item to be delete from the XQuery sequence that is the value of the parameter `seq`.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result	Explanation
1	`fn:remove(("a", "b", "c"), 0)`	("a", "b", "c")	Returns the value of parameter `seq` unchanged, because the value of parameter `pos` is less than `1`.
2	`fn:remove(("a", "b", "c"), 1)`	("b", "c")	Returns the XQuery sequence with the first XQuery item removed.
3	`fn:remove(("a", "b", "c"), 4)`	("a", "b", "c")	Returns the value of parameter `seq` unchanged, because the value of parameter `pos` (`4`) is greater than the number of XQuery items in the XQuery sequence that is the value of the parameter `seq`.

(36) reverse

(a) Function

Returns the XQuery sequence that results from sorting the XQuery items in an XQuery sequence in reverse order.

(b) Format

```
fn:reverse ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-143: fn:reverse function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	seq	XQuery sequence consisting of zero or more arbitrary XQuery items	Input XQuery sequence

2. The result is an XQuery sequence consisting of zero or more XQuery items.
3. If the parameter `seq` is an empty XQuery sequence, an empty XQuery sequence is returned.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result	Explanation
1	`fn:reverse((10, 20, 30, 40))`	(40,30,20,10)	Returns an XQuery sequence in which the XQuery items in the XQuery sequence in parameter `seq` have been sorted in reverse order.
2	`fn:reverse(("abcde"))`	("abcde")	Returns the XQuery sequence in parameter `seq` unchanged, because there is only one XQuery item in the XQuery sequence in parameter `seq`.
3	`fn:reverse(())`	Empty XQuery sequence	Returns an empty XQuery sequence because the XQuery sequence in parameter `seq` is an empty XQuery sequence.

(37) round

(a) Function

Returns the integer value closest to the value of an argument.

(b) Format

```
fn:round ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-144: fn:round function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	num	Numeric data type (including empty XQuery sequences)	Input numeric value

2. The result is an empty XQuery sequence, or a value of one of the numeric data

types shown below.

Table 1-145: XQuery data type of the result of the fn:round function

No.	XQuery data type of the value of the parameter num	XQuery data type of the result
1	xs:double	xs:double
2	xs:decimal	xs:decimal
3	xs:int	xs:int

3. If the value of the parameter `num` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. If there are two qualifying values, the larger one is returned.
5. The following table describes what is returned when the value of the parameter `num` is `xs:double` type and one of the special value is specified.

Table 1-146: fn:round function special value results

No.	Value of the parameter num	Result value
1	Positive 0	Positive 0
2	Negative 0	Negative 0
3	At least -0.5 but less than 0	Negative 0
4	Positive infinity	Positive infinity
5	Negative infinity	Negative infinity
6	`NaN` (not a number)	`NaN` (not a number)

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (XQuery data type in parentheses)	Explanation
1	`fn:round(2.5)`	3. (xs:decimal)	Returns `3`, the larger of `2` and `3`, which are the integers closest to `2.5`.
2	`fn:round(2.4999)`	2. (xs:decimal)	Returns `2`, the integer closest to `2.4999`.
3	`fn:round(-2.5)`	-2. (xs:decimal)	Returns `-2`, the larger of `-2` and `-3`, which are the integers closest to `-2.5`.

No.	Function invocation	Result (XQuery data type in parentheses)	Explanation
4	`fn:round(-4.5E-1)`	Negative 0 (`xs:double`)	Returns `-0`, because `-4.5E-1` is less than `0` but greater than `-0.5`.

(38) seconds-from-dateTime

(a) Function

Returns only the seconds part extracted from the time stamp.

(b) Format

```
fn:seconds-from-dateTime ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-147: fn:seconds-from-dateTime function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	dtm	`xs:dateTime` type (including empty XQuery sequences)	Time stamp to extract from

2. The result is an empty XQuery sequence or an `xs:decimal` type value.
3. If the value of the parameter `dtm` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. If the value of parameter `dtm` is an `xs:dateTime` type value, the value of the seconds part, including fractional seconds, is returned as an `xs:decimal` type.

(d) Example

The following table provides an example of a function invocation and its result:

No.	Function invocation	Result (xs:decimal type)	Explanation
1	`fn:seconds-from-dateTime(xs:dateTime("2006-09-26T18:44:58.153"))`	58.153	Returns `58.153`, which is the seconds part of the value of the parameter `dtm`.

(39) seconds-from-time

(a) Function

Returns only the seconds part extracted from the time.

(b) Format

```
fn:seconds-from-time ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

   Table 1-148: fn:seconds-from-time function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	tim	`xs:time` type (including empty XQuery sequences)	Time to extract from

2. The result is an empty XQuery sequence or an `xs:decimal` type value.
3. If the value of the parameter `tim` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. If the value of the parameter `tim` is an `xs:time` type value, the value of the seconds part as an `xs:decimal` type is returned.

(d) Example

The following table provides an example of a function invocation and its result:

No.	Function invocation	Result (xs:decimal type)	Explanation
1	`fn:seconds-from-time(xs:time("18:44:58"))`	58	Returns `58`, which is the seconds part of the value of the parameter `tim`.

(40) starts-with

(a) Function

Returns a value indicating whether the character string specified in *argument-1* starts with the character string specified in *argument-2*.

(b) Format

```
fn:starts-with ( argument-1, argument-2 )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-149: fn:starts-with function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument-1*	str1	`xs:string` type (including empty XQuery sequences)	Character string to be checked to determine if it starts with the substring
2	*argument-2*	str2	`xs:string` type (including empty XQuery sequences)	Substring to check against the start of the character string in parameter `str1`

2. The result is an `xs:boolean` type value.
3. Returns `FALSE` if the value of parameter `str1` is an empty XQuery sequence or a character string of length zero and the value of parameter `str2` is a character string of length 1 or greater.
4. If the value of parameter `str2` is an empty XQuery sequence or a character string of length zero, `TRUE` is returned regardless of the value of parameter `str1`.
5. Returns `TRUE` if the values of parameters `str1` and `str2` are character strings of length 1 or greater and the character string in parameter `str1` starts with the character string in parameter `str2`. Otherwise returns `FALSE`.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (xs:boolean type)	Explanation
1	`fn:starts-with("abcdefg", "abc")`	`TRUE`	Returns `TRUE` because `"abcdefg"` starts with `"abc"`.
2	`fn:starts-with("abcdefg", "cde")`	`FALSE`	Returns `FALSE` because `"abcdefg"` does not start with `"cde"`.

No.	Function invocation	Result (xs:boolean type)	Explanation
3	`fn:starts-with((), "")`	`TRUE`	Returns `TRUE` because the value of parameter `str2` is a character string of length zero.
4	`fn:starts-with((), "a")`	`FALSE`	Returns `FALSE` because the value of parameter `str1` is an empty XQuery sequence, and the value of parameter `str2` is a character string of length 1 or greater.
5	`fn:starts-with("abcdefg", ())`	`TRUE`	Returns `TRUE` because the value of parameter `str2` is an empty XQuery sequence.

(41) string

(a) Function

Returns the character string representation of the value of an XQuery item as an `xs:string` type value.

(b) Format

```
fn:string ( [argument] )
```

(c) Rules

1. This function takes zero or one of the parameter listed in the following table.

   Table 1-150: fn:string function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	item	Any XQuery item (including empty XQuery sequences)	Input XQuery item

2. The result is an `xs:string` type value.
3. If no argument is specified, the character string representation of the value of the context item as evaluated at the time this function was invoked is returned. In other words, the result is the same as specifying `fn:string()`.
4. If the parameter `item` is an empty XQuery sequence, a character string of length zero.
5. If the parameter `item` is a node, the result will be a value listed in the following

table, depending on the type of node.

Table 1-151: Results when the parameter item is a node

No.	Kind of node	Value returned by the fn:string function
1	Document node	Value of the `string-value` property
2	Element node	Value of the `string-value` property
3	Attribute node	Value of the `string-value` property
4	Processing instruction node	Value of the `content` property
5	Comment node	Value of the `content` property
6	Text node	Value of the `content` property

6. If the value of the parameter `item` is an atomic value, the atomic value converted to `xs:string` type is returned. For details about the rules for converting to `xs:string` type, see *1.15.2(3)(c) Convertible XQuery data types*.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (xs:string type)	Explanation
1	`fn:string(())`	`""` (a character string of length zero)	Returns a character string of length zero because the value of the parameter `item` is an empty XQuery sequence.
2	`fn:string("abcde")`	"abcde"	Returns the character string `"abcde"` unchanged.
3	`fn:string(1.234E4)`	"12340"	Returns the result of converting the `xs:decimal` type value `1.234E4` to the `xs:string` type value `"12340"`.
4	`fn:string(0.1234E10)`	"1.234E9"	Returns the result of converting the `xs:decimal` type value `0.1234E10` to the `xs:string` type value `"1.234E9"`.
5	`fn:string(fn:true())`	"true"	Returns the result of converting the `xs:boolean` type value `TRUE` to the `xs:string` type value `"true"`.

(42) string-length

(a) Function

Returns the length (in number of characters) of a character string.

(b) Format

```
fn:string-length ( [argument] )
```

(c) Rules

1. This function takes zero or one of the parameter listed in the following table:

Table 1-152: fn:string-length function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	str	`xs:string` type (including empty XQuery sequences)	Input character string

2. The result is an `xs:int` type value.
3. If no argument is specified, the length of the character string representation of the context item as evaluated at the time this function was invoked is returned. In other words, the result is the same as specifying `fn:string-length(fn:string(.))`.
4. Returns `0` if the value of the parameter `str` is an empty XQuery sequence.
5. If the value of the parameter `str` is an `xs:string` type value, the number of characters in its value is returned.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (xs:int type)	Explanation
1	`fn:string-length(())`	0	Returns `0`, because the value of parameter `str` is an empty XQuery sequence.
2	`fn:string-length("abcde")`	5	Returns `5`, which is the length of `"abcde"`.
3	`fn:string-length("あいうえお")`	5	Returns `5`, which is the length of `"あいうえお"`.

(43) subsequence

(a) Function

Returns an XQuery subsequence of an XQuery sequence.

(b) Format

```
fn:subsequence ( argument-1, argument-2[, argument-3] )
```

(c) Rules

1. The following table lists the parameters of this function. Note that the parameter `len` is optional.

   Table 1-153: fn:subsequence function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument-1*	seq	XQuery sequence consisting of zero or more XQuery items	XQuery sequence from which to extract the XQuery subsequence
2	*argument-2*	pos	`xs:double` type	Starting position from which to extract the XQuery subsequence
3	*argument-3*	len	`xs:double` type	Number of XQuery items to extract

2. The result is an XQuery sequence consisting of zero or more XQuery items.
3. If the parameter `seq` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. The starting position from which to extract the XQuery subsequence is specified in the parameter `pos`. Parameter `pos` is evaluated using the `fn:round` function, and the integer value of the result is the starting position. The first position is 1.

   If the result of using the `fn:round` function to evaluate the parameter `pos` is not a positive value, extraction starts from the first position.

   If the result of using the `fn:round` function to evaluate the parameter `pos` is greater than the number of XQuery items in the parameter `seq`, an empty XQuery sequence is returned.
5. The number of XQuery items in the XQuery subsequence to extract is specified in the parameter `len`. Parameter `len` is evaluated using the `fn:round` function,

and the integer value of the result is the number of XQuery items to extract into the XQuery subsequence.

If the result of using the `fn:round` function to evaluate the parameter `len` is not a positive value, an empty XQuery sequence is returned.

If the result of using the `fn:round` function to evaluate the parameter `len` is greater than the number XQuery items from the starting position to the end of the XQuery sequence in parameter `seq`, or if the parameter `len` is omitted, all XQuery items from the starting position to the end of the XQuery sequence in parameter `seq` are extracted.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result	Explanation
1	`fn:subsequence( ("a","b","c","d","e","f","g"), 5)`	("e","f","g")	Returns the XQuery sequence consisting of XQuery items from the fifth position until the end of the XQuery sequence in parameter `seq`.
2	`fn:subsequence( ("a","b","c","d","e","f","g"), 2, 2.5)`	("b","c","d")	Returns the XQuery sequence consisting of three XQuery items starting from the second position of the XQuery sequence in parameter `seq` because the result of evaluating the `fn:round` function on `2.5` is `3`.
3	`fn:subsequence((), 10)`	Empty XQuery sequence	Returns an empty XQuery sequence because parameter `seq` is an empty XQuery sequence.
4	`fn:subsequence (("a","b","c","d","e","f","g") , -10, 10)`	`("a","b","c","d", "e","f","g")`	Extraction starts from the first position because parameter `pos` is negative. Extraction continues to the end because the value of parameter `len` is greater than the number of XQuery items from the starting position until the end of parameter `seq`.

(44) substring

(a) Function

Returns a substring of a character string.

(b) Format

```
fn:substring ( argument-1, argument-2[, argument-3] )
```

(c) Rules

1. The following table lists the parameters of this function. Note that the parameter `len` is optional.

   Table 1-154: fn:substring function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument-1*	str	`xs:string` type (including empty XQuery sequences)	Character string from which to extract a substring
2	*argument-2*	pos	`xs:double` type	Starting position of the substring to be extracted
3	*argument-3*	len	`xs:double` type	Length (in number of characters) of the substring to be extracted

2. The result is an `xs:string` type value.
3. If the value of the parameter `str` is an empty XQuery sequence, a character string of length zero is returned.
4. If the value of the parameter `str` is an `xs:string` type value of length zero, a character string of length zero is returned.
5. The starting position of the substring to extract is specified in the parameter `pos`. The integer value of the result of evaluating the `fn:round` function on the parameter `pos` will be the starting position. The first position is 1.

   If the integer value of the result of evaluating the `fn:round` function on the parameter `pos` is not a positive value, extraction starts from the first position.

   If the integer value of the result of evaluating the `fn:round` function on the parameter `pos` is greater than the length of the value of the parameter `str`, a character string of length zero is returned.

6. The length (in number of characters) of the part of the character string to be extracted is specified in the parameter `len`. The integer value of the result of evaluating the `fn:round` function on the parameter `len` will be the length of the substring to extract.

   If the integer value of the result of evaluating the `fn:round` function on the parameter `len` is not a positive value, a character string of length zero is returned.

   If the integer value of the result of evaluating the `fn:round` function on the parameter `len` is greater than the length of the string from the starting position to the end of the value of the parameter `str`, or if parameter `len` is omitted, extraction continues to the end of the character string that is the value of the parameter `str`.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (xs:string type)	Explanation
1	`fn:substring ("abcdefg", 5)`	"efg"	Returns the substring from the fifth character until the end of `"abcdefg"`.
2	`fn:substring("abcdefg", 2, 2.5)`	"bcd"	Returns the three-character substring starting from the second character of `"abcdefg"`, because the result of using the `fn:round` function to evaluate `2.5` is `3`.
3	`fn:substring((), 10)`	`""` (a character string of length zero)	Returns a character string of length zero because the value of the parameter `str` is an empty XQuery sequence.
4	`fn:substring ("abcdefg",-10, 10)`	"abcdefg"	Extraction starts from the first position because the value of parameter `pos` is negative. Extraction continues to the end because the value of parameter `len` is greater than the length from the starting position to the end of `"abcdefg"`.

(45) substring-after

(a) Function

Returns a substring of the character string specified in *argument-1* that begins immediately after the position of the first occurrence of the character string specified

in *argument-2*.

(b) Format

```
fn:substring-after ( argument-1, argument-2 )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-155: fn:substring-after function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument-1*	str1	`xs:string` type (including empty XQuery sequences)	Character string in which to find the substring
2	*argument-2*	str2	`xs:string` type (including empty XQuery sequences)	Substring to find in the character string in parameter `str1`

2. The result is an `xs:string` type value.
3. If the value of parameter `str1` is an empty XQuery sequence or a character string of length zero, a character string of length zero is returned regardless of the value of parameter `str2`.
4. If the value of parameter `str1` is a character string of length 1 or greater, and the value of parameter `str2` is an empty XQuery sequence or a character string of length zero, return the value of parameter `str1`.
5. When the values of parameters `str1` and `str2` are character strings of length 1 or greater, and if the character string in parameter `str2` is found in the character string in parameter `str1`, the substring that starts immediately after the position of the first such occurrence and extends to the end of the string is returned. Otherwise, a character string of length zero is returned.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (xs:string type)	Explanation
1	`fn:substring-after("abcdefg", "efg")`	`""` (a character string of length zero)	Returns the substring after the position of the first occurrence of `"eft"` in `"abcdefg"`.

No.	Function invocation	Result (xs:string type)	Explanation
2	`fn:substring-after("abcdefg cde", "cde")`	"fgcde"	Returns the substring after the position of the first occurrence of `"cde"` in `"abcdefgcde"`.
3	`fn:substring-after((), "abc")`	`""` (a character string of length zero)	Returns a character string of length zero because the value of parameter `str1` is an empty XQuery sequence.
4	`fn:substring-after("abcdefg", ())`	"abcdefg"	Returns `"abcdefg"` unchanged, because the value of parameter `str2` is an empty XQuery sequence.

(46) substring-before

(a) Function

Returns a substring of the character string specified in *argument-1* that ends immediately before the position of the first occurrence of the character string specified in *argument-2*.

(b) Format

```
fn:substring-before ( argument-1, argument-2 )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-156: fn:substring-before function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument-1*	str1	`xs:string` type (including empty XQuery sequences)	Character string in which to find the substring
2	*argument-2*	str2	`xs:string` type (including empty XQuery sequences)	Substring to find in the character string in parameter `str1`

2. The result is an `xs:string` type value.

3. If the value of parameter `str1` is an empty XQuery sequence or a character string of length zero, a character string of length zero is returned regardless of the value of parameter `str2`.
4. If the value of parameter `str1` is a character string of length 1 or greater, and the value of parameter `str2` is an empty XQuery sequence or a character string of length zero, the value of parameter `str1` is returned.
5. When the values of parameters `str1` and `str2` are character strings of length 1 or greater, and if the character string in parameter `str2` is found in the character string in parameter `str1`, the substring that starts at the beginning of the string and ends immediately before the position of the first such occurrence is returned. Otherwise, a character string of length zero is returned.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (xs:string type)	Explanation
1	`fn:substring-before("abcdefg", "abc")`	`""` (a character string of length zero)	Returns the substring before the position where `"abc"` occurs in `"abcdefg"`.
2	`fn:substring-before("abcdefgdef", "def")`	"abc"	Returns the substring before the position where `"def"` occurs in `"abcdefgdef"`.
3	`fn:substring-before((), "abc")`	`""` (a character string of length zero)	Returns a character string of length zero because the value of parameter `str1` is an empty XQuery sequence.
4	`fn:substring-before("abcdefg", ())`	"abcdefg"	Returns `"abcdefg"` unchanged, because the value of parameter `str2` is an empty XQuery sequence.

(47) sum

(a) Function

Returns the sum of the atomic values that make up an XQuery sequence.

(b) Format

```
fn:sum ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-157: fn:sum function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	seq	XQuery sequence with zero or more numeric data type values	Input XQuery sequence

2. The result is an empty XQuery sequence or a numeric data type value.
3. If the value of the parameter `seq` is an empty XQuery sequence, the `xs:int` type value `0` is returned.
4. Thee following table indicates the XQuery data type of the result for each XQuery sequence that is the value of the parameter `seq`.

Table 1-158: XQuery data type of the result of the fn:sum function

No.	XQuery data type of the value of the XQuery sequence that is the value of the parameter seq			XQuery data type of the result
	xs:double	xs:decimal	xs:int	
1	N	N	N	xs:int
2	Y	--	--	xs:double
3	N	Y	--	xs:decimal
4		N	Y	xs:int

Legend:

Y: Includes.

N: Does not include.

--: Does not affect the data type of the result.

5. If the XQuery sequence that is the value of the parameter `seq` includes `NaN` (not a number), result will be `NaN`.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (XQuery data type in parentheses)	Explanation
1	`fn:sum((3,4,5))`	12 (xs:int)	Returns `12`, which is the value of the sum of `3`, `4`, and `5`.
2	`fn:sum(())`	0 (xs:int)	Returns `0`, because the parameter `seq` is an empty XQuery sequence.
3	`fn:sum((1.0E2, 2.0E3))`	2.1E3 (xs:double)	Returns `2.1E3`, which is the value of the sum of `1.0E2` and `2.0E3`.

(48) translate

(a) Function

Returns the result of replacing the characters specified in *argument-2* with the characters specified in *argument-3* in the character string specified in *argument-1*.

(b) Format

```
fn:translate ( argument-1, argument-2, argument-3 )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-159: fn:translate function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument-1*	str	`xs:string` type (including empty XQuery sequences)	Input character string
2	*argument-2*	from	`xs:string` type	Character string consisting of characters to be replaced
3	*argument-3*	to	`xs:string` type	Character string consisting of replacement characters

2. The result is an `xs:string` type value.
3. If the value of the parameter `str` is an empty XQuery sequence, a character string of length zero is returned.

4. If the value of the parameter `str` is an `xs:string` type value, the characters in that string are evaluated against the characters in the string that is the value of the parameter `from`. Any matching characters are replaced with the character in the corresponding position in the character string that is the value of the parameter `to`, and the result is returned. If the character to be replaced is the *n*th character in the character string that is the value of the parameter `from`, the *n*th character in the character string that is the value of the parameter `to` will replace it. If the character to be replaced is the *m*th character in the character string that is the value of the parameter `from`, and the length of the character string that is the value of the parameter `to` is less than *m*, the character to be replaced is removed.
5. If the same letter appears multiple times in the character string that is the value of the parameter `from`, it is replaced by the first replacement character specified.

(d) Example

The following table provides examples of function invocations and their results:

No.	Function invocation	Result (xs:string type)	Explanation
1	`fn:translate("abcdef", "ace", "ACE")`	"AbCdEf"	Returns the character string in which each of `a`, `c`, `e` in `"abcdef"` is replaced with `A`, `C`, `E`, respectively.
2	`fn:translate("abcdefg", "aceg", "ACE")`	"AbCdEf"	Returns the character string in which each of `a`, `c`, `e` in `"abcdefg"` is replaced with `A`, `C`, `E`, respectively, and `g` is removed.
3	`fn:translate("abcdef", "acea", "ACEB")`	"AbCdEf"	Returns the character string in which each of `a`, `c`, `e` in `"abcdef"` is replaced with `A`, `C`, `E`, respectively.

(49) true

(a) Function

Returns `TRUE`.

(b) Format

```
fn:true ( )
```

(c) Rules

1. This function has no parameters.
2. The result is the `xs:boolean` type value `TRUE`.

(50) year-from-date

(a) Function

Returns only the year part extracted from the date.

(b) Format

```
fn:year-from-date ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

   Table 1-160: fn:year-from-date function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	dat	`xs:date` type (including empty XQuery sequences)	Date to extract from

2. The result is an empty XQuery sequence or an `xs:int` type value.
3. If the value of the parameter `dat` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. If the value of parameter `dat` is an `xs:date` type value, the value of the year part is returned as an `xs:int` type.

(d) Example

The following table provides an example of a function invocation and its result:

No.	Function invocation	Result (xs:int type)	Explanation
1	`fn:year-from-date(xs:date("2006-09-26"))`	2006	Returns `2006`, the year part of the value of parameter `dat`.

(51) year-from-dateTime

(a) Function

Returns only the year part extracted from the time stamp.

(b) Format

```
fn:year-from-dateTime ( argument )
```

(c) Rules

1. The following table lists the parameters of this function.

Table 1-161: fn:year-from-dateTime function parameters

No.	Corresponding argument	Parameter	XQuery data type	Explanation
1	*argument*	dtm	`xs:dateTime` type (including empty XQuery sequences)	Time stamp to extract from

2. The result is an empty XQuery sequence or an `xs:int` type value.
3. If the value of the parameter `dtm` is an empty XQuery sequence, an empty XQuery sequence is returned.
4. If the value of parameter `dtm` is an `xs:dateTime` type value, the value of the year part is returned as an `xs:int` type.

(d) Example

The following table provides an example of a function invocation and its result:

No.	Function invocation	Result (xs:int type)	Explanation
1	`fn:year-from-dateTime(xs:dateTime("2006-09-26T18:44:58.153"))`	2006	Returns `2006`, which is the year part of the value of the parameter `dtm`.

Chapter
2. Details of Constituent Elements

This chapter explains in detail the constituent elements that are involved in the use of SQL.

This chapter contains the following sections:

2.1 Cursor specification

2.1.1 Cursor specification: Format 1

(1) Function

A cursor specification enables the user to retrieve data from one or more tables or to sort the results of a retrieval.

A cursor specification is made in a cursor declaration or in the dynamic `SELECT` statement.

For details about the `SELECT` statement, see *Dynamic SELECT statement Format 1 (Retrieve dynamically)* in Chapter 4.

(2) Privileges

Users who can use all query specifications that are included in a cursor specification or who can use subqueries are authorized to use a cursor specification.

See *2.3 Query specification* for query specification privileges and *2.4 Subqueries* for subquery privileges.

(3) Format

Format 1: Searching one or more tables

query-expression
```
[ORDER BY {column-specification
          |sort-item-specification-number}
[{ASC | DESC}]
[,{column-specification | sort-item-specification-number}
[{ASC | DESC}]]... ]
[LIMIT {[offset-of-first-row-to-return,] {limit-row-count | ALL}
       | {limit-row-count | ALL} [OFFSET offset-of-first-row-to-return]}]
```

(4) Operands

- *query-expression*

A query expression specifies either a query specification or the union set or the disjunction set between derived tables that are obtained as a result of a query specification.

For details of query expressions, see *2.2 Query expressions*.

- `ORDER BY` {*column-specification*|*sort-item-specification-number*}

Specifies the sort method to be used when the retrieval results produced by a query

expression are to be sorted in ascending or descending order.

If the ORDER BY clause is omitted, the rows in the derived table may appear in random order.

The following rules apply to the ORDER BY clause:

1. A maximum of 255 columns can be specified in a sort key.
2. A column can be specified only once.
3. If an AS column name is specified in the outermost query selection expression and a table derived from the query expression contains a column with the same name, that column name cannot be specified in a sort key.
4. None of the following can be specified as a sort key:
   - Repetition columns and value expressions that contain a repetition column
   - BLOB
   - BINARY with a minimum defined length of 32,001 bytes
   - BOOLEAN
   - Abstract data type
   - WRITE specification
   - GET_JAVA_STORED_ROUTINE_SOURCE specification
5. When the ORDER BY clause is specified, an SQL variable, SQL parameter, or literal of any of the following data types cannot be specified by itself in a selection expression:
   - BLOB
   - BINARY with a minimum defined length of 32,001 bytes
   - Abstract data type

■ *column-specification*

Specifies a column to be used as a sort key.

The following rules apply to column specifications:

1. If the query expression contains either UNION [ALL] or EXCEPT [ALL], a column cannot be specified. However, if the table derived from the query expression contains no column with the same name, that column name can be specified.
2. If SELECT DISTINCT is specified in the outermost query, the column specified in the ORDER BY clause must be an item that is output as a retrieval result (specified in the selection expression).

3. If the outermost query specification does not specify SELECT DISTINCT but instead specifies either a GROUP BY clause or a set function, the column specified in the ORDER BY clause must be a GROUPing column. In other cases, any column in the table specified in the outermost query specification can be specified.

- *sort-item-specification-number*

When UNION [ALL] or EXCEPT [ALL] is specified, specifies the ordinal positional number in the derived table of a column to be used as a sort key. A column in the derived table obtained as a result of the query expression must be specified.

When UNION [ALL] or EXCEPT [ALL] is not specified and a column in the derived table obtained as a result of a set function, windows function, literal, arithmetic operation, date operation, scalar operation, CASE expression, CAST specification, function invocation, component specification, or concatenation operation is used as a sort key, this variable specifies the position in the derived table of the column that is to be used as the sort key.

A sort item specification number cannot be specified in the ORDER BY clause of a query expression in which ROW is specified in the selection expression.

- {ASC|DESC}

ASC

Specifies that the retrieval results are to be sorted in ascending order.

DESC

Specifies that the retrieval results are to be sorted in descending order.

- LIMIT {[*offset-of-first-row-to-return*,] {*limit-row-count*|ALL} | {*limit-row-count*|ALL} [OFFSET *offset-of-first-row-to-return*]}

Of search results in a query expression, specify the number of rows to be skipped from the beginning and the number of rows to be acquired. The LIMIT clause can improve SQL search performance. For a description of under what conditions the LIMIT clause must be specified, see the section on the facility for acquiring *n* rows of search results from the beginning in the *HiRDB Version 9 UAP Development Guide*.

offset-of-first-row-to-return

In *offset-of-first-row-to-return*, specifies the number of rows to be skipped from the beginning of the results in a search expression.

limit-row-count

If an offset is not specified, specifies the number of rows to be acquired from the results of the query expression.

If an offset is specified, specifies the number of rows to be acquired by skipping the offset from the beginning of the results of the query expression.

ALL

If an offset is not specified, acquires all the results from the query expression.

If an offset is specified, acquires all the results by skipping the offset from the beginning of the results of the query expression.

The LIMIT clause is subject to the following rules:

1. In *offset-of-first-row-to-return* and *limit-row-count*, you can specify an integer literal, an embedded variable, a ? parameter, an SQL variable, and an SQL parameter.
2. Set the data type for *offset-of-first-row-to-return* and *limit-row-count* as an integer (either SMALLINT or INTEGER).
3. The null value cannot be specified in either *offset-of-first-row-to-return* or *limit-row-count*.
4. If either a ? parameter or an embedded variable is specified in *offset-of-first-row-to-return* or *limit-row-count*, the ? parameter or the embedded variable is assumed to have the INTEGER (without an indicator variable) data type.
5. The range of values that can be specified in *offset-of-first-row-to-return* is 0 to 2,147,483,647.
6. Specifying 0 in *offset-of-first-row-to-return* has the same effect as when nothing is specified in *offset-of-first-row-to-return*.
7. The range of values that can be specified in *limit-row-count* is -1 to 2,147,483,647.
8. If the value -1 is specified in *limit-row-count*, it must be specified as a literal.
9. The maximum total value that can be specified in *offset-of-first-row-to-return* and *limit-row-count* is 2,147,483,647.
10. If *limit-row-count* ≥ 0, the search result row count will be:

    Max (Min (*query-expression-row-count* - *offset-of-first-row-to-return*, *limit-row-count*), 0)
11. If either *limit-row-count* = -1 or ALL is specified, the search result row count will be:

    Max (*query-expression-row-count* - *offset-of-first-row-to-return*, 0)
12. If an offset is specified, the retrieval result is not uniquely identified unless there are multiple rows containing the same sort key value as the last row of the rows that are skipped, or unless a sort key is specified.
13. If the result of the query expression is greater than *limit-row-count*, the retrieval

result is not uniquely identified unless there are multiple rows containing the same sort key as the last row of *limit-row-count* or a sort key is specified. The following example shows a case in which retrieval results are not uniquely identified.

Example:

If a table (STOCK) is retrieved using the cursor declared in the following SQL statement, multiple rows have the third smallest value, 3640, in the sort key column (PRICE). Therefore, whether the retrieval result is 1 or 2 is indefinite:

```
DECLARE CR1 CURSOR FOR
  SELECT * FROM STOCK ORDER BY PRICE LIMIT 3
```

Table name: STOCK

PCODE	PNAME	PRICE	SQUANTITY
101M	Blouse	35.00	85
201M	Polo shirt	36.40	29
302S	Skirt	36.40	65
591S	Socks	2.50	280

• Result 1

PCODE	PNAME	PRICE	SQUANTITY
591S	Socks	2.50	280
101M	Blouse	35.00	85
201M	Polo shirt	36.40	29

• Result 2

PCODE	PNAME	PRICE	SQUANTITY
591S	Socks	2.50	280
101M	Blouse	35.00	85
302S	Skirt	36.40	65

(5) Notes

1. Specifying ORDER BY can cause HiRDB to create a work table. In this case, the processing of ORDER BY can be restricted depending on the row length of the work table. For details about work table row lengths, see the *HiRDB Version 9 Installation and Design Guide*.

(6) Specification examples

1. Provide a cursor specification in a cursor declaration:

```
DECLARE CR1 CURSOR FOR
  SELECT PNAME,SUM(SQUANTITY)
    FROM STOCK
```

```
GROUP BY PNAME
ORDER BY PNAME ASC
```

2. Specify a LIMIT clause in a cursor specification in a cursor declaration:

```
DECLARE CR1 CURSOR FOR
  SELECT PCODE,SQUANTITY
    FROM STOCK
    WHERE SQUANTITY>20
    ORDER BY 2,1
    LIMIT 10
```

2.1.2 Cursor specification: Format 2

(1) Function

This format is used when a table is to be retrieved by means of a list.

A cursor specification for a table retrieved by means of a list is specified in a dynamic SELECT statement (for details about the dynamic SELECT statement, see *Dynamic SELECT statement Format 1 (Retrieve dynamically)* in *4. Data Manipulation SQL*).

(2) Privileges

A user with the SELECT privilege for a base table can execute cursor specification: format 2 to retrieve that base table by means of a list.

(3) Format

```
SELECT {{value-expression | WRITE specification | GET_JAVA_STORED_ROUTINE_SOURCE specification}
 [[AS] column-name]
[,{value-expression | WRITE specification|GET_JAVA_STORED_ROUTINE_SOURCE specification}
 [[AS] column-name]]...
 |*}
FROM LIST list-name
[ORDER BY {column-name | sort-item-specification-number} [{ASC | DESC}]
[,{column-name | sort-item-specification-number} [{ASC | DESC}]]... ]
[LIMIT {[offset-of-first-row-to-return,] {limit-row-count | ALL}
| {limit-row-count | ALL} [OFFSET offset-of-first-row-to-return]}]
```

(4) Operands

- {{*value-expression* | WRITE *specification*
  |GET_JAVA_STORED_ROUTINE_SOURCE *specification*} [[AS] *column-name*]

  [, {*value-expression* | WRITE *specification*
  |GET_JAVA_STORED_ROUTINE_SOURCE *specification*} [[AS] *column-name*]]...

  |*}

value-expression

Specifies the value expression to be retrieved.

The following rules apply to the value expressions in the SELECT clause:

1. In *column-name*, specify the column name of the base table of the list to be retrieved.
2. In a repetition column, specify a repetition column in the base table of the list to be retrieved.
3. If a repetition column is specified singly, the ANY subscript cannot be specified.
4. In an attribute name, specify the attribute of the abstract data type for the base table of the list to be retrieved.
5. The following items cannot be specified in a value expression in the SELECT clause:
   - Tables specified in column specifications of a base table
   - External references to (a column in) a base table
   - Set functions
   - Window function
   - Subqueries
6. Some receive functions for passing inter-function values can be specified without specifying a receive function for passing inter-function values for a plug-in provided function. For details about receive functions for passing inter-function values that can be specified, see the *HiRDB Version 9 UAP Development Guide* and various plug-in manuals.
7. Only one receive function for passing inter-function values for a plug-in provided function can be specified.

WRITE *specification*

Specifies a WRITE specification to enable BLOB data retrieval results to be output to a file (for details about the WRITE specification, see *2.22 WRITE specification*).

GET_JAVA_STORED_ROUTINE_SOURCE *specification*

Specify when a Java class source file is to be extracted from the JAR file (for details, see *2.23 GET_JAVA_STORED_ROUTINE_SOURCE specification*).

[AS] *column-name*

Specifies a name to be assigned to *value-expression*, WRITE *specification*, or GET_JAVA_STORED_ROUTINE_SOURCE *specification*.

*

Specifies that all columns in the base table are to constitute the list that is to be retrieved.

- *list-name*

Specifies the name of the list storing the set to be retrieved.

- ORDER BY {*column-name* | *sort-item-specification-number*} {ASC | DESC}

For rules on the ORDER BY clause, see Format 1.

- LIMIT {[*offset-of-first-row-to-return*,] {*limit-row-count* | ALL} | {*limit-row-count* | ALL} [OFFSET *offset-of-first-row-to-return*]}

For rules on the LIMIT clause, see Format 1.

(5) Common rules

1. If a row that was found when a list was created is not found during retrieval processing, the system returns SQL code +110 and continues the retrieval. However, when one of the following conditions is applicable, the system does not return SQLCODE + 110:
   - A column name in the base table for the list other than an argument for a receive function for passing inter-function values is specified, and a * is not specified for a column in the base table for the list.
   - An ORDER BY clause is specified.
2. If the LIMIT clause specifies an offset, the row that returns SQLCODE + 110 is included in the number of rows to be skipped.
3. If the LIMIT clause specifies a limit row count, the row that returns SQLCODE + 110 is included in the number of rows to be acquired.
4. A user cannot manipulate lists by connecting to multiple copies of HiRDB at the same time.

(6) Notes

1. A row in the base table that is deleted after a list is created is not retrieved. However, if only the following items are specified in a selection expression, the system retrieves the data in the row in the base table before the row is deleted:
   - A value expression that does not include the column name of the table
   - A receive function for passing inter-function values
2. If a row in the base table is updated after a list is created, the updated data will be retrieved. Note that a receive function for passing inter-function values, specified in a selection expression, retrieves un-updated data from a row in the base table.
3. If a row in the base table is deleted and then inserted after a list is created, the inserted row may be retrieved.

4. From the time the SQL that retrieves a table by means of a list is processed by a PREPARE statement to the time the OPEN statement is executed, you must not execute an ASSIGN LIST statement that creates a list with the same list name.
5. When you specify a WRITE specification, you can specify in the first argument the name of the base table's BLOB type column or the BLOB type attribute name of the abstract data type.

(7) Specification example: Specifying a cursor specification in a cursor declaration

```
PREPARE P1 FROM
 'SELECT PCODE,PNAME FROM LIST LIST1'
DECLARE CRL1 CURSOR FOR P1
```

2.2 Query expressions

2.2.1 Query expression format 1 (general-query-expression)

(1) Function

A query specification specifies a combination of a derived query expression and the query expression itself in the `WITH` clause. When the `WITH` clause is used, the derived table produced by the derived query expression can be the query name, which can be specified in the query expression itself.

The query expression body specifies either a query specification or a set operation in order to determine the union set or the disjunction set between the derived tables that are obtained as a result of a query specification. `UNION [ALL]` must be specified when a union set is to be determined. `EXCEPT [ALL]` must be specified when a disjunction set is to be determined. A query expression can be specified in either a cursor declaration or a cursor specification in a dynamic `SELECT` statement.

(2) Format

query-expression ::= [WITH *query-name* [(*column-name* [, *column-name*] ...)]
AS (*derived-query-expression*)
[, *query-name* [(*column-name* [, *column-name*] ...)]
AS (*derived-query-expression*)] ...]
query-expression-body
derived-query-expression ::= *query-expression-body*
query-expression-body ::= {*query-specification*
| (*query-expression-body*)
| *query-expression-body* {UNION | EXCEPT} [ALL]
{*query-specification* | (*query-expression-body*) }}

(3) Operands

- `WITH` *query-name* [(*column-name*[, *column-name*]...)]

`WITH` *query-name*

Specifies the name of the derived table to be specified in the table expression of the query expression body. Multiple query names specified in a `WITH` clause must all be unique.

column-name

Specifies a column name that corresponds to a column to be derived from the derived table specified by one of the derived query expressions in the `WITH` clause.

The following rules apply to column names:

1. If no column name is specified, the default is determined as follows:
   - If no set operation is specified in the derived query expression in the WITH clause, the column name of the column in the derived table specified by the query specification in the WITH clause (if AS *column-name* is specified, the column name specified in the AS clause) becomes the column name of the query name for the WITH clause.
   - If a set operation is specified in the derived query expression in the WITH clause, the column name of the column in the derived table specified by the first query specification in the derived query expression (if AS *column-name* is specified, the column name specified in the AS clause) becomes the column name of the query name for the WITH clause.

   If the derived table specified in the derived query expression in the WITH clause contains two or more columns with the same column name or contains a column that does not have a name, *column-name* cannot be omitted.
2. If a column in the derived table has been derived from any of the items listed below and AS *column-name* is omitted, that column becomes a nameless column:
   - Scalar operation
   - Function call
   - Set function
   - Literal
   - USER
   - CURRENT_DATE
   - CURRENT_TIME
   - CURRENT_TIMESTAMP[(*p*)]
   - SQL variable
   - SQL parameter
   - Component specification
3. Each query in a WITH clause must have a unique name.
4. The number of columns specified by column names in a query name in a WITH clause must be the same as the number of columns in the derived table obtained as a result of the derived query expression in the WITH clause.
5. A maximum of 30,000 columns can be specified by column names for a single query name in a WITH clause.

■ *derived-query-expression*::=*query-expression-body*

Specify a table expression, a column to be derived as a query name, a set operation, or whether duplicate-row exclusion is on. In the FROM clause contained in a table expression in *query-expression-body*, specify either a table name or a view table name.

- *query-expression-body*::={*query-specification*|(*query-expression-body*)|*query-expression-body* {UNION|EXCEPT}[ALL] {*query-specification*|(*query-expression-body*)}}

When the WITH clause is used, specifies a table name, a view table name, or a query name in the FROM clause contained in the table expression in the query expression body.

When the WITH clause is used, a single-row SELECT statement cannot be specified in the query expression body.

- *query-specification*

Specifies the table expressions, the columns to be selected, or whether or not duplicates elimination is to be in effect for the rows to be selected.

See *2.3 Query specification* for details about query specifications.

- (*query-expression-body*)

Specifies the order in which the set operations in the query expressions are to be evaluated when two or more query specifications joined by UNION or EXCEPT are specified (*query-expression-body* must be specified in parentheses).

- *query-expression-body* {UNION|EXCEPT} [ALL] {*query-specification*|(*query-expression-body*)}

Specifies that the union set or the difference set of derived tables produced by two query specifications or query expression bodies is to be determined. For details about set operations, see *(6)* below.

(4) Rules for derived query expressions in WITH clauses

1. The attributes of columns specified as query names for a derived query expression in a WITH clause (presence of data type, data length, and NOT NULL constraint) are the same as the attributes of columns corresponding to the derived table specified for a derived query expression in a WITH clause.
2. Specify the base table name or view table name to be used in the FROM clause of a derived query expression in a WITH clause. A query name cannot be specified.
3. [*table-specification*.]ROW cannot be specified in a selection expression of a derived query expression specified in the WITH clause.
4. A subscripted repetition column cannot be specified in a directly included SELECT clause in a derived query expression specified in the WITH clause.
5. The following value expressions cannot be specified in a derived query

expression specified in the WITH clause:

- XML constructor function
- SQL/XML scalar function
- SQL/XML predicate
- SQL/XML set function

6. A value expression other than a column specification cannot be specified in a GROUP BY clause of a derived query expression specified in a WITH clause.
7. If a CASE expression is specified in a selection expression of the outermost query specification specified in a WITH clause, a repetition column cannot be specified in that CASE expression's search conditions.
8. The following items cannot be specified in a selection expression in a derived query expression specified in a WITH clause:
   - WRITE specification
   - GET_JAVA_STORED_ROUTINE_SOURCE specification
   - Window function

(5) Rules for the query expression body

1. When the WITH clause is used, a table name, view table name, or query name must be specified in the FROM clause contained in the table expression in the query expression body.
2. When the WITH clause is used, single-row SELECT statements cannot be specified in the query expression body.
3. When a column derived by a derived query expression in a view definition or WITH clause is a set function whose argument is a value expression, the derived column cannot be used for external referencing in the query expression body.

   Example

```
WITH QRY1 (QC1,QC2) AS (SELECT MAX(C1+C2),AVG(C1+C2)
                              FROM T1),
     QRY2 (QC1,QC2) AS (SELECT C1+C2,C3-C4 FROM T2)
  SELECT * FROM QRY1 X
     WHERE QC1>(SELECT QC1 FROM QRY2 WHERE QC2 = X.QC2)
```

4. If the column derived from a view definition or a derived query expression specified in a WITH clause is COUNT(*) or COUNT_FLOAT(*), that derived column cannot be used as an external reference in the query expression body.

   Example

```
WITH QRY1 (QC1) AS (SELECT COUNT(*) FROM T1),
     QRY2 (QC1,QC2) AS (SELECT C1+C2,C3/C4 FROM T2)
```

```
SELECT * FROM QRY1 WHERE EXISTS
   (SELECT * FROM QRY2 WHERE QC1 = QRY1.QC1)
```

5. When you specify a table derived by specifying a value expression with any of the following attributes in the query expression body in a selection expression in the outermost query specification in a WITH clause, you cannot specify a query that creates an internally derived table (for the circumstances under which an internally derived table can be created, see *2.21 Inner derived tables*):
   - BLOB
   - BINARY with minimum length of 32,001 bytes
   - Abstract data type
   - Repetition column
6. If a column derived from a function invocation, component specification, SQL function, or SQL parameter with the following attributes is specified in a selection expression in the query expression body, FOR READ ONLY cannot be specified in a query that directly includes the SELECT clause:
   - BLOB
   - BINARY with a minimum defined length of 32,001 bytes
   - Abstract data type
7. If a column derived from a view definition or a derived query expression specified in a WITH clause is derived from a value expression other than a column specification, a component specification cannot be specified for that derived column.
8. If a column derived from a view definition or a derived query expression specified in a WITH clause is derived with the component specification, that derived column cannot be used as an external reference in the query expression body.
9. An error occurs if either a function that receives inter-function values (receive function for passing inter-function values) or a function that sends inter-function values (send function for passing inter-function values) of the functions provided by a plug-in is specified and any of the following applies to the query in the defined view table:
   - The CASE clause in the view definition or the receive function for passing inter-function values specifies the scalar function VALUE.
   - The view definition specifies a receive function for passing inter-function values for which there are multiple instances of the send function for passing inter-function values, and the first arguments of the send functions for passing inter-function values are the same.

- The view definition specifies either the receive function for passing inter-function values or the send function for passing inter-function values, and the query specification that specified the view table creates an inner derived table. For details about the conditions for creating an inner derived table, see *2.21 Inner derived table*.
- The send function for passing inter-function values that corresponds to the receive function for passing inter-function values specified in the view definition does not exist in either the view definition or the query specification that specified the defined view table.
- The send function for passing inter-function values that corresponds to the receive function for passing inter-function values in the query specification that specified the view table does not exist in either the query specification or the query that defined the view table.

(6) Rules common to set operations

1. When a set operation is performed to obtain a union of sets or the difference between sets, the system performs the set operation by treating the two query specifications, derived query expressions, or query expression bodies that are subject to the operation as a set of rows in the derived table that is obtained as the result.

   Therefore, the derived tables that are subject to these set operations must have the same number of constituent columns in the same order. Similarly, the corresponding columns must have compatible data types for comparison operations. However, set operations cannot be performed between the following items: character string representations of date data items; character string representations of time data items; character string representations of time stamp data items; decimal representations of date interval data items; and decimal representations of time interval data items.
2. If the columns of a derived table that are objects of a set operation are of the character string data type, make sure the character set specified for all of the corresponding columns is the same. However, if the columns of a derived table specified in a second or later query specification are the value expression listed below, they are converted to the character set of the columns of the derived table specified in the first query specification:
   - Character string literal
3. The column names of a table derived by a set operation become the column names of the columns of the derived table specified by the first query specification of the query expression (if `AS` *column-name* is specified, the column name specified in the `AS` clause).
4. If a column in the derived table has been derived from any of the items listed below and `AS` *column-name* is omitted, that column becomes a nameless column:

- A value expression that includes an arithmetic operation, date operation, time operation, concatenation operation, or system built-in scalar function
- `CASE` expression
- `CAST` specification
- Set function
- Literal
- `USER`
- `CURRENT_DATE`
- `CURRENT_TIME`
- `CURRENT_TIMESTAMP[(`*p*`)]`
- SQL variable
- SQL parameter
- Component specification
- Function call

5. A derived table obtained as a result of a set operation has the same number of constituent columns in the same order as the derived table that was subject to the set operation.
6. Literals and the results of a set function, date operation, time operation, arithmetic operation, `CASE` expression, or `CAST` specification cannot be `NOT NULL` constrained (the null value must be allowed).
7. In a set operation, a query expression body cannot be specified in the `FROM` clause of a query specification.
8. When a set operation is performed, a value expression that produces a result with any of the following data types cannot be specified in a constituent column in the derived table that is subject to the set operation:
   - `BLOB`
   - `BINARY` with a minimum length of 32,001 bytes
   - `BOOLEAN`
   - Abstract data type
9. When a set operation is specified, a repetition column cannot be included in the columns in the derived table that are subject to the set operation.
10. When a set operation is specified, you cannot specify a `WRITE` specification, a `GET_JAVA_STORED_ROUTINE_SOURCE` specification, or the window function

for a column of the target-derived table.

11. The results of a set operation produce the following data types and data lengths:

    For character data, national character data, or mixed character data:

    - If any of the value expressions contains variable-length data, the result will be variable-length data.
    - The data length of the result will be the data length of the value expression with the largest data length.
    - If value expressions contain both character data and mixed character data, the data type of the result will be mixed character data.

    For numeric value data:

    The following table indicates the data types of the results from set operations involving numeric data:

Operand 1	Operand 2				
	SMALLINT	**INTEGER**	**DECIMAL**	**SMALLFLT**	**FLOAT**
SMALLINT	SMALLINT	INTEGER	DECIMAL	SMALLFLT	FLOAT
INTEGER	INTEGER	INTEGER	DECIMAL	FLOAT	FLOAT
DECIMAL	DECIMAL	DECIMAL	DECIMAL	FLOAT	FLOAT
SMALLFLT	SMALLFLT	FLOAT	FLOAT	SMALLFLT	FLOAT
FLOAT	FLOAT	FLOAT	FLOAT	FLOAT	FLOAT

    Set operation results of the `DECIMAL` data type have the following precision and scaling, where p and s denote the precision and scaling of operand 1, and p_2 and s_2 denote the precision and scaling of operand 2:

    $\texttt{Precision} = \max(p_1 - s_1, p_2 - s_2) + \max(s_1, s_2)$

    $\texttt{Scaling} = \max(s_1, s_2)$

    The specified values of the `pd_sql_dec_op_maxprec` operand are used to determine the precision as follows.

pd_sql_dec_op_maxprec operand value	Precision of result column of set operation operand and any corresponding columns	Precision obtained with $max(p_1 - s_1, p_2 - s_2) + max(s_1, s_2)$	Precision of result column of set operation operand
29 or omitted	All columns are 29 or fewer digits wide	29 or fewer digits	Precision obtained with $max(p_1 - s_1, p_2 - s_2) + max(s_1, s_2)$
		30 or more digits	Error
	Includes columns that are 30 or more digits wide	38 or fewer digits	Precision obtained with $max(p_1 - s_1, p_2 - s_2) + max(s_1, s_2)$
		39 or more digits	Error
38	Any	38 or fewer digits	Precision obtained with $max(p_1 - s_1, p_2 - s_2) + max(s_1, s_2)$
		39 or more digits	Error

For details about the pd_sql_dec_op_maxprec operand, see the manual *HiRDB Version 9 System Definition*.

INTEGER is treated as DECIMAL (10,0); SMALLINT is treated as DECIMAL (5,0). If all corresponding constituent columns are NOT NULL constrained, the results are treated as NOT NULL constrained. Otherwise, the results are treated as not being NOT NULL constrained.

Time stamp data

Set operations can be performed between time stamp data items, producing the same data type as the source data type. If terms of a set operation include a fractional second precision, the fractional second precision of the result is max(p_1, p_2), where, given Q_1 *set operation* Q_2, p_1 is the fractional second precision of Q_1, and p_2 is the fractional second precision of Q_2.

Binary data

The data length of the result is the data length of the value expression with the largest data length.

12. When ALL is specified, duplicated rows are left intact and handled as separate rows. When ALL is omitted, duplicated rows are consolidated into a single row (duplications are eliminated).

The number of duplicated rows resulting from the operation "Q1 set-operation Q2" depends on whether ALL is specified, where Q1 denotes a query expression body (derived query expression) and Q2 denotes a query expression body (derived query expression) or a query specification.

Let *R* represent the duplicate rows in Q1 or Q2, or Q1 and Q2; let m be the number of rows in Q1 in the duplicate row *R*; and let *n* be the number of rows in Q2 ($m \geq 0, n \geq 0$). If Q1 or Q2, or Q1 and Q2 do not contain duplicated rows, the system performs the set operation by assuming that there is one duplicated row *R* ($m = 1, n = 0$, or $m = 0, n = 1$). The following table indicates the number of duplicate rows *R* resulting from each set operation:

Set operation	ALL not specified	ALL specified
UNION	min(1, $n + m$)	$m + n$
EXCEPT	1 (m>0 and n=0) 0 (m=0 or n>0)	max($m - n$, 0)

(7) Note

1. HiRDB may create a work table when the following conditions are satisfied:

   - Either UNION[ALL] or EXCEPT[ALL] is specified.

   In this operation, the above processing can be restricted depending upon the row length of the work table. For details about work table row lengths, see the *HiRDB Version 9 Installation and Design Guide*.

(8) Specification examples

WITH clause used

```
DECLARE CR1 CURSOR FOR
  WITH QRY1(QPCODE,QPNAME,QCOLOR,QSALES) AS
    (SELECT PCODE,PNAME,COLOR,PRICE*SQUANTITY FROM STOCK)
      SELECT QPNAME,MAX(QSALES) FROM QRY1 GROUP BY QPNAME
```

WITH clause not used

```
DECLARE CR1 CURSOR FOR
  SELECT PCODE,PNAME,COLOR,PRICE,SQUANTITY
     FROM STOCK
```

2.2.2 Query expression format 2 (unnesting query expression for repetition columns)

(1) Function

This facility allows you to split a given repetition column by element and retrieve each element as a separate row.

In addition, this facility, which is called the *unnesting facility for repetition columns*, can retrieve elements in a repetition column having the same subscript as the same row.

(2) Format

```
SELECT [ALL | DISTINCT] {selection-expression
                [,selection-expression] ...| * }
FROM [authorization-identifier.] table-identifier
                [IN (RDAREA-name-specification)]
      (FLAT (column-name [,column-name] ...))
     [[AS] correlation-name]
      [used-index-SQL-optimization-specification]
      [WHERE search-condition]
      [GROUP BY column-specification [,column-specification]...]
      [HAVING search-condition]
```

(3) Operands

For details about selection expressions, FLAT, and operands other than search conditions, see *2.2.1 Query expression format 1 (general-query-expression)*.

- {*selection-expression* [,*selection-expression*]...| * }

Specifies the item to be output as a search result.

For selection expressions, see *2.3 Query specification*.

- [IN (*RDAREA-name-specification*)]

IN

Specifies the RDAREA to be accessed. This operand cannot be specified in the following cases:

- If the table specified in the table identifier is a read-only view table
- While in a derived query expression of a view definition

RDAREA-name-specification::=*value-specification*

Use a value specification of the VARCHAR type, CHAR type, MVARCHAR type, or MCHAR type to specify the RDAREA name to be used as the RDAREA for storing the table specified in the table identifier. To specify multiple RDAREA names, separate them with commas (,). The same name cannot be used to specify another RDAREA name; doing so results in an error. For details about the characters allowed in an RDAREA name specified using a value specification, see *1.1.7 Specification of names*. Spaces before or after an RDAREA name specified using a value specification are ignored. However, if the RDAREA name is enclosed in double quotation marks ("), only the spaces outside the double quotation marks

are ignored.

When specifying an RDAREA while using the inner replica facility, specify the original RDAREA name. To target a replica RDAREA, use the change current database command (`pddbchg` command) or `PDDBACCS` operand of the client environment definition to change the RDAREA to be accessed to the replica RDAREA.

- (`FLAT` (*column-name* [, *column-name*]...))

Specifies the repetition column or the list of columns containing repetition columns that are the object of flattening.

If multiple repetition columns are specified, the flattening is performed so that elements of the same subscript are contained in the same row.

If the specified repetition column also contains a normal column (a non-repetition structure column), the system generates rows having the same value with respect to any element in the repetition column.

After being flattened, any repetition column or a set of columns containing a repetition column becomes a normal column. Therefore, the columns must be specified as a normal column in that query expression.

- *search-condition*

In *search-condition*, you can specify items that have been flattened.

For search conditions, see *2.7 Search conditions*.

(4) Rules

1. The following table describes the relationship between columns to be flattened and the index.

<table>
<tr><th colspan="4">Relationship between a column specified in a column name and an index</th><th>SQL statement executable?</th></tr>
<tr><td rowspan="4">Index definition provided</td><td rowspan="3">Has an index that contains a repetition column.</td><td rowspan="2">There is an index that covers all the columns specified in the SQL statement among the columns specified in a <code>FLAT</code> column name.</td><td>The <code>FLAT</code> column name contains at least one repetition column.</td><td>Y</td></tr>
<tr><td>The <code>FLAT</code> column name does not contain any repetition column.</td><td>N</td></tr>
<tr><td colspan="2">There is no index that covers all columns specified in the SQL statement, among the columns that are specified in a <code>FLAT</code> column name.</td><td>N</td></tr>
<tr><td colspan="3">No index that contains a repetition column</td><td>N</td></tr>
<tr><td colspan="4">No index definition</td><td>N</td></tr>
</table>

Legend: Y: executable, N: error

2. If WITHOUT INDEX is specified in the SQL optimization specification for the index being used, the system ignores the optimization specification and performs retrievals using the index that is available to HiRDB.
3. When specifying WITH INDEX in the SQL optimization specification for the index being used, specify an index that satisfies the SQL statement execution-enabling condition given in Rule 1. If the specified index does not satisfy the SQL statement execution-enabling condition, the system ignores the optimization specification and performs retrievals using the index that is available to HiRDB.
4. In the FROM clause, a FLAT specification cannot be specified in the SELECT statement of INSERT SELECT, a WITH clause query, a view definition, a derived table in the FROM clause, or a subquery.
5. The maximum number of column names that can be specified in FLAT is 16.
6. FLAT must be used to specify column names in an SQL statement that specifies flattening.
7. In *table-identifier*, a view table cannot be specified.
8. The following items cannot be specified with flattening:
   - FOR UPDATE clause
   - FOR READ ONLY
   - LIMIT clause
   - Component specifications
   - Function calls
   - Subqueries
   - Set functions with a FLAT specification
   - Subscripted columns
9. The following items cannot be specified in a search condition with flattening:
   - Structured repetition predicates
10. If you specify the RDAREA name specification, you cannot use an index that has a different number of partitions than the table has. When defining an index for a query that specified an RDAREA name, define an index with the same number of partitions as the table has.

(5) Example

Flatten a table of test scores and retrieve a list of students who scored 70 points or higher.

```
SELECT name, subject, score, FROM test-score-table (FLAT
(name, subject, score))
    WHERE score >=70
```

Score table

Name	Subject	Grade
Yates	Math	90
	English	65
Saunders	Math	50
	English	90
Somner	Math	85
	English	70

Retrieval results

Name	Subject	Grade
Yates	Math	90
Saunders	English	90
Somner	Math	85
Somner	English	70

The following table and index definition apply to the test score table:

```
Table definition:
    CREATE TABLE test-score-table (name,MCHAR(10),
                        subject MCHAR(10) ARRAY[4],
                        score SMALLINT ARRAY[4]);
Index definition:
    CREATE INDEX subject score ON test-score-table (name,
subject, score);
```

2.3 Query specification

(1) Function

A query specification derives a table composed of retrieval results containing columns of selection expressions, by specifying retrieval conditions (table expressions) for a table and items (selection expressions) in which the retrieval results are to be output. The table obtained in this manner is called a derived table.

(2) Privileges

A user with the SELECT privilege for a table can execute a query specification to retrieve that table.

(3) Format

```
SELECT[{ALL|DISTINCT}]{selection-expression[, selection-expression]...
      |*}table-expression
```

(4) Operands

- [{ALL|DISTINCT}]

For a retrieval that produces duplicated rows (identical rows composed of items specified in the selection expressions), specifies whether the duplicates are to be retained or eliminated.

Eliminating duplicate rows is called duplicates exclusion.

ALL

Specifies that duplicated rows are to be retained in the retrieval results.

DISTINCT

Specifies that duplicated rows are to be output as a single row.

- {*selection-expression*[, *selection-expression*]...|*}

Specifies the items to be output as retrieval results.

selection-expression

The following can be specified as a selection expression:

- Column specification
- [*table-specification*.]ROW
- Set functions
- Window function
- Scalar functions

- CASE expressions
- CAST specification
- Literals
- USER
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP[(*p*)]
- SQL variable or SQL parameter
- Value expressions # [[AS] *column-name*]
- *table-specification*.#
- Component specification
- Function call#
- WRITE specification
- GET_JAVA_STORED_ROUTINE_SOURCE specification
- Scalar subquery

#: The ? parameter and embedded variables can be specified as arguments in a function call. For the specification details, see 3 in *(5)Rules* below. ? parameters and embedded variables can be specified in the arguments for the scalar functions LENGTH, SUBSTR, and POSITION. See the rules on these functions for specification methods.

The following rules apply to selection expressions:

1. The ? parameter and the embedded variable can be specified in the arguments for the scalar functions LENGTH, SUBSTR, and POSITION. The ? parameter and the embedded variable can be specified in the argument for LENGTH, in the first argument of SUBSTR, or in the first and second arguments of POSITION only if their data type is BLOB or BINARY, and if the AS data type is specified.
2. Value expressions that yield the following type of result cannot be specified in a selection expression:
   - BOOLEAN
3. Value expression that yield results of the abstract data type cannot be specified in a cursor declaration, a dynamic SELECT statement, a single-row SELECT statement, a derived table in a FROM clause, or a subquery in a predicate.
4. When SELECT DISTINCT is specified, a set function with the DISTINCT

specification cannot be specified in a selection expression or table expression.

5. When a repetition column is specified in a selection expression, none of the following items can be specified:
   - SELECT DISTINCT
   - UNION [ALL] with respect to that query
   - EXCEPT [ALL] with respect to that query
6. [*table-specification*.]ROW can be specified only for a base table with the FIX attribute. ROW refers to an entire row, and specifying ROW retrieves into one area an entire row as a single data item. Regardless of the data types of the columns that compose the row, the data type of the row that is retrieved is the ROW type. (Either a variable associated with CHAR(*n*), where *n* denotes the row length, or a structure can be specified for the ROW type; however, the structure specified for the ROW type must not contain spaces for boundary alignment.) The data length of the resulting row is its row length (sum of the data lengths of the columns that compose the row). If ROW is specified in a selection expression, none of the following can be specified in a query specification related to the selection expression:
   - Set functions
   - GROUP BY clause
   - UNION [ALL] or EXCEPT [ALL]
7. [*table-specification*.]ROW cannot be specified for a table that includes columns of character string data type that use other than default character set.
8. When using the ROW type, make sure that the platform on which the UAP runs and the platform on which the HiRDB server runs have the same endian. The ROW type cannot be used between applications using different endian types. For example, if the ROW type is used in a Windows UAP, the HiRDB server must also use the same endian order in the Windows edition.
9. [*table-specification*.]ROW cannot be specified for an inner table of a joined table.
10. Any column specified in a SELECT clause must reference the table that is specified in the FROM clause of that table expression.
11. If a set function, GROUP BY clause, or HAVING clause is specified in a table expression, any column specification in the SELECT clause must be one of the following:
    - Grouping column (value expression specified in the GROUP BY clause)
    - Specification in an argument of a set function
12. If an SQL variable or SQL parameter of any of the following data types is

specified in a selection expression, the functions and clauses indicated below cannot be specified in a query specification that directly includes the SELECT clause or in a query that includes the SELECT clause:

Data type of SQL variable or SQL parameter

- BLOB
- BINARY with a minimum defined length of 32,001 bytes
- BOOLEAN
- Abstract data type

Functions and clauses that cannot be specified simultaneously in a query specification

- Set functions
- GROUP BY clause
- HAVING clause
- DISTINCT
- Specifying in the FROM clause a view table specifying any of the facilities described previously or clauses in a derived query expression in a view definition.

Functions and clauses that cannot be specified simultaneously in a query

- Set operations
- ORDER BY clause
- FOR UPDATE clause
- FOR READ ONLY clause

13. Specifying *table-specification*.* means that all columns in the specified table are to be output in the order in which they were specified when the table was defined.
14. AS *column-name* is specified in order to assign a name to a selection expression.
15. When AS *column-name* is specified in the n^{th} selection expression, the column name will be the n^{th} column name in the table that is derived by the query specification that specified the selection expression.
16. When AS *column-name* is not specified in the n^{th} selection expression, the column name that is specified in the selection expression will be the n^{th} column name in the table that is derived by the query specification. If the selection expression contains more than the column specification, the n^{th} column will be a column without a name.

17. If a scalar subquery is specified in an *n*-th selection expression, the *n*-th column name derived by the query specification will be the column name that is derived by a query specification of the scalar subquery. However, if an AS column name is specified, the column name will be the one specified in AS *column-name*. If the column derived by the query specification in the scalar subquery is a nameless column, the *n*-th column that is derived by the query specification will be a nameless column.
18. A WRITE specification can be specified alone in a selection expression.
19. A GET_JAVA_STORED_ROUTINE_SOURCE specification can be specified as follows:
    - Specified alone in a selection expression that is specified in the outermost query
    - Specified as an argument in the LENGTH scalar function in a selection expression that is specified in the outermost query
20. DISTINCT is mutually exclusive with a WRITE specification or GET_JAVA_STORED_ROUTINE_SOURCE specification.
21. A WRITE specification and GET_JAVA_STORED_ROUTINE_SOURCE specification cannot be specified in a selection expression in a query specification in an INSERT statement.
22. If a GROUP BY clause specifying a value expression other than a column specification is specified in a table expression, that grouping column cannot be referenced from the scalar subquery that is specified in the selection expression.
23. When the window function is specified as a selection expression, you must specify at least one selection expression in addition to the window function.
24. You cannot specify the window function in a scalar operation.
25. When the window function is specified, you cannot specify a set function as a selection expression.

*

Specifies that all table columns are to be output.

Specification of an asterisk indicates that all columns in all the tables specified in the FROM clause of the query specification are specified in the order of their specification in the table specified in the FROM clause. In this case, the table columns are arranged in the same order as when they were specified during table definition.

- *table-expression*

A table expression specifies one or more tables or externally joined tables that are the objects of retrieval. In a table expression it is also possible to specify the conditions

under which tables are to be searched or joined (search conditions and grouping). For details about table expressions, see *2.5 Table expressions*.

For details about the restrictions for a derived query expression in a view definition, see *CREATE [PUBLIC] VIEW (Define view, define public view)* in Chapter 3.

(5) Rules

1. In a query specification with respect to the named derived table that has been derived from a value expression specifying the following attributes in the selection expression in the outermost query, a query for creation of an internally derived table cannot be specified (for the conditions under which an internally derived table can be created, see *2.21 Inner derived tables*):
   - `BLOB`
   - `BINARY` with a minimum length of 32,001 bytes
   - Abstract data type
   - Repetition column
2. When a `?` parameter or an embedded variable is specified as an argument in a function call, the argument must be specified in the following format:
   - `?` `AS` *data-type*
   - :*embedded-variable* [`:` *indicator-variable*] `AS` *data-type*

(6) Notes

1. When retrieving or updating entire rows (`ROW` specification), the length of the date data type that is specified in an embedded variable, an SQL variable, or an SQL parameter relative to `ROW` is four bytes in `X'`*YYYYMMDD*`'` format.

   If date data is passed or received in a character expression using a row-by-row basis (`ROW` specification) interface, the column must be defined as `CHAR(10)`, rather than as a date data column. Any date operation must be specified by converting the data into the date data type with the `DATE` scalar function.
2. When retrieving or updating entire rows (`ROW` specification), the length of the time data type that is specified in an embedded variable, an SQL variable, or an SQL parameter relative to `ROW` is three bytes in `X'`*HHMMSS*`'` format.

   If time data is passed or received in a character expression using a row-by-row basis (`ROW` specification) interface, the column must be defined as `CHAR(8)`, rather than as a time data column. Any time operation must be specified by converting the data into the time data type with the `TIME` scalar function.
3. When retrieving or updating entire rows (`ROW` specification), the length of the time stamp data type that is specified in an embedded variable, an SQL variable, or an SQL parameter relative to `ROW` is (7 + (*p* ÷ 2)) bytes in

X'*YYYYMMDDhhmmss*[*nn...n*]' format. For passing or receiving time stamp data in character representations using a row-by-row interface (ROW specification), columns must be defined in CHAR of length 19, 22, 24, or 26 bytes instead of the time stamp data type.

4. HiRDB may create a work table when any of the following conditions is satisfied:

   - DISTINCT is specified.
   - A value expression containing a set function is specified in a selection expression.
   - A value expression containing the window function is specified in a selection expression.

   In this process, the above processing may be subject to some restrictions depending on the row length of the work table. For details about work table row lengths, see the *HiRDB Version 9 Installation and Design Guide*.

2.4 Subqueries

(1) Function

A subquery specifies the value, or the set of values, that can be obtained by retrieving a table.

The following types of subqueries are available:

- Scalar subquery

  A scalar subquery is a subquery that yields a result column count of 1 and a result row count of 1 or less.

- Row subquery

  A row subquery is a subquery that yields a result column count of 2 or greater and a result row count of 1 or less.

- Table subquery

  A table subquery is a subquery that yields a result column count of 1 or greater, and a result row count of 0 or greater.

Subqueries can be specified in the following locations:

Scalar subquery

- Anywhere a value expression can be specified

Row subquery

- Anywhere a row value constructor can be specified
- `SET` clause in the `UPDATE` statement

Table subquery

- Right-hand side of the `IN` predicate
- Right-hand side of a quantified predicate
- `EXISTS` predicate
- Derived table in the `FROM` clause

(2) Privileges

A user with the `SELECT` privilege for a table can execute a subquery to retrieve that table.

(3) Format

scalar-subquery::=*subquery*
row-subquery::=*subquery*

table-subquery::=*subquery*

subquery::=([*SQL-optimization-specification-for-subquery-execution-method*]*query-expression-body*)

(4) Operands

- ([*SQL-optimization-specification-for-subquery-execution-method*]*query-expression-body*)

For SQL optimization specification for a subquery execution method, see *2.24 SQL optimization specification*. For details about a query specification body, see *2.2 Query expressions*.

(5) Rules on subqueries for a derived table in a predicate (IN predicate, comparison predicate, quantified predicate, or EXISTS predicate) or in a FROM clause

1. Value expressions that yield results of the following data types cannot be specified in a selection expression in a predicate or in a subquery in a derived table:
   - BLOB
   - BINARY with a minimum length of 32,001 bytes
   - Abstract data type
2. If either * or *table-specification.** is specified in a subquery in the EXISTS predicate, the specification means the following:
   - When a set operation is not specified:

   The specification means any one column that is allowed in the subquery other than a set function.
   - When a set operation is specified:

   The * specification means specifying all column in all the tables specified in the FROM clause of that query specification in the order of the tables specified in the FROM clause. In each table, columns are ordered in the sequence that was specified at the time of table definition.

   The *table-specification*.* specification means specifying all columns in all the specified tables in the order in which they were specified at the time of table definition.

(6) Rules

1. The results of the subquery is returned without the NOT NULL constraint (allowing null values). However, the results of a table subquery specified in a derived table in the FROM clause are subject to the same constraint as the results of that query

expression.

2. The maximum number of columns of the results of a row subquery specified in a row value constructor is 255.
3. The maximum number of columns in the results of a row subquery specified in the SET clause of UPDATE is 30,000.
4. The maximum number of columns in the results of a table subquery is 255. However, the maximum number of columns in the results of a table subquery for a derived table in the FROM clause is 30,000.
5. When a scalar subquery or a row subquery is specified, the maximum allowable number of rows in the results is 1.
6. If the results of a scalar subquery are zero rows, the result is the null value.
7. If the results of a row subquery are zero rows, the result is a row in which all members are the null value.
8. Value expressions that yield results with the following data types cannot be specified in a selection expression in a subquery:
   - BLOB
   - BINARY with a maximum length of 32,001 bytes or greater
   - Abstract data type
   - BOOLEAN

   The above restrictions, however, do not apply to a selection expression in a scalar subquery or a row subquery that is directly specified as an update value in the SET clause of the UPDATE statement.

   An XMLAGG set function can be specified as an XML type value expression in a selection expression in a scalar subquery that is specified in the value expression of an XML query variable of an XMLQUERY function.
9. [*table-specification* .]ROW cannot be specified in a selection expression in a subquery.
10. You cannot specify a WRITE specification, a GET_JAVA_STORED_ROUTINE_SOURCE specification, or the window function as a selection expression in a subquery.
11. Unsubscripted repetition columns cannot be specified in a selection expression in a subquery.

    The above restriction, however, does not apply to selection expressions for a scalar subquery or a row subquery that is directly specified as an update value in the SET clause of the UPDATE statement.

12. A subquery cannot be specified in a value expression that is specified as an argument in a set function.

(7) Specification examples

1. Specifying a table subquery in a quantified predicate in a SELECT statement:

```
SELECT DISTINCT PNAME FROM STOCK
   WHERE SQUANTITY > ALL
   (SELECT SQUANTITY FROM STOCK
   WHERE PNAME = N'socks')
```

2. Specifying a scalar subquery in the SET clause in an UPDATE statement:

   Change the column of stock level (SQUANTITY) with a product code (PCODE) column in the stock table (STOCK) is 302S to the stock level (SQUANTITY) of the product for which the product code (PCODE) column of stock table 2 (STOCK2) with the same column definition information as the stock table is 302S.

```
UPDATE STOCK
  SET SQUANTITY =
    (SELECT SQUANTITY FROM STOCK2 WHERE PCODE = '302S')
  WHERE PCODE = '302S'
```

3. Specifying a row subquery in the SET clause of an UPDATE statement

   Change the stock level (SQUANTITY) column and the unit price (PRICE) column of a stock table (STOCK) with a product code column (PCODE) of 302S to the stock level (SQUANTITY) and unit price (PRICE) of the product for which the product code (PCODE) column in stock table 2 (STOCK2) with the same column definition information as the stock table is 302S.

```
UPDATE STOCK
  SET (PRICE,SQUANTITY) =
        (SELECT PRICE,SQUANTITY FROM STOCK2
          WHERE PCODE = '302S')
```

2.5 Table expressions

(1) Function

A table expression specifies one or more tables to be retrieved, tables to be inner-joined or outer-joined, or the query name specified in a WITH clause of a query expression. The conditions under which a table is to be retrieved or joined (search conditions or grouping conditions) can also be specified in a table expression. A table expression is specified in either a subquery or a query specification or in a single-line SELECT statement.

(2) Format

```
FROM table-reference [,table-reference]...
  [WHERE search-condition]
  [GROUP BY value-expression [,value-expression]...]
  [HAVING search-condition]
```

(3) Operands

(a) Not using a subquery

- FROM *table-reference* [, *table-reference*]...

Specifies the table, query name, derived table, or joined table to be retrieved. For details about table references, see *2.6 Table reference*.

- [WHERE *search-condition*]

Omitting a search condition causes the system to retrieve all rows that are derived from a table (a specified table, joined table, derived table, or a table that is derived as a derived query expression in a WITH clause).

Embedded variables can be specified in a search condition. In a SELECT statement prepared by the PREPARE statement, ? parameters are used in place of embedded variables.

Either an SQL variable or an SQL parameter is used in the SQL procedure. For details about Java procedures, see the section on the JDBC driver in the *HiRDB Version 9 UAP Development Guide*.

- [GROUP BY *value-expression*[, *value-expression*] ...]

Specifies grouping. In a grouping operation, all rows that have the same value in the result of the value expression specified by the GROUP BY clause are treated as a group and are output as one row.

Only the following can be specified in a selection *expression*:*column* names used for

grouping, set functions, literals, value expressions that include these items as primaries, and value expressions to be grouped. In other words, value expressions that include value expressions to be grouped as primaries (except when value expressions are column specifications) cannot be specified. A value expression specified by a GROUP BY clause is called a grouping column.

The following rules apply to the GROUP BY clause:

1. If the columns that serve as a grouping condition contain a row that has null values, all the null values are treated alike and grouping is performed accordingly.
2. Duplicate value expressions of the same format cannot be specified in a value expression specified in a GROUP BY clause.
3. A set function cannot be included in a value expression specified in a GROUP BY clause.
4. The window function cannot be included in a value expression specified in a GROUP BY clause.
5. A component specification cannot be included in a value expression specified in a GROUP BY clause.
6. A repetition column cannot be specified in a value expression specified in a GROUP BY clause.
7. A subquery cannot be contained in a value expression specified in the GROUP BY clause.

- [HAVING *search-condition*]

A set function (AVG, MAX, MIN, SUM, COUNT or COUNT_FLOAT) can be specified.

The following rules apply to the HAVING clause:

1. Only grouped value expressions can be specified in a search condition other than a set function.
2. Any grouped value expression must have a format exactly identical to the value expression that is specified in the GROUP BY clause.
3. If a search condition is omitted, all groups are output.

(b) Using a subquery

- FROM *table-reference* [, *table-reference*]...

Specifies the table, query name, derived table, or joined table to be retrieved. For details about table references, see *2.6 Table reference*.

If tables are added to the FROM clause, the rows that are taken from the tables, one row per table, and joined in the order in which the tables are specified, become the rows of the table that is the result of the FROM clause. The number of rows in the resulting table

is the product of the numbers of rows in the original tables.

- [WHERE *search-condition*]

A column specification in a search condition in a subquery can reference the columns of the tables specified outside the subquery.

In the case of nested queries, referencing from an inner query to (a column in) a table specified by an outer query is called outer referencing.

The following rules apply to the WHERE clause:

1. COUNT(*) and COUNT_FLOAT(*) cannot be specified in the WHERE clause.
2. A set function can be specified in a WHERE clause only if the WHERE clause belongs to a HAVING clause.
3. If a set function is specified in a WHERE clause belonging to a HAVING clause, any column specification in the set function must reference (in an external reference) the table that is specified in the FROM clause preceding the HAVING clause.
4. In a subquery in a WHERE clause with a query specification in which a value expression other than a column specification is specified in the GROUP BY clause, a value expression other than a column specification cannot be specified in the GROUP BY clause.

- [GROUP BY *value-expression*[, *value-expression*] ...]

Specifies grouping. A column to be specified in a GROUP BY clause is listed in the table specified by the FROM clause in the table expression containing that GROUP BY clause.

In a grouping operation, all rows that have the same value in the result of the value expression specified by the GROUP BY clause are treated as a group and are output as one row.

Only the following can be specified in a selection expression: column names used for grouping, set functions, literals, value expressions that include these items as primaries, and value expressions to be grouped. In other words, value expressions that include value expressions to be grouped as primaries (except when value expressions are column specifications) cannot be specified. A value expression specified by a GROUP BY clause is called a grouping column.

In addition, if an item other than a column specification is specified in a value expression, the grouping column cannot be referenced from the scalar subquery specified in a selection expression.

The following rules apply to the GROUP BY clause:

1. If the columns that serve as a grouping condition contain a row that has null values, all null values are treated alike and the grouping is performed accordingly.
2. Duplicate value expressions of the same format cannot be specified in a value

expression specified in a GROUP BY clause.

3. A set function cannot be included in a value expression specified in a GROUP BY clause.
4. The window function cannot be included in a value expression specified in a GROUP BY clause.
5. A component specification cannot be included in a value expression specified in a GROUP BY clause.
6. A repetition column cannot be specified in a value expression specified in a GROUP BY clause.
7. A subquery cannot be contained in a value expression specified in a GROUP BY clause.

- [HAVING *search-condition*]

The HAVING clause specifies the condition by which groups that are obtained as a result of preceding GROUP BY, WHERE, or FROM clauses are to be selected. If a GROUP BY clause is not specified, the result of a WHERE clause or a FROM clause forms a group that does not contain any grouping columns.

The following rules apply to the HAVING clause:

1. The group that yields the TRUE result from the search condition specified in the HAVING clause is selected.
2. If a HAVING clause is omitted, all groups of the results of the preceding GROUP BY clause, WHERE clause, or FROM clause are selected.
3. Any grouped value expressions must have a format exactly identical to the value expression that is specified in the GROUP BY clause.
4. When a column specification is specified in a HAVING clause, the following provisions must be made:
   - Either reference the table in the FROM clause in the table expression, or reference the table in the FROM clause in an outer table expression (in an external reference).
   - If the table in the FROM clause in the table expression is referenced, specify the table in either a grouping column (the value expression that is specified in the GROUP BY clause) or in an argument of a set function.

(4) Common rules

1. Embedded variables can be specified in a search condition.
2. The ? parameter must be specified in place of an embedded variable in SQL statements that are pre-processed by the PREPARE statement.

(5) Join rules

A retrieval that specifies multiple tables or query names in a single FROM clause (a retrieval that encompasses multiple tables) is called a join.

1. All tables or query names to be joined must be specified in the FROM clause. If no join conditions are specified in the WHERE clause, SQL extracts one row at a time from each of the tables to be joined and combines the rows to produce a result.

   Thus, if three tables composed of l rows, *m* rows, and *n* rows are joined unconditionally, l x *m* x *n* rows will result.

2. If a condition that expresses the relationship between tables is specified in the WHERE clause (join condition), those rows that satisfy the condition are selected from the results of the concatenation operation described in (1) above.

3. Columns that are specified to be joined in a join condition must have mutually convertible data types (one numeric value can be converted into another numeric value; one character can be converted into another character).

4. A row that has the null value in its columns to be joined does not satisfy any condition relative to any row.

5. When specifying a column when joining a table containing columns of the same name or a table derived as a query name from a derived query expression of the WITH clause, assign a table name or correlation name to uniquely specify the column.

6. The total number of base tables and derived tables that can be joined in a FROM clause is 64.

   Also, the total number of base tables that can be specified in one SQL statement (including the total number of base tables specified in a subquery) is 64. The total number of correlation names that can be specified is 65.

   When using a named derived table (view table or query of a WITH clause), the number of joined tables with respect to one of the named derived tables is equal to the total number of base tables specified in the view definition statement or in a derived query expression in a WITH clause plus the number of derived tables in the FROM clause. Similarly, the number of tables specified in an SQL statement with respect to one of the named derived tables is equal to the total number of base tables specified in the view definition statement or in a derived query expression in a WITH clause. The number of correlation names specified in an SQL statement with respect to one of the named derived tables is equal to the number of correlation names specified in a view definition statement or in a derived query expression in a WITH clause.

   If a named derived table derived by specifying a set operation in a derived query expression in a view definition statement or a WITH clause is specified in an SQL statement, and the named derived table does not satisfy any of the conditions

under which it can be an inner derived table, the following rules apply to the number of tables or the number of correlation names:

number of tables =

(*total number of base tables specified in a derived query expression*)

+ ((*number of set operations in derived query expression* + 1)

x (*total number of base tables in subquery*))

number of correlation names =

(*total number of correlation names specified in derived query expression*)

+ ((*number of set operations in derived query expression* + 1) x (*total number of correlation names in subquery*))

For details about the restrictions for a derived query expression in a view definition, see *CREATE [PUBLIC] VIEW (Define view, define public view)* in Chapter 3. For restrictions on derived query expressions in a `WITH` clause, see *2.2 Query expressions*. Also, for conditions under which a named derived table becomes an inner derived table, see *2.21 Inner derived tables*.

7. In a query specification for the creation of an inner derived table, the joining of a named derived table to itself, which is the object of inner derived table creation, cannot be specified. For conditions under which an inner derived table can be created, see *2.21 Inner derived tables*.

(6) Rules for the GROUP BY clause and HAVING clause

1. For the `GROUP BY` clause in a view definition or a derived query expression in the `WITH` clause, specify a value expression with column specification.
2. If the window function is specified for a query specification, you cannot specify a `GROUP BY` or `HAVING` clause.

(7) Notes

1. HiRDB may create a work table when any of the following conditions is satisfied:
   - Multiple tables are joined.
   - A `GROUP BY` clause is specified.
   - An inner derived table is created by specifying either a view table or a query name in the table primary. (For inner derived tables, see *2.21 Inner derived tables*.)

   In this process, the above processing may be subject to restrictions depending on the row length of the work table. For details about work table row lengths, see the *HiRDB Version 9 Installation and Design Guide*.

(8) Specification example: Specifying a table expression in a dynamic SELECT statement

```
SELECT SUM(SQUANTITY) FROM STOCK
   WHERE PRICE >= (SELECT AVG(PRICE)
   FROM STOCK)
```

(9) Usage examples

Table A is the product unit price table, table B is the table of quantities of product ordered, and table C is the table of quantities of products ordered in the last month.

Table name: A

Product code	Price
PCODE	PRICE
10	1.00
20	2.00
30	3.00
40	4.00
50	5.00
60	6.00

Table name: B

Product code	Quantity
PCODE	OQUANTITY
10	10
40	40
50	50

Table name: C

Product code	Previous month's orders
PCODE	LMORDER
10	30
20	20
40	10
50	40

1. Select the product codes with a unit price greater than 200, their unit prices, and quantities ordered:

```
SELECT A.PCODE, PRICE, OQUANTITY FROM A
   LEFT OUTER JOIN B
   ON A.PCODE = B.PCODE WHERE PRICE > 200
```

Execution results

PCODE	PRICE	OQUANTITY
30	300	
40	400	40
50	500	50
60	600	

Note

The quantity of a product for which an order has not been received is the null value.

2. Select all product codes, unit prices, and products with a quantity ordered of 40 or greater:

```
SELECT A.PCODE, PRICE, OQUANTITY FROM A
   LEFT OUTER JOIN B
   ON A.PCODE = B.PCODE AND OQUANTITY >= 40
```

Execution results

PCODE	PRICE	OQUANTITY
10	100	
20	200	
30	300	
40	400	40
50	500	50
60	600	

Note

The quantity for a product for which an order has not been received is the null value.

3. Select all product codes, unit prices, and quantities ordered for products with a unit price of greater than or equal to 400:

```
SELECT A.PCODE, PRICE, OQUANTITY FROM A
  LEFT OUTER JOIN B
  ON A.PCODE = B.PCODE AND PRICE >= 400
```

Execution results

PCODE	PRICE	OQUANTITY
10	100	
20	200	
30	300	
40	400	40
50	500	50
60	600	

Note

The quantity for a product for which an order has not been received is the null

value.

4. For each product with a unit price of 500 or less, determine its product code, unit price, orders received for the current month where the minimum size order is at least 40, and orders received for past month where the maximum size order is 30 or fewer:

```
SELECT A.PCODE,A.PRICE,B.OQUANTITY,C.LMORDER
   FROM A LEFT OUTER JOIN B ON A.PCODE=B.PCODE AND
   B.OQUANTITY >= 40
         LEFT OUTER JOIN C ON A.PCODE=C.PCODE AND
           C.LMORDER=30
          WHERE A.PRICE<=500
```

Execution results

PCODE	PRICE	OQUANTITY	LMORDER
10	100		30
20	200		20
30	300		
40	400	40	10
50	500	50	

Note

Products with a unit price greater than 500 are not retrieved.

NULL will be set as the ordered quantities for products that did not receive any orders, as the ordered quantity for the current month if all orders were for less than 40 (OQUANTITY), and as the ordered quantity for the past month if all orders were for more than 30 (LMORDER).

5. For each product with a unit price of 400 or less, determine its product code, unit price, and ordered quantities for the current and past months.

```
SELECT A.PCODE,A.PRICE,B.OQUANTITY,C.LMORDER
  FROM A LEFT OUTER JOIN B ON A.PCODE = B.PCODE
           AND A.PRICE <= 400
         LEFT OUTER JOIN C ON A.PCODE = C.PCODE
           AND A.PRICE <= 400
```

Execution results

PCODE	PRICE	OQUANTITY	LMORDER
10	100	10	30
20	200		20
30	300		
40	400	40	10
50	500		
60	600		

Note

NULL will be set as the ordered quantities for products that did not receive any orders (OQUANTITY, LMORDER) and for products with a unit price greater than 400.

6. Determine all product codes and the percentage of the orders (in quantity) received this month against the orders received last month for each product. The quantity of the product for which an order has not been received this month or last month is the null value.

```
SELECT A.PCODE,100.0*B.OQUANTITY/C.LMORDER
   FROM A LEFT OUTER JOIN (B INNER JOIN C ON B.PCODE=C.PCODE)
            ON A.PCODE = B.PCODE
```

Execution results

PCODE	100.0*B.SQUANTITY/C.LMORDER
10	33.3333333333333333
20	
30	
40	400.0000000000000000
50	125.0000000000000000
60	

7. Determine sales for the current and preceding months from all product codes, unit prices, the orders received this month, and the orders received last month:

```
SELECT PCODE,PRICE*OQTY,PRICE*LMORDER
  FROM(SELECT A.PCODE,A.PRICE,B.OQTY,C.LMORDER
```

```
        FROM A LEFT OUTER JOIN B ON A.PCODE = B.PCODE
               LEFT OUTER JOIN C ON A.PCODE = C.PCODE)
    AS DT1(PCODE,PRICE,OQUANTITY,LMORDER)
```

Execution results

PCODE	PRICE*OQTY	PRICE*LMORDER
10	1000	3000
20		4000
30		
40	16000	4000
50	25000	20000
60		

Note: Sales figures (PRICE*OQTY, PRICE*LMORDER) for products not ordered are indicated as null values.

2.6 Table reference

(1) Function

A table reference specifies a table to be retrieved or a join of tables to be retrieved. A table reference is specified in a table expression.

(2) Format

table-reference : := {*table-primary* | *joined-table*}
table-primary : :=
{[*authorization-identifier*.] *table-identifier*
[IN (*RDAREA-name-specification*)]
[[AS] *correlation-name*]
[*used-index-SQL-optimization-specification*]
| *query-name* [[AS] *correlation-name*]
[*used-index-SQL-optimization-specification*] | (*joined-table*)
| *derived-table* [[AS] *correlation-name* [(*derived-column-list*)]]}
derived-table : := *table-subquery*
derived-column-list : := *column-name-list*
column-name-list : := *column-name* [,*column-name*]...
joined-table : := *table-reference* [{INNER | LEFT [OUTER]}]
JOIN [*join-method-SQL-optimization-specification*]
table-reference ON *search-condition*

(3) Operands

■ *table-primary* : :=
{[*authorization-identifier*.] *table-identifier*
[IN (*RDAREA-name-specification*)]
[[AS] *correlation-name*]
[*used-index-SQL-optimization-specification*]
| *query-name* [[AS] *correlation-name*]
[*used-index-SQL-optimization-specification*]
| (*joined-table*)
| *derived-table* [[AS] *correlation-name* [(*derived-column-list*)]]}

authorization-identifier

Specifies the authorization identifier for the owner of the table. When retrieving a data dictionary table, specify MASTER in *authorization-identifier*.

table-identifier

Specifies the name of the table to be retrieved.

[IN (*RDAREA-name-specification*)]

IN

Specifies the RDAREA to be accessed. This cannot be specified in the following cases:

- If the table specified in the table identifier is a read-only view table
- While in a derived query expression of a view definition

RDAREA-name-specification::=*value-specification*

Use a value specification of the VARCHAR type, CHAR type, MVARCHAR type, or MCHAR type to specify the RDAREA name to be used as the RDAREA for storing the table specified in the table identifier. To specify multiple RDAREA names, separate them with commas (,). The same name cannot be used to specify another RDAREA name; doing so results in an error. For details about the characters allowed in an RDAREA name specified using a value specification, see *1.1.7 Specification of names*. Spaces before or after an RDAREA name specified using a value specification are ignored. However, if the RDAREA name is enclosed in double quotation marks ("), only the spaces outside the double quotation marks are ignored.

When specifying an RDAREA while using the inner replica facility, specify the original RDAREA name. To target a replica RDAREA, use the change current database command (pddbchg command) or PDDBACCS operand of the client environment definition to change the RDAREA to be accessed to the replica RDAREA.

query-name

Specifies the name of the table derived from the derived query expression in the WITH clause.

[AS]*correlation-name*

When joining a table to itself or referencing columns in the same table from an inner subquery, specifies a name to be assigned to the table to distinguish it from each other. The word AS is optional.

If a given table or query name is specified multiple times in a FROM clause, specify a correlation name so that the table or the query name can be uniquely identified. If the query name is identical to the table name, specify a correlation name so that they can be uniquely identified.

The correlation name specified in *correlation-name* must be distinct from any other correlation names specified in a FROM clause.

In a single FROM clause, you cannot specify a name for *correlation-name* that is the same as a table name that specifies that correlation name, or a name, other than

a query name, that is the same as a table identifier.

If you specify a correlation name that is the same as a correlation name specified by a table name or a query name in the same FROM clause, the table name or query name in the FROM clause that specifies the correlation name does not have a valid scope.

The scope of a correlation name is the query specification that contains in a FROM clause a table reference specifying the correlation name not through a derived table, a single-row SELECT statement, and any subqueries that are internal to them.

SQL optimization specification for the index being used

For SQL optimization specification for the index being used, see *2.24 SQL optimization specification*.

(*joined-table*)

When specifying the order of evaluation of joined tables, specify the joined tables enclosed in parentheses. If joined tables are not enclosed in parentheses, they are evaluated in sequence beginning with the leftmost table reference.

- *joined-table*::= *table-reference* [{INNER |LEFT [OUTER]}] JOIN

  [*join-method-SQL-optimization-specification*]

  table-reference ON *search-condition*

In *joined-table*, specify the table that is derived by an inner or outer join.

In inner join, rows are fetched one by one from the outer and inner tables, and, of those rows, rows that satisfy the search condition are retrieved. In outer join, all rows in the outer table and rows in the inner table that satisfy the search condition are retrieved. If successive outer joins are specified, the outermost two tables are evaluated first, and the resulting table is treated as an outer table, the table to the right of it is treated as an inner table, and this evaluation is repeated until all right-side tables are exhausted.

table-reference [{INNER | LEFT [OUTER]}] JOIN *table-reference*

Specifies this operation when processing two tables, outer and inner tables, by matching them (creating an inner or outer join).

For *table-reference-1* [INNER JOIN] *table-reference-2*, the table (*table-reference-1*) of the result of the table reference specified to the left of the [INNER JOIN] becomes the outer table, and the table (*table-reference-2*) of the result of the table reference specified to the right becomes the inner table. Of the results of the matching, the rows that satisfy the search condition are derived.

For *table-reference-1* LEFT [OUTER] JOIN *table-reference-2*, the table (*table-reference-1*) of the result of the table reference specified to the left of the LEFT [OUTER] JOIN becomes the outer table, and the table (*table-reference-2*)

of the result of the table reference specified to the right becomes the inner table. All rows in the outer table are derived irrespective of whether the result of the matching is TRUE or FALSE. With regard to the rows in the inner table, only those rows that satisfy the search condition are derived.

SQL optimization specification for join methods

For SQL optimization specification for join methods, see *2.24 SQL optimization specification*.

ON *search-condition*

Specifies a join condition for an inner or outer join.

In *search-condition*, a column of an outer or inner table can be specified.

Column specifications in a search condition in a subquery can reference columns of the table specified outside the subquery. For nested queries, referencing a table or column that is specified in an outside query from an inner query is called an outer reference.

When qualifying a column specification in a search condition with a table name, table columns for which a correlation name is specified must be qualified with a correlation name.

COUNT(*), COUNT_FLOAT(*), and window functions cannot be specified in a subquery's ON search condition. A set function can be specified in an ON search condition only in the ON search condition in a FROM clause belonging to a HAVING clause. If a set function is specified in an ON search condition in a FROM clause that belongs to a HAVING clause, any table specification in the set function must reference (in an outer reference) the table that was specified in a FROM clause preceding the HAVING clause.

- *derived-table* [AS] *correlation-name* [(*derived-column-list*)]]

Specifies a table subquery. The table derived by this query is called a *derived table of the FROM clause*. The *n*-th column in the table subquery becomes the *n*-th column in the derived table.

The query specification containing a derived table is a read-only specification.

When specifying a derived table, observe the following notes:

- If a derived table is specified for table reference, a correlation name for the derived table must be specified. The specification [AS] *correlation-name* [(*derived-column-list)*]] can be omitted only if the following format is specified in the outermost query:
  - SELECT COUNT(*) FROM *derived-table*
  - SELECT COUNT_FLOAT(*) FROM *derived-table*

- The following structure or data type cannot be specified as a constituent column of a table which is derived as a result of a table subquery in which a value expression is specified in a selection expression:
  - Repetition structure
  - BLOB
  - BINARY with a minimum length of 32,001 bytes
  - BOOLEAN
  - Abstract data type
- A row interface (ROW) cannot be specified on a derived table.
- The window function cannot be specified in a derived table.

[AS] *correlation-name*

Specifies the name of a derived table.

[(*derived-column-list*)]

Specifies the names of columns in a derived table.

If a derived column list is omitted, the column name derived as a result of the outermost query in the derived table becomes the column name for the derived table. Consequently, the column specification (if an AS clause is specified, the column name in the AS clause) in the selection expression will be the column name of the derived table. In other cases, HiRDB assumes a column name that is distinct from any column name used in the SQL statement.

If a derived column list is omitted, care must be taken that no columns of duplicate names are derived as a result of the table subquery.

Column names specified in a derived column list must all be distinct.

When specifying a derived column list, the number of column names used in the derived column list must be the same as the number of columns in the table that is derived by the derived table.

The number of columns used in a derived column list or derived by a table subquery must be 30,000 or less.

(4) Common Rules

If you specify the RDAREA name specification, you cannot use an index that has a different number of partitions than the table does. When defining an index for a query that specifies an RDAREA name, define an index with the same number of partitions as the table has.

(5) Rules on joined tables

1. Specifying an outer join permits the specification of the null value in the columns of the inner table that results from the outer join.
2. Only columns in an outer or inner table or outer reference columns can be specified in ON *search-condition*.
3. Joined tables are read-only tables.
4. Tables derived from an inner join are composed of concatenated rows that satisfy the ON search condition for the outer and inner tables.
5. Tables derived from an outer join are composed of concatenated rows that satisfy the ON search condition for the outer and inner tables, and rows to which the null value is added, in numbers equal to the columns in the inner table, to the rows that do not satisfy the ON search condition for the outer table.
6. For a query specification containing a joined table, ROW cannot be specified in the selection expression that is derived from the inner table of the outer join.

(6) Rules on derived tables in a FROM clause

1. ROW cannot be specified in a selection expression in a derived table in a FROM clause.
2. An unsubscripted repetition column cannot be specified in a selection expression for a derived table in a FROM clause.
3. WRITE or GET_JAVA_STORED_ROUTINE_SOURCE cannot be specified in a selection expression for a derived table in a FROM clause.
4. A format that can omit [AS] *correlation-name*[(*derived-column-list*)] (SELECT COUNT(*) FROM *derived-table* or SELECT COUNT_FLOAT(*) FROM *derived-table*) cannot be specified in a WITH clause query, view definition, set operation, INSERT statement, or subquery.

(7) Notes

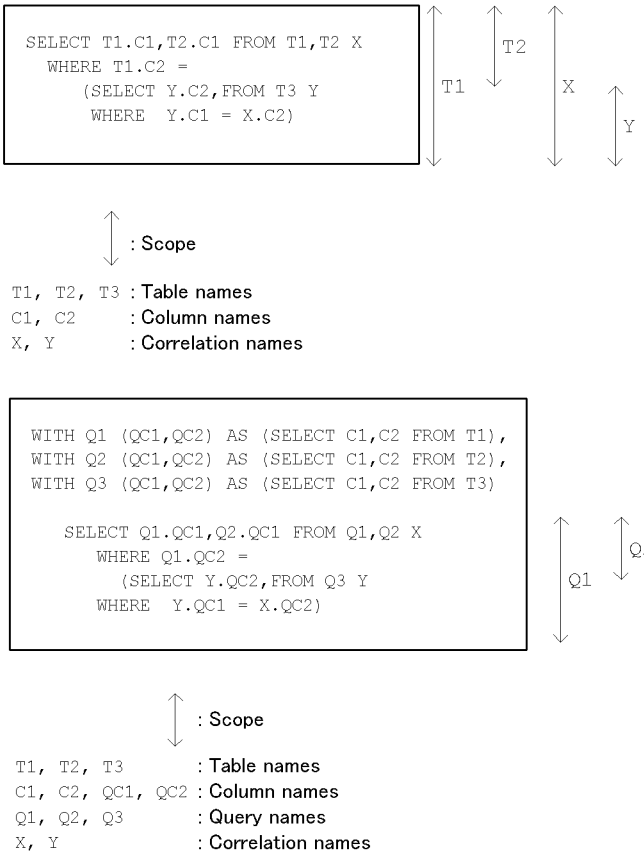
1. The scope for correlation names specified in a FROM clause, tables names specified without a correlation name, or query names is the innermost subquery including the FROM clause, the query specification, or the single-row SELECT statement. The scope also includes subqueries that are interior to these elements.
2. If a correlation name is specified in a FROM clause in a subquery, the table name or query name does not have a valid scope.
3. The valid scope of a table name or a query name that specifies a correlation name in the outermost query specification, or that specifies a correlation name in the FROM clause directly under a single-row SELECT statement, consists of the query specifications other than the innermost subquery, or the single-row SELECT statement. However, if a specified correlation name is the same as a table name or

a query name in the FROM clause, the table name or query name in the FROM clause that specifies the correlation name does not have a valid scope.

The following figures show examples of scopes of correlation names, table names, and query names.

If a table name or query name specified in a FROM clause in a subquery is associated with a correlation name, the table name or query name does not have a valid scope (T3 in the figure). If a table (T2 X in the figure) with a correlation name that is directly specified in the query is referenced by an inner query, the column name must be qualified with a column name (X), rather than with a table name (T2).

Figure 2-1: Examples of valid scopes for correlation names, table names, and query names

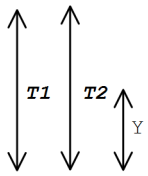


If a correlation name, table name, or query name is referenced within the scope in which the same valid name is specified more than once, the one with

the greatest local valid scope is specified. To reference an outer table in such a situation, specify a different correlation name in the FROM clause, and use that name to specify the reference.

Figure 2-2: Examples of valid scopes of correlation names, table names, and query names (when a specified correlation name is the same as a table name or a query name in a FROM clause)

```
SELECT T1.C1,T2.C1 FROM T1 T1,T2 T2
  WHERE T1.C2 =
     ( SELECT Y.C2,FROM T3 Y
       WHERE  Y.C1 = T1.C2      )
```



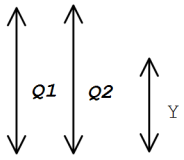
Legend:

: Scope

T1, T2, T3 : Table name
C1, C2 : Column name
T1, T2, Y : Correlation name

```
WITH Q1 (QC1,QC2) AS (SELECT C1,C2 FROM T1),
WITH Q2 (QC1,QC2) AS (SELECT C1,C2 FROM T2)


   SELECT Q1.QC1,Q2.QC1 FROM Q1 Q2,Q1 Q1
      WHERE Q1.QC2 =
        ( SELECT Y.QC2,FROM Q2 Y
      WHERE  Y.QC1 = Q2.QC2      )
```



Legend:

: Scope

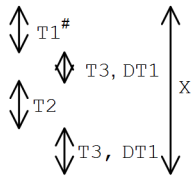
T1, T2 : Table name
C1, C2, QC1, QC2 : Column name
Q1, Q2 : Query name
Q1, Q2, Y : Correlation name

If a specified correlation name is the same as a table name or a query name in the FROM clause, the table name or query name in the FROM clause that specifies the correlation name does not have a valid scope.

4. The following figure shows an example of the scopes for correlation names and table names with a FROM derived table specified.

Figure 2-3: Example of valid scopes for correlation names and table names with a FROM derived table specified

```
SELECT * FROM T1 X
   WHERE (C1,C2) IN
     (SELECT DT1.DC1,T3.C1
       FROM (SELECT C1,C2 FROM T2 WHERE T2.C1> X.C1) AS DT1(DC1,DC2)
            ,T3
       WHERE DT1.DC1=T3.C1
         AND X.C1 = DT1.DC2)
```



Legend:

: Scope

T1, T2, T3 : Table name
C1, C2, DC1, DC2 : Column name
X, DT1 : Correlation name

#: T1 is not valid ISO SQL code, but it is compatible with earlier versions of SQL. It is valid only in a main query.

2.7 Search conditions

2.7.1 Function

- When outer join is not specified, a logical operation is performed based on the search conditions specified in the SQL, and the system retrieves only those rows for which the result of the evaluation of the search condition is TRUE. When outer join is specified, the rows in the outer table for which the result of the evaluation of the search condition is FALSE are also retrieved.
- A search condition can be specified in any of the following locations:
  - ON search condition in the FROM clause
  - WHERE clause
  - HAVING clause
  - WHEN in a search CASE expression
  - WHEN in a trigger action condition
  - IF clause
  - WHILE clause

Format

search-condition::={[NOT] {(*search-condition*)|*predicate*}
|*search-condition* OR{(*search-condition*)|*predicate*}
|*search-condition* AND{(*search-condition*)|*predicate*}
predicate::={*NULL-predicate*|*IN-predicate*|*LIKE-predicate*|*XLIKE-predicate*|*SIMILAR-predicate*|*BETWEEN-predicate*|*comparison-predicate*|*quantified-predicate*|*EXISTS-predicate*|*logical-predicate*|*structured-repetition-predicate*}

2.7.2 Logical operations

Logical operations are preformed according to the following rules:

- The order of evaluation of logical operation is items inside parentheses, NOT, AND, and OR.
- The maximum number of logical operation nesting levels is 255.
- The number of logical operation nesting levels is the number of parenthesized nestings, when the parentheses for specifying the order of evaluation of the logical operators AND and OR (exclusive of NOT) are specified explicitly.

If a named derived table is specified in a query specification and the specified named derived table does not create an inner derived table, the maximum allowable number of nesting levels for logical operations may be exceeded when the search condition for

the query that derived the named derived table is joined by the AND logical operation.

2.7.3 Results of a predicate

The figure below shows results of predicates on which logical operations are performed.

The result of a predicate (other than the NULL predicate) that contains the null value is undefined.

A predicate that produces the null value is ineligible for retrieval.

Figure 2-4: Results of predicates on which logical operations are performed

AND logical operation

AND	TRUE	FALSE	UNDEFINED
TRUE	TRUE	FALSE	UNDEFINED
FALSE	FALSE	FALSE	FALSE
UNDEFINED	UNDEFINED	FALSE	UNDEFINED

OR logical operation

OR	TRUE	FALSE	UNDEFINED
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNDEFINED
UNDEFINED	TRUE	UNDEFINED	UNDEFINED

NOT logical operation

NOT	Results
TRUE	FALSE
FALSE	TRUE
UNDEFINED	UNDEFINED

2.7.4 Rules common to predicates

1. For types of data that can be compared, see *1.2.2 Data types that can be converted (assigned or compared)*. However, if a character string literal is specified as an object of comparison of national character data, the character string literal is treated as a national character literal. If a character string literal is treated as a national character string literal, the character data receives a length check only, and the character code is not checked.
2. Character data cannot be compared if the character sets are different. If the conditions in the following table are met, however, the character set can be converted to a character set for which comparison is supported, allowing comparisons to be performed.

   Table 2-1: Conditions for being able to compare different character sets (for EBCDIK and the default character set (SJIS))

EBCDIK value expression	Default character set (SJIS) value expression		
	Character string literal	Embedded variable, ? parameter	Character string literal, embedded variable, ? parameter
Character string literal	--	--	--
Embedded variable, ? parameter	N	N	N

EBCDIK value expression	Default character set (SJIS) value expression		
	Character string literal	Embedded variable, ? parameter	Character string literal, embedded variable, ? parameter
Character string literal, embedded variable, ? parameter	Y	Y	N

Legend:

Y: Converted to an EBCDIK character code and then compared.

N: Cannot be compared.

--: Not supported.

Table 2-2: Conditions for being able to compare different character sets (for UTF-16 and default character set (UTF-8))

UTF-16 value expression	Default character set (UTF-8) value expression#		
	Character string literal	Embedded variable, ? parameter	Character string literal, embedded variable, ? parameter
Character string literal	--	--	--
Embedded variable, ? parameter	Yes	N	Yes
Character string literal, embedded variable, ? parameter	Y	Y	N

#

Includes mixed character data.

Legend:

Yes: Converted to UTF-16 character encoding and then compared.

Y: Converted to UTF-8 character encoding and then compared.

N: Cannot be compared.

--: Not supported.

3. Before comparing two fixed-length data items or a fixed-length data item and variable-length data item of different lengths, the shorter data item is right-filled with the character set's space character to make the strings the same length.

4. When comparing two variable-length character data items of different lengths, HiRDB performs the comparison from the left, for a length equal to the length of the shorter data item. If the results are equal, HiRDB further compares the character string lengths.
5. When comparing two numeric data items of different data types, HiRDB performs the comparison by using the data type of the wider range. Different numeric data types have the following range widths:

   FLOAT > SMALLFLT > DECIMAL > INTEGER > SMALLINT

   If one of the items to be compared with is the SMALLFLT, it is compared as a FLOAT type irrespective of the width of its range.
6. When comparing two time stamp data items of different fractional second precisions, HiRDB extends the fractional precision of the lower precision item to match the precision of the higher precision item, and zero-fills the extended fractional second part.
7. When comparing two BINARY type items of different data lengths, HiRDB performs the comparison from the left, for a length equal to the length of the shorter data item. If the results are equal, HiRDB further compares the data lengths.

2.7.5 Predicates

(1) Comparison predicate

Format

row-value-constructor
{= | <> | ^= | != | < | <= | > | >=} *row-value-constructor*

Case in which the predicate is TRUE

- Case involving one row value constructor

  The predicate is TRUE if the right and left row value constructors satisfy the comparison condition.

  If either row value constructor is the null value, the result of the comparison is indefinite.

- Case involving two or more row value constructors

  (=)

  The result of the comparison is TRUE if the relationships between the corresponding elements in the right and left row value constructors are all =.

  The result of the comparison is FALSE if one or more combinations of elements exist in which the relationship <> holds.

The result of the comparison is indefinite if one or more of the elements being compared is the null value, even when there is no combination of elements in which the relationship <> holds.

- (Examples of the result of a comparison being TRUE)

```
(1,2,3) = (1,2,3)
```

```
('A','B','C') = ('A','B','C')
```

(<> | ^= | !=)

The corresponding elements in the right and left row value constructors are compared, and the result is TRUE if at least one combination exists in which the relationship <> holds.

The result is FALSE if the relationship between the corresponding elements is entirely =.

The result is FALSE if there is no combination in which the relationship <> holds and one or more elements being compared has the null value.

- (Examples of the result of a comparison being TRUE)

```
(1,2,3) <> (1,5,3)
```

```
('A','B','C') <> ('C','A','B')
```

(<)

The corresponding elements in the right and left row value constructors are compared from left to right, as long as the relationship = holds. The result is TRUE if the relationship < holds between the first elements for which = does not hold.

The result is FALSE if the relationship between the first elements for which = does not hold is >, and if the relationship = holds between all corresponding elements.

The result is indefinite if the first elements for which = does not hold contain the null value.

- (Examples of the result of a comparison being TRUE)

```
(1,2,3) < (3,1,2)
```

This is TRUE because the relationship between the first elements is 1< 3.

```
('A','B','C','D') < ('A','B','E','A')
```

The elements are compared from left to right, and the result is TRUE because the relationship between the first elements for which = does not hold is 'C' < 'E'.

(>)

The corresponding elements in the right and left row value constructors are compared from left to right as long as = holds. The result is TRUE if the relationship > holds between the first elements for which = does not hold.

The result is FALSE if the relationship between the first elements for which = does not hold is a <, or if the relationship = holds between all corresponding elements.

The result is indefinite if the first elements for which = does not hold contain the null value.

- (Examples of the result of a comparison being TRUE)

  ```
  (1,2,3) > (1,1,5)
  ```

  The elements are compared from left to right. The result is TRUE because the relationship between the first elements for which = does not hold is `2 > 1`.

  ```
  ('A','A','C') > ('A','A','A')
  ```

  The elements are compared from left to right. The result is TRUE because the relationship between the first elements for which = does not hold is `'C' > 'A'`.

(<=)

The corresponding elements in the right and left row value constructors are compared from left to right as long as the relationship = holds. The result is TRUE if the relationship between the first elements for which = does not hold is a <, or if the relationship = holds between all corresponding elements.

Notice that the result is FALSE if the relationship between the first elements for which = does not hold is a >.

The result is indefinite if the first elements for which = does not hold contain the null value.

(>=)

The corresponding elements in the right and left row value constructors are compared from left to right as long as the relationship = holds. The result is TRUE if the relationship between the first elements for which = does not hold is a >, or if the relationship = holds between all corresponding elements.

Notice that the result is FALSE if the relationship between the first elements for which = does not hold is a <.

The result is indefinite if the first elements for which = does not hold contain the null value.

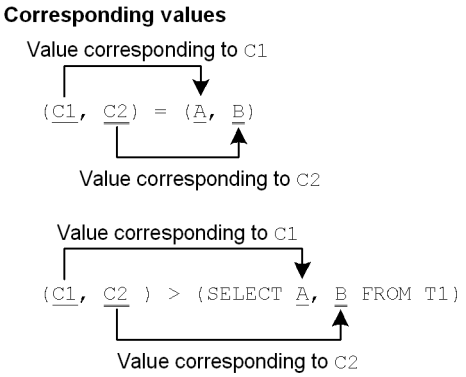
Each comparison predicate can be expanded into a form using Boolean

operations. The following table lists forms in which comparison predicates are expanded using Boolean operations.

Operator	Coding using a row value constructor	Coding using Boolean operations
=	(Rx1, Rx2, ..., Rxn) = (Ry1, Ry2, ..., Ryn)	Rx1 = Ry1 AND Rx2 = Ry2 AND . . . AND Rxn = Ryn
<>	(Rx1, Rx2, ..., Rxn) <> (Ry1, Ry2, ..., Ryn)	Rx1 <> Ry1 OR Rx2 <> Ry2 OR ... OR Rxn <> Ryn
<	(Rx1, Rx2, Rx3, ..., Rxn) < (Ry1, Ry2, Ry3, ..., Ryn)	Rx1 < Ry1 OR (Rx1 = Ry1 AND Rx2 < Ry2) OR (Rx1 = Ry1 AND Rx2 = Ry2 AND Rx3 < Ry3) OR ... OR (Rx1 = Ry1 AND Rx2 = Ry2 AND Rx3 = Ry3 AND ... AND Rxn-1 = Ryn-1 AND Rxn < Ryn)
>	(Rx1, Rx2, Rx3, ..., Rxn) > (Ry1, Ry2, Ry3, ..., Ryn)	Rx1 > Ry1 OR (Rx1 = Ry1 AND Rx2 > Ry2) OR (Rx1 = Ry1 AND Rx2 = Ry2 AND Rx3 > Ry3) OR ... OR (Rx1 = Ry1 AND Rx2 = Ry2 AND Rx3 = Ry3 AND ... AND Rxn-1 = Ryn-1 AND Rxn > Ryn)

Rules

1. On both sides of the comparison operators (=, <>, ^=, !=, <, <=, >, >=), row value constructors consisting solely of literals can be specified.
2. Values that yield any of the following data types as a result of an operation cannot be specified:
   - BLOB
   - BINARY with a minimum defined length of 32,001 bytes
   - BOOLEAN
   - Abstract data type
3. For row value constructors, see *2.8 Row value constructors*.
4. Row value constructor elements that are located in the corresponding positions in the right and left row value constructors are treated as corresponding values. The corresponding values must have data types that are comparable.



5. A subquery cannot be specified in a comparison predicate in any of the following locations:
   - A search condition in an `IF` statement
   - A search condition in a `WHILE` statement
   - A `WHEN` search condition in `CREATE TRIGGER` (trigger action search condition)
6. When specifying a repetition column, specify a subscript. If a subscripted repetition column is specified and if its elements satisfy the condition, the result of the comparison with the corresponding value is `TRUE`.
7. The word `ANY` can be specified as a repetition column subscript. If `ANY` is specified and if at least one element in the column satisfies the condition, the result of the comparison is `TRUE`. If the result of the comparison is not `TRUE` and if the condition specified on at least one element of the column is indefinite, the result of the comparison is indefinite. If the result of the comparison is not `TRUE` or indefinite, the result of the comparison is `FALSE`.
8. If a subscripted repetition column is specified and the subscripted elements do not exist, the result of the comparison with the corresponding value is indefinite.
9. The row value constructors that are to be compared must have the same number of result columns.
10. A leap second is treated as a value smaller than one minute.

    Example:

    '00:00:61' < '00:01:00'

(2) NULL predicate

Format

value-expression IS [NOT] NULL

Conditions under which a predicate is TRUE

The NULL predicate is TRUE for rows in which the value of the specified value expression is a null value. If NOT is specified, it is TRUE for rows in which the value of the specified value expression is not a null value. For details about the NULL value, see *1.7 Null value*.

Rules

1. Value expressions of the following data types cannot be specified:
   - BLOB (except for the ? parameter or when specified with only one embedded variable)
   - BINARY with a minimum defined length of 32,001 bytes (except for the ? parameter or when specified with only one embedded variable)
   - BOOLEAN
2. If an unsubscripted repetition column is specified and the column does not contain any elements (in the case of the NOT specification, if at least one element is present), the NULL predicate will be TRUE. Even if all the elements in the column are NULL, the NULL predicate will not be TRUE.
3. If a subscripted repetition column is specified and a specified element is NULL, the NULL predicate will be TRUE.
4. If a subscripted repetition column is specified and the column does not contain any elements, the NULL predicate will be UNKNOWN.
5. When a subscripted repetition column is specified, ANY can be specified as a subscript. When ANY is specified and at least one element in the column satisfies the specified conditions, the NULL predicate will be TRUE.

Note

- The NULL predicate is TRUE if overflow error suppression is set or if there is a null value due to an overflow of the operation result of the value expression.

(3) IN predicate

Format

row-value-constructor [IS] [NOT] IN
{(*row-value-constructor* [,*row-value-constructor*]...)
 |(*table-subquery)*
([*SQL-optimization-specification-for-subquery-execution-method*]
SELECT[{ALL | DISTINCT}]{*selection-expression*|*}
(*table-expression*)

```
FROM table-reference [,table-reference]...
[WHERE search-condition]
[GROUP BY value-expression [,value-expression]...]
[HAVING search-condition])}
```

Conditions under which a predicate is TRUE

The IN predicate is TRUE if any of the following conditions is satisfied:

- The left-hand side row value constructor matches any row value constructor on the right-hand side.
- The left-hand side row value constructor matches any result row for a table subquery.

If NOT is specified, the IN predicate is TRUE with respect to rows for which the row value constructor on the left-hand side does not match any of the result rows for any row value constructor or table subquery that is specified on the right-hand side.

Rules

1. Values in which the data items as a result of a row value constructor or table subquery take any of the following data types:
   - BLOB
   - BINARY with a minimum defined length of 32,001 bytes
   - BOOLEAN
   - Abstract data type
2. A maximum of 30,000 row value constructors can be specified on the right-hand side.
3. A row value constructor consisting solely of a value specification cannot be specified on the left side of an IN predicate that is not a table subquery.
4. For table subqueries, see *2.4 Subqueries*.
5. Some IN predicates have the same meaning as a quantified predicate. The following predicates are synonymous:

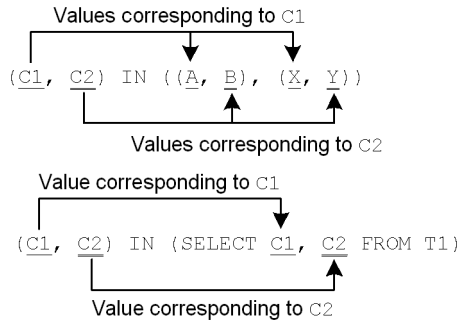
IN predicate	Quantified predicate
row-value-constructor IN *table-subquery*	*row-value-constructor* = ANY *table-subquery* or *row-value-constructor* = SOME *table-subquery*
row-value-constructor NOT IN *table-subquery*	*row-value-constructor* <> ALL *table-subquery*

6. If the result of a table subquery is the empty set, the result of the IN predicate is

FALSE; the result, however, is TRUE if NOT is specified.

7. A table subquery cannot be specified in an IN predicate in the following locations:
   - Search condition of an IF statement
   - Search condition of a WHILE statement
   - WHEN search condition (trigger action condition) of CREATE TRIGGER
8. In the row value constructors, the row value constructor elements and the selection expressions of table subqueries in the corresponding positions are treated as corresponding values. The corresponding values must have mutually convertible or comparable data types. If, however, the result data type of a row value constructor element in a row value constructor specified on the left-hand side is national character data, and a character string literal is specified as the corresponding value, the character string literal is treated as a national character string literal. If a character string literal is treated as a national character string literal, HiRDB only checks the length of the character data without checking the character code.

Corresponding values



9. When specifying a repetition column, a subscript must be specified. If a subscripted repetition column is specified and its elements satisfy specified conditions, the IN predicate is TRUE.
10. The word ANY can be specified as a subscript for a repetition column. If ANY is specified and if at least one element in the column satisfies the specified condition, the IN predicate is TRUE. If the IN predicate is not TRUE and if a condition specified with respect to at least one element in the column is indefinite, the IN predicate becomes indefinite. Any IN predicate that is not TRUE or that is not indefinite is FALSE.
11. A repetition column with an ANY subscript cannot be specified in the row value constructor on the right side.
12. If a repetition column is specified with a subscript and the column contains no

elements, the IN predicate is indefinite.

13. If the result of a row value constructor is the null value, the result of the comparison of the corresponding value is indefinite.

14. The row value constructors that are subject to comparison must have the same number of result columns.

Note

1. If a table subquery is specified in an IN predicate, in some cases HiRDB creates a work table. In this process, the processing of the subquery in the IN predicate may be subject to restrictions depending on the row length of the work table. For details about work table row lengths, see the *HiRDB Version 9 Installation and Design Guide*.

(4) LIKE predicate

Format

value-expression [NOT] LIKE *pattern-character-string* [ESCAPE *escape-character*]

Conditions under which a predicate is TRUE

The LIKE predicate is TRUE for a row in which the value of a specified value expression matches the pattern represented by a pattern character string. If NOT is specified, the LIKE predicate is TRUE for those rows for which the value of the specified column does not match the pattern expressed by the pattern character string.

value-expression

1. Specifies the value expression that will be the object of a character string pattern comparison.

   However, value expressions in which only values other than SQL variables or SQL parameters are specified cannot be specified.

2. The following data types can be specified in a value expression or a pattern character string: character string data, national character string data, mixed character string data, or BINARY with a maximum defined length of 32,000 bytes.

3. When a repetition column is specified, a subscript must be specified. If a repetition column is specified with a subscript and its elements meet specified conditions, the LIKE predicate will be TRUE.

4. ANY can be specified as a subscript for a repetition column. When ANY is specified and at least one element in the column meets specified conditions, the LIKE predicate will be TRUE. If the LIKE predicate is not TRUE and a condition specified for at least one element in the column is UNKOWN, the LIKE predicate will be UNKNOWN. If the LIKE predicate is neither TRUE nor

UNKNOWN, it will be FALSE.

5. If a repetition column is specified with a subscript and the column contains no elements, the LIKE predicate will be UNKNOWN.

pattern-character-string

1. Specifies a value specification.
2. A specifiable pattern character string must have the same data type as the data type of the value expression.
3. The following combinations of value expression data types and pattern character string data types are allowed:

Value expression data types		Pattern character string or escape character string data type					
		Character string data			National character string data	Mixed character string data	Binary data
		Default character set	EBCDIK	UTF16			
Character string data	Default character set	Y	N	N	N	Y	N
	EBCDIK	R[#3]	Y	N		N	
	UTF16	R[#3]	N	Y			
National character string data		R[#1]	N	N	Y	N	N
Mixed character string data		Y	N	N	N	Y	N
Binary data		R[#2]	N	N	N	N	Y

Legend:

Y: Can be specified.

R: Can be specified with restrictions.

N: Cannot be specified.

#1

Only a character string literal can be specified in a pattern character string. In this case, the character string literal is treated as a national character string literal. When a character string literal is treated as a national character string literal, the length of the character string data is checked, but the character encoding is not checked. When specifying a special character, make sure to specify it using a double-byte character.

#2

Only a hexadecimal character string literal can be specified.

#3

Only a character string literal, embedded variable, or ? parameter can be specified.

4. Use the same character set for the value expression, pattern character string, and escape character string. If the pattern character string and escape character string are one of those listed below, however, they are converted to the character set of the value expression, enabling comparison:
   - Character string literal
   - Embedded variable (default character set)
   - ? parameter
5. The special characters that can be included in the pattern character string include the underscore (_), percent sign (%), and escape character. Specify the code for the special characters as binary data. The following table describes how the special characters underscore and percent sign are interpreted in the pattern character string.

Table 2-3: Meaning of special characters in the pattern character string (LIKE predicate)

Special character	Item specification data type		Meaning of the special character	Method of specifying the special character
Underscore (_)	Character string data	Default character set#1	Any single character	Single-byte underscore (_)
		EBCDIK#1		
		UTF16		
	National character string data			Double-byte underscore (__)
	Mixed character string data			Single-byte underscore (_)
	Binary data		Any single byte	5f (hex code for an underscore)#3

Special character	Item specification data type		Meaning of the special character	Method of specifying the special character
Percent sign (%)	Character string data	Default character set#1	A character string of any length (one or more characters)	Single-byte percent sign (%)
		EBCDIK#1		
		UTF16		
	National character string data#2			Double-byte percent sign (％)
	Mixed character string data			Single-byte percent sign (%)
	Binary data		A byte string of any length (one or more bytes)	25 (hex code for percent sign)#3

#1

All characters or data values are treated as single-byte characters.

#2

All characters or data values are treated as double-byte characters.

#3

When specifying special characters in binary data, specify the code indicating the special character (_ or %) using the character encoding specified when HiRDB was set up.

6. If the pattern character string does not contain the % sign and the data lengths of the column to be compared and the pattern character string are different, the LIKE predicate is not TRUE.
7. If the character string and the pattern character string specified in *value-expression* are variable-length data (VARCHAR, NVARCHAR, MVARCHAR, or BINARY), HiRDB compares the data lengths in addition to the data and the pattern character string data in the *value-expression*.
8. When an embedded variable, an SQL variable, or an SQL parameter is specified as a pattern character string point, the following must be observed:

   If a pattern character string formed with a fixed-length embedded variable, SQL variable, or SQL parameter and if a pattern character string shorter than the length of the variable is set in that variable, the variable may be filled with trailing spaces or with any remaining invalid characters as values. If a

retrieval is attempted using this type of pattern character string, target data that does not have matching trailing spaces or matching leftover characters will not be retrieved.

To avoid this problem when a fixed-length variable is used as a pattern character string, we recommend that the percent sign be set at the end of the variable.

Example:

In the following example, character string data containing `'ABCD'` gives the indicated results if pattern character strings `'AB%'` and `'AB%%'` are assigned to variables:

Data type of variable	Pattern character string	Character string data	Result of comparison
Variable-length character string	`'AB%'`	`'ABCD'`	Matches.
	`'AB%%'`	`'ABCD'`	Matches.
Fixed-length character string (4 bytes)	`'AB%Δ'`	`'ABCD'`	Does not match.
	`'AB%%'`	`'ABCD'`	Matches.

Example:

A comparison between binary data 52454452554d and a variable in which a pattern character string, `X'52454425'`, is set gives the following results:

Data type of variable	Pattern character string	Binary data	Result of comparison
`BINARY` type	`X'52454425'`	`52454452554d`	Matches
	`X'5245442525'`	`52454452554d`	Matches

Examples of pattern character strings

The following table provides examples of typical pattern character strings used in the `LIKE` predicate.

Table 2-4: Typical pattern character strings used in the LIKE predicate

Item	Pattern character string	Meaning	Example	
			Pattern character string	Pattern-matching character string
Front match	*nnn*%	The leading portion of the character string is *nnn*.	'ACT%'	Character strings beginning with "ACT", such as ACT, ACTOR, and ACTION.
Rear match[#1]	%*nnn*	The trailing portion of the character string is *nnn*.	'%ING'	Character strings ending with "ING", such as ING, BEING, and HAVING.
Any match	%*nnn*%	The character string contains *nnn* at any position.	N'%OR%'	Character strings containing "or", such as OR, More, and CoLoR.[#2]
Complete match	*nnn*	The character string is equal to *nnn*.	'EQUAL'	EQUAL
Partial match	_ . . . _*nnn*_ . . . _	A specific portion of the character string is equal to *nnn*; the other portions of the character string contain any characters.	'_I_'	3-letter character strings, in which the second character is "I", such as BIT, HIT, and KIT.
Other	*nnn*%*mmm*	The leading portion of the character string is *nnn* and the trailing portion is *mmm*.	'O%N'	Character strings that begin with O and end with N such as ON, OWN, and ORIGIN.
	%*nnn*%*mmm*%	The character string contains *nnn* at any position and *mmm* at a subsequent position.	'%O%N%'	Character strings that contain O, and have an N in any subsequent position, such as ON, ONE, DOWN, and COUNT.
	*nnn*_ . . . _ *mmm*%	The leading portion of the character string is *nnn* and the trailing portion is *mmm*.	'CO_ _ ECT%'	Character strings that begin with "CO" and contain the string "ECT" in the 5th through 7th character positions, such as CORRECT, CONNECTER, and CONNECTION

#1: Because the space is regarded as a character for comparison purposes, a comparison with data that has trailing spaces produces the FALSE result.

#2: In a national character string, the _ and % special characters are coded using the _ and % national characters.

Note

nnn and *mmm* denote any character strings that do not contain % or _.

escape character

Any underscore or percent sign coded in a pattern character string is unconditionally treated as a special character; these characters cannot be treated as regular characters. When a special character is to be specified as a regular character, an escape character must be specified. Specifying any character after the ESCAPE keyword (an escape character) causes the special character following the escape character coded in the pattern character string to be treated as a regular character.

Example 1

A character string containing a '5%', such as '5%' and '25%':

```
'%5?%%' ESCAPE '?'
```

Example 2

A character string ending with 'PRINT_REC', such as 'SQLPRINT_REC':

```
'%PRINT@_REC' ESCAPE '@'
```

Example 3

A hexadecimal character string containing X'48695244425f' in the binary string, such as X'48695244425f':

```
X'4869524442ee5f' ESCAPE X'ee'
```

The following characters can be specified as escape characters:

Data type of item		Character that can be specified
Character string data (CHAR, VARCHAR)	Default character set	Any single character (single-byte character)#1
	EBCDIK	
	UTF16	Any single character#2
Mixed character string data (MCHAR, MVARCHAR)		Any single character#2
National character data (NCHAR, NVARCHAR)		Any single character (double-byte character)#3

Data type of item	Character that can be specified
Binary data (BINARY)	Any single byte value

Note

Care must be taken that the special character is specified following the escape character.

#1

All characters or data values are treated as single-byte characters.

#2

Data values that are not characters are treated as characters with the smallest length allowed for the character encoding (two bytes for UTF-16 and one byte for UTF-8 and SJIS).

#3

All characters or data values are treated as double-byte characters.

Notes

1. For improved performance, the defined length of the column used in a value expression on the left-hand side of LIKE should be either a maximum of 255 bytes (CHAR, VARCHAR, MCHAR, MVARCHAR, or BINARY) or a maximum of 127 characters (NCHAR or NVARCHAR).
2. Multi-byte characters stored in a CHAR or VARCHAR-type column are evaluated byte by byte. Consequently, if the character code for a single-byte character specified in a pattern character string is included in the character codes for the multi-byte characters, the result of LIKE predicate is true.

   Example

   Execute the following query in a condition where the query is set up in sjis character code, a CHAR-type column C1 is in Table T1, and a row ア is in column C1:

   ```
   SELECT C1 FROM T1 WHERE C1 LIKE '%A%' ;
   ```

   The character code for the character ア in hexadecimal is 8341. The character code in hexadecimal for the character A in the pattern character string is 41. Therefore, because the character code for ア, which is a multi-byte character, includes the character code for the single-byte character A, the result of the LIKE predicate is true.

(5) XLIKE predicate

Format

value-expression [NOT] XLIKE *pattern-character-string* [ESCAPE *escape-character*]

Conditions under which a predicate is TRUE

The XLIKE predicate is TRUE for a row in which the value of a specified value expression matches the pattern represented by a pattern character string. If NOT is specified, the predicate is TRUE for a rows for which the value does not match the pattern character string. The comparison performed is not case-sensitive.

value-expression

1. Specifies the value expression that will be the object of a character string pattern comparison.

   However, value expressions in which only values other than SQL variables or SQL parameters are specified cannot be specified.

2. Character string data, national character data, or mixed character string data can be specified as the data type of a value expression.

3. When a repetition column is specified, a subscript must be specified. If a repetition column is specified with a subscript and its elements meet specified conditions, the XLIKE predicate will be TRUE.

4. ANY can be specified as a subscript for a repetition column. If ANY is specified and at least one element in the column meets specified conditions, the XLIKE predicate will be TRUE. If the XLIKE predicate is not TRUE and a condition specified for at least one element in the column is UNKOWN, the XLIKE predicate will be UNKNOWN. If the XLIKE predicate is neither TRUE nor UNKNOWN, it will be FALSE.

5. If a repetition column is specified with a subscript and the column contains no elements, the XLIKE predicate will be UNKNOWN.

pattern-character-string

1. A value specification must be specified in the pattern character string.

2. Any data type that can be specified in an value expression can be specified in a pattern character string.

3. The following combinations of value expression data types and pattern character string data types are allowed:

Data type of value expression		Data type of pattern character string or escape character				
		Character string data			National character string data	Mixed character string data
		Default character set	EBCDIK	UTF16		
Character string data	Default character set	Y	N	N	N	Y
	EBCDIK	R[#2]	Y	N		N
	UTF16	R[#2]	N	Y		
National character string data		R[#1]	N	N	Y	N
Mixed character string data		Y	N	N	N	Y

Legend:

Y: Can be specified.

R: Can be specified with restrictions.

N: Cannot be specified.

#1

Only a character string literal can be specified in a pattern character string. In this case, the character string literal is treated as a national character string literal. When a character string literal is treated as a national character string literal, the length of the character string data is checked, but the character encoding is not checked. When specifying a special character, be sure to specify it using a double-byte character.

#2

This can specify only a character string literal, embedded variable, or ? parameter.

4. Use the same character set for the value expression, pattern character string, and escape character string. If the pattern character string and escape character string are one of those listed below, however, they are converted to the character set of the value expression, enabling comparison:
   - Character string literal
   - Embedded variable (default character set)
   - ? parameter

5. The following special characters can be used in a pattern character string: the underscore, the percent sign, and escape characters. The following table describes how the special characters underscore and percent sign are interpreted in the pattern character string.

Table 2-5: Meanings of special characters in pattern character strings (XLIKE predicate)

Special character	Data type of item specification		Meaning of the special character	Method of specifying the special character
Underscore (_)	Character string data	Default character set#1	Any single character	Single-byte underscore (_)
		EBCDIK#1		
		UTF16		
	National character string data			Double-byte underscore (＿)
	Mixed character string data			Single-byte underscore (_)
Percent sign (%)	Character string data	Default character set#1	A character string of any length (one or more characters)	Single-byte percent sign (%)
		EBCDIK#1		
		UTF16		
	National character string data#2			Double-byte percent sign (％)
	Mixed character string data			Single-byte percent sign (%)

#1

All characters or data values are treated as single-byte characters.

#2

All characters or data values are treated as double-byte characters.

6. Comparison of the following characters with pattern character string data is not case-sensitive:

   Alphanumeric national characters and mixed characters

7. This predicate is not TRUE if a percent sign does not occur in the pattern

character string or the column data and the pattern character string differ in length.

8. If the character string and pattern character string specified in the value expression are both variable-length character strings (VARCHAR, NVARCHAR, or MVARCHAR), HiRDB compares the character string lengths, as well as the character string data and the pattern character string data.

9. When an embedded variable, an SQL variable, or an SQL parameter is to be specified as a pattern character string, the following point must be observed:

   If a pattern character string is formed with a fixed-length embedded variable, SQL variable, or SQL parameter and if a pattern character string shorter than the length of the variable is set in that variable, the variable may be filled with trailing spaces or with any remaining invalid characters as values. When this type of pattern character string is used as a search string, data that does not similarly contain trailing spaces or invalid characters cannot be retrieved. Therefore, when using a fixed-length variable as a pattern character string, the variable must be filled with trailing percent signs.

Examples of pattern character strings

The following table provides examples of typical pattern character strings used in the XLIKE predicate.

Table 2-6: Examples of pattern character strings used in the XLIKE predicate

Item	Pattern character string	Meaning	Example	
			Pattern character string	Pattern-matching character string
Front match	*nnn*%	The leading portion of the character string is *nnn*.	'ACT%'	Character strings beginning with "ACT"[#1], such as ACT, Actor, and action.
Rear match	%*nnn*	The trailing portion of the character string is *nnn*.	'%ING'	Character strings ending with "ING"[#2], such as Ing, Being, and HAVING.
Any match	%*nnn*%	The character string contains *nnn* at any position.	'%OR%'	Character strings containing "or"[#3], such as OR, More, and CoLoR.
Complete match	*nnn*	The character string is equal to *nnn*.	'MAX'	Character strings such as MAX, max, and mAx[#4]

Item	Pattern character string	Meaning	Example	
			Pattern character string	Pattern-matching character string
Partial match	_...*nnn*_...	A specific portion of the character string is equal to *nnn*, the other portions of the character string contain any characters.	'_I_'	A three-character character string in which the second character is either "I" or "i", such as Bit, HIT, and Kit.
Other	*nnn*%*mmm*	The leading portion of the character string is *nnn* and the trailing portion is *mmm*.	'O%N'	Character strings that begin with "O" or "o" and end with "N" or "n", such as on, Own, and ORIGIN.
	%*nnn*%*mmm*%	The character string contains *nnn* at any position and *mmm* at a subsequent position.	'%O%N%'	Character strings that contain "O" or "o" and have "N" or "n" at a subsequent position, such as ON, one, DowN, and Count.
	*nnn*_..._*mmm*%	The leading portion of the character string is *nnn* and the trailing portion is *mmm*.	'CO__ECT%'	Character strings that begin with "CO"[#5], and whose 5th through the 7th characters are "ECT"[#6], such as correct, Connecter, and CONNECTION.

Note 1: *nnn* and *mmm* are any character strings that do not contain % or _.

Note 2: Because the space character is also used as a comparison character, comparison is with data containing trailing spaces, this string yields the FALSE result.

Note 3: In a national character string, the special characters (% and _) must be coded as the national character "%" or "_".

#1: One of the character strings ACT, ACt, Act, aCT, aCt, acT, or act

#2: One of the character strings ING, INg, Ing, InG, iNG, iNg, inG, or ing

#3: One of the character strings OR, Or, oR, or or

#4: One of the character strings MAX, MAx, Max, MaX, mAX, mAx, maX, or max

#5: One of the character strings cO, Co, cO, or co

#6: One of the character strings ECT, ECt, Ect, EcT, eCT, eCt, ecT, or ect

Escape-character

Any underscore or percent sign coded in a pattern character string is unconditionally treated as a special character; these characters cannot be treated as regular characters. When a special character is to be specified as a regular character, an escape character must be specified. Specifying any character after the ESCAPE keyword (escape character) causes the special character following the escape character coded in the pattern character string to be treated as a regular character.

Example 1

A character string containing a '5%', such as '5%' and '25%':

```
'%5?%%' ESCAPE '?'
```

Example 2

A character string ending with 'PRINT_REC', such as 'SQLPRINT_REC':

```
'%PRINT@_REC' ESCAPE '@'
```

The following characters can be specified as escape characters:

Data type of item		Character that can be specified
Character string data (CHAR, VARCHAR)	Default character set	Any single character (single-byte character)#1
	EBCDIK	
	UTF16	Any single character#2
Mixed character string data (MCHAR, MVARCHAR)		Any single character#2
National character data (NCHAR, NVARCHAR)		Any single character (double-byte character)#3

Note

Care must be taken that the special character is specified following the escape character.

#1

All characters or data values are treated as single-byte characters.

#2

Data values that are not characters are treated as characters with the smallest length allowed for the character encoding (two bytes for UTF-16 and one byte for UTF-8 and SJIS).

#3

All characters or data values are treated as double-byte characters.

Notes

1. For improved performance, the defined length of the column used in a value expression on the left-hand side of XLIKE should be either a maximum of 255 bytes (CHAR, VARCHAR, MCHAR, or MVARCHAR) or a maximum of 127 characters (NCHAR or NVARCHAR).
2. Multi-byte characters stored in a CHAR or VARCHAR-type column are evaluated byte by byte. Consequently, if the character code for a single-byte character specified in a pattern character string is included in the character codes for the multi-byte characters, the result of the XLIKE predicate is true.

Example

Execute the following query in a condition where the query is set up in sjis character code, a CHAR-type column C1 is in Table T1, and a row ア is in column C1:

```
SELECT C1 FROM T1 WHERE C1 XLIKE '%A%' ;
```

The character code for the character ア in hexadecimal is 8341. The character code in hexadecimal for the character A in the pattern character string is 41. Therefore, because the character code for ア, which is a multi-byte character, includes the character code for the single-byte character A, the result of the XLIKE predicate is true.

(6) SIMILAR predicate

Format

value-expression [NOT] SIMILAR TO *pattern-character-string* [ESCAPE *escape-character*]

Conditions under which a predicate is TRUE

The SIMILAR predicate is TRUE for a row in which the value of a specified value expression matches the pattern expressed by a pattern character string. If NOT is specified, the predicate is TRUE for a row in which the value of a specified value expression does not match the pattern expressed by the pattern character string. Note that when the length of the pattern character string is 0, the SIMILAR predicate is TRUE when the length of the value expression is 0.

Rules

value-expression

1. Specifies the value expression that is to be compared with the character string pattern. However, you cannot specify a value expression that specifies only the ? parameter, or a value of an embedded variable.

2. The following data types can be specified in a value expression or a pattern character string: character string data, national character string data, mixed character string data, or BINARY with a maximum defined length of 32,000 bytes.
3. When a repetition column is specified, a subscript must be specified. If a repetition column is specified with a subscript and its elements meet specified conditions, the SIMILAR predicate will be TRUE.
4. ANY can be specified as a subscript for a repetition column. When ANY is specified, the SIMILAR predicate will be TRUE as long as at least one of the elements of the column satisfies the condition. If the SIMILAR predicate is not TRUE and if the condition specified for at least one of the elements of the column is unknown, the SIMILAR predicate will be FALSE. When the SIMILAR predicate is neither TRUE nor unknown, it is FALSE.
5. If a repetition column is specified with a subscript and the column contains no elements, the SIMILAR predicate will be unknown.

pattern-character-string

1. A value expression must be specified in the pattern character string.
2. You cannot specify a repetition column for a value expression.
3. The following combinations of value expression data types and pattern character string data types are allowed:

Data type of value expression		Data type of pattern character string or escape character					
		Character string data			National character string data	Mixed character string data	Binary data
		Default character set	EBC DIK	UTF16			
Character string data	Default character set	Y	N	N	N	Y	N
	EBCDIK	R[#3]	Y	N		N	
	UTF16	R[#3]	N	Y			
National character string data		R[#1]	N	N	Y	N	N
Mixed character string data		Y	N	N	N	Y	N
Binary data		R[#2]	N	N	N	N	Y

Legend:

Y: Can be specified.

R: Can be specified with restrictions.

N: Cannot be specified.

#1

Only a character string literal can be specified in a pattern character string. In this case, the character string literal is treated as a national character string literal. When a character string literal is treated as a national character string literal, the length of the character string data is checked, but the character encoding is not checked. When specifying a special character, make sure to specify it using a double-byte character.

#2

This can specify only a hexadecimal character string literal.

#3

This can specify only a character string literal, embedded variable, or ? parameter.

4. Use the same character set for the value expression, pattern character string, and escape character string. If the pattern character string and escape character string are one of those listed below, however, they are converted to the character set of the value expression, enabling comparison:
   - Character string literal
   - Embedded variable (default character set)
   - ? parameter
5. The format of a regular expression to be specified for a pattern character string is shown below:

```
regular-expression ::= normal-term | regular-expression | regular-expression
normal-term ::= normal-factor | normal-term normal-factor
normal-factor ::=    normal-primary
                   | normal-primary *
                   | normal-primary +
                   | normal-primary ?
                   | normal-primary repetition-factor
repetition-factor ::= { lower-limit [ upper-limit-specification ] }
upper-limit-specification ::= , [ upper-limit ]
normal-primary ::=    character-specifier
                    | %
                    | normal-character-set
                    | normal-character-set-identifier-specification
                    | ( regular-expression )
character-specifier ::=    non-escape-character
                         | escape-character
normal-character-set ::=    _
                          | [ character-list... ]
                          | [ ^ character-list... ]
character-list ::=    character-specifier
                    | character-specifier - character-specifier
                    | normal-character-set-identifier-specification
normal-character-set-identifier-specification ::= [ : normal-character-set-identifier : ]
```

6. The syntax rules for a regular expression to be specified for a pattern character string are described below:

   • Specify one of the following for the normal character set identifier:

   `'ALPHA'`, `'UPPER'`, `'LOWER'`, `'DIGIT'`, `'ALNUM'`, `'SPACE'`, `'WHITESPACE'`

   • The non-escaped characters include all the individual characters other than the following special characters:

Special character	Code representation in binary data		
	No character set specified	Character set specified	
		EBCDIK	UTF16
_ (underscore)	X'5F'	X' 6D'	U+005F
% (percent sign)	X' 25'	X' 6C'	U+0025
* (asterisk)	X' 2A'	X' 5C'	U+002A
+ (plus sign)	X' 2B'	X' 4E	U+002B
? (question mark)	X' 3F'	X' 6F'	U+003F

Special character	Code representation in binary data		
	No character set specified	Character set specified	
		EBCDIK	UTF16
| (vertical line)	X' 7C'	X' 4F'	U+007C
((left parenthesis)	X' 28'	X' 4D'	U+0028
) (right parenthesis)	X' 29'	X' 5D'	U+0029
{ (left curly bracket)	X' 7B'	X' C0'	U+007B
} (right curly bracket)	X' 7D'	X' D0'	U+007D
[(left square bracket)	X' 5B'	X' 4A'	U+005B
] (right square bracket)	X' 5D'	X' 5A'	U+005D
Escape character	Value specified in ESCAPE		
- (minus sign)#	X' 2D'	X' 60'	U+002D
: (colon)#	X' 3A'	X' 7A'	U+003A
^ (circumflex)#	X' 5E'	X' 5F'	U+005E

#: These symbols are treated as special characters only within a character string.

- To specify a special character as a regular character (that is, to escape it), you must specify it following the escape character.
- For the lower and upper limits, specify integers that satisfy the following condition: 0 ≤ lower limit ≤ upper limit ≤ 256.

7. The following table explains the meaning of each regular expression specification used in a pattern character string.

Table 2-7: Meaning of each regular expression specification

Regular expression specification	Meaning
Character specifier	Means a character (character string with a length of 1) specified by a character specifier.
_ (underscore)	Means a character with a length of 1.
% (percent sign)	Means a character string with a length of 0 or greater.
Normal primary*	Means 0 or more repetitions of the preceding normal primary.

Regular expression specification	Meaning
Normal primary+	Means one or more repetitions of the preceding normal primary.
Normal primary?	Means 0 or one repetition of the preceding normal primary.
regular-expression|*regular-expression*	Means either of the regular expressions specified before and after the vertical bar (|).
(*regular-expression*)	Means grouping of the regular expression specified within the parentheses. When a regular expression is to be used, this specification is used to clarify that it is a regular expression. It is used primarily when the vertical bar (|) is used.
normal-primary{*n*} *normal-primary*{*n*,*m*} *normal-primary*{*n*,}	Means that the preceding normal primary is repeated. The following shows how a repetition count is specified: {*n*}: Repeats the preceding regular expression *n* times. {*n*,*m*}: Repeats the preceding regular expression at least *n* times but not more than *m* times. {*n*,}: Repeats the preceding regular expression at least *n* times.
[*character-list*...]	Means any of the characters listed.
[^*character-list*...]	Means any character except those listed.
character-specifier-1-*character-specifier-2*	When specified in a character string list, means any character between the character indicated by *character-specifier-1* and the character indicated by *character-specifier-2* (character code range).
`[:ALPHA:]`	Any upper-case alphabetic character (excluding \, @, and #) or lower-case alphabetic character
`[:UPPER:]`	Any upper-case alphabetic character (excluding \, @, and #)
`[:LOWER:]`	Any lower-case alphabetic character
`[:DIGIT:]`	Any numeric digit
`[:ALNUM:]`	Any upper-case alphabetic character (excluding \, @, and #), lower-case alphabetic character, or numeric digit
`[:SPACE:]`	A single-byte space (double-byte space when the value expression is national character string data)
`[:WHITESPACE:]`	Any single character from among the characters whose character codes are listed in the table below (what is meant by `[:WHITESPACE:]` depends on the type of character codes)

The following table lists the character codes of the characters included in `[:WHITESPACE:]`:

Unicode(UTF-8)		Shift JIS kanji		EUC Japanese kanji, EUC Chinese kanji	Chinese kanji (GB18030)	LANG-C	Character name in the Unicode standard character set
Default character set	UTF16	Default character set	EBCDIK				
X'09'	U+0009	X'09'	X'05'	X'09'	X'09'	X'09'	Horizontal Tabulation
X'0A'	U+000A	X'0A'	X'15'	X'0A'	X'0A'	X'0A'	Line Feed
X'0B	U+000B	X'0B	X'0B'	X'0B	X'0B	X'0B	Vertical Tabulation
X'0C'	U+000C	X'0C'	X'0C'	X'0C'	X'0C'	X'0C'	Form Feed
X'0D'	U+000D	X'0D'	X'0D'	X'0D'	X'0D'	X'0D'	Carriage Return
X'20'	U+0020	X'20'	X'40'	X'20'	X'20'	X'20'	Space
X'C285'	U+0085	--	--	--	X'81308135'	--	Next Line[#]
X'C2A0'	U+00A0	--	--	--	X'81308432'	--	No-Break Space[#]
X'E19A80'	U+1680	--	--	--	X'814AC34'	--	Ogham Space Mark[#]
X'E28080'	U+2000	--	--	--	X'816A336'	--	En Quad[#]

Unicode(UTF-8)		Shift JIS kanji		EUC Japanese kanji, EUC Chinese kanji	Chinese kanji (GB18030)	LANG-C	Character name in the Unicode standard character set
Default character set	UTF16	Default character set	EBCDIK				
X'E28081'	U+2001	--	--	--	X'816A337'	--	Em Quad#
X'E28082'	U+2002	--	--	--	X'816A338'	--	En Space#
X'E28083'	U+2003	--	--	--	X'816A339'	--	Em Space#
X'E28084'	U+2004	--	--	--	X'816A430'	--	Three-Per-Em Space#
X'E28085'	U+2005	--	--	--	X'816A431'	--	Four-Per-Em Space#
X'E28086'	U+2006	--	--	--	X'816A432'	--	Six-Per-Em Space#
X'E28087'	U+2007	--	--	--	X'816A433'	--	Figure Space#
X'E28088'	U+2008	--	--	--	X'816A434'	--	Punctuation Space#
X'E28089'	U+2009	--	--	--	X'816A435'	--	Thin Space#

Unicode(UTF-8)		Shift JIS kanji		EUC Japanese kanji, EUC Chinese kanji	Chinese kanji (GB18030)	LANG-C	Character name in the Unicode standard character set
Default character set	UTF16	Default character set	EBCDIK				
X'E2808A'	U+200A	--	--	--	X'8136A436'	--	Hair Space#
X'E280A8'	U+2028	--	--	--	X'8136A635'	--	Line Separator#
X'E280A9'	U+2029	--	--	--	X'8136A636'	--	Paragraph Separator#
X'E280AF'	U+202F	--	--	--	X'8136A732'	--	Narrow No-Break Space#
X'E38080'	U+3000	0x8140	--	X'A1A1'	X'A1A1'	--	Ideographic Space#

Legend:

--: Not applicable

#

Not included in `[:WHITESPACE:]` when the value expression is character string type.

8. Binary data cannot be specified for a normal character set identifier.
9. You should note the following about specifying an embedded variable, SQL variable, or SQL parameter as a pattern character string:

   When a fixed-length embedded variable, SQL variable, or SQL parameter is specified as a pattern character string, setting a pattern character string that

is shorter than the length of the variable may cause spaces to be entered following the variable, or may set the remaining invalid characters as a value. When such a pattern character string is used for a search, data that is not followed by similar spaces or that does not contain the same value as the invalid characters is not retrieved. Therefore, when using a fixed-length variable as a pattern character string, you should fill the excess positions with the percent sign (%).

Example 1:

The following table lists the comparison results when AB% and AB%% are specified for the pattern character string and the character string data is ABCD.

Variable data type	Pattern character string	Character string data	Comparison result
Variable-length character string	'AB%'	'ABCD'	Matches.
	'AB%%'	'ABCD'	Matches.
Fixed-length character string (4 bytes)	'AB%Δ'	'ABCD'	Does not match.
	'AB%%'	'ABCD'	Matches.

Example 2:

The following table lists the comparison results when X'52454425 is specified for the pattern character string and the binary data is 52454452554d.

Variable data type	Pattern character string	Binary data	Comparison result
BINARY type	X'52454425'	52454452554d	Matches.
	X'5245442525'	52454452554d	Matches.

Invalid pattern character string

1. The following table describes the conditions that make a pattern character string invalid (KFPA11424-E message is issued):

Related item	Condition	Examples of invalid pattern character strings
Normal primary* Normal primary+ Normal primary?	Normal primary preceding * , + , or ? is not specified.	(*), (+), (?)

Related item	Condition	Examples of invalid pattern character strings
regular-expression|*regular-expression*	A regular expression is not specified on both sides of the vertical bar (|).	`a|`, `(a|)`, `(a||b)`
(regular-expression)	No regular expression is specified within the parentheses.	`()`
	Left and right parentheses do not match.	`(abc`, `abc)`
normal-primary{*n*} *normal-primary*{*n*,*m*} *normal-primary*{*n*,}	The normal primary that should precede the repetition factor is not specified.	`{4}`
	The repetition count specified for the repetition factor is invalid.	`a{-1}`, `a{4,2}`
	Left and right curly brackets do not match.	`a{4`, `a4}`
[*character-list*...] [^*character-list*...]	Character list contains a non-escaped special character. Note that a normal character set identifier can be specified.	`[a%c]`
	The character specified before or after the minus sign (`-`) is invalid.	`[-]`, `[c-a]`, `[a--]`
	No character list is specified within the square brackets.	`[]`, `[^]`
	Left and right square brackets do not match.	`[a-c`, `a-c]`
Escape character	The last character of a pattern character string is an escape character.	`abc\` (when \ is specified as the escape character)
Normal character set identifier specification	The normal character set identifier is invalid.	`[:INVALID:]`

Examples of pattern character strings

The following table provides examples of typical pattern character strings used in the `SIMILAR` predicate.

Table 2-8: Examples of pattern character strings used in the SIMILAR predicate

Item	Pattern character string	Meaning	Example	
			Pattern character string	Pattern-matching character string
Front match	*nnn*%	The leading portion of the character string is *nnn*.	'ACT%'	Character strings beginning with ACT, such as ACT, ACTOR, ACTION.
Rear match[#1]	%*nnn*	The trailing portion of the character string is *nnn*.	'%ING'	Character strings ending with ING, such as ING, BEING, and HAVING.
Any match	%*nnn*%	Any portion of the character string contains *nnn*.	N'%A%'	Character strings containing A, such as A, ACT, CA, and TACT[#2]
Complete match	*nnn*	Character string is the same as *nnn*.	'EQUAL'	EQUAL
Partial match	_...*nnn*_... _	A specific portion of the character string is the same as nnn, but the remaining characters are different.	'_I_'	Three-letter character strings in which the second letter is I, such as BIT, HIT, and KIT.
Repetition of at least once	*mmm*[0-9]+ or *mmm*[:DIGIT:]+	The leading portion of the character string is *mmm* and the trailing portion is a numerical value.	'KFPA11[0-9]+-E' or 'KFPA11[:DIGIT:]+-E'	Character strings beginning with KFPA11 in which numerics begin at the seventh character, followed by -E, such as KFPA11104-E and KFPA11901-E.
Several selected characters	*mmm*(*n*|*o*) or *mmm*[*no*]	The leading portion of the character string is *mmm* and the i^{th} character is *n* or *o* (*i* is any number).	'KFPA%-(W|E)' or 'KFPA%-[WE]'	Character strings beginning with KFPA in which the last two characters are -W or -E, such as KFPA20008-W and KFPA11901-E.
Repetition of between zero and one time	*nnno*?*mmm*	The leading and trailing portions of the character string are *nnn* and *mmm*, respectively, and the character W may or may not be present between the two character strings.	'OW?N'	Character strings beginning with O and ending with N in which the character W may or may not be present between O and N, such as ON and OWN.

Item	Pattern character string	Meaning	Example	
			Pattern character string	Pattern-matching character string
Repetition of at least zero times	*nnno***mmm*	The leading and trailing portions of the character string are *nnn* and *mmm*, respectively, and the character `0` is repeated at least 0 times between the character strings.	`10*1`	Character strings beginning with `1`, followed by `0` at least zero times, and ending with `1`, such as `11`, `101`, and `1001`.
n repetitions	*mmm*{*n*}	The leading portion of the character string is *mmm*, followed by *n* repetitions.	`[1-9]0{3}`	Character strings beginning with `1` through `9`, followed by `0` repeated three times, such as `1000`, `2000`, and `3000`.
Other	*nnn*%*mmm*	The leading and trailing portions of the character string are *nnn* and *mmm*, respectively.	`'O%N'`	Character strings beginning with `O` and ending with `N`, such as `ON`, `OWN`, and `ORIGIN`.
	%*nnn*%*mmm*%	Character string containing *nnn* anywhere within it, and containing *mmm* anywhere in the following portion.	`'%O%N%'`	Character strings containing the character `O` and also containing the character `N` in the following portion, such as `ON`, `ONE`, `DOWN`, and `COUNT`.
	*nnn*_...*_mmm*%	The leading portion of the character string is *nnn* and the trailing portion is *mmm*.	`'CO_ _ECT%'`	Character strings beginning with `CO` and in which the fifth through seventh characters are `ECT`, such as `CORRECT`, `CONNECTOR`, and `CONNECTION`.

Note:

nnn and *mmm* are any character strings that do not contain a special character.

#1

Because the space character is also used as a comparison character, comparison with data containing trailing spaces produces the `FALSE` result.

#2

Special characters appropriate to individual national characters are used for the special characters in a national character string.

escape-character

Special characters within a pattern character string cannot be handled as regular characters. When a special character needs to be specified as a regular character, an escape character must also be specified. The escape character is any character that you specify following the ESCAPE keyword. You can then specify the defined escape character before the special character in the pattern character string, which causes the special character to be handled as a regular character. You can specify as the escape character a character literal, ? parameter, embedded variable, SQL variable name, or SQL parameter name.

Example 1

A character string containing '5%', such as '5%' or '25%':

```
'%5\%%' ESCAPE '\'
```

Example 2

A character string ending with 'PRINT_REC', such as 'SQLPRINT_REC':

```
'%PRINT\_REC' ESCAPE '\'
```

Example 3

A hexadecimal character string containing X'48695244425f' in the binary string, such as X'48695244425f':

```
X'4869524442ee5f' ESCAPE X'ee'
```

The following characters can be specified as escape characters:

Data type of item		Character that can be specified
Character string data (CHAR, VARCHAR)	Default character set	Any single character (single-byte character)#1
	EBCDIK	
	UTF16	Any single character#2
Mixed character string data (MCHAR, MVARCHAR)		Any single character#2
National character data (NCHAR, NVARCHAR)		Any single character (double-byte character)#3
Binary data (BINARY)		Any single byte value

Note

You must be sure to specify the special character following the escape character.

#1

All characters or data values are treated as single-byte characters.

#2

Data values that are not characters are treated as characters with the smallest length allowed for the character encoding (two bytes for UTF-16 and one byte for UTF-8 and SJIS).

#3

All characters or data values are treated as double-byte characters.

Notes

1. For improved performance, the defined length of the column used in a value expression on the left-hand side of `SIMILAR` should be either a maximum of 255 bytes (`CHAR`, `VARCHAR`, `MCHAR`, `MVARCHAR`, or `BINARY`) or a maximum of 127 characters (`NCHAR` or `NVARCHAR`).
2. Multi-byte characters stored in a `CHAR` or `VARCHAR`-type column are evaluated byte by byte. Consequently, if the character code for a single-byte character specified in a pattern character string is included in the character codes for the multi-byte characters, the result of the `SIMILAR` predicate is true.

   Example:

   Execute the following query in a condition where the query is set up in `sjis` character codes, the `CHAR`-type column `C1` is in Table `T1`, and the row ア is in column `C1`:

   ```
   SELECT C1 FROM T1 WHERE C1 LIKE '%A%' ;
   ```

   The character code for the character ア in hexadecimal is `8341`. The character code in hexadecimal for the character `A` in the pattern character string is `41`. Therefore, because the character code for ア, which is a multi-byte character, includes the character code for the single-byte character `A`, the result of the `SIMILAR` predicate is true.
3. If the pattern character string is extremely long or if the special characters { } are specified consecutively, search performance may deteriorate or the amount of memory used may increase.

(7) BETWEEN predicate

Format

row-value-constructor-1 [NOT] BETWEEN
row-value-constructor-2 AND *row-value-constructor-3*

Conditions under which predicate is TRUE

The BETWEEN predicate is TRUE for those rows that satisfy the following condition:

row-value-constructor-2 ≤ *row-value-constructor-1* ≤ *row-value-constructor-3*

If NOT is specified, the BETWEEN predicate is TRUE for those rows that do not satisfy the above condition.

Rules

(*row-value-constructor-1*)

1. A row value constructor element consisting solely of a value specification cannot be specified.
2. When a repetition column is specified, a subscript must be specified. If a repetition column is specified with a subscript and its elements meet specified conditions, the BETWEEN predicate will be TRUE.
3. ANY can be specified as a subscript for a repetition column. If ANY is specified and at least one element in the column meets specified conditions, the BETWEEN predicate will be TRUE. If the BETWEEN predicate is not TRUE and a condition specified for at least one element in the column is UNKNOWN, the BETWEEN predicate will be UNKNOWN. If the BETWEEN predicate is neither TRUE nor UNKNOWN, it will be FALSE.
4. If a repetition column is specified with a subscript and the column contains no elements, the BETWEEN predicate will be UNKNOWN.

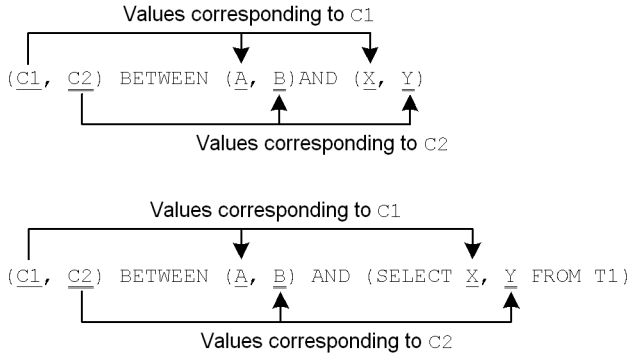
(*row-value-constructor-2* and *row-value-constructor-3*)

1. Repetition columns cannot be specified.

Common

1. Row value constructor elements located in corresponding positions in row value constructors are treated as corresponding values. Corresponding values must have data types that are mutually convertible. If, however, national character data is specified in *row-value-constructor-1*, and character string literals are specified in the corresponding values for *row-value-constructor-2* or *row-value-constructor-3*, the character string literals are treated as national character string literals. If a character string literal is treated as a national character string literal, HiRDB only checks the length of the character data, not the character code.

Corresponding values



2. Values of any of the following data types cannot be specified in *row-value-constructor-1*, *row-value-constructor-2*, or *row-value-constructor-3*:
   - BLOB
   - BINARY with a minimum length of 32,001 bytes
   - BOOLEAN
   - Abstract data type

(8) Quantified predicate

Format

```
row-value-constructor {= | <> | ^= | != | < | <=|> | >=}
     {{ANY | SOME} | ALL}
(table-subquery)
([SQL-optimization-specification-for-subquery-execution-method]
SELECT [{ALL | DISTINCT}] {selection expression |*}
(table-expression)
FROM table-reference [,table-reference]...
[WHERE search-condition]
[GROUP BY value-expression [,value-expression]...]
[HAVING search condition])
```

Conditions under which a predicate is TRUE

If either ANY or SOME is specified and if any one row in the results of a table subquery satisfies the comparison condition with respect to a row value constructor, the result of the quantified predicate is TRUE.

If ALL is specified and if all rows in the results of a table subquery satisfy the comparison condition with respect to a row value constructor, or if the result of a table

subquery is the empty set, the result of the quantified predicate is TRUE.

Rules

1. Values that yield any of the following data types as a result of an operation cannot be specified:
   - BLOB
   - BINARY with a minimum defined length of 32,001 bytes
   - BOOLEAN
   - Abstract data type
2. This predicate is indefinite with respect to rows in which the result of the row value constructor is null.
3. For table subqueries, see *2.4 Subqueries*.
4. The SOME quantified predicate and the ANY quantified predicate produce the same results.
5. Some quantified predicates have the same meaning as the IN predicate, as follows:

Quantified predicate	IN predicate
row-value-constructor = *row-value-constructor* = ANY *table-subquery* or *row-value-constructor* = SOME *table-subquery*	*row-value-constructor* IN *table-subquery*
row-value-constructor <> ALL *table-subquery*	*row-value-constructor* NOT IN *table-subquery*

6. The following table lists the results of a quantified predicate in which either ANY or SOME is specified. If any row in the results of a table subquery satisfies specified conditions, the result of the quantified predicate is TRUE. If the results of comparison of all rows are all FALSE or if the result of a table subquery is the empty set, the result of the quantified predicate is FALSE. If neither is the case, the result of the quantified predicate is indefinite.

   Table 2-9: Result of a quantified predicate with ANY or SOME specified

Result of comparing rows in subquery		Result of quantified predicate (ANY or SOME)
TRUE rows found		TRUE
No TRUE rows	Indefinite	Indefinite
	Not indefinite	FALSE
Empty set		FALSE

7. The table below lists the results of a quantified predicate with ALL specified. The result of the quantified predicate is TRUE if the comparison results of all rows in the results of a table subquery are TRUE or if the results of a table subquery are TRUE. If neither of the above two conditions is applicable, the quantified predicate is indefinite.

Table 2-10: Result of a quantified predicate with ALL specified

Result of comparing rows in subquery		Result of quantified predicate (ALL)
FALSE rows found		FALSE
No FALSE rows	Indefinite	Indefinite
	Not indefinite	TRUE
Empty set		TRUE

8. A qualified predicate cannot be specified in the following locations:
   - Search condition of an IF statement
   - Search condition of a WHILE statement
   - WHEN search condition (trigger action condition) of CREATE TRIGGER
9. When specifying a repetition column in a row value constructor element, specify a subscript. If a repetition column is specified with a subscript and its elements meet specified conditions, the quantified predicate will be TRUE.
10. ANY can be specified as a subscript for a repetition column. If ANY is specified and at least one element in the column meets specified conditions, the quantified predicate will be TRUE. If the quantified predicate is not TRUE and a condition specified for at least one element in the column is UNKNOWN, the quantified predicate will be UNKNOWN. If the quantified predicate is neither TRUE nor UNKNOWN, it will be FALSE.
11. If a repetition column is specified with a subscript and the column contains no elements, the quantified predicate will be UNKNOWN.
12. The number of columns in the row value constructor specified on the left side of a quantified predicate must have the same number of columns as the results of a table subquery.

Note

1. Specifying a quantified predicate may cause HiRDB to create a work table. In this case, the processing of the quantified predicate may be subject to restrictions, depending on the row length of the work table. For details about work table row lengths, see the *HiRDB Version 9 Installation and Design Guide*.

(9) EXISTS predicate

Format

```
EXISTS
  (table-subquery)
     ([SQL-optimization-specification-for-subquery-execution-method]
       SELECT [{ALL|DISTINCT}] {selection-expression|*}
      (Table-Expression)
      FROM table-reference [,table-reference]...
          [WHERE search-condition]]...
          [GROUP BY value-expression [, value-expression]...]
          [HAVING search-condition])
```

Conditions under which a predicate is TRUE

The results of the EXISTS predicate are TRUE unless the results of the table subquery are the empty set.

Rules

1. For table subqueries, see *2.4 Subqueries*.
2. The EXISTS predicate is used to determine whether the results of a table subquery are the empty set.
3. The following table lists the results of an EXISTS predicate. The result of the EXISTS predicate is TRUE if the results of a table subquery are one or more rows. The EXISTS predicate is FALSE if the result of the subquery is the empty set.

   Table 2-11: Result of EXISTS predicate

Number of rows meeting query condition as result of subquery	Result of EXISTS predicate
One or more rows	TRUE
0 rows	FALSE

4. The EXISTS predicate cannot be specified in the following locations:
   - Search condition of an IF statement
   - Search condition of a WHILE statement
   - WHEN search condition (trigger action condition) of CREATE TRIGGER

(10) Boolean predicate

Format

value-expression IS [NOT] {TRUE|FALSE|UNKNOWN}

Conditions under which a predicate is TRUE

If the logical value of a value expression matches the specified TRUE, FALSE, or UNKNOWN, the Boolean predicate will be TRUE. If NOT is specified and the logical value of the value expression does not match the specified TRUE, FALSE, or UNKNOWN, the Boolean predicate will be TRUE.

Rules

1. The following table lists the results of a predicate obtained by evaluating a Boolean predicate. If NOT is specified, the logical values shown in Table 2-10 are reversed.

   Table 2-12: Results of a predicate obtained by evaluating a Boolean predicate

Logical value of value expression	IS TRUE	IS FALSE	IS UNKNOWN
TRUE	TRUE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE
UNKNOWN	FALSE	FALSE	TRUE

2. Values in the following data type can be specified in a value expression:
   - BOOLEAN
3. An undefined Boolean value is the same as a null value.

(11) Structured repetition predicate

Format

```
ARRAY (column-specification [, column-specification]...)
    [ANY] (search-condition)
```

Conditions under which a predicate is TRUE

If the repetition column specified in ARRAY (*column-specification* [, *column-specification*]...) is treated as repetitions of multiple items that are a set of elements with the same subscript and if any of the elements meet specified search conditions, the structured repetition predicate will be TRUE.

Rules

```
ARRAY (column-specification [,column-specification]...)
```

1. *column-specification* specifies a repetition column to be structured.
2. The column specifications must be entirely from one index constituent column.
3. The following cannot be specified in a column specification:
   - Columns of different tables

- Columns derived from different tables

"Different tables" includes tables with the same base table but different correlation names.

4. Columns that make external references cannot be specified.
5. The same column cannot be specified more than once.
6. A maximum of 16 columns can be specified.

search-condition

Specifies a search condition. Structured repetition predicates are subject to the following rules.

1. None of the following items can be specified in a search condition:
   - Subscripted column specifications
   - Columns other than a repetition column specified in `ARRAY` (*column-specification* [`,` *column-specification*]...)
   - Predicates containing a system-defined scalar function, a function call, or `IS_USER_CONTAINED_IN_HDS_GROUP`
   - Structured repetition columns
   - `XMLEXISTS` predicate
   - Predicates not containing a column specification
   - Subqueries
2. If the `NULL` predicate is specified and the column does not contain any elements, the `NULL` predicate will be `UNKNOWN`. If a specified element is the `NULL` value, the `NULL` predicate will be `TRUE`.

Common rules

1. Search conditions that contain a structured repetition predicate cannot be negated by `NOT`.
2. Structured repetition predicates cannot be specified in a search condition in the `IF` or `WHILE` statement.
3. Structured repetition predicates cannot be specified in the `HAVING` clause.
4. Structured repetition predicates cannot be specified in a search condition in a `CASE` expression.
5. An `OR` containing a structured repetition predicate and any of the following columns (excluding external reference columns) in its operand search condition cannot be specified:

- Columns of different tables
- Columns derived from different tables

"Different tables" includes tables with the same base table but different correlation names.

6. Structured repetition predicates cannot be specified in a search condition in a derived query expression in a view definition.
7. When a structured repetition predicate in the ON search condition for a query specification including an outer-joined joined table is specified, specify columns of the inner table in the column specification.
8. When a structured repetition predicate in the WHERE clause of a query specification including an outer-joined joined table is specified, specify columns of the outermost table in the outer join in the column specification.

Usage example

From a listing of students' grades, find the names of students who had a minimum score of 85 in mathematics. The grades list consists of a repetition column of 10 elements composed of subjects and scores.

```
SELECT name FROM grades-list
   WHERE ARRAY (subject,score) [ANY]
      (subject='mathematics' AND score>=85)
```

Grades list

Name	Subject	Score
Bryant	Mathematics	90
	English	65
Smith	Mathematics	50
	English	90
Johnson	Mathematics	85
	English	70

Search results

Name
Bryant
Johnson

Note

The grades list has the following table definition and index definition:

Table definition:

```
CREATE TABLE grades-list (name MCHAR(10),
                          subject MCHAR(10) ARRAY[4],
                          score SMALLINT ARRAY[4];
```

Index definition:

```
CREAT INDEX subject-score
            ON grades-list (subject, score);
```

(12) XMLEXISTS predicate

A predicate that can be used with HiRDB XML Extension.

For details, see *1.14.4(1) XMLEXISTS predicate*.

2.8 Row value constructors

(1) Function

A row value constructor specifies either rows or a list of ordered columns.

(2) Format

row-value-constructor : := {*row-value-constructor-element*
| (*row-value-constructor-element*
[,*row-value-constructor-element*] ...) | *row-subquery*}
row-value-constructor-element : := *value-expression*

(3) Rules

1. The maximum number of row value constructor elements that can be specified in a row value constructor is 255.
2. The following data types cannot be specified for columns of the results of a row value constructor:
   - BLOB
   - BINARY with a minimum defined length of 32,001 bytes
   - Abstract data type
   - BOOLEAN
3. In the specification of two or more row value constructor elements, if a repetition column is specified as a row value constructor element, the subscript ANY cannot be specified.
4. For details about row subqueries, see *2.4 Subqueries*.

2.9 Value expressions, value specifications, and item specifications

(1) Function

Values can be specified in SQL in the formats shown below.

(2) Format

value-expression : := { [+ | -] *primary* | *value-expression* { + | - | * | / } [{ + | - }] *primary* | *value-expression* * *primary* | *value-expression* / *primary* | *value-expression* | | *primary* }

primary : := { (*value-expression*) | *item-specification*
| *unsigned-value-specification*
| *set-function* | *window-function* | *scalar-function*
| *CASE-clause* | *CAST-specification* | *labeled-interval*
| *function-call* | *scalar-subquery* | *NEXT VALUE-expression* }

value-expression : := { *literal* | ? *parameter* | : *embedded-variable*
[: *indicator-variable*]
| USER | CURRENT DATE | CURRENT_DATE
| CURRENT_TIME | CURRENT_TIMESTAMP [(*p*)]
| CURRENT DATE | CURRENT DATE
| CURRENT TIME | CURRENT TIMESTAMP [(*p*)]
| [*statement-label* .] *SQL-variable-name*
| [[*authorization-identifier* .] *routine-identifier* .]
SQL-parameter-name
| SQLCODE | SQLCOUNT
| SQLCODE_OF_LAST_CONDITION
| SQLERRM_OF_LAST_CONDITION }

unsigned-value-specification : := { *unsigned-numeric-literal* | *general-literal*
| ? *parameter* | : *embedded-variable*
[: *indicator-variable*]
| USER | CURRENT DATE | CURRENT_DATE
| CURRENT_TIME
| CURRENT_TIMESTAMP [(*p*)]
| CURRENT DATE | CURRENT DATE
| CURRENT TIME
| CURRENT TIMESTAMP [(*p*)]
| [*statement-label* .] *SQL-variable-name*
| [[*authorization-identifier* .]
routine-identifier .]
SQL-parameter-name
| SQLCODE | SQLCOUNT
| SQLCODE_OF_LAST_CONDITION
| SQLERRM_OF_LAST_CONDITION }

literal : := { *numeric-literal* | *general-literal* }

general-literal : : = { *character-string-literal* | *hexadecimal-character-string-literal*
| *national-character-string-literal*
| *mixed-character-string-literal* }
item-specification : : = { *column-specification*
| [*statement-label* .] *SQL-variable-name*
| [[*authorization-identifier* .] *routine-identifier* .]
SQL-parameter-name
| *component-specification* }

(3) Common rules

1. Of the operations that can be specified in a value expression, the arithmetic operations, date operations, time operations, concatenation operations, the CASE expression, the CAST specification, window functions, scalar functions, and the NEXT VALUE expression are collectively called scalar operations.
2. A value expression is specified in terms of a comparison predicate, comparison value, BETWEEN predicate, IN predicate, LIKE predicate, XLIKE predicate, quantified predicate, Boolean predicate, component specification, function call, column specification, update value, primary, set function, or scalar operation.
3. Scalar operations are evaluated in the following order:

   (1) Inside the parentheses

   (2) * or /

   (3) +, -, or | |

   However, multiple scalar operations of the same order of evaluation occurring in a value expression are evaluated from left to right.
4. The maximum number of allocatable nesting levels for scalar operations is 255. The number of nesting levels for scalar operations is the number of nesting levels of parentheses when the parentheses indicating the order of evaluation of the operators +, -, *, /, or | | are specified explicitly. The number of nesting levels associated with a scalar function, depending on the type of scalar function involved, is as follows:

   - When the scalar function SUBSTR, VARCHAR_FORMAT, TIMESTAMP_FORMAT, DATE (with a datetime format specified), TIME (with a datetime format specified), or TIMESTAMP (function 3) is specified, the number is 2.
   - For the scalar function VALUE, the number is the *number-of-value-expressions-of-arguments* + 1.
   - For the other scalar functions, the number is 1.

   For simple CASE expressions and search CASE expressions, the number is the

number of WHEN statements; for the CASE abbreviation COALESCE, it is the *number-of-value-expressions-of-arguments* + 1; for the CASE abbreviation NULLIF, it is 2.

If a scalar operation for which the operand is a column in a named derived table is specified, the column is derived from the scalar operation, and the named derived table does not create an inner derived table; such a specification is equivalent to specifying a scalar operation deriving a column of a named derived table for which the scalar operation is an operand. In this case, the maximum allowable number of nesting levels for scalar operations may be exceeded.

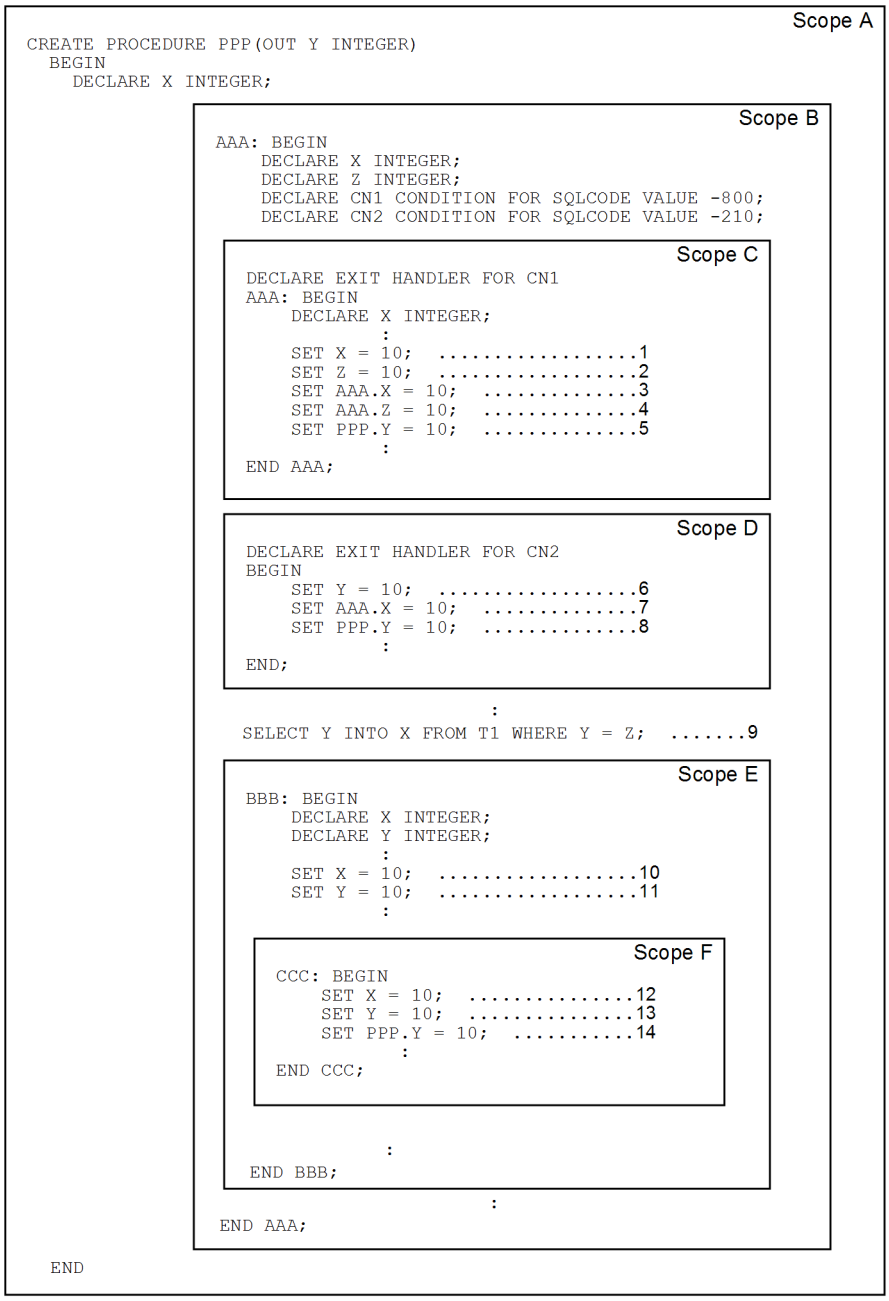
If scalar functions and function calls are specified in a value expression, the maximum allowable sum of scalar operation nesting levels and function call nesting levels is 255.

5. If specified data has the null value, the result of an arithmetic, date, time, or concatenation operation also has the null value.
6. In an arithmetic, date, or time operation involving division, an error results if 0 is specified as the value of the second operand.
7. An error results if overflow occurs during an operation.

   For value expressions, see *2.10 Arithmetic operations* through *2.13 Concatenation operation*.
8. An SQL parameter name is used in a procedure definition to reference the SQL parameter for that procedure or function.
9. An SQL variable name is used in a compound statement in a procedure definition or a function definition to reference the SQL variable declared in that compound statement.
10. If a column, an SQL variable, or an SQL parameter with the same name exists, these names must be qualified with a table specification, a statement label or with [*authorization-identifier* .]*routine-identifier*. If these items are not qualified or if they are qualified with the same set of qualifiers, they will be identified on a priority basis in ascending order of the scopes of the names. Thus, the following priority will be effective: columns, SQL variables, then SQL parameters. If an item is valid as a column name, it is identified as a column. If an item, though invalid as a column, is valid as an SQL variable, it is identified as an SQL variable. If an item, though invalid as a column or an SQL predicate, is valid as an SQL parameter, it is identified as an SQL parameter. If an item is invalid as a column, an SQL variable, or an SQL parameter, a syntax error results.

    However, in a handler declaration, statement labels and [*authorization-identifier* .] *routine-identifier* outside the handler declaration are not inherited. Consequently, if an SQL variable or an SQL parameter is qualified with a statement label or [*authorization-identifier* .] *routine-identifier* within the

handler declaration, only the statement label declared in the applicable handler declaration takes effect. The following shows examples of scopes of columns, SQL variables, and SQL parameters. The following examples assume that a table `T1` containing columns `Y` and `Z` of the `INTEGER` type is defined:



Explanation

1. The value 10 is assigned to the SQL variable X defined in scope C.
2. The value 10 is assigned to the SQL variable Z defined in scope B.
3. The value 10 is assigned to the SQL variable X defined in scope C.
4. An error occurs (a valid Z is not found in the handler in question).
5. An error occurs (PPP is not inherited because a handler is being declared).
6. The value 10 is assigned to the SQL parameter Y.
7. An error occurs (AAA is not inherited because a handler is being declared).
8. An error occurs (PPP is not inherited because a handler is being declared).
9. The value of column Y of table T1 is assigned to the SQL variable X defined in scope B. For Y and Z that are specified in the WHERE clause of the single-row SELECT statement, columns Y and Z are used from table T1.
10. The value 10 is assigned to the SQL variable X defined in scope E.
11. The value 10 is assigned to the SQL variable Y defined in scope E.
12. The value 10 is assigned to the SQL variable X defined in scope E.
13. The value 10 is assigned to the SQL variable Y defined in scope E.
14. The value 10 is assigned to the SQL parameter Y.

11. The following value specifications can only be used in routine control SQL, but not in an SQL procedure statement. The data types are indicated in parentheses:

    - SQLCODE (INTEGER)
    - SQLCODE_OF_LAST_CONDITION (INTEGER)
    - SQLCOUNT (INTEGER)
    - SQLERRM_OF_LAST_CONDITION (VARCHAR(254))

    SQLCODE and SQLCOUNT are used to determine whether the return code resulting from the execution of the immediately preceding SQL procedure statement without routine control SQL is 100, and to reference the number of updated rows.

    SQLCODE_OF_LAST_CONDITION and SQLERRM_OF_LAST_CONDITION are used to reference the last return code and message in the exception handler, respectively, if the return code resulting from the execution of an SQL procedure statement without routine control SQL is not 0.

12. A repetition column without a subscript cannot be specified in a scalar operation that is directly specified in a CASE expression search condition and that does not contain a repetition column.

13. The subscript ANY cannot be specified in a scalar operation that is directly

specified in a `CASE` expression search condition and that does not contain a repetition column.

14. For details about scalar subqueries, see *2.4 Subqueries*.

2.10 Arithmetic operations

(1) Types of arithmetic operations

Retrievals involving arithmetic operations can be specified in a value expression.

The following table lists the types of arithmetic operations and their functions.

Table 2-13: Types of arithmetic operations

Arithmetic operation		Meaning	Function
+	(Unary operation)	Positive sign	Leaves the sign unchanged.
	(Binary operation)	Addition	Adds the second operand to the first operand.
-	(Unary operation)	Negative sign	Reverses the sign.
	(Binary operation)	Subtraction	Subtracts the second operand from the first operand.
*	(Binary operation)	Multiplication	Multiplies the first operand by the second operand.
/	(Binary operation)	Division	Divides the first operand by the second operand.

(2) Data types of results of arithmetic operations

The following table indicates the relationships between the data types of the operands used in arithmetic operations (binary operations) and the data type of the result when the first operand data type is SMALLINT, INTEGER, DECIMAL, SMALLFLT, or FLOAT.

Table 2-14: Relationships between the data types of operands of arithmetic operations (binary operations) and the data type of a result

Data type of operand 1	Data type of operand 2				
	SMALLINT	INTEGER	DECIMAL	SMALLFLT	FLOAT
SMALLINT	INTEGER	INTEGER	DECIMAL	SMALLFLT	FLOAT
INTEGER	INTEGER	INTEGER	DECIMAL	FLOAT	FLOAT
DECIMAL	DECIMAL	DECIMAL	DECIMAL	FLOAT	FLOAT
SMALLFLT	SMALLFLT	FLOAT	FLOAT	SMALLFLT	FLOAT
FLOAT	FLOAT	FLOAT	FLOAT	FLOAT	FLOAT

This describes the relationships between the data types of the operands used in arithmetic operations (binary operations) and the data type of the result when the first operand data type is CHAR, VARCHAR, MCHAR, or MVARCHAR, or when the first operand

data type is SMALLINT, INTEGER, DECIMAL, SMALLFLT, or FLOAT and the second operand data type is CHAR, VARCHAR, MCHAR, or MVARCHAR.

Character string data is converted to the following numeric data types that represents the value of the character string. The resulting data type of the arithmetic operation is determined using the numeric data type after conversion, according to *Table 2-14 Relationships between the data types of operands of arithmetic operations (binary operations) and the data type of a result*:

- A character string that represents an integer is converted to INTEGER.
- A character string that represents a decimal is converted to DECIMAL.
- A character string that represents a floating point number is converted to FLOAT.

Arithmetic operations are performed using the data format already indicated. The data format of the result for a prefix operation is the same as the data format of the operand. The following table describes the resulting precision and decimal scaling position when the data type of a result of an arithmetic operation is DECIMAL.

Table 2-15: Precision and the decimal scaling position of a result when the data type of a result of an arithmetic operation is DECIMAL

Data type of result of arithmetic operation	Precision and decimal scaling position		
	Addition and subtraction	Multiplication	Division
DECIMAL(p,s)	$p = 1+\max(p_1 - s_1, p_2 - s_2) + s$ $s = \max(s_1,s_2)$	$p = p_1 + p_2$ $s = s_1 + s_2$	p = max_prec $s = \max(0,\text{max\_prec} - ((p_1 - s_1) + s_2))$
DECIMAL(p',s') (p > max_prec)	p' = max_prec $s' = \max(s_1,s_2)$		Not applicable

Note 1: Data type of operand 1: DECIMAL(p_1,s_1)

Data type of operand 2: DECIMAL(p_2,s_2)

Note 2:

max_prec is the maximum precision value of the DECIMAL type. The table below provides more details about max_prec. For details about the pd_sql_dec_op_maxprec operand, see the manual *HiRDB Version 9 System Definition*.

Table 2-16: Maximum precision value of the DECIMAL type

System common definition pd_sql_dec_op_maxprec operand	First operand precision p_1 and second operand precision p_2	max_prec value
29 or omitted	$p_1 \leq 29$ and $p_2 \leq 29$	29
	$p_1 > 29$ or $p_2 > 29$	38
38	Any	38

Note 3: `INTEGER` is treated as `DECIMAL`(10,0).

`SMALLINT` is treated as `DECIMAL`(5,0).

(3) Rules

For the rules for using arithmetic operations, see the rules below and the rules in Section *2.9 Value expressions, value specifications, and item specifications*.

1. Arithmetic operations are executed using numeric data. If character string data is specified, arithmetic operations are executed after the character string data is converted to numeric data.
2. The following value expressions, in which the result is character string data, cannot be specified in an arithmetic operation:
   - USER
   - SQLERRM_OF_LAST_CONDITION
3. When specifying an arithmetic operation that includes character string data in the following locations, only a character string literal or mixed character string literal can be specified as character string data:
   - Selection expression
   - Argument of a set function
   - Argument of a user-defined function (includes system-defined scalar functions)
   - `GROUP BY` clause
   - Value expression of a `VALUES` clause corresponding to a division column
   - Value expression of a `VALUES` clause in an `INSERT` statement or a `SET` clause in an `UPDATE` statement referenced with a new value correlation name
   - Value expression of a `SET` clause in an `UPDATE` statement referenced with an old correlation name

4. Embedded variables and value expressions involving the ? parameter only cannot be specified on both sides of an arithmetic operation (+, -, *, /).
5. When arithmetic operations of different data types are nested, the intermediate results must be handled without any loss in accuracy.
6. The window function cannot be specified in the operation term.

(4) Notes

1. The rules below apply to arithmetic operations; however, they do not apply (no error results) when the overflow error suppression feature is set:

   - An error occurs when 0 is specified as the value of the second operand in division
   - An error results when overflow occurs during computation

   For details about the operational results that are produced when the overflow error suppression feature is set, see *2.18 Operational results with overflow error suppression specified*.

2. When dividing DECIMAL values, if the total of the number of digits in the integer part of a dividend ($p_1 - s_1$) and the number of digits in the decimal fraction part of a divisor ($s2$) is greater than or equal to max_prec ($(p_1 - s_1) + s_2 \geq$ max_prec), the decimal scaling position of the division result is 0. To obtain the digits after the decimal point, use the scalar function DECIMAL before the division to decrease the precision (p_1) of the dividend or the decimal scaling position ($s2$) of the divisor.

   Reference

   The following expressions can be obtained from *Table 2-15 Precision and the decimal scaling position of a result when the data type of a result of an arithmetic operation is DECIMAL*.

   $p1$=max_prec-s+$s1$-$s2$
   $s2$=max_prec-s+$s1$-$p1$

   Example

   If the pd_sql_dec_op_maxprec operand of the system common definition is 29 or this operand is omitted, dividing the sum (SUM) of column C1 (DECIMAL(12,2)) and column C2 (DECIMAL(12,2)) in Table T1 results in the division of DEC(29,2) and DEC(29,2), and the result is DEC(29,0).If two places following the decimal point are desired, the DECIMAL scalar function can be used to reduce the precision of the dividend as follows:

```
SELECT DEC(SUM(C1),27,2)/SUM(C2) FROM T1
```

3. If arithmetic operations and concatenation operations are mixed, the data type of the result is determined by the order of operation and result of each operation.

Example:

Figure 2-5: Data type of result

For `1 + 2 - '1' || '3' + 5`

```
                    1   +   2   -   '1'   ||   '3'   +   5   →    28
Operation order  |INTEGER||INTEGER||VARCHAR| |VARCHAR||INTEGER| (INTEGER)
      ↓          |←    INTEGER   →||VARCHAR| |VARCHAR| |INTEGER|
      ↓          |←       INTEGER       →|   |VARCHAR| |INTEGER|
      ↓          |←           VARCHAR            →|   |INTEGER|
      ↓          |←                INTEGER                  →|
```

For `1 + 2 - '1' || '3' * 5`

```
                    1   +   2   -   '1'   ||   '3'   *   5   →    '215'
Operation order  |INTEGER||INTEGER||VARCHAR| |VARCHAR |INTEGER|    (VARCHAR)
      ↓          |←    INTEGER   →||VARCHAR| |VARCHAR| |INTEGER|
      ↓          |←       INTEGER       →|   |VARCHAR| |INTEGER|
      ↓          |←       INTEGER       →|     |←  INTEGER    →|
      ↓          |←               VARCHAR                   →|
```

2.11 Date operations

(1) Function

Date operations enable retrieval and updating involving dates and date intervals to be performed.

The date function supports the following operations: subtraction on dates; addition and subtraction of the dates between date intervals; addition and subtraction on date intervals; multiplication and division of a date interval by an integer.

(2) Data eligible for operations

The following table lists the date data or date interval data and applicable date operation values.

Table 2-17: Date operation data

Operation	Applicable date operation values	
	Date data	**Date interval data**
Addition	• Date interval data (INTERVAL YEAR TO DAY) • Literals that express date intervals as decimal numbers • Labeled duration (YEAR[S], MONTH[S], DAY[S])	• Date data (DATE) • Literals in which a date is specified in a predefined character string representation • Date interval data (INTERVAL YEAR TO DAY) • Literals that express date intervals as decimal numbers
Subtraction	• Date data (DATE) • Literals in which a date is specified in a predefined character string representation • Date interval data (INTERVAL YEAR TO DAY) • Literals that express date intervals as decimal numbers • Labeled duration (YEAR[S], MONTH[S], DAY[S])	• Date interval data (INTERVAL YEAR TO DAY) • Literals that express date intervals as decimal numbers
Multiplication, Division	Not specifiable	Integer data (INTEGER, SMALLINT)
Unary operation	Not specifiable in date data or as a labeled duration	

(3) Labeled duration

A labeled duration used in date operations expresses specific units of time as numeric

values followed by interval keywords. A labeled duration can be specified only in the second operand for addition or subtraction of date interval data relative to date data.

(4) Format

(*value-expression*){YEAR [S]|MONTH [S]|DAY [S]}

(5) Explanation

1. The following items can be specified in a value expression:
   - Integer literals
   - Column specification
   - Component specification
   - SQL variable or SQL parameter
   - Arithmetic operations
   - Set functions
   - Scalar functions
   - CASE expression
   - CAST specification
   - Function call
   - Scalar subquery
2. The value expression must have an integer data type (SMALLINT or INTEGER).
3. YEAR [S], MONTH [S], and DAY [S] indicate the units of years, months, and days, respectively. The S suffix is optional.

   Specification examples are given below:

   1 year: 1 YEAR

   11 months: 11 MONTHS

   100 days: 100 DAYS
4. The following ranges of values can be specified in a value expression:

   YEAR [S]: -9998 to 9998

   MONTH [S]: -199987 to 119987

   DAY [S]: -3652058 to 3652058

(6) Format of a date operation and the data type of the result

The following table indicates the relationships between the format of a date operation and the data type of the result.

Table 2-18: Relationships between the format of a date operation and the data type of the result

Operation format	Data type of result
date-data - *date-data*	Date interval data type
date-data {+|-} {*date-interval-data*|*labeled-duration*}	Date data type
date-interval-data + *date-data*	Date data type
date-interval-data {+|-} *date-interval-data*	Date interval data type
date-interval-data {*|/} *integer-data*	Date interval data type

Note: Embedded variables, indicator variables, and ? parameters cannot be specified in a date operation involving date data or date interval data.

(7) Rules for date operations

For the rules for performing date operations, see the rules below and the rules provided in Section *2.9 Value expressions, value specifications, and item specifications*.

(a) Rules for subtracting one date data item from another

- The data type of the result of subtraction involving two date data items is the date interval data type that expresses the number of years, months, and days.
- The result of the operation expression (*date1* - *date2*) is computed according to the following rules:

date1 ≥ *date2*:

Result = *date1* - *date2*

date1 < *date2*:

Result = - (*date2* - *date1*)#

Day of *date1* ≥ day of *date2*:

Day of result = day of *date1* - day of *date2*

Day of *date1* < day of *date2*:

Day of result = day of *date1* - day of *date2* + last day of month of *date2*

Month of *date2* = month of *date2* + 1

Month of *date1* ≥ month of *date2*:

Month of result = month of *date1* - month of *date2*

Month of *date1* < month of *date2*:

Month of result = month of *date1* - month of *date2* + 12

Year of *date2* = year of *date2* + 1

Year of result = year of *date1* - year of *date2*

#: The result of subtraction will be the result of *date2* - *date1* with a minus sign.

Example

Determine the result of the subtraction
`DATE('1995-10-15')-DATE('1989-12-16')`:

```
 1995-10-15 ← Date 1
-1989-12-16 ← Date 2
-------------------
 0005 years, 09 months, 30 days ← Computational result
```

(b) Explanation of the algorithm

(1) Day of result = 15 − 16 + 31 = 30
└ Last day of December

Month of date 2 = 12 + 1 = 13

(2) Month of result = 10 − 13 + 12 = 9
└ Month of date 2 determined in (1)

Year of date 2 = 1989 + 1 = 1990

(3) Year of result = 1995 − 1990 = 5
└ Year of date 2 determined in (2)

(4) From formulas (1), (2), and (3) above, the result is a date interval of 5 years, 9 months, 30 days.

(8) Rules for addition and subtraction of date data and date interval data

1. The result of addition or subtraction of date data or date interval data is the date data type.
2. The allowable range of the result of a computation is from 1/1/0001 through 12/31/9999.
3. Date interval data (not a labeled duration) is computed in the order of year, month, and day.
4. If the result of an operation on a year and month is a non-existent date (31st day of a 30-day month or February 29 of a non-leap-year), the date is changed to the last day of the month.# When a non-existent date is generated and the resulting date has been so modified, `'W'` is set in the `SQLWARNA` variable.

5. If the result of an operation on a year and month is beyond the last date of the month or is before the first day (1st) of the month, the year and month are rounded up or down, as appropriate.

#: Adding of months to the last day of a month does not necessarily produce the last day of the resulting month. Likewise, adding months to a day and subtracting the same number of months from the result does not necessarily produce the original date.

Example

```
DATE('1995-01-31') + 1 MONTH → DATE('1995-02-28')

DATE('1995-02-28') - 1 MONTH → DATE('1995-01-28')
```

Adding 1 month to 1/31/1995 results in 2/28/1995. However, subtracting 1 month from 2/28/1995 results in 1/28/1995, rather than the original 1/31/1995.

(9) Rules for addition or subtraction between date interval data, and multiplication or division of date interval data by an integer

1. The results of these computations take the date interval data type.
2. If the result of a date computation falls outside the range 00 to 99, the "W" warning is set to the SQLWARNC area. In this case, 00 is set as the number of days if the result is less than 00, and 99 is set if the result is greater than 99.
3. In the result of division, any digits following the decimal point are rounded off.

(10) Notes

The rules below for date operations do not result in an error when the overflow error suppression feature is set:

- Overflow occurs in the date data type
- Overflow occurs in the date interval data type
- Overflow occurs in the labeled duration data type

For details about the operational results produced when the overflow error suppression feature is set, see *2.18 Operational results with overflow error suppression specified*.

2.12 Time operations

(1) Function

Time operations enable retrieval and updating involving times and time intervals to be performed.

The time function supports the following operations: subtraction on times; addition and subtraction of the time between time intervals; addition and subtraction on time intervals; multiplication and division of a time interval by an integer.

(2) Data eligible for operations

The following table lists the time data or time interval data and applicable time operation values.

Table 2-19: Time operation data

Operation	Applicable time operation values	
	Time data	**Time interval data**
Addition	• Time interval data (INTERVAL HOUR TO SECOND) • Literals that express time intervals as decimal numbers • Labeled duration (HOUR[S], MINUTE[S], SECOND[S])	• Time data (TIME) • Literals in which time is specified in a predefined character string representation • Time interval data (INTERVAL HOUR TO SECOND) • Literals that express time intervals as decimal numbers
Subtraction	• Time data (TIME) • Literals in which time is specified in a predefined character string representation • Time interval data (INTERVAL HOUR TO SECOND) • Literals that express time intervals as decimal numbers • Labeled duration (HOUR[S], MINUTE[S], SECOND[S])	• Time interval data (INTERVAL HOUR TO SECOND) • Literals that express time intervals as decimal numbers
Multiplication, Division	Not specifiable	Integer data (INTEGER, SMALLINT)
Unary operation	Not specifiable in time data or as a labeled duration	

(3) Labeled duration

A labeled duration used in time operations expresses specific units of time as numeric values followed by interval keywords. A labeled duration can be specified only in the

second operand for addition or subtraction of time interval data relative to time data.

(4) Format

(*value-expression*) {HOUR[S]|MINUTE[S]|SECOND[S]}

(5) Explanation

1. The following items can be specified in a value expression:
   - Integer literals
   - Column literals
   - SQL variable or SQL parameter
   - Arithmetic operations
   - Set functions
   - Scalar functions
   - CASE expressions
   - CAST specification
   - Function call
   - Scalar subquery
2. The value expression must have an integer data type (SMALLINT or INTEGER).
3. HOUR[S], MINUTE[S], and SECOND[S] indicate the units of hours, minutes, and seconds, respectively. The S suffix is optional.

   Specification examples are given below:

   1 hour: 1 HOUR

   23 minutes: 23 MINUTES

   100 seconds: 100 SECONDS
4. The following ranges of values can be specified in a value expression:

   HOUR[S]: -23 to 23

   MINUTE[S]: -1439 to 1439

   SECOND[S]: -86399 to 86399

(6) Format of a time operation and the data type of the result

The following table indicates the relationships between the format of a time operation and the data type of the result.

Table 2-20: Relationships between the format of time operation and the data type of the result

Operation format	Data type of result
time-data - *time-data*	Time interval data type
time-data {+|-} {*time-interval-data*|*label-duration*}	Time data type
time-interval-data + *time-data*	Time data type
time-interval-data {+|-} *time-interval-data*	Time interval data type
time-interval-data {*|/} *integer-data*	Time interval data type

Note: Embedded variables, indicator variables, and ? parameters cannot be specified in a time operation involving time data or time interval data.

(7) Rules for time operations

For the rules for performing time operations, see the rules below and the rules provided in section *2.9 Value expressions, value specifications, and item specifications*.

(a) Rules for subtracting one time data item from another

- The data type of the result of subtraction involving two time data items is the time interval data type that expresses the number of hours, minutes, and seconds.
- When *time1* or *time2* is greater than or equal to 60 seconds, this is implicitly changed to 59 seconds before the operation is executed.
- The result of the operation expression (*time1* - *time2*) is computed according to the following rules:

  time1 ≥ *time2*:

  Result = *time1* - *time2*

  time1 < *time2*:

  Result = - (*time2* - *time1*)

  Second of *time1* ≥ second of *time2*:

  Second of result = second of *time1* - second of *time2*

  Second of *time1* < *second-of-time2*:

  Second of result = second of *time1* - second of *time2* + 60

  Minute of *time2* = minute of *time2* + 1

  Minute of *time1* ≥ minute of *time2*:

  Minute of result = minute of *time1* - minute of *time2*

Minute of *time1* < *minute-of-time2*:

Minute of *result* = minute of *time1* - minute of *time2* + 60

Hour of *time2* = hour of *time2* + 1

Hour of result = hour of *time1* - hour of *time2*

Example

Determine the result of the subtraction

```
TIME('13:10:15')-TIME ('11:50:59'):
```

```
  13:10:15 ← Time 1
- 11:50:59 ← Time 2
-----------
  1 hour, 19 minutes, 16 seconds ← Computational result
```

(b) Explanation of the algorithm

(1) Second of result = 15 − 59 + 60 = 16
└ 1 minute

Minute of time 2 = 50 + 1 = 51

(2) Minute of result = 10 − 51 + 60 = 19
└ Minute of time 2 determined in (1)

Hour of time 2 = 11 + 1 = 12

(3) Hour of result = 13 − 12 = 1
└ Hour of time 2 determined in (2)

(4) From formulas (1), (2), and (3) above, the result is a time interval of 1 hour, 19 minutes, 16 seconds.

(8) Rules for addition and subtraction of time data and time interval data

1. The result of addition or subtraction of time data or time interval data is the time data type.
2. The allowable range of the result of a computation is from 0:0:0 through 23:59:59.
3. Time interval data (not a labeled duration) is computed in the order of hour, minute, and second.
4. The result of operations using time data that includes a leap second is as follows:
   - If the number of seconds is greater than or equal to 60, this is implicitly changed to 59 before the operation is executed.

Examples:

The result of the operation `'12:34:60' + 1 MINUTE` is `'12:35:59'`.

The result of the operation `'12:34:60' + 1 SECOND` is `'12:35:00'`.

- Leap seconds are not included in the results of operations.

(9) Rules for addition or subtraction between time interval data, and multiplication or division of time interval data by an integer

1. The results of these computations take the date interval data type.
2. The result of an operation must be in the following range: -99 hours, 59 minutes, 59 seconds to 99 hours, 59 minutes, 59 seconds.
3. If the result of a time computation is greater than 60 minutes or 60 seconds, the result is carried to the next hour or minute, respectively.
4. Division is performed by converting a given time value into seconds. Any digits following the decimal point in the result are rounded off.

(10) Notes

The rules below for time operations do not result in an error when the overflow error suppression feature is set:

- Overflow occurs in the time data type.
- Overflow occurs in the time interval data type.
- Overflow occurs in the labeled duration data type.

For details about the operational results produced when the overflow error suppression feature is set, see *2.18 Operational results with overflow error suppression specified*.

2.13 Concatenation operation

(1) Function

When you specify a concatenation operation in a value expression, HiRDB concatenates multiple data strings (character strings or binary columns) in the specified order to create a single data string.

Note that you can specify concatenation operations on BLOB type data, and on BINARY type data that has a maximum length of 32,001 bytes or greater, only as an update value in the SET clause of the UPDATE statement. For details about concatenating BLOB type or BINARY type data with a defined length of 32,001 bytes or greater, see *Rules on updating a column of the BLOB type or the BINARY type with a defined length of 32,001 bytes or greater, using concatenation operations* under *Update statement Format 1 (Update data)* in Chapter 4. The format is the same as that for concatenation operations on character string data.

The following table lists the data types that can be concatenated and the data types resulting from concatenation operations for character string data and for BINARY type data that has a maximum length of no more than 32,000 bytes.

Table 2-21: Data types eligible for concatenation and the data type of a concatenation operation result (1/2)

Data type of operand 1		Data type of operand 2						
		Character data		National character data		Mixed character data		Binary data
		CHAR	VARCHAR	NCHAR	NVARCHAR	MCHAR	MVARCHAR	
Char data	CHAR	CHAR[#1]	VARCHAR	N	N	MCHAR[#1]	MVARCHAR	BINARY
	VARCHAR	VARCHAR	VARCHAR	N	N	MVARCHAR	MVARCHAR	N
Nat'l char data	NCHAR	N	N	NCHAR[#1]	NVARCHAR	N	N	BINARY[#2]
	NVARCHAR	N	N	NVARCHAR	NVARCHAR	N	N	N

Data type of operand 1		Data type of operand 2						
		Character data		National character data		Mixed character data		Binary data
		CHAR	VARCHAR	NCHAR	NVARCHAR	MCHAR	MVRCHAR	
Mixed char data	MCHAR	MCHAR[#1]	MVARCHAR	N	N	MCHAR[#1]	MVARCHAR	N
	MVARCHAR	MVARCHAR	MVARCHAR	N	N	MVARCHAR	MVARCHAR	N
Binary data	BINARY	N	BINARY[#2]	N	N	N	N	BINARY
Numeric data[#3, #4]	INTEGER, SMALLINT, or DECIMAL	VARCHAR	VARCHAR	N	N	MVARCHAR	MVARCHAR	N
	FLOAT or SMALLFLT	CHAR[#1]	VARCHAR	N	N	MCHAR[#1]	MVARCHAR	N

N: Cannot be specified.

Char: Character

Nat'l: National

#1

If the result length after concatenation is greater than the maximum length of CHAR, NCHAR or MCHAR, the result is treated as either VARCHAR, NVARCHAR or MVARCHAR.

#2

For the BINARY type, only a hexadecimal character string literal can be specified in the operation term.

#3

Numeric data is converted as follows:

- INTEGER type numeric data is converted to VARCHAR(11) character string data.
- SMALLINT type numeric data is converted to VARCHAR(6) character string data.
- DECIMAL(p,0) type numeric data is converted to VARCHAR(p+1)

character string data.

- DECIMAL(p,s) type numeric data is converted to VARCHAR(p+2) character string data.
- FLOAT or SMALLFLT type numeric data is converted to CHAR(23) character string data.

#4

When numeric data is concatenated, the numeric data is converted as follows:

- If numeric data and character string data are concatenated, numeric data is converted to the character set of the character string data.
- If numeric data and mixed character string data are concatenated, numeric data is converted to the character set of the mixed character string data.
- If numeric data and numeric data are concatenated, numeric data is converted to the default character set.

Table 2-22: Data types eligible for concatenation and the data type of a concatenation operation result (2/2)

First operand data type		Second operand data type	
		Numeric data#2, #3	
		INTEGER, SMALLINT, and DECIMAL	FLOAT and SMALLFLT
Character string data	CHAR	VARCHAR	CHAR#1
	VARCHAR	VARCHAR	VARCHAR
National character data	NCHAR	N	N
	NVARCHAR	N	N
Mixed character data	MCHAR	MVARCHAR	MCHAR#1
	MVARCHAR	MVARCHAR	MVARCHAR
Binary data	BINARY	N	N
Numeric data#2	INTEGER, SMALLINT, and DECIMAL	VARCHAR	VARCHAR
	FLOAT and SMALLFLT	VARCHAR	CHAR#1

Legend:

N: Cannot be specified.

#1

See footnote #1 following in *Table 2-21 Data types eligible for concatenation and the data type of a concatenation operation result (1/2)*.

#2

See footnote #3 following *Table 2-21 Data types eligible for concatenation and the data type of a concatenation operation result (1/2)*.

#3

See footnote #4 following *Table 2-21 Data types eligible for concatenation and the data type of a concatenation operation result (1/2)*.

Table 2-23: Data length of the result of concatenation

Data type of result of concatenation	Data length (maximum length for variable-length data)
CHAR(*n*)	$n = n_1 + n_2$ (if $n > 255$, the data type of the result is VARCHAR)
VARCHAR(*n*)	$n = n_1 + n_2$ (if $n > 32000$, an error results)
NCHAR(*n*)	$n = n_1 + n_2$ (if $n > 127$, the data type of the result is NVARCHAR)
NVARCHAR(*n*)	$n = n_1 + n_2$ (if $n > 16000$, an error results)
MCHAR(*n*)	$n = n_1 + n_2$ (if $n > 255$, the data type of the result is MVARCHAR)
MVARCHAR(*n*)	$n = n_1 + n_2$ (if $n > 32000$, an error results)
BINARY(*n*)	$n = n_1 + n_2$ (provided $n > 32{,}000$; an error in clauses other than the SET clause of the UPDATE statement)

n_1: Data length of operand 1

n_2: Data length of operand 2

Note 1: *n* is expressed in bytes for data types CHAR, VARCHAR, MCHAR, MVARCHAR, and BINARY. For NCHAR and NVARCHAR, *n* is expressed as the number of characters.

Note 2: If operands 1 and 2 are character string literals (including national and mixed character string literals) whose length is 0, n_1 or n_2 must be set to 0. However, if the data length of the concatenation result is 0, *n* is 1.

(2) Format

value-expression | | *primary*

(3) Rules

For concatenation operations on character string data, also see the common rules given in *2.9 Value expressions, value specifications, and item specifications*.

1. An embedded variable or a `?` parameter cannot be directly specified in an operand of a concatenation operation.
2. The null value is allowed in the result of a concatenation operation, regardless of the `NOT NULL` constraint on primaries or value expressions.
3. An error results if the data length (maximum length) of the result of a concatenation operation exceeds the allowable maximum length for variable-length data.
4. If the data type of the result of a concatenation operation is the `BINARY` type and the data length (maximum length) of the result is greater than 32,000 bytes, an error may occur in clauses other than the `SET` clause of the `UPDATE` statement.
5. If the character sets are different, you cannot perform concatenation operations on different character data. Concatenation operations can be performed, however, by converting the following value expression to the character set of the concatenation object:
   - Character string literal

2.14 Set functions

(1) Function

Specifying a set function in SQL allows you to calculate the average, sum, maximum value, minimum value, and the number of rows, as well as to concatenate XML type values.

For details about the XMLAGG set function, see *1.14.5(1) XMLAGG*.

The following table lists the functions of the set functions.

Table 2-24: Functions of set functions

Item	Set function						
	AVG	SUM	MAX	MIN	COUNT	COUNT_FLOAT	XMLAGG
Function	Calculates an average.	Calculates a sum.	Calculates a maximum value.	Calculates a minimum value.	Calculates the number of rows.	Calculates the number of rows.	Joins XML type values.
Treatment of null value	Ignored	Ignored	Ignored	Ignored	Ignored[#1]	Ignored[#1]	Ignored
Meaning of DISTINCT specification	Average value obtained by removing rows that contain a duplicated value in specified columns.	Sum obtained by removing rows that contain a duplicated value in specified columns.	Has no meaning.	Has no meaning.	Number of rows after removing rows that contain an identical value in specified value expressions.	Number of rows obtained by removing rows that contain a duplicated value in specified columns.	Cannot be specified.

Item	Set function						
	AVG	SUM	MAX	MIN	COUNT	COUNT_FLOAT	XMLAGG
Function value for a set in which the target of application is either zero[#3] or only the null value	Null value	Null value	Null value	Null value	0	0	Null value
Whether a repetition column can be specified in an argument[#2]	Cannot be specified.	Cannot be specified.	Can be specified.	Can be specified.	Can be specified.	Cannot be specified.	Can be specified.

#1: A null value is ignored in most set functions, except for `COUNT(*)` and `COUNT_FLOAT(*)`, where the total number of rows that satisfy the conditions is calculated regardless of null values.

#2: With the set functions `MAX` and `MIN`, a repetition column can be specified in an argument by coding a `FLAT` specification. If a repetition column is specified, the set functions `MAX` and `MIN` calculate either a maximum or a minimum from all the elements in the rows within the scope of the operation. This process excludes any columns in which the values of the entire column are the null value (a repetition column in which the number of elements is 0).

#3: If a `GROUP BY` or `HAVING` clause is specified, groups for which the number of rows is 0 are excluded from the calculation.

Table 2-25: Relationships between data types of columns and data types of function values

Data type of set function argument	AVG	SUM	MAX and MIN	COUNT	COUNT_FLOAT	XMLAGG
`INTEGER`	`INTEGER`	`INTEGER`	`INTEGER`	`INTEGER`	FLOAT	--
`SMALLINT`	`INTEGER`	`INTEGER`	`SMALLINT`	`INTEGER`	FLOAT	--
`DECIMAL`(*p*,*s*)	DECIMAL(`max_prec`[#3], `max_prec`[#3] - *p* + *s*)	DECIMAL(`max_prec`[#3], *s*)	`DECIMAL`(*p*,*s*)	`INTEGER`	FLOAT	--

Data type of set function argument	AVG	SUM	MAX and MIN	COUNT	COUNT_FLOAT	XMLAGG
FLOAT	FLOAT	FLOAT	FLOAT	INTEGER	FLOAT	--
SMALLFLT	SMALLFLT	SMALLFLT	SMALLFLT	INTEGER	FLOAT	--
INTERVAL YEAR TO DAY	--	--	INTERVAL YEAR TO DAY	INTEGER	FLOAT	--
INTERVAL HOUR TO SECOND	--	--	INTERVAL HOUR TO SECOND	INTEGER	FLOAT	--
CHAR(*n*)	--	--	CHAR(*n*)#1	INTEGER	FLOAT	--
VARCHAR(*n*)	--	--	VARCHAR(*n*)#1	INTEGER	FLOAT	--
NCHAR(*n*)	--	--	NCHAR(*n*)	INTEGER	FLOAT	--
NVARCHAR(*n*)	--	--	NVARCHAR(*n*)	INTEGER	FLOAT	--
MCHAR(*n*)	--	--	MCHAR(*n*)	INTEGER	FLOAT	--
MVARCHAR(*n*)	--	--	MVARCHAR(*n*)	INTEGER	FLOAT	--
DATE	--	--	DATE	INTEGER	FLOAT	--
TIME	--	--	TIME	INTEGER	FLOAT	--
TIMESTAMP	--	--	TIMESTAMP	INTEGER	FLOAT	--
BINARY(*n*)#2	--	--	BINARY(*n*)	INTEGER	FLOAT	--
BLOB(*n*)	--	--	--	--	--	--
BOOLEAN	--	--	--	--	--	--
XML	--	--	--	--	--	XML
Abstract data type(not including XML type)	--	--	--	--	--	--

--: Cannot be used.

#1

The character set of the result is the same as the character set of the value expression specified in the argument of the set function.

#2

n must be less than or equal to 32,000.

#3

max_prec is the maximum precision value of the DECIMAL type. The table below lists the values of max_prec. For details about the pd_sql_dec_op_maxprec operand, see the manual *HiRDB Version 9 System Definition*.

Table 2-26: Maximum precision value of the DECIMAL type

System common definition pd_sql_dec_op_maxprec operand	Precision p of set function argument	max_prec value
29 or omitted	$p \leq 29$	29
	$p > 29$	38
38	Any	38

(2) Format

set-function::={COUNT(*)|COUNT FLOAT(*)|ALL *set-function*|DISTINCT *set-function*|*XMLAGG-set-function*}
ALL *set-function*::={AVG|SUM|MAX|MIN|COUNT|COUNT FLOAT}([ALL] {*value-expression*|*FLAT-specification*})
DISTINCT *set-function*::= {AVG|SUM|MAX|MIN|COUNT|COUNT FLOAT} (DISTINCT {*value-expression*|*FLAT-specification*})
FLAT-specification:: = FLAT (*column-specification*)

For details about the format of the XMLAGG set function, see *1.14.5(1) XMLAGG*.

(3) Rules for when a subquery is not used

1. When GROUP BY, WHERE, or FROM clauses are specified, the groups that are obtained as a result of the last specified clause are used as the input to the set function.

   However, if the GROUP BY clause is not specified, the result of either the WHERE clause or the FROM clause becomes a group that does not have grouped columns (value expressions that are specified in the GROUP BY clause).

   The results of an operation by a set function are obtained for each group.

2. A set function can be specified only in the SELECT or HAVING clause.

3. When the GROUP BY clause, the HAVING clause, or a set function is specified, one of the following column specifications must be used in the SELECT or HAVING clause:

   - Grouped columns (value expressions specified in the GROUP BY clause)

- Specified in the argument of a set function

4. The SELECT DISTINCT specification and DISTINCT set-function are mutually exclusive.

(4) Rules for when a subquery is used

1. When a set function is specified in a subquery, the following can be used as input to the set function:

   - A set function can be specified for each query specification (including specifications inside the parentheses in a subquery). The sets that can be used as input to the set function are determined for each query specification.
   - The set function that is the input object of COUNT(*) or COUNT_FLOAT(*) is determined by the query specification that directly includes COUNT(*) or COUNT_FLOAT(*). Other sets that can be used as input to the set function are determined on the basis of the query specification that uses the FROM clause to specify the table that is referenced in the argument.
   - If a GROUP BY clause, a WHERE clause, or a FROM clause is specified in a query specification, the group that is obtained as the result of the last specified clause is used as the input to the set function.

   If a GROUP BY clause is not specified, the group that results from the WHERE or FROM clause does not have a grouping column (value expressions specified in a GROUP BY clause). The results of operation by a set function are obtained on a group-by-group basis.

2. A set function for query specification Q can be specified only in the SELECT or HAVING clause in query specification Q. A set function can also be specified in the ON search condition in a FROM clause, the WHERE clause, or HAVING clause in a subquery of the HAVING clause by referencing a column in the table for query specification Q as an argument (outer referencing).

3. When a GROUP BY clause or a HAVING clause is specified in a query specification, or when a set function is specified in a SELECT clause, any column specification in a SELECT or HAVING clause in the query specification must meet the following conditions:

   Columns specified in a SELECT clause

   - The column specification must reference the table in the FROM clause in the query specification.
   - The column specification either must be a grouping column or must be specified in an argument in the set function.

   Columns specified in a HAVING clause

   - The column specification references either the table in the FROM clause in the

query specification or a table in the FROM clause in an outer query specification (outer referencing).

- The column specification references a table in the FROM clause in the query specification, is a grouping column, or is specified in an argument in the set function.

4. If SELECT DISTINCT is specified in a query specification, a set function specifying DISTINCT for the query specification cannot be specified. A set function specifying the DISTINCT option is called a DISTINCT set function.
5. In a given query, an argument in a set function cannot specify an operation that includes an outer-referencing column.

(5) Common rules

1. A set function cannot be specified in any of the following clauses and statements: SET clause, IF statement, WHILE statement, SET statement, RETURN statement, WRITE LINE statement, ADD clause, GROUP BY clause.
2. Embedded variables and ? parameters cannot be specified in the argument of a set function.

   However, if an XMLQUERY function is specified in the value expression of an XMLAGG set function, embedded variables and the ? parameter can be specified in the value expression of an XML query variable in that XMLQUERY function.
3. Value expressions containing column specifications must be specified in the argument of a set function.
4. A set function or the window function cannot be specified in the argument of a set function.
5. The collating sequence for character string data is based on the character encoding used by the character set of the value expression specified in the argument of the set function.
6. The collating sequence of national character string data is based on the national character code being used (Shift JIS code, EUC Japanese kanji code, or EUC Chinese kanji code).
7. The collating sequence of mixed character string data is based on the ASCII encoding and national character encoding being used (Shift JIS code, EUC Japanese kanji code, EUC Chinese kanji code, Chinese kanji code (GB18030), or Unicode (UTF-8)).
8. In averages (AVG), the digits following the significant digits are rounded off.
9. An error results if overflow occurs during a computation, unless the overflow error suppression feature is set. The overflow error suppression feature is applicable to the following set functions:

- AVG
- SUM
- COUNT
- COUNT FLOAT

For details about the operational results produced when the overflow error suppression feature is set, see *2.18 Operational results with overflow error suppression specified*.

10. If the value of COUNT_FLOAT is 2^{53} (16 digits) or greater, the value is rounded off.
11. Component specifications cannot be specified in an argument of a set function.
12. The following rules apply to the specification of repetition columns in an argument in a set function:
    - Repetition columns can be specified only in the MAX or MIN set function for which a FLAT specification is coded.
    - Only unsubscripted repetition columns can be specified in a column specification that is coded in a FLAT specification.
    - Set functions for which a FLAT specification is coded cannot be specified in a query specification in which a value expression other than a column specification is specified in the GROUP BY clause.
13. Normally, only one DISTINCT set function can be specified per query specification.

    However, in the following cases, multiple DISTINCT set functions can be specified:
    - When different value expressions can be specified in the MAX and MIN set functions
    - Values described using the same format can be specified in set functions AVG, SUM, COUNT, and COUNT_FLOAT.
14. A subquery cannot be specified in a value expression that is specified as an argument of a set function.

(6) Usage examples

1. Determine the average stock quantities (SQUANTITY) of all products in a stock table (STOCK):
```
SELECT AVG(SQUANTITY) FROM STOCK
```
2. Determine the sum of the stock quantities (SQUANTITY) of the rows whose product name (PNAME) is skirt in a stock table (STOCK):
```
SELECT N'skirt',SUM(SQUANTITY)
```

```
    FROM STOCK
    WHERE PNAME=N'skirt'
```

3. Determine the total sum of each product's quantity in stock (SQUANTITY) times its unit price (PRICE) from a stock table (STOCK):

```
SELECT SUM(PRICE*SQUANTITY)
    FROM STOCK
```

4. Determine the maximum, minimum, and total number of entries for the quantities in stock (SQUANTITY) in a stock table (STOCK):

```
SELECT COUNT(*),MAX(SQUANTITY),MIN(SQUANTITY)
    FROM STOCK
```

5. From the following table of competition results, determine the highest score (maximum value) and the lowest score (minimum value) for the event with an event code 0001; assume that the column of scores is a repetition column for which the number of elements is 3:

Table of competition results

Player code	Event code	Score
A001	0001	3.0
		2.5
		4.5
A001	0002	6.5
		6.5
		7.0
A002	0001	5.0
		6.8
		6.0
A002	0002	6.5
		6.5
		7.0

SQL statement to be executed

```
SELECT MAX(FLAT(score)),MIN(FLAT(score))
   FROM table-of-competition-results
   WHERE event-code='0001'
```

Retrieval result

MAX(EXP)	MIN(EXP)
6.8	2.5

2.15 Window function

(1) Function

The window function determines a result from a window frame that specifies a window associated with the window function. The function of the window function is described below.

In the window frame, specify the portion of a set of rows from which data is to be collected. In a version that supports only `()` for the window specification, the window frame indicates the entire range of a table derived as the result of a `WHERE` or `FROM` clause.

Table 2-27: Function of the window function

Window function type	Explanation
`COUNT(*)`	Sets the number of rows to be input into the window frame.

(2) Format

window-function `::=COUNT(*)`
`OVER` *window-specification*
window-specification `::=()`

(3) Operands

- *window-function* `::=COUNT(*)`
  `OVER` *window-specification*

  window-specification `::=()`

The following rules apply to the window function:

1. The window function can be specified in a selection expression.
2. The result data type of the window function `COUNT(*) OVER()` is `INTEGER`.
3. An error results if overflow occurs during an operation, unless the overflow error suppression feature is set. For details about the operational results when overflow error suppression is specified, see *2.18 Operational results with overflow error suppression specified*.
4. The window function cannot be specified in the following locations:
   - Query expression body of an `INSERT` statement
   - Subquery

- Derived table
- Derived query expression of a view definition
- Derived query expression inside a `WITH` clause

5. The window function cannot be specified in a query specification, derived query expression, or query expression body that is the target of a set operation.
6. When the window function is specified, a `GROUP BY` or `HAVING` clause cannot be specified.
7. When the window function is specified in a selection expression, at least one selection expression must be specified, in addition to the window function.
8. The window function cannot be specified in a scalar operation.
9. When the window function is specified, a set function cannot be specified in the selection expression.

(4) Notes

1. If the input to the function is an empty set, set function `COUNT(*)` outputs `0`, but if the window function `COUNT(*) OVER()` is specified, no search result row is output.

(5) Usage examples

A usage example of the window function is described below. The table used in this example has the following structure:

Scores table (`SCORES`)

ID	ROUND	POINTS
A	1st round	5
A	2nd round	6
B	1st round	3
B	2nd round	7
C	1st round	5
C	2nd round	5

1. From the scores table (`SCORES`), determine numbers (`ID`), points (`POINTS`), and total (`TOTAL`), in descending order of the points.

```
SELECT "ID", "POINTS", COUNT(*) OVER() AS "TOTAL"
  FROM "SCORES" ORDER BY "POINTS" DESC
```

<Execution result>

ID	POINTS	TOTAL
B	7	6
A	6	6
A	5	6
C	5	6
C	5	6
B	3	6

2.16 Scalar functions

Scalar functions are used as data-type variables, for the partial extraction of data, and for value conversions; they are specified in selection expressions in an SQL query or in search conditions.

HiRDB provides the following three types of scalar functions:

- System built-in scalar functions

  These scalar functions can be specified anywhere a scalar function can be specified.

- System-defined scalar functions

  These scalar functions can be specified in locations where a function call can be specified.

- Plug-in definition scalar function

  The plug-in definition scalar function can be specified anywhere a function can be invoked.

The following table lists the scalar functions.

Table 2-28: List of scalar functions

Classification	Scalar code function	Function	Type of scalar function
Conversion function	INTEGER	Converts numeric data into integer data.	Built-in
	DECIMAL	Converts numeric data into decimal data.	Built-in
	FLOAT	Converts numeric data into floating-point data.	Built-in
	DIGITS	Extracts the numeric part of integer, decimal, date interval, or time interval data and converts it into a character string representation.	Built-in
	NUMEDIT	Edits a numeric value and converts it into a character string representation.	Defined
	STRTONUM	Converts the character string representation of a numeric value into a numeric data type.	Defined
	CHARACTER	Converts date data, time data, or time stamp data into a character string representation.	Built-in

Classification	Scalar code function	Function	Type of scalar function
	VARCHAR_FORMAT	Converts date data, time data, or time stamp data into a character string representation in a specified format.	Built-in
	DATE	Converts the character string representation of a date in a specified format into date data. Converts a given cumulative number of days from January 1, year 1 (A.D.) into date data representing that date.	Built-in
	DAYS	Converts either given date data or time stamp data into a cumulative number of days from January 1, year 1 (A.D.).	Built-in
	TIME	Converts the character string representation of a given time in a specified format into time data.	Built-in
	TIMESTAMP	Converts the predefined character string representation of a time stamp into the time stamp data. Converts the cumulative number of days from January 1, year 1 (A.D.) into time stamp data that it represents. Given date data and time data, converts the results into time stamp data combining the data items.	Built-in
	TIMESTAMP_FORMAT	Converts the character string representation of a time stamp in a specified format into time stamp data.	Built-in
	MIDNIGHTSECONDS	Determines the number of seconds from 00:00:00 a.m. to a specified time value.	Defined
	HEX	Converts a numeric expression into a hexadecimal character string representation.	Built-in
	ASCII	Converts a given character into its ASCII code.	Defined
	CHR	Converts a given ASCII code into its character equivalent.	Defined
	RADIANS	Converts a given angle from degrees to the equivalent radian measure.	Defined
	DEGREES	Converts a given angle from a radian measure to the equivalent degrees.	Defined

Classification	Scalar code function	Function	Type of scalar function
	CAST specification	Converts value expression data into a specified data type. For CAST specification, see *2.25 CAST specification*.	Not applicable
Extraction function	YEAR	Extracts the year part from date data, time stamp data, or date interval data.	Built-in
	MONTH	Extracts the month part from date data, time stamp data, or date interval data.	Built-in
	DAY	Extracts the day part from date data, time stamp data, or date interval data.	Built-in
	HOUR	Extracts the hour part from time data, time stamp data, or time interval data.	Built-in
	MINUTE	Extracts the minute part from time data, time stamp data, or time interval data.	Built-in
	SECOND	Extracts the second part from time data, time stamp data, or time interval data.	Built-in
Mathematical function	ABS	Returns the absolute value of a given numeric expression.	Built-in
	MOD	Returns the remainder of a division.	Built-in
	CEIL	Determines the smallest integer greater than or equal to a given numeric value.	Defined
	FLOOR	Determines the largest integer less than or equal to a given numeric value.	Defined
	TRUNC	Truncates a given numeric value below a specified digit.	Defined
	ROUND	When a boundary between rounding up and rounding off is specified, rounds a given numeric value to a specified number of digits, or performs rounding.	Defined
	SIGN	Determines the sign of a given numeric value in terms of 1 (positive), 0, and -1 (negative).	Defined
	SQRT	Determines the square root of a given numeric value.	Defined
	POWER	Determines the power of a given numeric value.	Defined

Classification	Scalar code function	Function	Type of scalar function
	EXP	Determines the power to the base of the natural logarithm.	Defined
	LN	Determines the natural logarithm of a given numeric value.	Defined
	LOG10	Determines the common logarithm of a given numeric value.	Defined
	SIN	Determines the sine (trigonometric function) of an angle specified in radian measure.	Defined
	COS	Determines the cosine (trigonometric function) of an angle specified in radian measure.	Defined
	TAN	Determines the tangent (trigonometric function) of an angle specified in radian measure.	Defined
	ASIN	Determines the inverse sine (trigonometric function) of an angle specified in radian measure.	Defined
	ACOS	Determines the inverse cosine (trigonometric function) of an angle specified in radian measure.	Defined
	ATAN	Determines the inverse tangent (trigonometric function) of an angle specified in radian measure.	Defined
	ATAN2	Determines the inverse sine (trigonometric function) of a given point (x, y) in terms of radian measure.	Defined
	SINH	Determines the hyperbolic sine of a given numeric value.	Defined
	COSH	Determines the hyperbolic cosine of a given numeric value.	Defined
	TANH	Determines the hyperbolic tangent of a given numeric value.	Defined
	PI	Determines the circle ratio π .	Defined

Classification	Scalar code function	Function	Type of scalar function
Character string manipulation function	SUBSTR	Determines a partial data string of a specified number of characters or length from a specified position in a given data string (character string or binary string).	Built-in
	LEFTSTR	Determines a partial character string of a specified number of characters from the beginning of a given character string.	Defined
	RIGHTSTR	Determines a partial character string of a specified number of characters from the end of a given character string.	Defined
	UPPER	Converts lower case characters into upper case characters in given character string data.	Built-in
	LOWER	Converts upper case characters into lower case characters in given character string data.	Built-in
	TRANSL (TRANSL_LONG)	Translates a specified character in a character string into another, equivalent character.	Defined
	LTRIM	Trims either spaces or specified characters from the left.	Defined
	RTRIM	Trims either spaces or specified characters from the right.	Defined
	LTRIMSTR	Trims a specified character string from the left.	Defined
	RTRIMSTR	Trims a specified character string from the right.	Defined
	REPLACE (REPLACE_LONG)	Repeatedly replaces partial character strings in a given character string with another character string.	Defined
	INSERTSTR (INSERTSTR_LONG)	Deletes a partial character string of a specified number of characters from a specified position, and inserts another character string into that position.	Defined
	POSSTR	Determines the character position of a specified n^{th} partial character string that occurs after a specified position in a given character string.	Defined

Classification	Scalar code function	Function	Type of scalar function
	POSITION	Determines the position of the first data substring that occurs at a specified position or beyond in a data string (either a character string or a binary string).	Built-in
	REVERSESTR	Determines a right-left reversed character string.	Defined
Date manipulation function	NEXT_DAY	Determines the date of a specified day of week following a given date.	Defined
	LAST_DAY	Determines the last day of the year in which a specified date falls.	Defined
	DAYOFWEEK	Given a date in day of week, determines the ordinal number of the day in that week.	Defined
	DAYOFYEAR	Given a date, determines the ordinal number of the date in that year.	Defined
	DAYNAME	Determines the day of week of a specified date in English.	Defined
	WEEK	Given a date, determines the ordinal number of the week in which the date falls in that year.	Defined
	WEEKOFMONTH	Given a date, determines the ordinal number of the week in that month.	Defined
	MONTHNAME	Determines the name of the month of a specified date in English.	Defined
	ROUNDMONTH	Given a smallest number of days beyond which days are to be rounded up, determines the year and month of a specified date by rounding the day.	Defined
	TRUNCYEAR	Given the first month and day of a fiscal year, determines the first date of that fiscal year.	Defined
	QUARTER	Given the first month and day of a fiscal year, determines the quarter in which that date falls.	Defined
	HALF	Given the first month and day of a fiscal year, determines whether that date falls in the first half or the second half.	Defined
	CENTURY	Determines the century of a specified date.	Defined

Classification	Scalar code function	Function	Type of scalar function
	MONTHS_BETWEEN	Determines the number of months between given dates as a real number.	Defined
	YEARS_BETWEEN	Determines the number of years between given dates as a real number.	Defined
Datetime manipulation function	DATE_TIME	Concatenates date data and time data, and converts the result into a predefined character string representation of a time stamp.	Defined
	INTERVAL_DATETIMES	Determines the date and time interval between time stamps that are given in a predefined character string representations.	Defined
	ADD_INTERVAL	Adds a given date and time interval to a date stamp in a predefined character string representation.	Defined
Inspection function	ISDIGITS	Determines whether all characters in a given character string are digits.	Defined
	IS_DBLBYTES	Determines whether all characters in a given character string are double-byte characters.	Defined
	IS_SNGLBYTES	Determines whether all characters in a given character string are single-byte characters.	Defined
XML type value manipulation function	XMLQUERY	Evaluates XQuery expressions and generates an XML type value as the result.	Plug-in
	XMLSERIALIZE	Generates a VARCHAR or BINARY type value from an XML type value.	Plug-in
	XMLPARSE	Generates an XML type value from an XML document.	Plug-in
Other functions	LENGTH	Determines the data length of a value expression.	Built-in
	VALUE	Extracts the first non-null value expression from a list of value expressions.	Built-in
	GREATEST	Determines the maximum value of arguments.	Defined
	LEAST	Determines the minimum value of arguments.	Defined
	BIT_AND_TEST	Determines the logical product of specified arguments bit-by-bit and returns the result in terms of true or false.	Built-in

Classification	Scalar code function	Function	Type of scalar function
	CASE expressions	Specifies a conditional value. For CASE expressions, see *2.17 CASE expressions*.	Not applicable

Legend:

Built-in: System built-in scalar function

Defined: System-defined scalar function

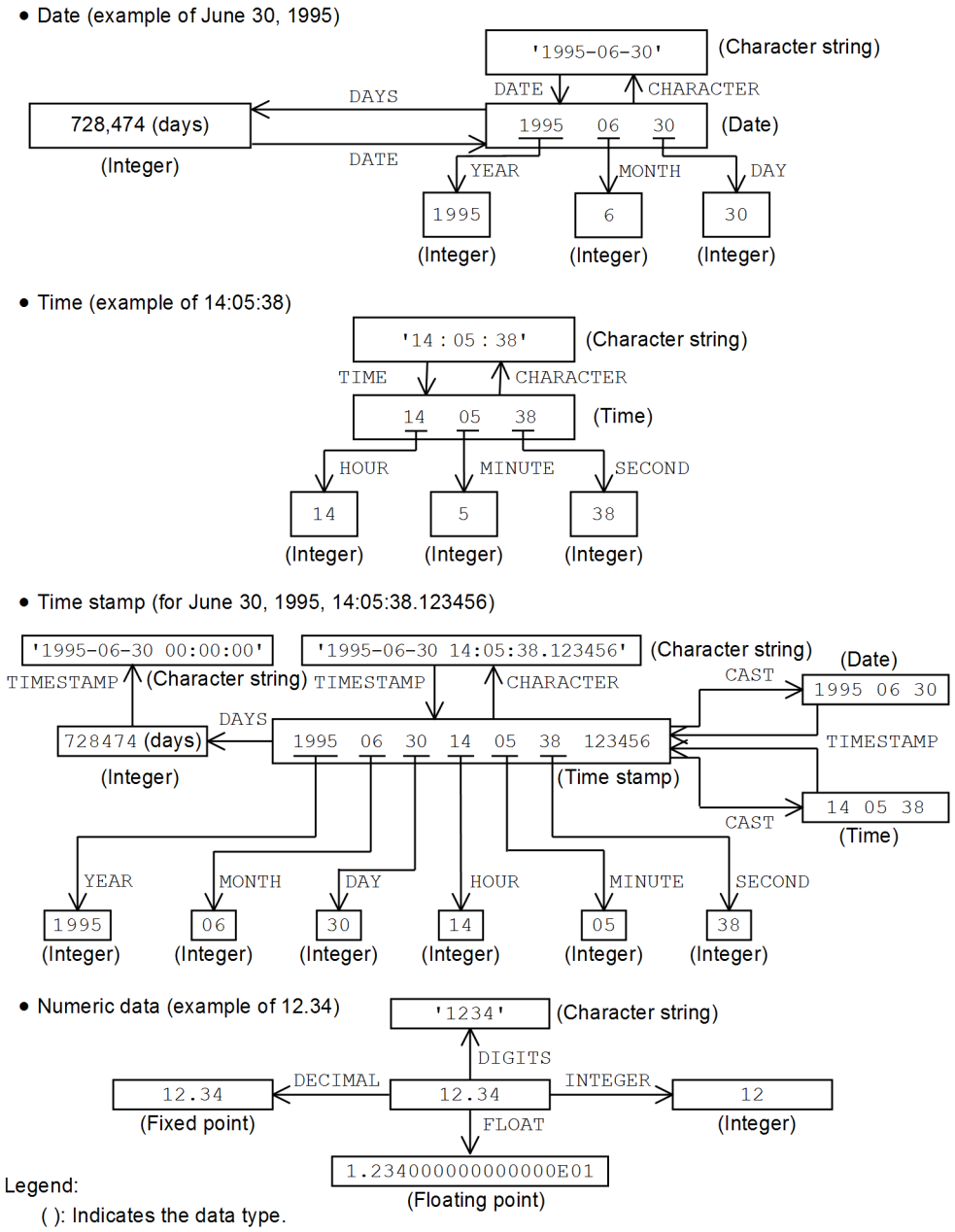
Plug-in: Plug-in defined scalar function

2.16.1 System built-in scalar functions

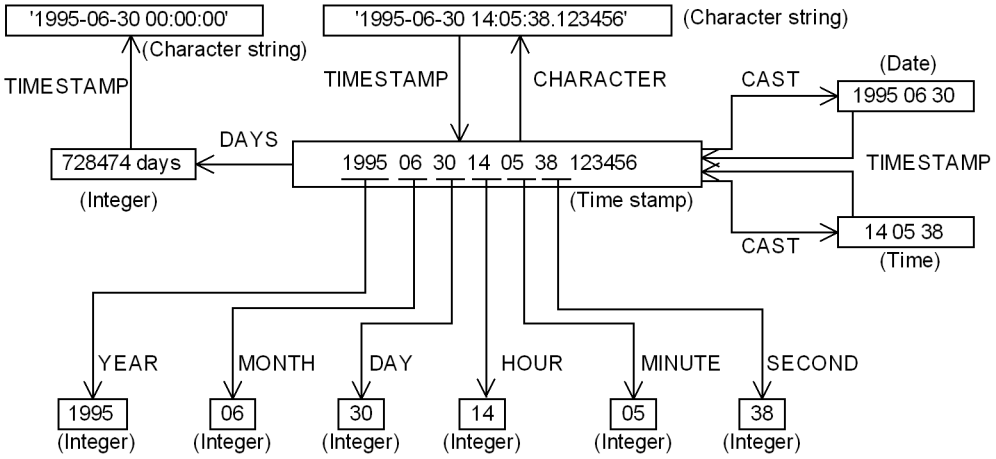
This subsection explains the syntax of system built-in scalar functions.

The following figure shows examples of executing system built-in scalar functions with respect to date data, time data, time stamp data, and numeric data.

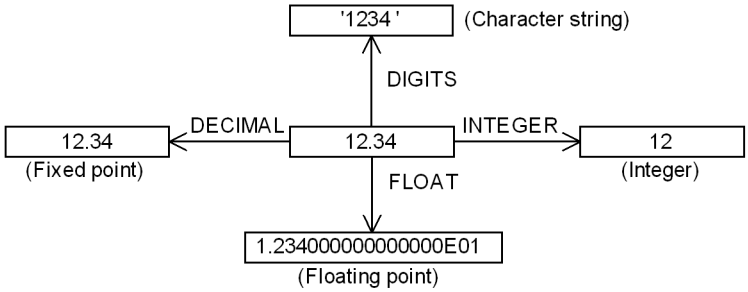
Figure 2-6: Execution examples of system built-in scalar functions with respect to date data, time data, time stamp data, and numeric data



• Time stamp (June 30, 1995, 14:5:38.123456)



• Numeric data (Example: 12.34)



( ): Indicates the data type.

Common rules

The following rules apply to system built-in scalar functions:

1. An embedded variable or the ? parameter cannot be specified in a value expression by itself. However, they can be specified in value expressions involving arithmetic operations (except for unary operators).
2. When a repetition column is specified as a value expression in an argument, a subscript must also be specified; however, the ANY subscript cannot be specified.

(1) ABS

(a) Function

The ABS scalar function returns the absolute value of a value expression.

(b) Format

ABS(*value-expression*)

(c) Rules

1. The following items can be specified as the value expression:
   - Numeric literals
   - Column specifications
   - Component specification
   - SQL variables or SQL parameters
   - Arithmetic operations
   - Set functions
   - Scalar functions
   - CASE expressions
   - CAST specification
   - Function call
   - Scalar subquery
2. The data type of a value expression must be numeric data, date interval data, or time interval data.
3. The data type of the result will be the same as the data type of the value expression.
4. The NOT NULL constraint does not apply to the result value (the null value is allowed). If the value expression is the null value, the result will also be the null value.
5. The result must be a value that can be expressed as the absolute value of the value expression. If a value that cannot be expressed as the absolute value is specified, an overflow error occurs (for the result when overflow error suppression is set, see *2.18 Operational results with overflow error suppression specified*).

(2) BIT_AND_TEST

(a) Function

Determines a bit-by-bit logical product of *value-expression-1* and *value-expression-2*, and returns the BOOLEAN value TRUE if any of the bits in the results of the logical product is 1.

(b) Format

```
BIT_AND_TEST(value-expression-1, value-expression-2)
```

(c) Rules

1. The following items can be specified in *value-expression-1* and *value-expression-2*:
   - Literals
   - USER
   - Column specification
   - Component specification
   - SQL variables or SQL parameters
   - Concatenation operations
   - Set functions
   - Scalar functions
   - CASE expressions
   - CAST specification
   - Embedded variables or ? parameters
   - Function calls
   - Scalar subquery
2. The data type of *value-expression-1* and *value-expression-2* must be either a character data type (CHAR or VARCHAR) or BINARY with a maximum length of 32,000 bytes. The following table indicates the combinations of data types that can be specified in *value-expression-1* and *value-expression-2*.

   Table 2-29: Combinations of data types that can be specified in value-expression-1 and value-expression-2 (system built-in scalar function BIT_AND_TEST)

Data type of value expression 1	Data type of value expression 2	
	Character data (CHAR and VARCHAR)	Binary data (BINARY)
Character data (CHAR and VARCHAR)	Y	N#
Binary data (BINARY)	N#	Y

Legend:

Y: Can be specified.

N: Cannot be specified.

#: Only hexadecimal character string literals can be specified as a character data value expression.

1. Value expressions consisting solely of embedded variables or ? parameters cannot be specified in both *value-expression-1*and *value-expression-2*.
2. If one value expression is an embedded variable or a ? parameter, HiRDB assumes that the data type of the embedded variable or ? parameter is VARCHAR, provided that the data type of the other value expression is character data, and BINARY if the data type of the other value expression is binary data. Similarly, HiRDB assumes that the data length of the embedded variable or ? parameter is equal to the data length of the other value expression.
3. If *value-expression-1* and *value-expression-2* are both character string data types, use the same character set for *value-expression-1* and *value-expression-2*. However, if either *value-expression-1* or *value-expression-2* is one of the value expressions listed below, it is converted to the character set of the corresponding value expression:
   - Character string literal
   - Embedded variable (default character set)
   - ? parameter
4. If *value-expression-1*and *value-expression-2* have different data lengths, HiRDB fills the shorter data with X'00' on the right and determines a bit-by-bit logical product after making the two value expressions equal in character string length.
5. The data type of the result is the BOOLEAN type.
6. The value of the result is not NOT NULL constrained (the null value is allowed). If either *value-expression-1*or *value-expression-2* is the null value, the result also is the null value.
7. Determines a bit-by-bit logical product of *value-expression-1* and *value-expression-2*, and the result is TRUE if any of the bits in the results of the logical product is 1; it is FALSE otherwise.
8. If both *value-expression-1*and *value-expression-2* are character strings of a length 0, the result is FALSE.

(d) Notes

The BIT_AND_TEST scalar function can be specified in the following locations:

- Value expressions in a logical predicate in a search condition

- RETURN statement in CREATE FUNCTION for which the data type of the return value is the BOOLEAN type

(e) Example

Performs a test to determine if C1 in a column of table T1 (data type: VARCHAR(2)) contains bits.

```
SELECT * FROM T1
    WHERE BIT_AND_TEST(C1,X'FFFF') IS TRUE
```

(3) CHARACTER

(a) Function

Converts date data, time data, or time stamp data into a character string representation.

(b) Format

CHAR[ACTER](*value-expression*)

(c) Rules

1. The following items can be specified as the value expression:
   - CURRENT_DATE
   - CURRENT_TIME
   - CURRENT_TIMESTAMP[(*p*)]
   - Columns of the date data type, time data type, or time stamp data type
   - Column specifications
   - Component specification
   - SQL variables or SQL parameters
   - Date operations that produce date data type results
   - Time operations that produce time data type results
   - Set functions
   - Scalar functions
   - CASE expressions
   - CAST specification
   - Function call
   - Scalar subquery
2. The data type of *value-expression* must be the date data type (DATE), time data type (TIME), or time stamp data type (TIMESTAMP).

3. The data types of the result are as follows:

   If *value-expression* is the date data type:

   `CHAR(10)`

   If *value-expression* is the time data type:

   `CHAR(8)`

   If *value-expression* is the time stamp data type:

   `CHAR(19)`, `CHAR(22)`, `CHAR(24)`, or `CHAR(26)`

4. The value of the result is a predefined character string representation of the data type of *value-expression*.
5. The value of the result is not `NOT NULL` constrained (null values are allowed). Therefore, if the source value expression is the null value, the result is also the null value.
6. The character set of the result is the default character set.

(d) Examples

- Update any dates older than 6 months in column `C1` of table `T1` (`CHAR` data type) to the date 02-06-1995:

```
UPDATE T1
   SET C1=CHAR(CURRENT_DATE)
   WHERE CHAR(CURRENT_DATE - 6 MONTHS) > C1
```

Table name: `T1`

Column `C1` (`CHAR(10)`)

1994-02-24
1994-05-17
1994-08-19
1994-11-21
1994-09-04

Execution result

1995-02-06
1995-02-06
1994-08-19
1994-11-21
1994-09-04

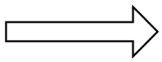
- Update the time 0:0:0 in column `C1` of table `T2` (`CHAR` data type) to the time 14:24:45:

```
UPDATE T2
   SET C1=CHAR(CURRENT_TIME)
   WHERE C1='00:00:00'
```

Table name: T2

Column C1 (CHAR(10))

C1
00:00:00
08:40:10
10:03:00
00:00:00
13:50:30



Execution result

14:24:45
08:40:10
10:03:00
14:24:45
13:50:30

(4) DATE

(a) Function

The DATE scalar function performs the following conversions:

1. Converts the character string representation of a date in a specified format into date data.
2. Converts the cumulative number of days since January 1, 1 (Gregorian calendar) into equivalent date data.

(b) Format

Format of function 1
DATE(*value-expression*[,*datetime-format*])

Format of function 2
DATE(*value-expression*)

(c) Rules for function 1

1. The following items can be specified as the value expression:
   - Literals that are character string expressions of dates
   - CURRENT_DATE
   - Column specifications
   - Component specification
   - SQL variables or SQL parameters
   - Date operations that produce date data type results
   - Concatenation operation
   - Set functions (MAX, MIN)

- Scalar functions
- CASE expressions
- CAST specification
- Function call
- Scalar subquery

2. The value expression must be one of the following data types:
   - A datetime format is specified:
     - Character string data type with a defined length of 8 to 255 bytes (CHAR, VARCHAR). However, if the character set is UTF-16, it is a character string data type with a defined length of 16 to 510 bytes.
     - Mixed character string data type (MCHAR, MVARCHAR)
   - A datetime format is not specified:
     - Character string data type with a defined length of 10 bytes (CHAR, VARCHAR). However, if the character set is UTF-16, it is a character string data type with a defined length of 20 bytes (CHAR, VARCHAR).
     - Date data type (DATE)
3. If the value expression is a character string data type (CHAR, VARCHAR), specifying the character set of the data type is optional.
4. The value expression must be the character string representation of a date in a format specified in a datetime format. If a datetime format is omitted, the value expression must be the predefined character string representation of a date.

   Examples:

   Datetime format '*YYYY*/*MM*/*DD*' → '1995/06/30'

   Datetime format omitted → '1995-06-30'
5. If the value expression is of the date data type, the result will be the equivalent date.
6. For datetime formats, see *1.11 Specifying a datetime format*.
7. When specifying a datetime format, use the same character set for the value expression and datetime format. However, if the datetime format is the value expression listed below, it is converted to the character set of the following value expression:
   - Character string literal

(d) Rules for function 2

1. The following items can be specified as the value expression:
   - Numeric literals
   - Column specifications
   - Arithmetic operations
   - Set functions
   - Scalar functions
   - CASE expressions
   - CAST specification
   - Function call
   - Scalar subquery
2. The data type of the value expression must be an integer (INTEGER).

   If an arithmetic operation, set function, or CASE expression is specified, the result of the operation must be the integer data type.
3. The allowable range of values is 1 to 3652059.
4. The result is the date (specified numeric value - 1) from January 1, 1 (Gregorian calendar).

   Example: If the value expression is 35, then the date is February 4, 1 (Gregorian calendar).

(e) Common rules

1. The data type of the result is the date data type (DATE).
2. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression or the datetime format is the null value, the result also is the null value.

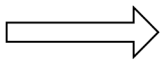
(f) Example

1. The following uses the DATE scalar function to perform the same processing as the example in the section on the CHARACTER scalar function:
```
UPDATE T1
   SET C1=CHAR(CURRENT DATE)
   WHERE CURRENT DATE - 6 MONTHS > DATE(C1)
```
2. Obtains date data from the character string, in a format other than a predefined character string representation ('*DD/MON/YYYY*') of a date in column C1 (data type: CHAR) in table T2:
```
SELECT DATE(C1,'DD/MON/YYYY') FROM T2
```

Table name: T2

Column C1 (CHAR(11))

01/JAN/2000
16/JUL/2002



Execution result

2000-01-01
2002-07-16

(5) DAY

(a) Function

Extracts the day part from date data, time stamp data, or date interval data.

(b) Format

DAY (*value-expression*)

(c) Rules

1. The following items can be specified as the value expression:
   - CURRENT_DATE
   - CURRENT_TIMESTAMP[(*p*)]
   - Column specifications
   - Component specification
   - SQL variables or SQL parameters
   - Date operations
   - Set functions (MAX, MIN)
   - Scalar functions
   - CASE expressions
   - CAST specification
   - Function call
   - Scalar subquery
2. The data type of the value expression must be either the date data type (DATE), time stamp data type (TIMESTAMP), or the date interval data type (INTERVAL YEAR TO DAY).
3. The data type of the result is integer (INTEGER).
4. If the value expression is of the date data type or time stamp data type, the results will be in the range 1 to 31.

5. If the value expression is of the date interval data type, the result will be in the range -99 to 99.

   If the result is non-zero, the result has the same sign as the value expression.

6. The value of the result is not NOT NULL constrained (null values are allowed). Therefore, if the source value expression is the null value, the result is also the null value.

(d) Example

Retrieve all rows with the current date from the rows in column C1 (date data type data type) of table T1:

```
SELECT * FROM T1
   WHERE DAY(C1)=DAY(CURRENT_DATE)
```

(6) DAYS

(a) Function

Converts date data or time stamp data into a cumulative number of days since January 1, year 1 (A.D.).

(b) Format

```
DAYS (value-expression)
```

(c) Rules

1. The following items can be specified as the value expression:
   - Predefined character string representation literals of a date
   - CURRENT_DATE
   - CURRENT_TIMESTAMP[(*p*)]
   - Column specifications
   - Component specification
   - SQL variables or SQL parameters
   - Date operations that produce date data type results
   - Concatenation operations producing a result that is the predefined character string representation literal of a date
   - Set functions (MAX, MIN)
   - Scalar functions
   - CASE expressions
   - CAST specification

- Function call
- Scalar subquery

2. The data type of the value expression must be the date data type (DATE) or time stamp data type (TIMESTAMP).
3. The data type of the result must be an integer (INTEGER).
4. The result of executing the DAYS scalar function on a specified date is the cumulative number of days, including the specified date, since January 1, 1 (Gregorian calendar).
5. The value of the result is not NOT NULL constrained (null values are allowed). Therefore, if the value expression is the null value, the result is also the null value.

(d) Example

Determine the number of days through the current date (06-30-1995) since the value in column C1 of table T1:

```
SELECT DAYS(CURRENT_DATE) - DAYS(C1)
   FROM T1
```

(7) DECIMAL

(a) Function

The DECIMAL scalar function converts numeric data into decimal data.

(b) Format

```
DEC[IMAL] (value-expression [, precision [, decimal-scaling-position]])
```

(c) Rules

1. The following items can be specified as the value expression:
   - Numeric literals
   - Column specifications
   - Component specification
   - SQL variables or SQL parameters
   - Arithmetic operations
   - Set functions
   - Scalar functions
   - CASE expressions
   - CAST specification
   - Function call

- Scalar subquery

2. The following items can be specified as the data type of *value-expression*:
   - Numeric data type
3. The precision must be an integer in the range `1` to `38`.

   If the precision is not specified, the assumed value varies depending on the data type of the specified value expression. The following table indicates the precision assumed when it is not specified.

   Table 2-30: Default precisions of the DECIMAL scalar function

Data Type	Precision (number of digit positions)
INTEGER	10
SMALLINT	5
DECIMAL	15
FLOAT SMALLFLT	When the pd_sql_dec_op_maxprec operand of the system common definition is 29 or omitted#: 29 When the pd_sql_dec_op_maxprec operand of the system common definition is 38#: 38

   #

   For details about the `pd_sql_dec_op_maxprec` operand of the system common definition, see the manual *HiRDB Version 9 System Definition*.

4. Scaling is specified in the range of values from 0 to the specified precision. The scaling must either be an integer or a character string representation of an integer. The default scaling is `0`.
5. The following table lists the data type of the result.

   Table 2-31: Data type of the result of the DECIMAL scalar function

Data Type	Precision of result	Decimal scaling position of the result
DECIMAL	Precision specified in the argument. If not specified, then the precision shown in *Table 2-30*.	Decimal scaling position specified in the argument. If not specified, then 0.

6. The integer part of the value expression must be expressed by a value that is within the specified precision and decimal scaling position. If the integer part exceeds the specified precision, an overflow error results.

7. Any digits following the specified decimal scaling position in the result are rounded off.
8. The value of the result is not NOT NULL constrained (null values are allowed). Therefore, if the value expression is the null value, the result is also the null value.

(d) Example

Ifthe pd_sql_dec_op_maxprec operand of the system common definition is 29, or if this operand was omitted, the result obtained by dividing column C1 (data type: DECIMAL(10,0)) by column C2 (data type: INTEGER) in table T1 is decimal data DEC(29,2), and the scalar function DECIMAL can be used to delete unneeded digits and convert the data to DEC(4,2).

```
SELECT DECIMAL(C1/C2,4,2)
   FROM T1
```

Table name: T1

Column C1 (DECIMAL(10,0))	Column C2 (INTEGER)
12	7
5	8
7	3

Execution result

1.71
0.62
2.33

(8) DIGITS

(a) Function

The DIGITS scalar function extracts the digits part of an integer, decimal number, date interval data, or time interval data and converts it into a character string expression.

(b) Format

DIGITS (*value-expression*)

(c) Rules

1. The following items can be specified as the value expression:
   - Integer or decimal literals
   - Column specifications
   - Component specification
   - SQL variables or SQL parameters
   - Arithmetic operations
   - Date operations that produce results that are of the date interval data type

- Time operations that produce results that are of the time interval data type
- Set functions
- Scalar functions
- CASE expressions
- CAST specification
- Function call
- Scalar subquery

2. The data type of the value expression must be one of the following:
   - Integer (INTEGER, SMALLINT)
   - Fixed-point number (DECIMAL)
   - Date interval data type (INTERVAL YEAR TO DAY)
   - Time interval data type (INTERVAL HOUR TO SECOND)
3. The data type of the result is a fixed-length character string (CHAR).
4. The length of the resulting data varies depending on the data type of the value expression. The following table lists the data lengths of the result.

   Table 2-32: Data lengths of the result of the DIGITS scalar function

Data type	Data length of result
INTEGER	10
SMALLINT	5
DECIMAL(*p*,*s*)	*p*
INTERVAL YEAR TO DAY	8
INTERVAL HOUR TO SECOND	6

   p: Precision.

   s: Decimal scaling position.

5. The result is a character string expression of the absolute value of the value expression, without a sign or decimal point. If the actual value has fewer digits than the data length of the destination field, the destination field is filled with leading zeros.

   Example: The data type of the values to be converted is DECIMAL(4,1):

```
15.   → '0150'
-12.4 → '0124'
```

6. The value of the result is not NOT NULL constrained (null values are allowed). Therefore, if the value expression is the null value, the result is also the null value.
7. The character set of the result is the default character set.

(d) Example

Retrieve data for which the value of column C1 (CHAR data type) and the value of column C2 (DECIMAL data type) in table T3 are equal:

```
SELECT * FROM T3
   WHERE C1 = DIGITS(C2)
```

Table name: T3

Column C1 (CHAR(3))	Column C2 (DECIMAL(3,0))
'△10'	10
'020'	20
'030'	-30

→ Column DIGITS(C2) (CHAR(3))

'010'
'020'
'030'

Execution result

'020'	20
'030'	-30

(9) FLOAT

(a) Function

The FLOAT scalar function converts numeric data into floating-point data.

(b) Format

FLOAT (*value-expression*)

(c) Rules

1. The following items can be specified as the value expression:
   - Numerical literals
   - Component specification
   - SQL variables or SQL parameters
   - Arithmetic operations

- Set functions
- Scalar functions
- CASE expressions
- CAST specification
- Function call
- Scalar subquery

2. The following items can be specified as the data type of *value-expression*:
   - Numeric data type
3. The data type of the result must be a double-precision floating-point number (FLOAT).
4. The value of the result is not NOT NULL constrained (null values are allowed). Therefore, if the value expression is the null value, the result is also the null value.

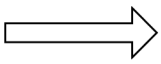
(d) Example

Obtain the result in floating-point data of dividing column C1 (INTEGER data type) by column C2 (INTEGER data type) in Table T1; use the FLOAT scalar function before the division operation to convert either operand to the FLOAT type:

```
SELECT FLOAT(C1)/C2 FROM T1
```

Table name: T1

Column C1 (INTEGER)	Column C2 (INTEGER)
1	3
8	100
-1	4



Execution result

Column C1/C2 (FLOAT)
3.333333333333333E-01
8.000000000000000E-02
-2.500000000000000E-01

(10) HEX

(a) Function

The HEX scalar function converts a value expression into a hexadecimal character string expression.

(b) Format

HEX （*value-expression*）

(c) Rules

1. The HEX scalar function converts the format of the value expression, represented internally in the system, into a hexadecimal character string expression.

The following table lists the formats of internal representations and provides examples of execution results.

Table 2-33: Formats of internal representations by the HEX scalar function and examples of execution results

Value expression	Internal representation format	HEX(value-expression)
'#AB12'	Data type: CHAR(5) 23|41|42|31|32	'2341423132'
1234	Data type: INTEGER In Windows: D2|04|00|00 In UNIX: 00|00|04|D2	In Windows: 'D2040000' In UNIX: '000004D2'
1234.	Data type: DECIMAL(4,0) 01| 23| 4C	'01234C'

2. The following items can be specified as the value expression:
   - Literals
   - USER
   - CURRENT_DATE
   - CURRENT_TIME
   - CURRENT_TIMESTAMP[(*p*)]
   - Column specifications
   - Component specification
   - SQL variables or SQL parameters
   - Arithmetic operations
   - Date operations
   - Time operations
   - Concatenation operations
   - Set functions
   - Scalar functions
   - CASE expressions
   - CAST specification
   - Function call
   - Scalar subquery

3. The following table indicates the relationship between the data types that can be specified in a value expression and the resulting data type and data length.

Table 2-34: Relationship between the data type of a value expression for the HEX scalar function and the data type and data length of a result

Value expression			Execution result		
Data type	**Defined length**	**Actual length**	**Data type**	**Defined length**	**Actual length**
CHAR(*n*)	1 ≤ *n* < 128	--	CHAR	*n**2	--
	128 ≤ *n* ≤ 16,000		VARCHAR		*n**2
NCHAR(*n*)	1 ≤ *n* < 64		CHAR	*n**4	--
	64 ≤ *n* ≤ 8,000		VARCHAR		*n**4
MCHAR(*n*)	1 ≤ *n* < 128		CHAR	*n**2	--
	128 ≤ *n* ≤ 16,000		VARCHAR		*n**2
VARCHAR(*n*)	1 ≤ *n* ≤ 16,000	[*r*]	VARCHAR	*n**2	*r**2
NVARCHAR(*n*)	1 ≤ *n* ≤ 8,000			*n**4	*r**4
MVARCHAR(*n*)	1 ≤ *n* ≤ 16,000			*n**2	*r**2

Value expression			Execution result		
Data type	**Defined length**	**Actual length**	**Data type**	**Defined length**	**Actual length**
INTEGER	--	--	CHAR	8	--
SMALLINT				4	
DECIMAL(*P*,*S*)	$1 \leq P \leq 38$ $0 \leq S \leq 38$ $S \leq P$			($\downarrow P/2 \downarrow + 1$) * 2	
FLOAT	--			16	
SMALLFLT				8	
DATE				8	
TIME				6	
TIMESTAMP(*p*)	*p* = 0, 2, 4, or 6			(7+*p*/2)*2	
INTERVAL YEAR TO DAY	--			10	
INTERVAL HOUR TO SECOND				8	
BINARY(*n*)	$1 \leq n \leq$ 16,000	[*r*]	VARCHAR	*n**2	*r**2

P: Precision

S: Decimal scaling position

p: Fractional second precision

--: Not applicable

Note: If only a character string literal (including national and mixed character string literals) whose length is 0 is specified in the value expression, the defined length of the execution result will be 1.

4. If the value expression is a character string data type (CHAR, VARCHAR), specifying the character set of the data type is optional.
5. The HEX scalar function cannot be specified for a value expression if execution of the value expression produces any of the following data types:
   - CHAR, VARCHAR, MCHAR, or MVARCHAR with a minimum length of 16,001 bytes
   - NCHAR or NVARCHAR with a minimum length of 8,001 characters

- BLOB
- BINARY with a minimum length of 16,001 bytes
- BOOLEAN

6. A value expression that contains embedded variables or ? parameters cannot be specified.
7. If an operand in the value expression or the argument of the function is composed solely of literals, the HEX scalar function cannot be specified if it produces a result with a length exceeding 255 bytes.
8. The value of the result is not NOT NULL constrained (null values are allowed). Therefore, if the value expression is the null value, the result is also the null value.
9. In the Windows edition, the result value (numeric data excluding the DECIMAL type) is represented internally in Little Endian. Specifically, the value 1234 of the INTEGER type is represented internally as D2 04 00 00 and the execution result is represented as D2040000.
10. In the UNIX edition, the result value depends on the internal expression of the server platform. For example, in the case of Linux running on an Intel family CPU, numeric data excluding the DECIMAL type is represented internally in Little Endian. Specifically, the value 1234 of the INTEGER type is represented internally as D2 04 00 00, and the execution result is represented as D2040000.
11. If the data type of the result is character string data, the character set of the result is the default character set.

(11) HOUR

(a) Function

Extracts the time part from time data, time stamp data, or time interval data.

(b) Format

HOUR (*value-expression*)

(c) Rules

1. The following items can be specified as the value expression:
   - CURRENT_TIME
   - CURRENT_TIMESTAMP[(*p*)]
   - Column specifications
   - Component specification
   - SQL variables or SQL parameters
   - Time operations

- Set functions (MAX, MIN)
- Scalar functions (ABS, TIME, VALUE)
- CASE expressions
- CAST specification
- Function call
- Scalar subquery

2. The data type of the value expression must be the time data type (TIME), time stamp data type (TIMESTAMP), or time interval data type (INTERVAL HOUR TO SECOND).
3. The data type of the result is integer (INTEGER).
4. If the value expression is of the time data type or the time stamp data type, the result is in the range 0 to 23.
5. If the value expression is of the time interval data type, the result will be in the range -99 to 99.

   If the result is non-zero, the result has the same sign as the value expression.
6. The value of the result is not NOT NULL constrained (null values are allowed). Therefore, if the value expression is the null value, the result is also the null value.

(d) Example

Retrieve all rows with the current time from the rows in column C1 (time data type) of table T1:

```
SELECT * FROM T1
   WHERE HOUR (C1) = HOUR(CURRENT TIME)
```

(12) INTEGER

(a) Function

The INTEGER scalar function converts numeric data into an integer.

(b) Format

INT[EGER] (*value-expression*)

(c) Rules

1. The following items can be specified as the value expression:
   - Numeric literals
   - Column specifications
   - Component specification

- SQL variables or SQL parameters
- Arithmetic operations
- Set functions
- Scalar functions
- CASE expressions
- CAST specification
- Function call
- Scalar subquery

2. The following items can be specified as the data type of *value-expression*:
   - Numeric data type
3. The data type of the result is an integer (INTEGER).
4. The result of executing the INTEGER scalar function must be a value that can be expressed in INTEGER.
5. Any numeric digits in the result that follow the decimal point are rounded off.
6. The value of the result is not NOT NULL constrained (null values are allowed). Therefore, if the value expression is the null value, the result is also the null value.

(d) Example

Obtain only the fractional part from column C1 (DECIMAL(4,3)) in table T1:

```
SELECT C1 - INTEGER(C1) FROM T1
```

Table name: T1

Column C1 (DECIMAL(4,3))

C1
1.125
8.849
3.

Execution result

0.125
0.849
0.

(13) LENGTH

(a) Function

The LENGTH scalar function determines the length of a value expression.

(b) Format

```
LENGTH ({value-expression
         |GET_JAVA_STORED_ROUTINE_SOURCE specification})
```

(c) Rules

1. The following items can be specified as the value expression:
   - Literals
   - `USER`
   - `CURRENT_DATE`
   - `CURRENT_TIME`
   - `CURRENT_TIMESTAMP[(`*p*`)]`
   - Column specifications
   - Component specification
   - SQL variables or SQL parameters
   - Arithmetic operations
   - Date operations
   - Time operation
   - Concatenation operations
   - Set functions
   - Scalar functions
   - `CASE` expressions
   - `CAST` specification
   - Function call
   - Scalar subquery
   - : *embedded-variable* `[`: *indicator-variable*`] AS` *data-type* (allowable data types: `BLOB` or `BINARY` types only)
   - `? AS` *data-type* (allowable data types: `BLOB` or `BINARY` types only)
2. The following data types cannot be specified in *value-expression*:
   - `BOOLEAN`
   - Abstract data type
3. When specifying an embedded variable or a `?` parameter in *value-expression*, its data type must be specified in the `AS` clause. An error may result if the actual length (for a locator, the actual length of the data allocated to the locator) of the data assigned to the embedded variable or `?` parameter is greater than the maximum length of the data type specified in the `AS` clause.

4. The data type of the result is integer (INTEGER).
5. The execution result varies depending on the data type of the value expression. The following table describes the execution result as a function of the data type of the value expression.

   Table 2-35: Execution result as a function of the data type of the value expression for the LENGTH scalar function

Data type of value expression	Execution result
Fixed-length character data	Defined length in bytes for the default character set. Otherwise, defined length in characters in that character set.
Variable-length character data	Actual length in bytes for the default character set. Otherwise, actual length in characters in that character set.
Fixed-length national character data	Definition length in characters
Variable-length national character data	Actual number of data characters
Fixed-length mixed character data	Actual number of data characters
Variable-length mixed character data	Actual number of data characters
Numeric data	Definition length in bytes For the DECIMAL data type, the length is (↓number of digits specified in precision ÷ 2↓ + 1). For details, see *1.2 Data types*.
Date, time, or time stamp data	Definition length in bytes
Date interval/time interval data	For details, see *1.2 Data types*.
Large-object data	Actual number of data bytes
Binary data	Actual number of data bytes

6. If the value expression is of the character string data type, a space is counted as one character.
7. If the value expression is a literal, it is processed according to the data type interpreted by HiRDB; see *1.4 Literals* for details.
8. If the value expression is USER, CURRENT_DATE, CURRENT_TIME, or CURRENT_TIMESTAMP[(*p*)], the value expression is processed according to the data type as interpreted by HiRDB. For details, see *1.5 USER, CURRENT_DATE value function, CURRENT_TIME value function, and CURRENT_TIMESTAMP value function*.
9. The value of the result is not NOT NULL constrained (null values are allowed).

Therefore, if the value expression is the null value, the result is also the null value.

(14) LOWER

(a) Function

The LOWER function converts the uppercase alphabetic characters in character data, national character data, or mixed character data into lowercase characters.

(b) Format

LOWER (*value-expression*)

(c) Rules

1. The following items can be specified as a value expression:
   - Literals
   - USER
   - Column specifications
   - Component specification
   - SQL variables or SQL parameters
   - Concatenation operations
   - Set functions
   - Scalar functions
   - CASE expressions
   - CAST specification
   - Function call
   - Scalar subquery
2. NULL, embedded variables, or the ? parameter cannot be specified in the value expression.
3. The data type of the value expression must be character string data type (CHAR or VARCHAR), national character string data type (NCHAR or NVARCHAR) or mixed character string data type (MCHAR or MVARCHAR).
4. If the value expression is a character string data type (CHAR, VARCHAR), specifying the character set of the data type is optional.
5. The execution result inherits the data type and the data length of the value expression.
6. The value of the result is not NOT NULL constrained (null values are allowed). Therefore, if the value expression is the null value, the result is also the null value.

7. The character set of the result is the character set of the value expression specified in the argument.

(15) MINUTE

(a) Function

Extracts the minute part from time data, time stamp data, or time interval data.

(b) Format

```
MINUTE (value-expression)
```

(c) Rules

1. The following items can be specified as the value expression:
   - `CURRENT_TIME`
   - `CURRENT_TIMESTAMP[(`*p*`)]`
   - Column specifications
   - Component specification
   - SQL variables or SQL parameters
   - Time operations
   - Set functions (`MAX`, `MIN`)
   - Scalar functions
   - `CASE` expressions
   - `CAST` specification
   - Function call
   - Scalar subquery
2. The data type of the value expression must be either the time data type (`TIME`) or the time interval data type (`INTERVAL HOUR TO SECOND`).
3. The data type of the result is integer (`INTEGER`).
4. If the value expression is the time data type or time stamp data type, the result is in the range 0 to 59.
5. If the value expression is of the time interval data type, the result will be in the range -59 to 59.

   If the result is non-zero, the result has the same sign as the value expression.
6. The value of the result is not `NOT NULL` constrained (null values are allowed). Therefore, if the value expression is the null value, the result is also the null value.

(d) Example

Retrieve the data for which the difference between column C2 (time data type) and column C3 (time data type) in table T1 is less than or equal to 30 minutes:

```
SELECT C1 FROM T1
   WHERE MINUTE(C3-C2) <= 30
```

Table name: T1

Column C1	Column C2 (time data type)	Column C3 (time data type)
A01	08:00:00	08:45:00
A02	09:00:00	09:28:00
A03	10:00:00	10:30:00

Execution result

Column C1
A02
A03

(16) MOD

(a) Function

The MOD scalar function returns the remainder from a division operation.

(b) Format

MOD (*value-expression-1*, *value-expression-2*)

(c) Rules

1. The following items can be specified as *value-expression-1* and *value-expression-2*:
   - Integer literals or decimal number literals
   - Column specifications
   - Component specification
   - SQL variables or SQL parameters
   - Arithmetic operations
   - Set functions
   - Scalar functions
   - CASE expressions
   - CAST specification
   - Function call
   - Scalar subquery
2. The numerator is specified as *value-expression-1*, and the denominator is

specified as *value-expression-2*.

3. The data types of *value-expression-1* and *value-expression-2* must be one of the following:
   - Integer (`INTEGER`, `SMALLINT`)
   - Fixed-point number (`DECIMAL`)
4. The value of the result is not `NOT NULL` constrained (the null value is allowed). If *value-expression-1* or *value-expression-2* is the null value, the result will also be the null value.
5. If *value-expression-1* or *value-expression-2* contains a decimal part, the result value will also contain a decimal part.
6. The sign of the result will be the same as the sign of *value-expression-1*.
7. If *value-expression-2* is 0, an error results (for the result when overflow error suppression is set, see *2.18 Operational results with overflow error suppression specified*).
8. If the following inequality holds, overflow will occur during the computation, resulting in an overflow error:

   $(p_1 - s_1) + s_2 > \text{max\_prec}$

   p_1: Effective precision of the value of *value-expression-1*

   s_1: Effective decimal scaling position of the value of *value-expression-1*

   p_2: Effective precision of the value of *value-expression-2*

   s_2: Effective decimal scaling position of the value of *value-expression-2*

   max_prec: The following table gives the maximum value `max_prec` of the precision.

Table 2-36: Maximum value max_prec of the precision

System common definition pd_sql_dec_op_maxprec operand#	p_1 and p_2	max_prec value
`29` or omitted	$p_1 \leq 29$ and $p_2 \leq 29$	29
	$p_1 > 29$ or $p_2 > 29$	38
38	Any	38

#

For details about the `pd_sql_dec_op_maxprec` operand of the system

common definition, see the manual *HiRDB Version 9 System Definition*.

For the result when overflow error suppression is set, see *2.18 Operational results with overflow error suppression specified*.

9. The following table indicates the relationship between a result data type and *value-expression-1* and *value-expression-2* data types.

Table 2-37: Relationship between a result data type and value-expression-1 and value-expression-2 data types

value-expression-1 Data Type	Value-expression-2 data type		
	SMALLINT	INTEGER	DECIMAL
SMALLINT	SMALLINT	INTEGER	DECIMAL
INTEGER	SMALLINT	INTEGER	DECIMAL
DECIMAL (p, 0)	SMALLINT	INTEGER	DECIMAL
DECIMAL (p, s) ($s > 0$)	DECIMAL	DECIMAL	DECIMAL

10. The following table gives the resulting precision and decimal scaling position when the data type of a result is DECIMAL.

Table 2-38: Precision and decimal scaling position of a result when the result data type is DECIMAL

Item	Precision or decimal scaling position
Precision (p)	p=MIN (p_2-s_2+s, max_prec)
Decimal scaling position (s)	s=MAX (s_1, s_2)

Note 1

value-expression-1 data type: DECIMAL(p_1,s_1)

value-expression-2 data type: DECIMAL(p_2,s_2)

Note 2

INTEGER is treated as DECIMAL(10,0).

SMALLINT is treated as DECIMAL(5,0).

Note 3

max_prec is the maximum value of the precision as shown in *Table 2-36 Maximum value max_prec of the precision*.

(17) MONTH

(a) Function

Extracts the month part from date data, time stamp data, or date interval data.

(b) Format

```
MONTH (value-expression)
```

(c) Rules

1. The following items can be specified as the value expression:
   - CURRENT_DATE
   - CURRENT_TIMESTAMP[(*p*)]
   - Column specifications
   - Component specification
   - SQL variables or SQL parameters
   - Date operations
   - Set functions (MAX, MIN)
   - Scalar functions
   - CASE expressions
   - CAST specification
   - Function call
   - Scalar subquery
2. The data type of the value expression must be the date data type (DATE), time stamp data type (TIMESTAMP), or date interval data type (INTERVAL YEAR TO DAY).
3. The data type of the result is integer (INTEGER).
4. If the value expression is the date data type or time stamp data type, the result is in the range 1 to 12.
5. If the value expression is of the date interval data type, the result will be in the range -11 to 11.

   If the result is non-zero, the result has the same sign as the value expression.
6. The value of the result is not NOT NULL constrained (null values are allowed). Therefore, if the value expression is the null value, the result is also the null value.

(d) Example

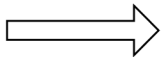
Delete all rows from table T1 that are not of the current month (September):

```
DELETE FROM T1
   WHERE MONTH(C1) <> MONTH(CURRENT_DATE)
```

Table name: T1

Column C1 (date data type)

1995-05-23
1995-09-27
1995-04-14
1995-02-03



Execution result

1995-09-27

(18) POSITION

(a) Function

Determines the starting position of the first part in a data string (a character string or binary string) that matches a given data substring.

(b) Format

```
POSITION (value-expression-1 IN value-expression-2 [ FROM value-expression-3])
```

(c) Rules

1. In *value-expression-1*, specify the search data substring. In *value-expression-2*, specify the data string to be searched for. Items that can be specified in *value-expression-1* and *value-expression-2* are listed below. Items that can be specified vary depending upon combinations of the data types of *value-expression-1* and *value-expression-2*. For specifiable combinations, see *Rule 2*.
   - Literals (character strings, national character strings, mixed character strings, or hexadecimal character strings)
   - Column specifications
   - Component specification
   - SQL variables or SQL parameters
   - Concatenation operations
   - Set functions

- Scalar functions
- CASE expressions
- CAST specification
- Function call
- Scalar subquery
- : *embedded-variable* [: *indicator-variable*] AS *data-type* (allowable data types: BLOB or BINARY types only)
- ? AS *data-type* (allowable data types: BLOB or BINARY types only)

2. The following table indicates the combinations of data types that can be specified in *value-expression-1* and *value-expression-2*:

Table 2-39: Combinations of data types that can be specified in value expression 1 and value expression 2 of the scalar function POSITION

Value expression 1	Value expression 2					
	Character string data type	**National character string data type**	**Mixed character string data type**	**BLOB type**	**BINARY type with a maximum length of 32,000 bytes**	**BINARY type with a maximum length of 32,001 bytes or greater**
Character string data type	Y[#3] (*Table 2-40*)	N	Y[#4] (*Table 2-40*)	R[#1] (*Table 2-41*)	R[#1] (*Table 2-40*)	R[#1] (*Table 2-41*)
National character string data type	N	Y (*Table 2-40*)	N	N	N	N
Mixed character string data type	Y[#4] (*Table 2-40*)	N	Y (*Table 2-40*)	N	N	N
BLOB type	R[#2] (*Table 2-42*)	N	N	Y (*Table 2-41*)	Y (*Table 2-42*)	Y (*Table 2-41*)
BINARY type with a maximum length of 32,000 bytes	R[#2] (*Table 2-40*)	N	N	Y (*Table 2-41*)	Y (*Table 2-40*)	Y (*Table 2-41*)
BINARY type with a maximum length of 32,001 bytes or greater	R[#2] (*Table 2-42*)	N	N	Y (*Table 2-41*)	Y (*Table 2-42*)	Y (*Table 2-41*)

Legend:

Y: Specifiable

R: Specifiable, subject to restrictions

N: Not specifiable

Table number: Table of combinations of corresponding item if item is specifiable

#1

Can be specified only if *value expression-1* is a hexadecimal character string literal.

#2

Can be specified only if *value expression-2* is a hexadecimal character string literal.

#3

If *value-expression-1* and *value-expression-2* are both character string data types, use the same character set for *value-expression-1* and *value-expression-2*. However, if *value-expression-1* is the value expression listed below, it is converted to the character set of *value-expression-2*:

- Character string literal

#4

Can only be specified if the character set of the value expression of a character string data type is the default character set.

3. The following table indicates the combinations of items that can be specified in *value expression-1* and *value expression-2*.

Table 2-40: Combinations of items that can be specified in value expression 1 and value expression 2 of the scalar function POSITION (where value expression 1 and value expression 2 are both character string data types, national character string data type, mixed character string data type, or BINARY type with a maximum length of 32,000 bytes)

Value expression 1	Value expression 2											
	Lit	Col spc	Cmp spc	SQL	Con	Set fnc	Scl fnc	CSE	CST	Fnc call	Scl sq	Emb var, ? para #
Literal	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Value expression 1	Value expression 2											
	Lit	**Col spc**	**Cmp spc**	**SQL**	**Con**	**Set fnc**	**Scl fnc**	**CSE**	**CST**	**Fnc call**	**Scl sq**	**Emb var, ? para #**
Column specification	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Component specification	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
SQL variable, SQL parameter	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Concatenation	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Set function	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Scalar function	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
`CASE` expression	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
`CAST` specification	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Function call	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Embedded variable, `?` parameter#	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Legend:

Y: Specifiable

Lit: Literal

Col spc: Column specification

Cmp spc: Component specification

SQL: SQL variable, SQL parameter

Con: Concatenation

Set fnc: Set function

Scl fnc: Scalar function

CSE: `CASE` expression

CST: CAST specification

Fnc call: Function call

Scl sq: Scalar subquery

Emb var: Embedded variable

? para: ? parameter

#: Embedded variables and ? parameters can be specified only if they are of the BINARY type.

Table 2-41: Combinations of items that can be specified in value expression 1 and value expression 2 of the scalar function POSITION (where value expression 2 is either the BLOB type or the BINARY type with a maximum length of 32,0001 bytes or greater)

Value expression 1	Value expression 2											
	Lit	Col spc	Cmp spc	SQL	Con	Set fnc	Scl fnc	CSE	CST	Fnc call	Scl sq	Emb var, ? para
Literal#	N	Y	N	Y	N	N	N	N	N	N	N	Y
Column specification	N	N	N	N	N	N	N	N	N	N	N	N
Component specification	N	N	N	N	N	N	N	N	N	N	N	N
SQL variable, SQL parameter	N	Y	N	Y	N	N	N	N	N	N	N	Y
Concatenation	N	N	N	N	N	N	N	N	N	N	N	N
Set function	N	N	N	N	N	N	N	N	N	N	N	N
Scalar function	N	N	N	N	N	N	N	N	N	N	N	N
CASE expression	N	N	N	N	N	N	N	N	N	N	N	N
CAST specification	N	N	N	N	N	N	N	N	N	N	N	N
Function call	N	N	N	N	N	N	N	N	N	N	N	N
Embedded variable, ? parameter	N	Y	N	Y	N	N	N	N	N	N	N	Y

Legend:

Y: Specifiable

N: Not specifiable

Lit: Literal

Col spc: Column specification

Cmp spc: Component specification

SQL: SQL variable, SQL parameter

Con: Concatenation

Set fnc: Set function

Scl fnc: Scalar function

CSE: `CASE` expression

CST: `CAST` specification

Fnc call: Function call

Scl sq: Scalar subquery

Emb var: Embedded variable

? para: ? parameter

#: Literals can be specified only if they are of the character string data type (hexadecimal character string literal).

Table 2-42: Combinations of items that can be specified in value expression 1 and value expression 2 of the scalar function POSITION (where value expression 1 is either the BLOB type or the BINARY type with a maximum length of 32,001 bytes or greater, and value expression 2 is either the character string data type or the BINARY type with a maximum length of 32,000 bytes)

Value expression 1	Value expression 2											
	Lit	Col spc	Cmp spc	SQL	Con	Set fnc	Scl fnc	CSE	CST	Fnc call	Scl sq	Emb var, ? para #
Literal	N	N	N	N	N	N	N	N	N	N	N	N
Column specification	N	N	N	N	N	N	N	N	N	N	N	N

Value expression 1	Value expression 2											
	Lit	**Col spc**	**Cmp spc**	**SQL**	**Con**	**Set fnc**	**Scl fnc**	**CSE**	**CST**	**Fnc call**	**Scl sq**	**Emb var, ? para #**
Component specification	N	N	N	N	N	N	N	N	N	N	N	N
SQL variable, SQL parameter	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Concatenation	N	N	N	N	N	N	N	N	N	N	N	N
Set function	N	N	N	N	N	N	N	N	N	N	N	N
Scalar function	N	N	N	N	N	N	N	N	N	N	N	N
`CASE` expression	N	N	N	N	N	N	N	N	N	N	N	N
`CAST` specification	N	N	N	N	N	N	N	N	N	N	N	N
Function call	N	N	N	N	N	N	N	N	N	N	N	N
Embedded variable, `?` parameter	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Legend:

Y: Specifiable

N: Not specifiable

Lit: Literal

Col spc: Column specification

Cmp spc: Component specification

SQL: SQL variable, SQL parameter

Con: Concatenation

Set fnc: Set function

Scl fnc: Scalar function

CSE: `CASE` expression

CST: `CAST` specification

Fnc call: Function call

Scl sq: Scalar subquery

Emb var: Embedded variable

? para: ? parameter

#: Embedded variables and `?` parameters can be specified only if they are of the `BINARY` type.

4. When specifying an embedded variable or a `?` parameter in *value-expression-1* or *value-expression-2*, specify its data type in the `AS` clause. An error may result if the actual length (for a locator, the actual length of the data allocated to the locator) of the data assigned to the embedded variable or `?` parameter is greater than the maximum length of the data type specified in the `AS` clause.
5. *value-expression-3* specifies the search start position. The following shows the relationship between the types and range of values for *value-expression-2* and *value-expression-3*.

   - If *value-expression-2* is the `BLOB` type, `BINARY` type, or a character string data type that uses the default character set

     Specify *value-expression-3* in bytes with a value in the following range:

     - If *value-expression-2* is a character string data type that uses the default character set, *value-expression-3* must be greater than or equal to 1 and less than or equal to the maximum length of *value-expression-2*.
     - If *value-expression-2* is the `BLOB` type or `BINARY` type, *value-expression-3* must be greater than or equal to `1`.

   - If *value-expression-2* is a national character data type or mixed character string data type

     Specify *value-expression-3* in characters. Specify the value of *value-expression-3* so that it is greater than or equal to 1 and less than or equal to the maximum length of *value-expression-2*. For a mixed character string data type, the maximum length of *value-expression-2* is in bytes.

   - If *value-expression-2* is a character string data type that uses a character set other than the default character set

     Specify *value-expression-3* in characters. Specify the value of *value-expression-3* so that it is greater than or equal to 1 and less than or equal to the maximum length of *value-expression-2* divided by *c*. The value of *c* is the minimum size, in bytes, of characters in the character set being used. This is 1 for EBCDIK and 2 for UTF-16. The maximum length of *value-expression-2* is in bytes.

   If *value-expression-3* is omitted, the search start position is assumed to be `1`.

6. The following items can be specified in *value-expression-3*:
   - Unsigned integer literal
   - Column specification
   - Component specification
   - Arithmetic operations
   - Set function
   - Scalar function
   - `CASE` expression
   - `CAST` specification
   - Function call
   - Scalar subquery
   - SQL variable or SQL parameter
   - Embedded variable or `?` parameter

   The following items can be specified in *value-expression-3* if *value-expression-1* or *value-expression-2* is a `BLOB` type or a `BINARY` type with a maximum length of 32,0001 bytes or greater:
   - Unsigned integer literal
   - SQL variable or SQL parameter
   - Embedded variable or `?` parameter
7. The data type of *value-expression-3* must be integer (`INTEGER`, `SMALLINT`).
8. The data type of a result will be integer (`INTEGER`).
9. If *value-expression-2* is the `BLOB` type, `BINARY` type, or a character string data type that uses the default character set, the result is in bytes. If *value-expression-2* is a national character string data type, mixed character string data type, or character string data type that uses a character set other than the default character set, the result is in characters.
10. If the actual length of *value-expression-1* is 0, the result will be the value of *value-expression-3*. If *value-expression-3* is omitted, the actual length will be 1.
11. If the actual length of *value-expression-1* is greater than 0 and *value-expression-3* is greater than the actual length of *value-expression-2*, the result will be 0.
12. If the data substring of *value-expression-1* is not found beyond the starting search position in the data string specified in *value-expression-2*, the result will be 0.
13. The value of the result will be the `NOT NULL` constraint (allows the null value). If

value-expression-1, *value-expression-2*, or *value-expression-3* is the null value, the result will also be the null value.

(d) Example

Find the first position where the character string `'TIME'` occurs at byte 6 or beyond in column `C1` (data type: `CHAR`) of table `T1`:

```
SELECT POSITION('TIME:' IN C1 FROM 6) FROM T1
```

Table name: `T1`

Column `C1` (`VARCHAR(50)`)

`TIME:10:15,TIME:12:00`	(Actual length: 21)	12
`TIME:13:00`	(Actual length: 10)	0
*	(Null value)	*

Execution result

Legend:

*: Null value

(19) SECOND

(a) Function

Extracts the second part from time data, time stamp data, or time interval data.

(b) Format

`SECOND` (*value-expression*)

(c) Rules

1. The following items can be specified as the value expression:
   - `CURRENT_TIME`
   - `CURRENT_TIMESTAMP[(`*p*`)]`
   - Column specifications
   - Component specification
   - SQL variables or SQL parameters
   - Time operations
   - Set functions (`MAX`, `MIN`)
   - Scalar functions

- CASE expressions
- CAST specification
- Function call
- Scalar subquery

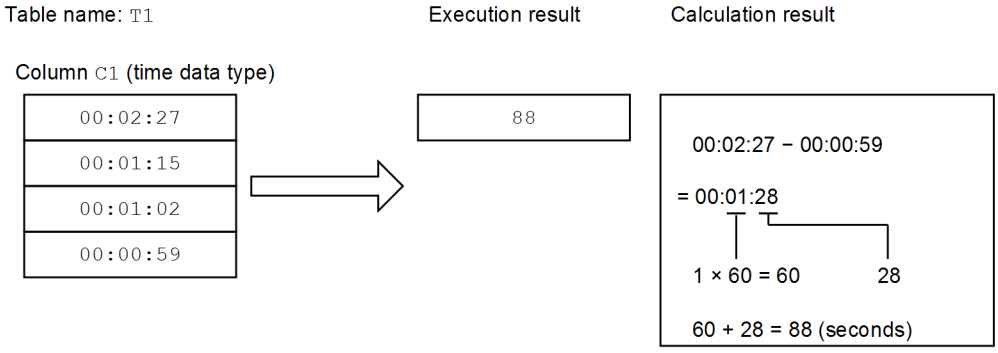
2. The data type of the value expression must be the time data type (TIME), time stamp data type (TIMESTAMP), or time interval data type (INTERVAL HOUR TO SECOND).
3. The data type of the result is integer (INTEGER).
4. If the value expression is the time data type or time stamp data type, the result is in the range 0 to 59. If the value expression is time data or time stamp data that includes a leap second, the result is in the range of 0 to 61. For details about how to specify time data or time stamp data that includes a leap second, see the description of the pd_leap_second operand in the manual *HiRDB Version 9 System Definition*.
5. If the value expression is of the time interval data type, the result will be in the range -59 to 59.

   If the result is non-zero, the result has the same sign as the value expression.
6. The value of the result is not NOT NULL constrained (null values are allowed). Therefore, if the value expression is the null value, the result is also the null value.

(d) Example

Determine in units of seconds the difference between the earliest time and the latest time in column C1 (time data type) in table T1:

```
SELECT MINUTE(MAX(C1)-MIN(C1))
             *60+SECOND(MAX(C1)
             -MIN(C1))
       FROM T1
```



(20) SUBSTR

(a) Function

Extracts a part of character string data, national character string data, mixed character string data, or binary data.

(b) Format

```
SUBSTR (value-expression-1, value-expression-2 [, value-expression-3])
```

(c) Rules

1. In *value-expression-1*, specify the data string (a character string or a binary string) to be processed. The following items can be specified in *value-expression-1*, subject to variation depending upon the data type of *value-expression-1*:

   - Literals (character strings, national character strings, or mixed character strings)
   - Column specification
   - Component specification
   - SQL variables or `?` parameters
   - Concatenation operation
   - Set functions
   - Scalar functions
   - `CASE` expression
   - `CAST` specification
   - Function calls
   - Scalar subquery
   - `:` *embedded-variable* [`:` *indicator-variable*] `AS` *data-type* (allowable data types: `BLOB` or `BINARY` types only)
   - `?` `AS` *data-type* (allowable data types: `BLOB` or `BINARY` types only)

   The following table lists the data types that can be specified.

Table 2-43: Items that can be specified depending on the data type of value expression 1 of the scalar function SUBSTR

Item	Data type of value expression 1		
	Character string data type, national character string data type, mixed character string data type	**BINARY type with a maximum length of 32,000 bytes**	**BLOB type, or BINARY type with a maximum length of 32,001 bytes or greater**
Literal	Y	N	N
Column specification	Y	Y	Y
Component specified	Y	Y	N
SQL variable, SQL parameter	Y	Y	Y
Concatenation Set function Scalar function `CASE` expression `CAST` specification	Y	Y	N
Function call	Y	Y	Y
Scalar subquery	Y	Y	N
Embedded variable `?` parameter	N	Y	Y

Legend:

Y: Specifiable

N: Not specifiable

1. Specify one of the following for the data type of *value-expression-1*:
   - Character string data type using any character set (`CHAR`, `VARCHAR`)
   - National character data type (`NCHAR`, `NVARCHAR`)
   - Mixed character string data type (`MCHAR`, `MVARCHAR`)
   - `BLOB` type or `BINARY` type
2. When specifying an embedded variable or a `?` parameter in *value-expression-1*, specify its data type in the `AS` clause. An error may result if the actual length (for a locator, the actual length of the data allocated to the locator) of the data assigned

to the embedded variable or ? parameter is greater than the maximum length of the data type specified in the AS clause.

3. In *value-expression-2*, specify the starting position of the partial data string to be extracted as a positive integer. The following shows the relationship between the type and range of values for *value-expression-1* and *value-expression-2*.

   - If *value-expression-1* is the BLOB type, BINARY type, or a character string data type that uses the default character set

     Specify *value-expression-2* in bytes with a value in the following range:

     • If f *value-expression-1* is a character string data type that uses the default character set, *value-expression-2* must be greater than or equal to 1 and less than or equal to the maximum length of *value-expression-1*.

     • If *value-expression-1* is a BLOB type or BINARY type, *value-expression-2* must be greater than or equal to 1.

   - If *value-expression-1* is a national character data type or mixed character string data type

     Specify *value-expression-2* in characters. Specify the value of *value-expression-2* so that it is greater than or equal to 1 and less than or equal to the maximum length of *value-expression-1*. For a mixed character string data type, the maximum length of *value-expression-1* is in bytes.

   - If *value-expression-1* is a character string data type that uses a character set other than the default character set

     Specify *value-expression-2* in characters. Specify the value of *value-expression-2* so that it is greater than or equal to 1 and less than or equal to the maximum length of *value-expression-1* divided by *c*. The value of *c* is the minimum size, in bytes, of characters in the character set being used. This is 1 for EBCDIK and 2 for UTF-16. The maximum length of *value-expression-1* is in bytes.

4. Specify the length of the data to be removed in *value-expression-3* as a positive integer. The following shows the relationship between the type and range of values for *value-expression-1* and *value-expression-3*.

   - If *value-expression-1* is the BLOB type, BINARY type, or a character string data type that uses the default character set

     Specify *value-expression-3* in bytes with a value in the following range:

     • If the data type of *value-expression-1* is a character string data type that uses the default character set, then 0 ≤ (*value-expression-3*) ≤ (maximum length of *value-expression-1*) - (*value-expression-2*) + 1.

     • If the data type of *value-expression-1* is the BLOB type or BINARY type,

value-expression-3 must be greater than or equal to 0.

- If *value-expression-1* is a national character data type or mixed character string data type

  Specify *value-expression-3* in characters with a value such that 0 ≤ (*value-expression-3*) ≤ (maximum length of *value-expression-1*) - (*value-expression-2*) + 1. For a mixed character string data type, the maximum length of *value-expression-1* is in bytes.

- If *value-expression-1* is a character string data type that uses a character set other than the default character set

  Specify *value-expression-3* in characters with a value such that 0 ≤ (*value-expression-3*) ≤ (maximum length of *value-expression-1*) ÷ *c* - (*value-expression-2*) + 1. The value of *c* is the minimum size, in bytes, of characters in the character set being used. This is 1 for EBCDIK and 2 for UTF-16. The maximum length of *value-expression-1* is in bytes.

A variable of `0` cannot be specified in *value-expression-3*.

5. The following items can be specified as *value-expression-2* and *value-expression-3*:

   - Unsigned integer literals
   - Column specifications
   - Component specification
   - Arithmetic operations
   - Set functions
   - Scalar functions
   - `CASE` expressions
   - `CAST` specification
   - Function call
   - SQL variables or SQL parameters
   - Embedded variables or `?` parameters
   - Scalar subquery

   If *value-expression-1* is of the `BLOB` type of the `BINARY` type with a minimum length of 32,001 bytes, the following items can be specified in *value-expression-2* and *value-expression-3*:

   - Unsigned integer literals
   - SQL variables or SQL parameters

- Embedded variables or ? parameters

6. The data type of *value-expression-2* and *value-expression-3* must be an integer (INTEGER or SMALLINT).
7. If *value-expression-2* is greater than the real length of *value-expression-1*, the result is the null value (the length of the result: 0).
8. If the data length of a result is 0, the result is the null value.
9. If *value-expression-3* is omitted and *value-expression-1* is fixed-length data, HiRDB extracts characters from the starting position indicated by *value-expression-2* through the last character indicated by the defined length. If *value-expression-1* is variable-length data, HiRDB extracts characters from the starting position indicated by *value-expression-2* through the last character indicated by the real length.
10. If *value-expression-1* is of the BLOB type and *value-expression-3* is not omitted, and if the data string in the specified range includes a part that does not contain real data, HiRDB only extracts the part that contains real data.
11. The value of the result is not NOT NULL constrained (the null value is allowed).
12. If *value-expression-1*, *value-expression-2*, or *value-expression-3* is a null value, the result is also a null value.
13. The data types and lengths of results are shown in *Table 2-44* to *Table 2-47*.

Table 2-44: Data types and lengths of results of the SUBSTR scalar function (when the data type of value-expression-1 is a character string type that uses the default character set, national character string type, mixed character string type, BLOB type, or BINARY type and value-expression-3 is specified)

Data type of character string (value-expression-1) from which partial string is extracted	Actual length	Length L (value-expression-3)		
		Literal (character string)		Non-literal
		$L_1 \leq 255$	$L_1 \geq 256$	
CHAR(*n*)	--	CHAR(*L*)	VARCHAR(*L*) [*L*]	VARCHAR[*L*]
VARCHAR(*n*)	[*r*]			
NCHAR(*n*)	--	NCHAR(*L*)	NVARCHAR(*L*) [*L*]	NVARCHAR[*L*]
NVARCHAR(*n*)	[*r*]			
MCHAR(*n*)	--	MCHAR(L_1)	MVARCHAR(L_1) [L_2]	MVARCHAR[L_2]
MVARCHAR(*n*)	[*r*]			
BLOB(*n*)	[*r*]	BLOB(*L*) [k_5]	BLOB(*L*) [k_5]	BLOB(*n*) [k_5]

Data type of character string (value-expression-1) from which partial string is extracted	Actual length	Length L (value-expression-3)		
		Literal (character string)		Non-literal
		$L_1 \leq 255$	$L_1 \geq 256$	
BINARY(n)	[r]	BINARY(L) [k_5]	BINARY(L) [k_5]	BINARY(n) [k_5]

[ ]

Value enclosed in square brackets is the actual length.

L_1

If data type is character string: Length L of *value-expression-3* (in bytes).

If data type is national character string: Length L of *value-expression-3* (in characters) x 2

If data type is mixed character string: MIN(length L of *value-expression-3* (in characters) x c, n).

L_2

Length (in bytes) of the extracted partial character string containing L characters ($L \leq L_2 \leq L_1$).

n

Definition length of the data string (*value-expression-1*) to be processed

For the character string data type, the mixed character string data type, or the BLOB type, in bytes

For the national character string data type, in units of characters

k_5

$\min(L, r - S + 1)$

c

Maximum number of bytes representing each character

The following table indicates the maximum number of bytes per character code.

Character code type specifying pdntenv or pdsetup	Maximum number of bytes
sjis	2
ujis	2

Character code type specifying pdntenv or pdsetup	Maximum number of bytes
chinese	2
chinese-gb18030	4
lang-c	2
utf-8#	3 to 6

#

When `utf-8` is specified for the character code type in the `pdntenv` command (`pdsetup` command in the case of UNIX), the following specifications are used:

- `pd_substr_length` in the system common definition
- `PDSUBSTRLEN` in the client environment definition
- `SUBSTR LENGTH` in the SQL compile option

--: Not applicable.

Note: If *value-expression-1* is a variable-length character string, the area where the base data is not included in the substring being removed is filled with spaces of that character set.

Example:

Executing `SUBSTR(`*character-string-1*`, 3, 5)` on character string 1 of `VARCHAR(8)[5]` causes spaces to be set in the two right-side characters in the character string being extracted.

If *value-expression-1* is of the `BLOB` type or the `BINARY` type and *value-expression-3* is not omitted, and if the binary string in the specified range includes a part that does not contain real data, HiRDB extracts only the part that contains real data, without setting spaces.

Example:

Executing `SUBSTR(`*binary-data-1*`, 101, 600)` on binary data 1 of `BLOB(1024)[512]` produces a result that is from bytes 101 to 512 of binary data 1.

Table 2-45: Data types and lengths of results of the SUBSTR scalar function (when the data type of value-expression-1 is a character string type that uses the

default character set, national character string type, mixed character string type, BLOB type, or BINARY type and value-expression-3 is omitted

Data type of character string (value-expression-1) from which partial string is extracted	Actual length	Begin position S (value-expression-2)			
		Literal (character string)			Non-literal
		value-expression-1: Fixed length		value-expression-1: Variable length	
		$k_0 \leq 255$	$k_0 \geq 256$		
CHAR(*n*)	--	CHAR(k_1)	VARCHAR(*n*) [k_1]	--	VARCHAR(*n*) [k_1]
VARCHAR(*n*)	[*r*]	--	--	VARCHAR(*n*) [k_2]	VARCHAR(*n*) [k_2]
NCHAR(*n*)	--	NCHAR(k_1)	NVARCHAR(*n*) [k_1]	--	NVARCHAR(*n*) [k_1]
NVARCHAR(*n*)	[*r*]	--	--	NVARCHAR(*n*) [k_2]	NVARCHAR(*n*) [k_2]
MCHAR(*n*)	--	MCHAR(k_1)	MVARCHAR(*n*) [k_3]	--	MVARCHAR(*n*) [k_3]
MVARCHAR(*n*)	[*r*]	--	--	MVARCHAR(*n*) [k_4]	MVARCHAR(*n*) [k_4]
BLOB(*n*)	[*r*]	--	--	BLOB(*n*) [k_2]	BLOB(*n*) [k_2]
BINARY(*n*)	[*r*]	--	--	BINARY(*n*) [k_2]	BINARY(*n*) [k_2]

[ ]

Value enclosed in square brackets is the actual length.

n

- If data type is character string: value in bytes
- If data type is mixed character string: value in bytes
- If data type is national character string: value in characters

k_0

If data type is character string or mixed character string: $n - S + 1$

If data type is mixed character string: $(n - S + 1) \times 2$

k_1

$n - S + 1$

k_2

$\max(r - S + 1, 0)$

k_3

Number of bytes in the partial character string from the S^{th} character through the n^{th} byte

$\max (n - (S - 1) \times c, 0) \leq k_3 \leq n - S + 1$

l_4

Number of bytes in the partial character string from the S^{th} character through the r^{th} byte

$\max (r - (S - 1) \times c, 0)) \leq k_4 \leq \max (r - S + 1, 0))$

c

Maximum number of bytes representing each character

For details, see *c* in *Legend* of *Table 2-44 Data types and lengths of the SUBSTR scalar function (when the data type of value-expression-1 is a character string type that uses the default character set, national character string type, mixed character string type, BLOB type, or BINARY type and value-expression-3 is specified)*.

--: Not applicable.

Note: When *value-expression-1* is a variable-length character string, *value-expression-3* is omitted, and l_2 or l_4 is 0, the result will be the null value.

Table 2-46: Data types and lengths of the SUBSTR scalar function (when the data type of value-expression-1 is a character string type that uses a character set other than the default character set and value-expression-3 is specified)

Data type of character string (value-expression -1) from which the substring is to be extracted	Actual length	Length L (value-expression-3)		
		Constant (character string)		Not a constant
		$L_1 \leq 255$	$L_1 \geq 256$	
CHAR(n)	--	CHAR (L_1)	VARCHAR (L_1) [L_2]	VARCHAR (n) [L_2]
VARCHAR(n)	--			

Legend:

[ ]: Value enclosed in square brackets is the actual length.

L_1

MIN(length L of *value-expression-3* x c, n)

L_2

Number of bytes in the character substring of the extracted L character ($L \leq L_2 \leq L_1$)

c

If the character set of the value expression result is EBCDIK, 1; if UTF-16, 4.

n

Definition length (in bytes) of the data column (*value-expression-1*) for processing

--

Not applicable

Note

If *value-expression-1* is a variable-length character string, the area where the base data is not included in the string being removed is filled with spaces of that character set.

Table 2-47: Data types and lengths of the SUBSTR scalar function (when the data type of value-expression-1 is a character string type that uses a character set other than the default character set and value-expression-3 is omitted)

Data type of character string (value-expression-1) from which the substring is to be extracted	Actual length	Start position S (value-expression-2)			
		Constant (character string)			Non-constant
		value-expression-1: Fixed length		value-expression-1: Variable length	
		$k_0 \leq 255$	$k_0 \geq 256$		
CHAR(n)	--	CHAR(k_0)	VARCHAR(n)[k_1]	--	VARCHAR(n)[k_1]
VARCHAR(n)	[r]	--	--	VARCHAR(n)[k_2]	VARCHAR(n)[k_2]

Legend:

[ ]: Value enclosed in square brackets is the actual length.

n

Definition length (in bytes) of the data column (*value-expression-1*) for processing

k_0

$n - (S - 1) \times c_1$

k_1

Number of bytes in the partial character string from the *S*th byte through the *n*th byte $\text{MAX}(n - (S - 1) \times c_{2,0}) \leq k_1 \leq (n - (S - 1) \times c_1)$

k_2

Number of bytes in the character substring from the *S*th byte through the *r*th byte $\text{MAX}(r - (S - 1) \times c_2, 0) \leq k_2 \leq \text{MAX}(r - (S - 1) \times c_1, 0)$

c_1

Minimum number of bytes representing each character

If the character set of the value expression result is EBCDIK, 1; if UTF-16, 2.

c_2

Maximum number of bytes representing each character

If the character set of the value expression result is EBCDIK, 1; if UTF-16, 4.

--

Not applicable

Note

If *value-expression-1* is a variable-length character string, *value-expression-3* is omitted, and k_2 is `0`, the result will be the null value.

(d) Example

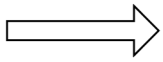
Retrieve the rows in which the two characters in column `C1` (`CHARACTER` data type) of table `T1` beginning with the second character are `95`:

```
SELECT C1 FROM T1
   WHERE SUBSTR(C1,2,2)='95'
```

Table name: `T1`

Column `C1` (`CHARACTER(6)`)

A95157
B93045
A94105
C95009
B95063



Execution result

A95157
C95009
B95063

(21) TIME

(a) Function

Converts the character string representation of time in a specified format into time data.

(b) Format

`TIME (`*value-expression*`[,`*datetime-format*`])`

(c) Rules

1. The following items can be specified as the value expression:
   - Literals that are character string expressions of the time
   - `CURRENT_TIME`
   - Column specifications
   - Component specification
   - SQL variables or SQL parameters

- Time operations that produce results that are of the time data type
- Concatenation operations
- Set functions (MAX, MIN)
- Scalar functions
- CASE expressions
- CAST specification
- Function call
- Scalar subquery

2. The value expression must be in one of the following data types:
   - A datetime format specified:
     - Character string data type with a defined length of 6 to 255 characters (CHAR, VARCHAR). However, if the character set is UTF-16, it is a character string data type with a defined length of 12 to 510 characters (CHAR, VARCHAR).
     - Mixed character string data type (MCHAR, MVARCHAR)
   - A datetime format not specified:
     - Character string data type with a defined length of eight bytes (CHAR, VARCHAR). However, if the character set is UTF-16, it is a character string data type with a defined length of 16 bytes (CHAR, VARCHAR).
     - Time data type (TIME)
3. The value expression must be a character string representation of time in the format specified in the datetime format. If the datetime format is a character string data type (CHAR, VARCHAR), specifying the character set of the data type is optional.
4. The value expression must be the character string representation of time in the format specified in *datetime-format*. If *datetime-format* is omitted, the predefined character string representation of time must be used.

   Examples:

   Datetime format `'HH-MI-SS'` → `'13-45-17'`

   Datetime format omitted → `'13:45:17'`
5. If the value expression is of the time data type, the result will be the equivalent time.
6. For datetime formats, see *1.11 Specifying a datetime format*.

7. The data type of the result is the time data type (TIME).
8. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression or the datetime format is the null value, the result also is the null value.
9. When specifying a datetime format, use the same character set for the value expression and datetime format. However, if the datetime format is the value expression listed below, it is converted to the character set of the value expression:
   - Character string literal

(d) Example

1. Obtains the difference between the earliest time and the latest time in column C1 (data type: CHAR) of table T1:

```
SELECT MAX(TIME(C1))-MIN(TIME(C1))
      FROM T1
```

Table name: T1

Column C1 (CHARACTER(8))

22:12:30
08:35:40
14:05:03
01:28:57

(Most recent time)

(Earliest time)

Execution result

204333

2. Obtains time data from a character string in column C1 (data type: CHAR) of table T2, expressed in a format ('HHMISS') other than the predefined character string representation of time:

```
SELECT TIME(C1,'HHMISS') FROM T2
```

Table name: T2

Column C1 (CHAR(6))

115930
203058

Execution result

11:59:30
20:30:58

(22) TIMESTAMP

(a) Function

1. Converts the predefined character expression of a time stamp into time stamp data.

2. Converts the cumulative number of days from January 1, year 1 (A.D.) into the equivalent time stamp data.
3. Converts date data and time data into time stamp data that is the combination of the two data items.

(b) Format

Format of function 1:

```
TIMESTAMP(value-expression)
```

Format of function 2:

```
TIMESTAMP(value-expression)
```

Format of function 3:

```
TIMESTAMP(value-expression-1, value-expression-2)
```

(c) Rules for function 1

1. The following items can be specified in *value-expression*:
   - Time stamp literals in predefined character string representation
   - CURRENT_TIMESTAMP[(*p*)]
   - Column specification
   - Component specification
   - SQL variables or SQL parameters
   - Concatenation operation
   - Set functions
   - Scalar functions
   - CASE expressions
   - CAST specification
   - Function calls
   - Scalar subquery
2. The following data types can be specified:

- Character string data type with a defined length of 19 to 26 characters (CHAR, VARCHAR). However, if the character set is UTF-16, it is a character string data type with a defined length of 38 to 52 characters (CHAR, VARCHAR).
- Time stamp data type (TIMESTAMP)

3. If the value expression is a character string data type (CHAR, VARCHAR), specifying the character set of the data type is optional.
4. When specifying a character data type in *value-expression*, specify the predefined character string representation of a time stamp.
5. If *value-expression* is of the time stamp data type, the result is that time stamp.
6. The data type of the result is the time stamp data type (TIMESTAMP) taking the following fractional second precision:
   - If *value-expression* is of the character data type, its fractional second precision is based on the predefined character string representation of the time stamp of the *value-expression*.
   - If *value-expression* is the time stamp data, its fractional second precision is that of the time stamp data.

(d) Rules for function 2

1. The following items can be specified in *value-expression*:
   - Numeric literals
   - Column specification
   - Component specification
   - SQL variables or SQL parameters
   - Arithmetic operations
   - Set functions
   - Scalar functions
   - CASE expressions
   - CAST specification
   - Function calls
   - Scalar subquery
2. The data type of *value-expression* must be an integer (INTEGER). When specifying an arithmetic operation, a scalar function, a CAST expression, a CASE specification, a function call, or a set function, ensure that the result of the operation is an integer data type.

3. *value-expression* must be in the range from 1 to 3,652,059.
4. The result is a time stamp after (*specified-numeric-value* - 1) since January 1, year 1 (A.D.). The time part of the result is 0:0:0.

   Example:

   If *value-expression* is 35 → February 4, year 1 (A.D.), 0:0:0 .

(e) Rules for function 3

1. The following items can be specified in *value-expression-1*:
   - Predefined character string representation literals of dates
   - `CURRENT_DATE`
   - Column specification
   - Component specification
   - SQL variables or SQL parameters
   - Date operations producing operation results that are of the date data type
   - Concatenation operation
   - Set functions
   - Scalar functions
   - `CASE` expressions
   - `CAST` specification
   - Function calls
   - Scalar subquery
2. The following data types can be specified in *value-expression-1*:
   - Character string data type with a defined length of 10 bytes (`CHAR`, `VARCHAR`). However, if the character set is UTF-16, it is a character string data type with a defined length of 20 bytes (`CHAR`, `VARCHAR`).
   - Date data type (`DATE`)
3. If *value-expression-1* is a character string data type (`CHAR`, `VARCHAR`), specifying the character set of the data type is optional.
4. In *value-expression-1*, specify the predefined character string representation of a date.

   Example:

   ```
   '1995-06-30'
   ```

5. The following items can be specified in *value-expression-2*:
   - Predefined character string representation literals of time
   - CURRENT_TIME
   - Column specification
   - Component specification
   - SQL variables or SQL parameters
   - Time operations producing results that are time data
   - Concatenation operation
   - Set functions
   - Scalar functions
   - CASE expressions
   - CAST specification
   - Function calls
   - Scalar subquery
6. The following data types can be specified in *value-expression-2*:
   - Character string data type with a defined length of eight bytes (CHAR, VARCHAR). However, if the character set is UTF-16, it is a character string data type with a defined length of 16 bytes (CHAR, VARCHAR).
   - Time data type (TIME)
7. If *value-expression-2* is a character string data type (CHAR, VARCHAR), specifying the character set of the data type is optional.
8. In *value-expression-2*, specify the predefined character string representation of time.

   Example:

   ```
   '13:45:17'
   ```
9. Use the same character set for *value-expression-1* and *value-expression-2*. However, if either *value-expression-1* or *value-expression-2* is the value expression listed below, it is converted to the character set of the corresponding value expression:
   - Character string literal

(f) Common rules

1. The data type of the result is of the time stamp data type (TIMESTAMP).

2. The value of the result is not `NOT NULL` constrained (the null value is allowed). If the value expression is the null value, the result also is the null value.

(g) Examples

1. From column `C1` (data type: time stamp data type) in table `T1`, retrieve data occurring since a specified time stamp:

```
SELECT C1 FROM T1
  WHERE C1 >= TIMESTAMP('2000-01-01 00:00:00.00')
```

Table name: T1

Column C1 (time stamp data type)
2002-01-01 00:00:00.00
2001-12-31 23:59:59.59
1999-12-31 23:59:40.35
1995-01-24 10:10:10.00

Execution result

Column C1 (time stamp data type)
2002-01-01 00:00:00.00
2001-12-31 23:59:59.59

2. Convert column `C1` (data type: numeric data type) in table `T1` into time stamp data and insert the results into column `C1` (data type: time stamp data type) in table `T2`:

```
INSERT INTO T2(C1) SELECT TIMESTAMP(C1) FROM T1
```

Table name: T1

Column C1 (numeric data type)
730000
735000
740000
745000

Execution result

Table name: T2

Column C1 (time stamp data type)
1999-09-03 00:00:00
2013-05-12 00:00:00
2027-01-19 00:00:00
2040-09-27 00:00:00

3. Combine the data from column `C1` (data type: date data type) in table `T1` with the data from column `C2` (data type: time data type) in table `T1`, and insert the results into column `C1` (data type: time stamp data type) in table `T2`:

```
INSERT INTO T2(C1) SELECT TIMESTAMP(C1,C2) FROM T1
```

Table name: T1

Column C1 (date data type)	Column C2 (time stamp data type)
2002-04-20	00:00:00
2002-05-10	10:10:10
2002-04-26	15:16:20
2002-04-29	18:03:12

Execution result

Table name: T2

Column C1 (time stamp data type)
2002-04-20 00:00:00
2002-05-10 10:10:10
2002-04-26 15:16:20
2002-04-29 18:03:12

(23) TIMESTAMP_FORMAT

(a) Function

Converts the character string representation of a time stamp based on a specified datetime format into time stamp data.

(b) Format

```
TIMESTAMP_FORMAT(value-expression, datetime-format)
```

(c) Rules

1. The following items can be specified in *value-expression*:
   - Character string representation literals of time stamps
   - Column specification
   - Component specification
   - SQL variables or SQL parameters
   - Concatenation operation
   - Set functions
   - Scalar functions
   - `CASE` expressions
   - `CAST` specification
   - Function calls
   - Scalar subquery
2. Use the following data types for *value-expression*:
   - Character string data type with a defined length of 14 to 255 characters (`CHARACTER`, `VARCHAR`). However, if the character set is UTF-16, it is a character string data type with a defined length of 28 to 510 characters (`CHARACTER`, `VARCHAR`).
   - Mixed character string data type (`MCHAR`, `MVARCHAR`)
3. If the value expression is a character string data type (`CHAR`, `VARCHAR`), specifying the character set of the data type is optional.
4. In *value-expression*, specify the character string representation of a time stamp in a format specified in the datetime format.

   Example:

Datetime format '*YYYY/MM/DD HH-MI-SS.NNNN*':

```
->'2002/06/30 10-45-30.1523'
```

5. For datetime formats, see *1.11 Specifying a datetime format*.
6. The data type of the result is of the time stamp data type (TIMESTAMP) with a fractional second precision of 6.
7. Specify the same character set in *value-expression* and *datetime-format*. However, if *datetime-format* is the datetime format listed below, it is converted to the character set of the corresponding value expression:
   - Character string literal

(d) Common rules

1. The value of the result is not NOT NULL constrained (the null value is allowed). If the value expression or the datetime format is the null value, the result also is the null value.

(e) Example

Obtain time stamp data from a character string in column C1 (data type: CHAR) in table T1, represented in a format ('*DD/MON/YYYY HH-MI-SS NNNN*') other than the predefined character string representation of a time stamp:

```
SELECT TIMESTAMP_FORMAT(C1,'DD/MON/YYYY HH-MI-SS NNNN')
  FROM T1
```

Table name: T1

Column C1 (CHAR (24))

01/JAN/2000 10-23-02 5620
16/JUL/2002 18-43-37 3415

Execution result

2000-01-01 10:23:02.562000
2002-07-16 18:43:37.341500

(24) UPPER

(a) Function

Converts the lowercase alphabetic characters in character data, national character data, or mixed character data into uppercase.

(b) Format

```
UPPER (value-expression)
```

(c) Rules

1. The following items can be specified as a *value-expression*:
   - Literals
   - USER

- Column specifications
- Component specification
- SQL variables or SQL parameters
- Concatenation operations
- Set functions
- Scalar functions (HEX, LOWER, SUBSTR, UPPER)
- CASE expressions
- CAST specification
- Function call
- Scalar subquery

2. NULL, embedded variables, or the ? parameter cannot be specified in the value expression.
3. The data type of the value expression must be the character string data type (CHAR or VARCHAR), the national character string data type (NCHAR or NVARCHAR) or the mixed character string data type (MCHAR or MVARCHAR).
4. If the data type of the value expression is a character string data type (CHAR, VARCHAR), specifying the character set of the data type is optional.
5. The execution result inherits the data type and the data length of value expression.
6. The value of the result is not NOT NULL constrained (null values are allowed). Therefore, if the value expression is the null value, the result is also the null value.
7. The character set of the result is the character set of the value expression specified in the argument.

(25) VALUE

(a) Function

The VALUE scalar function extracts the value indicated by the first non-null value expression from a list of value expressions.

(b) Format

VALUE (*value-expression* [, *value-expression*]...)

(c) Rules

1. The following items can be specified as a value expression:
   - Literals
   - USER

- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP[(*p*)]
- Column specifications
- Component specification
- SQL variables or SQL parameters
- Arithmetic operations
- Date operations
- Time operations
- Concatenation operations
- Set functions
- Scalar functions
- CASE expressions
- CAST specification
- ? parameters or embedded variables
- Function call
- Scalar subquery

2. The maximum allowable number of value expressions is 255.
3. The VALUE scalar function cannot be specified for a value expression if execution of the value expression produces any of the following data types:
   - BLOB
   - BINARY with a minimum length of 32,001 bytes
   - BOOLEAN
   - Abstract data type
4. NULL cannot be specified in a value expression.
5. The ? parameter or an embedded variable cannot be specified alone in the first value expression (including specification in monomial operational expressions).
6. All the value expressions must have data types that are compatible for comparison purposes.

   Example: If one value expression is the CHAR data type, all other value expressions must also be the CHAR data type.

For data types that can be compared, see *1.2 Data types*.

The following data types cannot be compared:

- Date data and a character string expression of date data
- Time data and a character string expression of time data
- Time stamp data and the character string representation of time stamp data
- Date interval data and a decimal expression of date interval data
- Time interval data and a decimal expression of time interval data
- Binary data and hexadecimal character string literals

7. If one or more value expressions of `VALUE` are `?` parameters or embedded variables, the data types of the `?` parameters or embedded variables will be assumed to be the same as in the first value expression.
8. The list of value expressions is evaluated sequentially from left to right. The first value that is not the null value is taken as the result.
9. The data type and the data length of the result are the same as data type and data length of the result of a set operation (`UNION ALL` or `EXCEPT ALL`). For details, see *2.2 Query expressions*.
10. The value of the result is not `NOT NULL` constrained (null values are allowed). Therefore, if the value expression is the null value, the result is also the null value.
11. If the value expression is a character string data type, use the same character set for all value expressions. However, if one of the following value expression is specified in the second or later argument, the value expression is converted to the character set of the value expression specified with the first argument.
    - Character string literal
    - Embedded variable (default character set)
    - `?` parameter
12. If the data type of the result is a character string data type, the character set of the value expression specified in the first argument becomes the character set of the result.

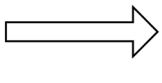
(d) Example

Extract the null value from column `C2` of table `T5`, and assign the value 0:

```
SELECT VALUE(C1,C2,C3,0)
   FROM T5
```

Table name: T5

Column C1	Column C2	Column C3
*	20	*
*	*	*



Execution result

20
0

Note: * indicates the null value.

(26) VARCHAR_FORMAT

(a) Function

Converts date data, time data, or time stamp data into a character string representation according to a specified datetime format.

(b) Format

```
VARCHAR_FORMAT(value-expression, datetime-format)
```

(c) Rules

1. The following items can be specified in *value-expression*:
   - CURRENT_DATE
   - CURRENT_TIME
   - CURRENT_TIMESTAMP[(*p*)]
   - Column specification
   - Component specification
   - SQL variables or SQL parameters
   - Date operations producing results that are of the date data type
   - Time operations producing results that are of the time data type
   - Set functions
   - Scalar functions
   - CASE expressions
   - CAST specification
   - Function calls
   - Scalar subquery

2. The data type of *value-expression* must be the date data type (DATE), time data type (TIME), or time stamp data type (TIMESTAMP).
3. For datetime formats, see *1.11 Specifying a datetime format*.
4. The results take the following data types:

   The datetime format is a character data type (CHAR or VARCHAR):

   VARCHAR(*n*)

   The datetime format is a mixed character data type (MCHAR or MVARCHAR):

   MVARCHAR(*n*)

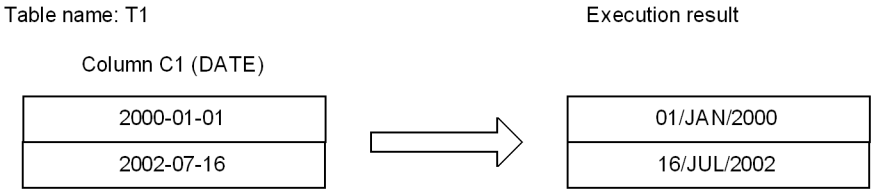
   The defined length *n* takes the following values:

   - If the value expression is specified as a non-literal value and the datetime format is specified in a literal, the defined length is the maximum length of the character string that can be converted according to a specified format.
   - If the value expression is specified as a literal and the datetime format is specified in a literal, the defined length is equal to the length of the character string that is converted according to the format.
   - If the datetime format is specified in a non-literal item, the defined length is equal to *definition-length* + 15 of the data type of the datetime format. However, if the datetime format is a character string data type that uses the UTF-16 character set, it is converted to the data type of *datetime-format* + 30.
5. The value of the result is the predefined character string representation of the data type of the value expression.
6. The value of the result is not NOT NULL constrained (the null value is allowed). If the value expression or the datetime format is the null value, the result also is the null value.
7. The character set of the result is the character set of the datetime format specified in the argument.

(d) Example

From column C1 (data type: DATE) in table T1, obtain a result in a character string represented in a specified datetime format ('*DD/MON/YYYY*'):

```
SELECT VARCHAR_FORMAT(C1,'DD/MON/YYYY') FROM T1
```



(27) YEAR

(a) Function

Extracts the year part from date data, time stamp data, or date interval data.

(b) Format

`YEAR` (*value-expression*)

(c) Rules

1. The following items can be specified in *value-expression*:
   - `CURRENT_DATE`
   - `CURRENT_TIMESTAMP[(`*p*`)]`
   - Column specifications
   - Component specification
   - SQL variables or SQL parameters
   - Date operations
   - Set functions (`MAX`, `MIN`)
   - Scalar functions
   - `CASE` expressions
   - `CAST` specification
   - Function call
   - Scalar subquery
2. The data type of *value-expression* must be the date data type (`DATE`), time stamp data type (`TIMESTAMP`) or date interval data type (`INTERVAL YEAR TO DAY`).
3. The data type of the result is integer (`INTEGER`).
4. If *value-expression* is of the date data type or the time stamp data type, the result is 1 to 9999.
5. If the value expression is of the date interval data type, the result will be in the

range -9999 to 9999. If the result is non-zero, the result has the same sign as the value expression.

6. The value of the result is not NOT NULL constrained (null values are allowed). Therefore, if the value expression is the null value, the result is also the null value.

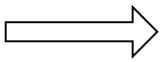
(d) Example

Obtain the year of column C1 (date data type) for which the value of column C2 (INTEGER data type) of table T1 is 221140; because column C1 expresses the end of a year, display the result as the year that is one year prior to the year indicated by column C1:

```
SELECT SUBSTR(DIGITS(YEAR(C1)-1),7,4),N'YEAR' FROM T1
   WHERE C2=221140
```

Table name: T1

Column C1 (date data type)	Column C2 (INTEGER)
1993-03-20	221140
1994-03-20	218030
1995-03-19	220100



Execution result

FY1992

2.16.2 System-defined scalar functions

This section explains the syntax of system-defined scalar functions.

The following rules apply to system-defined scalar functions:

1. Before you can use system-defined scalar functions, you must use pdinit or pdmod to create a data dictionary LOB RDAREA (for details of pdinit and pdmod, see the *HiRDB Version 9 Command Reference* manual).
2. When you specify a repetition column as a value expression for an argument, you must specify a subscript; however, the word ANY cannot be specified as a subscript.
3. When you specify an embedded variable or a ? parameter by itself in a value expression, you must specify the AS data type. When the AS data type is specified, items other than embedded variables or ? parameters cannot be specified.
4. The portable cursor cannot be used in queries on named derived tables derived by specifying a system-defined scalar function.
5. System-defined scalar functions cannot be specified in a derived query expression in a view definition.

(1) ACOS

(a) Function

Returns in the range 0 to π the angle (in radians) that is the inverse cosine of an argument.

(b) Format

```
[MASTER.]ACOS(argument)
```

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a numeric data type.
5. The data type of the result will be FLOAT.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.
7. Specifying in the argument a value whose function value is not defined causes a

definition area error (`domain error occurs`).

(2) ADD_INTERVAL

(a) Function

Returns the predefined character string representation of a time stamp, which is obtained by adding the datetime interval (± *YYYYMMDDhhmmss* .) in decimal representation specified in *argument-2* to the time stamp in predefined character string representation (`'`*YYYY-MM-DD hh:mm:ss*`'`) specified in *argument-1*.

(b) Format

`[MASTER.]ADD_INTERVAL(`*argument-1*`,` *argument-2*`)`

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [`AS` *data-type*]
2. The following can be specified in the value expression of *argument-1*:
   - Character string literal
   - Column specification
   - SQL variable or SQL parameter
   - Concatenation operation
   - Set function
   - Scalar function
   - Function call
   - `CASE` expression
   - `CAST` specification
   - Embedded variable or `?` parameter
   - Scalar subquery
3. The following can be specified in the value expression of *argument-2*:
   - Decimal or integer literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function

- Function call
- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

4. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
5. *argument-1* must be a character string data type (CHAR or VARCHAR), and its length must not exceed 19 bytes.
6. If *value-expression-1* is a character string data type (CHAR, VARCHAR), a character set other than UTF-16 must be used for the data type. If you specify a character string data type that uses the UTF-16 character set in *value-expression-1*, use the CAST specification to convert the data type to the character string data type of the default character set.
7. The data type of *argument-2* must be DECIMAL, INTEGER, or SMALLINT. An integer value must not exceed 14 digits (the decimal point is ignored if specified).
8. The data type of the result will be CHAR(19).
9. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of *argument-1* or *argument-2* is the null value, the result is also the null value.
10. The function result uses the character set of *argument-1*.
11. As the value of *argument-1*, specify the predefined character string representation ('*YYYY-MM-DD hh:mm:ss*') of a valid time stamp. A fractional second must not be used in the predefined character string representation of a time stamp.
12. *argument-2* must specify in the format ±*YYYYMMDDhhmmss*. the decimal representation of a date and time interval:
    - *YYYY*: Years
    - *MM*: Months
    - *DD*: Days
    - *hh*: Hours
    - *mi*: Minutes
    - *ss*: Seconds

If the value of *argument-2* is positive, the operation adds a datetime interval (*YYYYMMDDhhmmss*.) to the time stamp specified in *argument-1*. If the value of *argument-2* is negative, the operation results in the subtraction of the datetime interval (*YYYYMMDDhhmmss*.) from the time stamp specified in *argument-1*.

13. The date and time interval is added or subtracted in the following order: years, months, days, hours, minutes, seconds. If the result falls on a non-existent date (the 31st of a month with fewer than 31 days or on February 29 of a non-leap year), the result is adjusted to the last day of the month.

    Adding months to the last day of a month does not necessarily produce the last day of the result month. Also, adding a number of months to a date and subtracting the same number of months from the resulting date does not necessarily produce the original date.

    Note that subtracting the same number of months does not necessarily produce the original date.

14. If a leap second is specified when the time stamp is represented as a character string in the default format, results are calculated as follows:

    - If the result is 60 or more seconds in an operation that manipulates minutes, the result is changed to 59 seconds.
    - In an operation that manipulates seconds, 60 seconds is treated as 1 minute and 61 seconds is treated as 1 minute and 1 second.
    - A leap second is not included in the function result.

15. If the result of the ADD_INTERVAL function is not within the range 0001/01/01 00:00:00 - 9999/12/31 23:59:59, an overflow error occurs. For details about the operational results when overflow error suppression is in effect, see *2.18 Operational results with overflow error suppression specified*.

16. The result of the function is the predefined character string representation of a time stamp.

(d) Examples

Examples of the ADD_INTERVAL function are shown below:

```
ADD_INTERVAL('1999-12-31 23:59:59',10000000001.)
  ==> '2001-01-01 00:00:00'

ADD_INTERVAL(2001-01-01 00:00:00',-10000000001.)
  ==> '1999-12-31 23:59:59'

ADD_INTERVAL('1956-06-07 03:15:30',400313115450.)
  ==> '1996-09-20 15:10:20'

ADD_INTERVAL('1998-12-31 13:59:59',10200030405.)
  ==> '2000-02-29 17:04:04'
```

```
ADD_INTERVAL('2000-02-29 17:04:04',-10200030405.)
  ==> '1998-12-28 13:59:59'

ADD_INTERVAL('2000-03-05 17:04:60',100.)
  ==> '2000-03-05 17:05:59'
```

(If `set pd_leap_second = Y` is specified in system common definition `pdsys`)

```
ADD_INTERVAL('2000-03-05 17:04:60',-60.)
  ==> '2000-03-05 17:04:00'
```

(If `set pd_leap_second = Y` is specified in system common definition `pdsys`)

```
ADD_INTERVAL(CAST(CURRENT_TIMESTAMP AS CHAR(19)),-100000000.)
  ==> '2008-09-29 10:17:48'
```

(If `CURRENT TIMESTAMP` is 2008 October, 29, 10:17:48)

(3) ASCII

(a) Function

Returns the ASCII integer value for the first character in the character string specified in an argument.

(b) Format

`[MASTER.]ASCII(`*argument*`)`

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [`AS` *data-type*]
2. The following can be specified in the value expression:
   - Character string literal
   - Column specification
   - SQL variable or SQL parameter
   - Concatenation operation
   - Set function
   - Scalar function
   - Function call
   - `CASE` expression

- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a character string data type (CHAR or VARCHAR).
5. The character set used for the data type of the argument must use the default character set.
6. The data type of the result will be INTEGER.
7. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.
8. If the length of the argument value is 0 bytes, the result will be the null value.

(d) Example

An example of the ASCII function is shown below:

```
ASCII('ABC')       ==> 65
CHR(ASCII('ABC')) ==> 'A'
```

(4) ASIN

(a) Function

Returns in the range - π /2 to π /2 the angle (in radians) that is the inverse sine of an argument.

(b) Format

[MASTER.]ASIN(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter

- Arithmetic operation
- Set function
- Scalar function
- Function call
- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a numeric data type.
5. The data type of the result will be FLOAT.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.
7. Specifying in the argument a value whose function value is not defined causes a definition area error (domain error occurs).

(5) ATAN

(a) Function

Returns in the range - π /2 to π /2 the angle (in radians) that is the inverse tangent of an argument.

(b) Format

[MASTER.]ATAN(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Numeric literal
   - Column specification

- SQL variable or SQL parameter
- Arithmetic operation
- Set function
- Scalar function
- Function call
- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a numeric data type.
5. The data type of the result will be FLOAT.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.

(6) ATAN2

(a) Function

Returns in the range - π to π the angle (in radians) that is the inverse tangent of *x/y*, where *x* is *argument-1* and *y* is *argument-2*.

(b) Format

[MASTER.]ATAN2(*argument-1*, *argument-2*)

(c) Rules

1. The following items can be specified in *argument-1* and *argument-2*:
   - *value-expression* [AS *data-type*]
2. The following can be specified in each value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter

- Arithmetic operation
- Set function
- Scalar function
- Function call
- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a numeric data type.
5. The data type of the result will be FLOAT.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of *argument-1* or *argument-2* is the null value, the result is also the null value.

(7) CEIL

(a) Function

Returns the smallest integer that is greater than or equal to the value of an argument.

(b) Format

```
[MASTER.]CEIL(argument)
```

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function

- Scalar function
- Function call
- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a numeric data type.
5. The following table lists the data types of the results.

Table 2-48: Data type of the result of a CEIL system-defined scalar function

Data type of argument	Data type of result
SMALLINT	INTEGER
INTEGER	INTEGER
DECIMAL(*p*, *s*)	DECIMAL(*p*, *s*)
SMALLFLT	FLOAT
FLOAT	FLOAT

6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.
7. If the CEIL function's result is a value that cannot be represented by the result data type, an overflow error will result. For details about the operational results when overflow error suppression is in effect, see *2.18 Operational results with overflow error suppression specified*.

(8) CENTURY

(a) Function

Returns as an ordinal number the century of the date specified in an argument.

(b) Format

[MASTER.]CENTURY(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - CURRENT_DATE
   - Column specification
   - SQL variable or SQL parameter
   - Date operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The data type of the argument must be DATE.
5. The data type of the result will be INTEGER.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.

(d) Examples

Examples of the CENTURY function are shown below:

```
CENTURY(DATE'1900-12-31'))   ==> 19
CENTURY(DATE'1901-01-01'))   ==> 20
CENTURY(DATE'1999-12-31'))   ==> 20
CENTURY(DATE'2000-12-31'))   ==> 20
CENTURY(DATE'2001-01-01'))   ==> 21
```

(9) CHR

(a) Function

Returns the ASCII character represented by the integer value specified in an argument (returns the null value if the argument value is not in the range 0 to 255).

(b) Format

[MASTER.]CHR(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Integer literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The data type of the argument must be INTEGER or SMALLINT.
5. The data type of the result is CHAR(1).
6. The character set of the result is the default character set.
7. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.

(d) Example

An example of the CHR function is shown below:

```
CHR(65            ==> 'A'
ASCII(CHR(65))    ==> 65
```

(10) COS

(a) Function

Returns the cosine (COS trigonometric function) of an argument that is an angle specified in radians.

(b) Format

[MASTER.]COS(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a numeric data type.
5. The data type of the result will be FLOAT.

6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.

(11) COSH

(a) Function

Returns the hyperbolic cosine of an argument.

(b) Format

[MASTER.]COSH(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a numeric data type.
5. The data type of the result will be FLOAT.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null

value.

7. If the COSH function's result is a value that cannot be represented by the result data type, an overflow error will result. For details about the operational results when overflow error suppression is in effect, see *2.18 Operational results with overflow error suppression specified*.

(12) DATE_TIME

(a) Function

Returns the date of the DATE type specified in *argument-1* and the time of the TIME type specified in *argument-2* by converting them into the predefined character string representation ('*YYYY-MM-DD hh:mm:ss*') of a time stamp.

(b) Format

```
[MASTER.]DATE_TIME(argument-1, argument-2)
```

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression of *argument-1*:
   - CURRENT_DATE
   - Column specification
   - SQL variable or SQL parameter
   - Date operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. The following can be specified in the value expression of *argument-2*:
   - CURRENT_TIME
   - Column specification
   - SQL variable or SQL parameter

- Time operations
- Set function
- Scalar function
- Function call
- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

4. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
5. The data type of *argument-1* must be DATE; the data type of *argument-2* must be TIME.
6. The data type of the result is CHAR(19).
7. The character set of the result is the default character set.
8. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of *argument-1* or *argument-2* is the null value, the result is also the null value.
9. The predefined character string representation of the time stamp, which is the result of DATE_TIME(*date*, *time*), is a value that is obtained by concatenation of the predefined character string representation of *date*, one space character, and the predefined character string representation of *time*.
   CHAR(*date*) || ' ' || CHAR(*time*)

(d) Examples

Examples of the DATE_TIME function are shown below:

```
DATE_TIME(DATE('1999-12-31'), TIME('23:59:59'))
  ==> '1999-12-31 23:59:59'
DATE_TIME(CURRENT_DATE, CURRENT_TIME)
  ==> '1999-07-27 11:05:20'
```

(13) DAYNAME

(a) Function

Returns the character string (such as "Sunday" or "Monday") for the day of the week of the date specified in an argument.

(b) Format

[MASTER.]DAYNAME(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - CURRENT_DATE
   - Column specification
   - SQL variable or SQL parameter
   - Date operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The data type of the argument must be DATE.
5. The data type of the result is VARCHAR(18).
6. The character set of the result is the default character set.
7. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.

(d) Examples

Examples of the DAYNAME function are shown below:

```
DAYNAME(DATE('1999-06-06'))    ==> 'Sunday'
DAYNAME(DATE('1999-06-07'))    ==> 'Monday'
DAYNAME(DATE('1999-06-08'))    ==> 'Tuesday'
```

```
DAYNAME(DATE('1999-06-09'))   ==> 'Wednesday'
DAYNAME(DATE('1999-06-10'))   ==> 'Thursday'
DAYNAME(DATE('1999-06-11'))   ==> 'Friday'
DAYNAME(DATE('1999-06-12'))   ==> 'Saturday'
```

(14) DAYOFWEEK

(a) Function

Returns the integer indicating the day of the week (`1` for Sunday, `2` for Monday, and so on) for the date specified in an argument.

(b) Format

`[MASTER.]DAYOFWEEK(`*argument*`)`

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [`AS` *data-type*]
2. The following can be specified in the value expression:
   - `CURRENT_DATE`
   - Column specification
   - SQL variable or SQL parameter
   - Date operation
   - Set function
   - Scalar function
   - Function call
   - `CASE` expression
   - `CAST` specification
   - Embedded variable or `?` parameter
   - Scalar subquery
3. When you specify only an embedded variable or a `?` parameter in the value expression, you must specify the `AS` data type. When the `AS` data type is specified, items other than an embedded variable or a `?` parameter cannot be specified. Specifying the `AS` data type causes the function that has a parameter with the data type specified in the `AS` clause to be called.
4. The data type of the argument must be `DATE`.
5. The data type of the result will be `INTEGER`.

6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.

(d) Examples

Examples of the DAYOFWEEK function are shown below:

```
DAYOFWEEK(DATE('1999-06-06'))   ==> 1
DAYOFWEEK(DATE('1999-06-07'))   ==> 2
DAYOFWEEK(DATE('1999-06-08'))   ==> 3
DAYOFWEEK(DATE('1999-06-11'))   ==> 6
DAYOFWEEK(DATE('1999-06-12'))   ==> 7
  '1999-06-06' was a Sunday.
```

(15) DAYOFYEAR

(a) Function

Returns the integer (in the range 1 to 366) that represents the date specified in an argument as the number of days elapsed since January 1.

(b) Format

[MASTER.]DAYOFYEAR(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - CURRENT_DATE
   - Column specification
   - SQL variable or SQL parameter
   - Date operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. When you specify only an embedded variable or a ? parameter in the value

expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.

4. The data type of the argument must be DATE.
5. The data type of the result will be INTEGER.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.

(d) Examples

Examples of the DAYOFYEAR function are shown below:

```
DAYOFYEAR(DATE('1999-01-01'))   ==>   1
DAYOFYEAR(DATE('1999-02-28'))   ==>  53
DAYOFYEAR(DATE('1999-06-07'))   ==> 158
DAYOFYEAR(DATE('1999-12-31'))   ==> 365
DAYOFYEAR(DATE('2000-12-31'))   ==> 366
```

(16) DEGREES

(a) Function

Converts to degrees the angle specified in radians in an argument.

(b) Format

[MASTER.]DEGREES(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression

- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a numeric data type.
5. The data type of the result will be FLOAT.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.
7. If the DEGREES function's result is a value that cannot be represented by the result data type, an overflow error will result. For details about the operational results when overflow error suppression is in effect, see *2.18 Operational results with overflow error suppression specified*.

(17) EXP

(a) Function

Determines an exponent to the base of the natural logarithm.

(b) Format

[MASTER.]EXP(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call

- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a numeric data type.
5. The data type of the result will be FLOAT.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.
7. If the EXP function's result is a value that cannot be represented by the result data type, an overflow error will result. For details about the operational results when overflow error suppression is in effect, see *2.18 Operational results with overflow error suppression specified*.

(18) FLOOR

(a) Function

Returns the largest integer that is less than or equal to the value of an argument.

(b) Format

[MASTER.]FLOOR(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function

- Function call
- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a numeric data type.
5. The following table lists the data types of the results.

Table 2-49: Data type of the result of a FLOOR system-defined scalar function

Data type of argument	Data type of result
SMALLINT	INTEGER
INTEGER	INTEGER
DECIMAL(*p*, *s*)	DECIMAL(*p*, *s*)
SMALLFLT	FLOAT
FLOAT	FLOAT

6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.
7. If the FLOOR function's result is a value that cannot be represented by the result data type, an overflow error will result. For details about the operational results when overflow error suppression is in effect, see *2.18 Operational results with overflow error suppression specified*.

(19) GREATEST

(a) Function

Returns the largest value among specified arguments.

(b) Format

```
[MASTER.]GREATEST(argument, argument[, argument])
```

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. A maximum of three arguments can be specified.
3. The following can be specified in each value expression:
   - Literal
   - USER, CURRENT_DATE, or CURRENT_TIME
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic, date, time, or concatenation operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
4. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
5. All specified arguments must be of one of the following data types: numeric, character string, mixed character string, national character string, date, or time.
6. If the value expression specified in an argument is a character string data type, use the same character set for all value expressions.
7. The following table lists the data types of the results.

   Table 2-50: Data type of the result of a GREATEST system-defined scalar function

Data type of integer	Data type of result
INTEGER or SMALLINT	INTEGER

Data type of integer	Data type of result
DECIMAL [, INTEGER or SMALLINT]	DECIMAL#1
FLOAT, SMALLFLT [, DECIMAL, INTEGER, or SMALLINT]	FLOAT
CHAR or VARCHAR	VARCHAR#2
MCHAR or MVARCHAR	MVARCHAR#2
NCHAR or NVARCHAR	NVARCHAR#2
DATE	DATE
TIME	TIME

#1: The following precision and scaling apply, where p_i and s_i denote the precision and scaling of the i^{th} argument, respectively:

Precision = max(p_1-s_1, p_2-s_2, ...) + max(s_1, s_2, ...)

Scaling = max(s_1, s_2, ...)

A result whose precision exceeds 38 causes an error.

INTEGER is treated as DECIMAL(10,0); SMALLINT is treated as DECIMAL(5,0).

#2: The following maximum length applies, where n_i denotes the maximum length of the i^{th} argument (the defined length for a fixed-length data type):

Maximum length = max(n_1, n_2, ...)

8. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of any argument is the null value, the result is also the null value.
9. If the data type of the result is a character string data type, the character set used for the result is the character set used for the value expression specified in the argument.

(20) HALF

(a) Function

Based on a month specified in *argument-2* and a day specified in *argument-3* as the beginning of the fiscal year, returns an integer (1 or 2) indicating whether the date specified in *argument-1* is in the first half or the second half of the fiscal year.

(b) Format

```
[MASTER.]HALF(argument-1 [, argument-2 [, argument-3]])
```

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression of *argument-1*:
   - CURRENT_DATE
   - Column specification
   - SQL variable or SQL parameter
   - Date operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. The following can be specified in the value expressions of *argument-2* and *argument-3*:
   - Integer literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery

4. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
5. The data type of *argument-1* must be DATE; the data types of *argument-2* and *argument-3* must be INTEGER or SMALLINT.
6. The data type of the result will be INTEGER.
7. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of any argument is the null value, the result is also the null value.
8. For *argument-2*, the default is 1 and the range of specifiable values is 1 to 12. For *argument-3*, the default is 1 and the range of specifiable values is 1 to 29 when the value of *argument-2* is 2 and 1 to (number of days in the month specified in *argument-2*) in all other cases.
9. The result is determined according to the following rules:
   - If day of *argument-1-date* < *argument-3-day*:

     Number of month*s* = month of *argument-1* date - *argument-2-month* - 1
   - If day of *argument-1-date* ≥ *argument-3-day*:

     Number of months = month of *argument-1* date - *argument-2-month*
   - If number of months < 0:

     Result = (number of months + 12) ÷ 2 + 1
   - If number of months ≥ 0:

     Result = number of months ÷ 2 + 1

(d) Examples

Examples of the HALF function are shown below:

```
HALF(DATE('1999-01-01'))     ==> 1
HALF(DATE('1999-09-10'))     ==> 2
HALF(DATE('1999-12-31'))     ==> 2
HALF(DATE('1999-04-01'), 4)     ==> 1
HALF(DATE('1999-09-10'), 4)     ==> 1
HALF(DATE('1999-03-31'), 4)     ==> 2
HALF(DATE('1999-03-21'), 3, 21)     ==> 1
HALF(DATE('1999-09-20'), 3, 21)     ==> 1
HALF(DATE('1999-03-20'), 3, 21)     ==> 2
```

(21) INSERTSTR (INSERTSTR_LONG)

(a) Function

Deletes from the character string specified in *argument-1* the substring beginning at the character position specified in *argument-2* and consisting of the number of characters specified in *argument-3*, then sets the character string specified in *argument-4* at the *argument-2* position and returns the modified character string.

(b) Format

```
[MASTER.]INSERTSTR(argument-1, argument-2, argument-3, argument-4)
[MASTER.]INSERTSTR_LONG(argument-1, argument-2, argument-3,
argument-4)
```

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expressions of *argument-1* and *argument-4*:
   - Character string literal, mixed character string literal, or national character string literal
   - Column specification
   - SQL variable or SQL parameter
   - Concatenation operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. The following can be specified in the value expressions of *argument-2* and *argument-3*:
   - Integer literal
   - Column specification
   - SQL variable or SQL parameter

- Arithmetic operation
- Set function
- Scalar function
- Function call
- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

4. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
5. *argument-1* and *argument-4* must both be the same character string data type (CHAR or VARCHAR), mixed character string data type (MCHAR or MVARCHAR), or national character string data type (NCHAR or NVARCHAR).
6. If a character string type (CHAR or VARCHAR) value expression is specified in both *argument-1* and *argument-4*, the same character set must be used for both arguments.
7. The following table indicates the maximum length of *argument-4*.

   Table 2-51: Maximum length of argument-4 of the INSERTSTR system-defined scalar function

Data type of argument-1 (and argument-4)		Maximum length of argument-4
CHAR or VARCHAR	Default character set or a character set other than UTF-16	255
	UTF-16 character set	510
MCHAR or MVARCHAR		255
NCHAR or NVARCHAR		127

Table 2-52: Maximum length of argument-4 of the INSERTSTR_LONG system-defined scalar function

Data type of argument-1 (and argument-4)	Maximum length of argument-4
CHAR or VARCHAR	32000
MCHAR or MVARCHAR	32000
NCHAR or NVARCHAR	16000

8. The data types of *argument-2* and *argument-3* must be INTEGER or SMALLINT.
9. Specifying 0 in *argument-3* means that nothing is to be deleted.
10. The following table lists the data types of the results.

    Table 2-53: Data type of the result of the INSERTSTR system-defined scalar function

Data type of argument-1	Data type of result
CHAR(*n*) or VARCHAR(*n*)	VARCHAR(*n*)
MCHAR(*n*) or MVARCHAR(*n*)	MVARCHAR(*n*)
NCHAR(*n*) or NVARCHAR(*n*)	NVARCHAR(*n*)

Table 2-54: Data type of the result of the INSERTSTR_LONG system-defined scalar function

Data type of argument-1	Data type of result
CHAR(*n*) or VARCHAR(*n*)	VARCHAR(32000)
MCHAR(*n*) or MVARCHAR(*n*)	MVARCHAR(32000)
NCHAR(*n*) or NVARCHAR(*n*)	NVARCHAR(16000)

11. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of any argument is the null value, the result is also the null value.
12. If the data type of the result is a character string data type, the character set of the result is the character set of *argument-1*.
13. The following table indicates the ranges of values that can be specified in *argument-2* and *argument-3*. Note that *m* is the value of *argument-2*.

Table 2-55: Ranges of specifiable values in argument-2 and argument-3 of the INSERTSTR system-defined scalar function

Data type of argument-1		Range of values in argument-2	Range of values in argument-3
CHAR(*n*) or VARCHAR(*n*)	Default character set or a character set other than UTF-16	1 to *n*	0 to (*n* + 1 - *m*)
	UTF-16 character set	1 to *n* ÷ 2	0 to (*n* ÷ 2 + 1 - *m*)
MCHAR(*n*) or MVARCHAR(*n*)		1 to *n*	0 to (*n* + 1 - *m*)
NCHAR(*n*) or NVARCHAR(*n*)		1 to *n*	0 to (*n* + 1 - *m*)

Table 2-56: Ranges of specifiable values in argument-2 and argument-3 of the INSERTSTR_LONG system-defined scalar function

Data type of argument-1		Range of value of argument-2	Range of value of argument-3
CHAR(*n*) or VARCHAR(*n*)	Default character set or a character set other than UTF-16	1 to 32000	0 to (32001-*m*)
	UTF-16 character set	1 to 16000	0 to (16001-*m*)
MCHAR(*n*) or MVARCHAR(*n*)		1 to 32000	0 to (32001-*m*)
NCHAR(*n*) or NVARCHAR(*n*)		1 to 16000	0 to (16001-*m*)

14. If the length (number of characters) of the character string specified in *argument-1* is shorter (less) than character position *m* specified in *argument-2*, spaces as defined in the character set of *argument-1* (double-byte spaces if the data type of *argument-1* is NCHAR or NVARCHAR) are added until the number of characters reaches (*m* - 1), and no strings are deleted in any sections.

    If the character string specified in *argument-1* is longer than *m* but less than (*m-1+nd*), the characters beginning at position *m* through the last character are deleted (*nd* is the number of characters specified in *argument-3*).

15. The length of the result must not exceed the maximum length for the result data type. If the length of the result will exceed the length of the *argument-1* character string, we recommend that the INSERTSTR_LONG function be used.

(d) Examples

Examples of the INSERTSTR and INSERTSTR_LONG functions are shown below:

```
INSERTSTR('data warehouse system', 6, 9, 'base')
  ==> 'data base system'
INSERTSTR_LONG('data system', 6, 0, 'warehouse ')
  ==> 'data warehouse system'
INSRTSTR('data base management system', 11, 11, '')
 ==> 'data base system'
```

The character string specified in *argument-4* has a length 0.

```
INSERTSTR_LONG('data base system', 31, 0, '')
  ==> 'data base system               '
```

The result value contains 14 space characters following the string system.

(22) INTERVAL_DATETIMES

(a) Function

Returns the datetime interval between two time stamps in predefined character string representation ('*YYYY-MM-DD hh:mm:ss*') specified in the arguments, in terms of a decimal representation (± *YYYYMMDDhhmmss* .). If *time-stamp-of-argument-1* < *time-stamp-of-argument-2*, the result is a negative value.

(b) Format

[MASTER.]INTERVAL_DATETIMES(*argument-1*, *argument-2*)

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in each value expression:
   - Character string literal
   - Column specification
   - SQL variable or SQL parameter
   - Concatenation operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification

- Embedded variable or ? parameter
- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. *argument-1* and *argument-2* must both be a character string data type (CHAR or VARCHAR).
5. Use the same character set as was used for the value expression specified in *argument-1* and *argument-2*. In addition, use a character set other than UTF-16. If you specify a character string data type that uses the UTF-16 character set in *argument-1* or *argument-2*, use the CAST specification to convert the character string data type to the default character set.
6. The data type of the result will be DECIMAL(14).
7. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of *argument-1* or *argument-2* is the null value, the result is also the null value.
8. For the value of *argument*-1 and *argument-2*, specify a valid predefined character string representation ('*YYYY-MM-DD hh:mm:ss*') of the time stamp; do not specify a fractional second in the predefined character string representation of the time stamp.
9. The result of INTERVAL_DATETIMES(*time-stamp-1*, *time-stamp-2*) is calculated according to the following rules:
   - If *time-stamp-1* < *time-stamp-2*:

     Result = -INTERVAL_DATETIMES(*time-stamp-1*, *time-stamp-2*)
   - If *time-stamp-1* ≥ *time-stamp-2*:
     - If second of *time-stamp-1* ≥ second of *time-stamp-2*:

     Second of the result = second of *time-stamp-1* - second of *time-stamp-2*
     - If second of *time-stamp-1* < second of *time-stamp-2*:

     Second of the result = second of *time-stamp-1* - second of *time-stamp-2* + 60

     Minute of *time-stamp-2* = minute of *time-stamp-2* + 1
     - If minute of *time-stamp-1* ≥ minute of *time-stamp-2*:

     Minute of the result = minute of *time-stamp-1* - minute of *time-stamp-2*
     - If minute of *time-stamp-1* < minute of *time-stamp-2*:

Minute of the result = minute of *time-stamp-1* - minute of *time-stamp-2* + 60

Hour of *time-stamp-2* = hour of *time-stamp-2* + 1

- If hour of *time-stamp-1* ≥ hour of *time-stamp-2*:

Hour of the result = hour of *time-stamp-1* - hour of *time-stamp-2*

- If hour of *time-stamp-1* < hour of *time-stamp-2*:

Hour of the result = hour of *time-stamp-1* - hour of *time-stamp-2* + 24

Day of *time-stamp-2* = day of *time-stamp-2* + 1

- If day of *time-stamp-1* ≥ day of *time-stamp-2*:

Day of the result = day of *time-stamp-1* - day of *time-stamp-2*

- If day of *time-stamp-1* < day of *time-stamp-2*

Day of the result = day of *time-stamp-1* - day of *time-stamp-2* + last day of the month of *time-stamp-2*

Month of *time-stamp-2* = month of *time-stamp-2* + 1

- If month of *time-stamp-1* = month of *time-stamp-2*:

Month of the result = month of *time-stamp-1* - month of *time-stamp-2*

- If month of *time-stamp-1* < month of *time-stamp-2*:

Month of the result = month of *time-stamp-1* - month of *time-stamp-2* + 12

Year of *time-stamp-2* = year of *time-stamp-2* + 1

Year of the result = year of *time-stamp-1* - year of *time-stamp-2*

- Result = (years of result x 10000000000 + months of result x 100000000 + days of result x 1000000 + hours of result x 10000 + minutes of result x 100 + seconds of result)

(d) Examples

Examples of the INTERVAL_DATETIMES function are shown below:

```
INTERVAL_DATETIMES('2001-01-01 00:00:00',
                   '1999-12-31 23:59:59')
  ==>    10000000001.
INTERVAL_DATETIMES('1999-12-31 23:59:59',
                   '2001-01-01 00:00:00')
  ==>   -10000000001.
INTERVAL_DATETIMES('1996-09-20 15:10:20',
                   '1956-06-07 03:15:30')
  ==>   400313115450.
NUMEDIT(INTERVAL_DATETIMES('1996-09-20 15:10:20',
                           '1956-06-07 03:15:30'),
```

```
        '<9990" YEARS "90" MONTHS "90" DAYS "90" HOURS "
        90" MINUTES "90" SECONDS"')
==>  '40 years 3 months 13 days 11 hours 54 minutes 50 seconds'
```

(23) ISDIGITS

(a) Function

Returns a BOOLEAN value indicating whether or not the character string specified in an argument is composed solely of numeric digits.

(b) Format

[MASTER.]ISDIGITS(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Character string literal or mixed character string literal
   - Column specification
   - SQL variable or SQL parameter
   - Concatenation operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a character string data type (CHAR or VARCHAR) or mixed character string data type (MCHAR or MVARCHAR) and its length may not exceed 60 bytes.
5. If the data type of the argument is a character string type (CHAR, VARCHAR), use a

character set other than UTF-16 for the data type. If you specify a character string data type that uses the UTF-16 character set in the argument, use the CAST specification to convert the character string data type to the default character set.

6. The data type of the result will be BOOLEAN.
7. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.
8. If the character string specified in the argument consists solely of numeric digits (0 to 9), the result is TRUE; otherwise, the result is FALSE.

(d) Examples

Examples of the ISDIGITS function are shown below:

```
ISDIGITS('1234567890')  ==> true
ISDIGITS('123ABC')     ==> false
ISDIGITS('')           ==> false
```

(24) IS_DBLBYTES

(a) Function

Returns a BOOLEAN value indicating whether or not the character string specified in an argument is composed solely of double-byte characters.

(b) Format

```
[MASTER.]IS_DBLBYTES(argument)
```

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Character string literal or mixed character string literal
   - Column specification
   - SQL variable or SQL parameter
   - Concatenation operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification

- Embedded variable or ? parameter
- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a character string data type (CHAR or VARCHAR) or mixed character string data type (MCHAR or MVARCHAR).
5. If the data type of the argument is a character string data type (CHAR or VARCHAR), the default character set must be used when specifying the data type of the argument.
6. The data type of the result will be BOOLEAN.
7. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.
8. If the character string specified in the argument consists of only single-byte characters, the result is TRUE; otherwise, the result is FALSE.

(d) Examples

Examples of the IS_DBLBYTES function are shown below:

```
IS_DBLBYTES(M'database management system HiRDB')  ==> false
IS_DBLBYTES(M'データベース管理システムHiRDB')          ==> false
IS_DBLBYTES(M'データベース管理システム')                ==> true
```

(25) IS_SNGLBYTES

(a) Function

Returns a BOOLEAN value indicating whether or not the character string specified in an argument is composed solely of single-byte characters.

(b) Format

[MASTER.]IS_SNGLBYTES(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:

- Character string literal or mixed character string literal
- Column specification
- SQL variable or SQL parameter
- Concatenation operation
- Set function
- Scalar function
- Function call
- CAST specification
- CASE expression
- Embedded variable or ? parameter
- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a character string data type (CHAR or VARCHAR) or mixed character string data type (MCHAR or MVARCHAR).
5. If the data type of the argument is a character string type (CHAR or VARCHAR), the default character set must be used when specifying the data type of the argument.
6. The data type of the result will be BOOLEAN.
7. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.
8. If the character string specified in the argument consists solely of single-byte characters, the result is TRUE; otherwise, the result is FALSE.

(d) Examples

Examples of the IS_SNGLBYTES function are shown below:

```
IS_SNGLBYTES(M'database management system HiRDB')  ==> true
IS_SNGLBYTES(M'データベース管理システムHiRDB')           ==> false
IS_SNGLBYTES(M'データベース管理システム')                ==> false
```

(26) LAST_DAY

(a) Function

Returns the last day of the month for the date specified in an argument.

(b) Format

```
[MASTER.]LAST_DAY(argument)
```

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - CURRENT_DATE
   - Column specification
   - SQL variable or SQL parameter
   - Date operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The data type of the argument must be DATE.
5. The data type of the result will be DATE.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.

(d) Examples

Examples of the LAST_DAY function are shown below:

```
LAST_DAY(DATE('1999-01-01'))  ==> '1999-01-31'
LAST_DAY(DATE('1999-02-16'))  ==> '1999-02-28'
LAST_DAY(DATE('1999-06-10'))  ==> '1999-06-30'
LAST_DAY(DATE('1999-12-25'))  ==> '1999-12-31'
LAST_DAY(DATE('2000-02-03'))  ==> '2000-02-29'
```

(27) LEAST

(a) Function

Returns the smallest value among specified arguments.

(b) Format

[MASTER.]LEAST(*argument*, *argument* [, *argument*])

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. A maximum of three arguments can be specified.
3. The following can be specified in each value expression:
   - Literal
   - USER, CURRENT_DATE, or CURRENT_TIME
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic, date, time, or concatenation operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
4. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.

5. All specified arguments must be of one of the following data types: numeric, character string, mixed character string, national character string, date, or time.
6. If the value expression specified in an argument is a character string data type, use the same character set for all value expressions.
7. The following table lists the data types of the results.

Table 2-57: Data type of the result of the LEAST system-defined scalar function

Data type of integer	Data type of result
INTEGER or SMALLINT	INTEGER
DECIMAL [, INTEGER or SMALLINT]	DECIMAL#1
FLOAT, SMALLFLT [, DECIMAL, INTEGER, or SMALLINT]	FLOAT
CHAR or VARCHAR	VARCHAR#2
MCHAR or MVARCHAR	MVARCHAR#2
NCHAR or NVARCHAR	NVARCHAR#2
DATE	DATE
TIME	TIME

#1: The following precision and scaling apply, where p_i and s_i denote the precision and scaling of the i^{th} argument, respectively:

Precision = $\max(p_1 - s_1, p_2 - s_2, ...) + \max(s_1, s_2, ...)$

Scaling = $\max(s_1, s_2, ...)$

A result whose precision exceeds 38 causes an error.

INTEGER is treated as DECIMAL(10,0); SMALLINT is treated as DECIMAL(5,0).

#2: The following maximum length applies, where n_i denotes the maximum length of the i^{th} argument (the defined length for a fixed-length data type):

Maximum length = $\max(n_1, n_2, ...)$

8. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of any argument is the null value, the result is also the null value.
9. If the data type of the result is a character string data type, it uses the same character set as the argument.

(28) LEFTSTR

(a) Function

Returns from the beginning (the leftmost position) of the character string specified in *argument-1* the substring consisting of the number of characters specified in *argument-2*.

(b) Format

[MASTER.]LEFTSTR(*argument-1*, *argument-2*)

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression of *argument-1*:
   - Character string literal, mixed character string literal, or national character string literal
   - Column specification
   - SQL variable or SQL parameter
   - Concatenation operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. The following can be specified in the value expression of *argument-2*:
   - Integer literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call

- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

4. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
5. Specify one of the following for the data type of *argument-1*:
   - Character string data type using any character set (CHAR, VARCHAR)
   - National character data type (NCHAR, NVARCHAR)
   - Mixed character string data type (MCHAR, MVARCHAR)
6. The data type of *argument-2* must be INTEGER or SMALLINT.
7. The following table indicates the range of values that can be specified in *argument-2*.

   Table 2-58: Range of specifiable values in argument-2 of the LEFTSTR system-defined scalar function

Data type of argument-1		Range of values in argument-2
CHAR(*n*) or VARCHAR(*n*)	Default character set or a character set other than UTF-16	0 to *n*
	UTF-16 character set	0 to *n* ÷ 2
MCHAR(*n*) or MVARCHAR(*n*)		0 to *n*
NCHAR(*n*) or NVARCHAR(*n*)		0 to *n*

8. The following table lists the data types of the results.

   Table 2-59: Data type of the result of the LEFTSTR system-defined scalar function

Data type of argument-1	Data type of result
CHAR(*n*) or VARCHAR(*n*)	VARCHAR(*n*)
MCHAR(*n*) or MVARCHAR(*n*)	MVARCHAR(*n*)
NCHAR(*n*) or NVARCHAR(*n*)	NVARCHAR(*n*)

9. The value of the result is not `NOT NULL` constrained (null values are allowed). If the value expression of *argument-1* or *argument-2* is the null value, the result is also the null value.
10. If the number of characters in the *argument-1* character string is less than the number of characters specified in *argument-2*, the result will be the *argument-1* value.
11. If the data type of the result is a character string type, it uses the same character set as *argument-1*.

(d) Examples

Examples of the `LEFTSTR` function are shown below:

```
LEFTSTR('data base system', 9)  ==> 'data base'
LEFTSTR('DATA SYSTEM', 0)       ==> ''(Character string of 0 length)
```

(29) LN

(a) Function

Returns the natural logarithm of an argument.

(b) Format

`[MASTER.]LN(`*argument*`)`

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [`AS` *data-type*]
2. The following can be specified in the value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - `CASE` expression
   - `CAST` specification
   - Embedded variable or `?` parameter
   - Scalar subquery

3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a numeric data type.
5. The data type of the result will be FLOAT.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.

(30) LOG10

(a) Function

Returns the common logarithm of an argument.

(b) Format

[MASTER.]LOG10(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified,

items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.

4. The argument must be a numeric data type.
5. The data type of the result will be FLOAT.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.

(31) LTRIM

(a) Function

Beginning at the left end of the character string specified in *argument-1*, removes all instances of each of the characters in the character string specified in *argument-2* until a character not found in the *argument-2* character string is encountered.

(b) Format

[MASTER.]LTRIM(*argument-1* [, *argument-2*])

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in each value expression:
   - Character string literal, mixed character string literal, or national character string literal
   - Column specification
   - SQL variable or SQL parameter
   - Concatenation operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. When you specify only an embedded variable or a ? parameter in a value

expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.

4. Specify one of the following for the data type of *argument-1* and *argument-2*:
   - Character string type using any character set (CHAR, VARCHAR)
   - National character string type (NCHAR, NVARCHAR)
   - Mixed character string type (MCHAR, MVARCHAR)
5. If the data type is a character string data type or mixed character string data type, the length of the value of *argument-2* must be 30 bytes or less (60 bytes or less for a character string data type that uses the UTF-16 character set). If the data type is a national character string data type, the length of the value of *argument-2* must be 30 characters or less.
6. If the data type of both *argument-1* and *argument-2* is a character string type, use the same character set for the value expression.
7. The following table lists the data types of the results.

Table 2-60: Data type of the result of the LTRIM system-defined scalar function

Data type of argument-1 (and argument-2)	Data type of result
CHAR(*n*) or VARCHAR(*n*)	VARCHAR(*n*)
MCHAR(*n*) or MVARCHAR(*n*)	MVARCHAR(*n*)
NCHAR(*n*) or NVARCHAR(*n*)	NVARCHAR(*n*)

8. If the data type of the result is a character string type (CHAR or VARCHAR), it uses the same character set as *argument-1*.
9. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of *argument-1* or *argument-2* is the null value, the result is also the null value.
10. If *argument-2* is omitted, a space character of the appropriate data type is assumed.

(d) Examples

Examples of the LTRIM function are shown below:

```
LTRIM('abcabcabdata base', 'abc')  ==> 'data base'
LTRIM('    data base')             ==> 'data base'
```

(32) LTRIMSTR

(a) Function

Beginning at the left end of the character string specified in *argument-1*, deletes each successive occurrence of the character string specified in *argument-2* until it does not find that character string.

(b) Format

```
[MASTER.]LTRIMSTR(argument-1, argument-2)
```

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in each value expression:
   - Character string literal, mixed character string literal, or national character string literal
   - Column specification
   - SQL variable or SQL parameter
   - Concatenation operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. Specify one of the following for the data type of *argument-1* and *argument-2*:
   - Character string type using any character set (CHAR, VARCHAR)
   - National character string type (NCHAR, NVARCHAR)
   - Mixed character string type (MCHAR, MVARCHAR)

5. If the data type of both *argument-1* and *argument-2* is a character string type, use the same character set for the value expression.
6. The following table indicates the maximum length of *argument-2*.

   Table 2-61: Maximum length of argument-2 of the LTRIMSTR system-defined scalar function

Data type of argument-1 (and argument-2)		Maximum length of argument-2
CHAR or VARCHAR	Default character set or a character set other than UTF-16	255
	UTF-16 character set	510
MCHAR or MVARCHAR		255
NCHAR or NVARCHAR		127

7. The following table lists the data types of the results.

   Table 2-62: Data type of the result of the LTRIMSTR system-defined scalar function

Data type of argument-1 (and argument-2)	Data type of result
CHAR(*n*) or VARCHAR(*n*)	VARCHAR(*n*)
MCHAR(*n*) or MVARCHAR(*n*)	MVARCHAR(*n*)
NCHAR(*n*) or NVARCHAR(*n*)	NVARCHAR(*n*)

8. If the data type of the result is a character string type (CHAR or VARCHAR), it uses the same character set as *argument-1*.
9. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of *argument-1* or *argument-2* is the null value, the result is also the null value.

(d) Example

An example of the LTRIMSTR function is shown below:

```
LTRIMSTR('abcabcabdata base', 'abc')  ==> 'abdata base'
```

(33) MIDNIGHTSECONDS

(a) Function

Returns the number of seconds from midnight to the time specified in an argument.

(b) Format

```
[MASTER.]MIDNIGHTSECONDS(argument)
```

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - CURRENT_DATE
   - Column specification
   - SQL variable or SQL parameter
   - Time operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The data type of the argument must be TIME.
5. The data type of the result will be INTEGER.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.
7. The result can be obtained from the following formula:

   Result = (((hour of argument x 60) + minute of argument) x 60) + second of argument

(d) Examples

Examples of the MIDNIGHTSECONDS function are shown below:

```
MIDNIGHTSECONDS(TIME('23:59:59'))    ==> 86399
MIDNIGHTSECONDS(TIME('14:14:14'))    ==> 51254
```

(34) MONTHNAME

(a) Function

Returns the month name (such as "January" or "February") of the date specified in an argument.

(b) Format

```
[MASTER.]MONTHNAME(argument)
```

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - CURRENT_DATE
   - Column specification
   - SQL variable or SQL parameter
   - Date operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The data type of the argument must be DATE.
5. The data type of the result is VARCHAR(18).
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.
7. The character set of the result is the default character set.

(d) Examples

Examples of the MONTHNAME function are shown below:

```
MONTHNAME(DATE('1999-01-01'))   ==> 'January'
MONTHNAME(DATE('1999-02-28'))   ==> 'February'
MONTHNAME(DATE('1999-03-03'))   ==> 'March'
MONTHNAME(DATE('1999-04-01'))   ==> 'April'
MONTHNAME(DATE('1999-05-05'))   ==> 'May'
MONTHNAME(DATE('1999-06-07'))   ==> 'June'
MONTHNAME(DATE('1999-07-07'))   ==> 'July'
MONTHNAME(DATE('1999-08-15'))   ==> 'August'
MONTHNAME(DATE('1999-09-23'))   ==> 'September'
MONTHNAME(DATE('1999-10-10'))   ==> 'October'
MONTHNAME(DATE('1999-11-11'))   ==> 'November'
MONTHNAME(DATE('1999-12-31'))   ==> 'December'
```

(35) MONTHS_BETWEEN

(a) Function

Returns as a real number (FLOAT type) the number of months between two dates specified in arguments.

(b) Format

[MASTER.]MONTHS_BETWEEN(*argument-1*, *argument-2*)

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in each value expression:
   - CURRENT_DATE
   - Column specification
   - SQL variable or SQL parameter
   - Date operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter

- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The data types of *argument-1*and *argument-2* must be DATE.
5. The data type of the result will be FLOAT.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of *argument-1* or *argument-2* is the null value, the result is also the null value.
7. The number of months which is the result of MONTHS_BETWEEN is calculated according to the following rules:

   - If *date-1* < *date-2*:

     Result = -MONTHS_BETWEEN (*date-2*, *date-1*)

   - If *date-1* ≥ *date-2*:

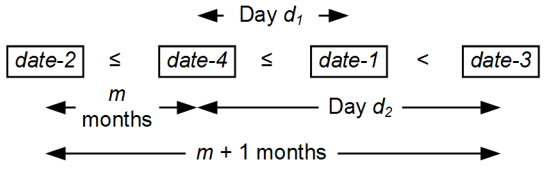
     Let *m* be an integer greater than or equal to 0, and let *date-4* and *date-3* be the date (*date-2* + *m* months) and (*date-2* + (*m* + 1) months), respectively, such that *date-4* ≤ *date-1* < *date-3*

     Let d_1 and d_2 be the number of days between *date-4* and *date-1* (DAYS (*date-1*) - DAYS (*date-4*)) and the number of days between *date-4* and *date-3* (DAYS (*date-3*) - DAYS (*date-4*)), respectively (where d_2 is one of the following values: 28, 29, 30, 31)

     The number of months in the result will be ($m + d_1 \div d_2$) months.

   The following figure shows the relationships between the dates *date-1*, *date-2*, *date-3*, and *date-4*.

   Figure 2-7: Relationships between the dates date-1, date-2, date-3, and date-4



(d) Examples

Examples of the MONTHS_BETWEEN function are shown below:

```
MONTHS_BETWEEN(DATE('1999-07-10'), DATE('1999-06-10'))
```

```
  ==> 1
MONTHS_BETWEEN(DATE('1999-07-11'), DATA('1999-06-10'))
  ==> 1.032258...
MONTHS_BETWEEN(DATE('1999-06-11'), DATA('1999-05-10'))
  ==> 1.033333...
MONTHS_BETWEEN(DATE('1999-02-11'), DATA('1999-01-10'))
  ==> 1.035714...
MONTHS_BETWEEN(DATE('2000-02-11'), DATA('2000-01-10'))
  ==> 1.034482...
MONTHS_BETWEEN(DATE('1999-09-09'), DATA('1999-06-10'))
  ==> 2.967741...
MONTHS_BETWEEN(DATE('1999-06-10'), DATA('1999-09-09'))
  ==> -2.967741...
```

(36) NEXT_DAY

(a) Function

Returns the next date after the date specified in *argument-1* that is the same day of the week as the weekday number specified in *argument-2* (where Sunday is weekday 1).

(b) Format

`[MASTER.]NEXT_DAY(`*argument-1*`,` *argument-2*`)`

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [`AS` *data-type*]
2. The following can be specified in the value expression for *argument-1*:
   - `CURRENT_DATE`
   - Column specification
   - SQL variable or SQL parameter
   - Date operation
   - Set function
   - Scalar function
   - Function call
   - `CASE` expression
   - `CAST` specification
   - Embedded variable or `?` parameter
   - Scalar subquery
3. The following can be specified in the value expression for *argument-2*:

- Integer literal
- Column specification
- SQL variable or SQL parameter
- Arithmetic operation
- Set function
- Scalar function
- Function call
- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

4. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
5. The data type of *argument-1* must be DATE; the data type of *argument-2* must be INTEGER or SMALLINT.
6. The data type of the result will be DATE.
7. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of *argument-1* or *argument-2* is the null value, the result is also the null value.
8. The following table lists the meanings of the integer values specified in *argument-2*.

   Table 2-63: Meanings of integer values specifiable in argument-2 (integer values for days of the week)

Integer value of argument-2	Day of week
1	Sunday
2	Monday
3	Tuesday
4	Wednesday
5	Thursday

Integer value of argument-2	Day of week
6	Friday
7	Saturday

9. If the NEXT_DAY function's result is a value that cannot be represented by the result data type, an overflow error will result. For details about the operational results when overflow error suppression is in effect, see *2.18 Operational results with overflow error suppression specified*.

(d) Examples

Examples of the NEXT_DAY function are shown below:

```
NEXT_DAY(DATE('1999-06-10'), 1)   ==> '1999-06-13'
NEXT_DAY(DATE('1999-06-10'), 4)   ==> '1999-06-16'
NEXT_DAY(DATE('1999-06-10'), 5)   ==> '1999-06-17'
NEXT_DAY(DATE('1999-06-10'), 6)   ==> '1999-06-11'
NEXT_DAY(DATE('1999-06-10'), 7)   ==> '1999-06-12'
   1999.06.10 was a Thursday.
```

(37) NUMEDIT

(a) Function

Converts the numeric value specified in *argument-1* into character string representation by editing it according to the format specified in *argument-2*.

(b) Format

```
[MASTER.]NUMEDIT(argument-1, argument-2)
```

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression for *argument-1*:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call

- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

3. The following can be specified in the value expression for *argument-2*:
   - Character string literal
   - Column specification
   - SQL variable or SQL parameter
   - Concatenation operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
4. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
5. *argument-1* must be a numeric data type.
6. *argument-2* must be a character string data type (CHAR or VARCHAR). The length of the value of *argument-2* must not exceed 250 bytes.
7. Use a character set other than UTF-16 for the data type of *argument-2*. If you specify a character string data type that uses the UTF-16 character set in *argument-2*, use the CAST specification to convert the character string data type to the default character set.
8. The data type of the result is VARCHAR(255).
9. The character set of the result is the character set used for *argument-2*.
10. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of *argument-1* or *argument-2* is the null value, the result is also the null value.

11. The value of *argument-2* must be specified in the following format:

```
["character-string"]   [+]
     {[{<|>}] {9|0|,|"character-string"}...
        [,{9|0|,|"character-string"}...]
       |{E|e} .integer ["character-string"]}
```

"*character-string*"

A character string enclosed in double quotation marks represents itself.

To specify double quotation marks within a character string enclosed in double quotation marks, specify two sets of double quotation marks in succession.

\+

The [+] sign specifies the method of sign representation.

\+ uses the minus sign (-) to represent *argument-1* as a negative value; otherwise, it uses a plus sign (+) to represent the value.

The default for the '+' symbol is a minus sign (-) if the value of *argument-1* is negative, and a space character otherwise.

{< | >}

The {< | >} symbols specify justification of the character representation of the numeric value.

The "<" symbol removes all space characters and shifts the character representation to the left.

The ">" symbol removes all space characters, shifts the character representation to the right, and pads the left side with space characters.

If neither "<" nor ">" is specified, the character representation is not shifted.

{ 9 | 0 | , | "*character-string*"} ... [, { 9 | 0 | , | "*character-string*"}] ...

The numbers "9" and "0" each represent one digit of the numeric value so that the number of digits associated with the numeric value of *argument-1* can be edited.

The total number of "9"s and "0"s represents the precision of the numeric value after being edited.

Specifying "0"s produces an edit result equal in digit positions to the number of "0"s specified.

Specifying a "," produces an edit result that is the comma itself if the edit results on both sides of "," are numeric characters; otherwise, the comma is replaced with one space character.

The symbol "." represents the decimal point.

The total number of "9"s and "0"s following the "." indicates the scaling of the numeric value after editing.

Specifying a "9" before a "." causes the result of editing by "9" to be a space character, provided that the corresponding digit is 0 and the "9" is the first digit, or the edit result to the left of the result of editing by "9" is not a numeric character (except when it is associated with a comma); otherwise, the result of editing by "9" will be a numeric character in the corresponding digit position.

Specifying a "9" after a "." causes the result of editing by "9" to be an empty (0) character, provided that the corresponding digit is 0 and the "9" is the last digit, or the edit result to the right of the result of editing by "9" is not a numeric character (except when it is associated with a comma); otherwise, the result of editing by "9" will be a numeric character in the corresponding digit position.

{ E | e }.*integer* is specified in floating-point decimal format.

The result of the editing is a floating-point decimal format with a scaling for the mantissa specified in *integer*. The scaling for the mantissa must not exceed 30.

12. If the value of *argument-1* is a value that cannot be represented by the format specified in *argument-2*, an overflow error will result. For details about the operational results when overflow error suppression is in effect, see *2.18 Operational results with overflow error suppression specified*.

13. If low-order digits are truncated by the editing, the result value is rounded off.

(d) Examples

Examples of the NUMEDIT function are shown below:

```
NUMEDIT(1234567.89, '99,999,990.00"$"')   ==>'  1,234,567.89$'
NUMEDIT(1000, '99,999,990.00"$"')         ==>'      1,000.00$'
NUMEDIT(1234567.89, '" $"99,999,990.00') ==>' $  1,234,567.89'
NUMEDIT(1000, '" $"99,999,990.00')        ==>' $      1,000.00'
NUMEDIT(1234567.89, '"$"+99,999,990.00') ==>'$+ 1,234,567.89'
NUMEDIT(-1000, '"$"+99,999,990.00')       ==>'$-     1,000.00'
NUMEDIT(1234567.89, '"$">99,999,990.00') ==>'  $1,234,567.89'
NUMEDIT(1000, '"$">99,999,990.00')        ==>'      $1,000.00'
NUMEDIT(1234567.89, '"$"<99,999,990.00') ==>'$1,234,567.89'
NUMEDIT(1000, '"$"<99,999,990.00')        ==>'$1,000.00'
NUMEDIT(0.5, '"$"<99,999,990.00')         ==>'$0.50'
NUMEDIT(1234567.89, '+E.10"$"')         ==>'+1.2345678900E+0.6$'
NUMEDIT(1234567.89, '"$"+e.10')         ==>'$+1.2345678900e+0.6'
```

(38) PI

(a) Function

Returns π, the value of the circle constant.

(b) Format

```
[MASTER.]PI()
```

(c) Rules

1. The data type of the result will be FLOAT.
2. The value of the result is not NOT NULL constrained (null values are allowed). However, because the value of π is always returned, the null value will never be returned.

(d) Example

An example of the PI function is shown below:

```
PI() ==> 3.14159265358979323846
```

(39) POSSTR

(a) Function

If in the character string specified in *argument-1* the substring specified in *argument-2* occurs at least the number of times (*nd*) specified in argument-4 at or after the character position specified in *argument-3*, returns the starting position (character position) of the nd^{th} substring.

(b) Format

```
[MASTER.]POSSTR(argument-1, argument-2, [, argument-3
                                [, argument-4]])
```

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in each of the value expressions of *argument-1* and *argument-2*:
   - Character string literal, mixed character string literal, or national character string literal
   - Column specification
   - SQL variable or SQL parameter
   - Concatenation operation

- Set function
- Scalar function
- Function call
- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

3. The following can be specified in each of the value expressions of *argument-3* and *argument-4*:
   - Integer literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
4. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
5. Specify one of the following for the data type of *argument-1* and *argument-2*:
   - Character string data type using any character set (CHAR, VARCHAR)
   - National character data type (NCHAR, NVARCHAR)
   - Mixed character string data type (MCHAR, MVARCHAR)
6. If the data type of both *argument-1* and *argument-2* is a character string data type (CHAR, VARCHAR), use the same character set for the value expression.
7. The following table indicates the maximum length of the value of *argument-2*.

Table 2-64: Maximum length of argument-2 of the POSSTR system-defined scalar function

Data type of argument-1 (and argument-2)		Maximum length of argument-2
CHAR or VARCHAR	Default character set or a character set other than UTF-16	255
	UTF-16 character set	510
MCHAR or MVARCHAR		255
NCHAR or NVARCHAR		127

8. The data types of *argument-3* and *argument-4* must be INTEGER or SMALLINT.
9. The data type of the result will be INTEGER.
10. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of any argument is the null value, the result is also the null value.
11. The following table indicates the ranges of values that can be specified in *argument-3* and *argument-4*.

    Table 2-65: Ranges of specifiable values in argument-3 and argument-4 of the POSSTR system-defined scalar function

Data type of argument-1 (and argument-2)		Range of values in argument-3	Range of values in argument-4
CHAR(*n*) or VARCHAR(*n*)	Default character set or a character set other than UTF-16	1 to *n*	1 to *n*
	UTF-16 character set	1 to *n* ÷ 2	1 to *n* ÷ 2
MCHAR(*n*) or MVARCHAR(*n*)		1 to *n*	1 to *n*
NCHAR(*n*) or NVARCHAR(*n*)		1 to *n*	1 to *n*

12. The default for *argument-3* is 1; the default for *argument-4* is also 1.
13. If the length (number of characters) of the character string in *argument-2* is 0, the result will be the value of *argument-3*.
14. If the substring matching the character string specified in *argument-2* occurs more than the number of times (*nd*) specified in *argument-4* at or after the character position (*m*) specified in *argument-3* in the character string specified in *argument-1*, the result will be a value greater than or equal to *m* and equal to the number of characters in the character string specified in *argument-1* that are to the

left of the beginning of the nd^{th} substring, plus 1. It should be noted that these substrings that match the character string specified in *argument-2* are not duplicate substrings. If the substring matching the character string specified in *argument-2* does not occur more than the number of times (*nd*) specified in *argument-4* at or after the character position (*m*) specified in *argument-3* in the character string specified in *argument-1*, the result will be 0.

(d) Examples

Examples of the POSSTR function are shown below:

```
POSSTR(data base system', 'a')        ==> 2
POSSTR(data base system', '')         ==> 1    (argument-2 is
POSSTR(data base system', 'a', 5)     ==> 7    character string
POSSTR(data base system', 'st')       ==> 13    of 0 length.)
POSSTR(data base system', 'a', 1, 3) ==> 7
POSSTR(data base system', 'manager') ==> 0
```

(40) POWER

(a) Function

Returns the n^{th} power of the value of *argument-1*, where *n* denotes the value of *argument-2*.

(b) Format

```
[MASTER.]POWER(argument-1, argument-2)
```

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in each value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification

- Embedded variable or ? parameter
- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. *argument-1* and *argument-2* must both be a numeric data type.
5. The following table lists the data types of the results.

   Table 2-66: Data type of the result of the POWER system-defined scalar function

Data type of argument-1	Data type of argument-2	
	SMALLINT or INTEGER	DECIMAL(p,s), SMALLFLT, or FLOAT
SMALLINT or INTEGER	INTEGER	FLOAT
DECIMAL(*p*, *s*), SMALLFLT, or FLOAT	FLOAT	FLOAT

6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of *argument-1* or *argument-2* is the null value, the result is also the null value.
7. Specifying a negative value in *argument-1* or a non-integer value in *argument-2* causes a definition area error (domain error occurs).
8. Specifying the value 0 in *argument-1* and a non-positive value in *argument-2* causes division by zero (division by zero). For details about the operational results when overflow error suppression is in effect, see *2.18 Operational results with overflow error suppression specified*.
9. If the POWER function's result is a value that cannot be represented by the result data type, an overflow error will result. For details about the operational results when overflow error suppression is in effect, see *2.18 Operational results with overflow error suppression specified*.

(41) QUARTER

(a) Function

Based on a month specified in *argument-2* and a day specified in *argument-3* as the beginning of the fiscal year, returns an integer (1, 2, 3, or 4) indicating the fiscal-year quarter in which the date specified in *argument-1* falls.

(b) Format

```
[MASTER.]QUARTER(argument-1 [, argument-2 [, argument-3]])
```

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression of *argument-1*:
   - CURRENT_DATE
   - Column specification
   - SQL variable or SQL parameter
   - Date operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. The following can be specified in the value expressions of *argument-2* and *argument-3*:
   - Integer literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery

4. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
5. The data type of *argument-1* must be DATE.
6. The data types and *argument-2* and *argument-3* must be INTEGER or SMALLINT.
7. The data type of the result will be INTEGER.
8. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of any argument is the null value, the result is also the null value.
9. For *argument-2*, the default is 1 and the range of specifiable values is 1 to 12.
10. For *argument-3*, the default is 1 and the range of specifiable values is 1 to 29 when the value of *argument-2* is 2 and 1 to (number of days in the month specified in *argument-2*) in all other cases.
11. The result is determined according to the following rules:
    - If day of *argument-1* date < *argument-3* day:

      Number of months = month of *argument-1* date - *argument-2* month - 1
    - If day of *argument-1* date ≥ *argument-3*:

      Number of months = month of *argument-1* date - *argument-2* month
    - If number of months < 0:

      Result = (number of months + 12) ÷ 4 + 1
    - If number of months ≥ 0:

      Result = number of months ÷ 4 + 1

(d) Examples

Examples of the QUARTER function are shown in the following:

```
QUARTER(DATE('1999-01-01'))          ==> 1
QUARTER(DATE('1999-09-10'))          ==> 3
QUARTER(DATE('1999-12-31'))          ==> 4
QUARTER(DATE('1999-04-01'), 4)       ==> 1
QUARTER(DATE('1999-09-10'), 4)       ==> 2
QUARTER(DATE('1999-03-31'), 4)       ==> 4
QUARTER(DATE('1999-03-21'), 3, 21)   ==> 1
QUARTER(DATE('1999-09-20'), 3, 21)   ==> 2
QUARTER(DATE('1999-03-20'), 3, 21)   ==> 4
```

(42) RADIANS

(a) Function

Converts to radians the angle specified in degrees in an argument.

(b) Format

```
[MASTER.]RADIANS(argument)
```

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a numeric data type.
5. The data type of the result will be FLOAT.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.

(43) REPLACE (REPLACE_LONG)

(a) Function

Replaces in the character string specified in *argument-1* all instances of the character string specified in *argument-2* with the character string specified in *argument-3*.

(b) Format

```
[MASTER.]REPLACE(argument-1, argument-2[, argument-3])
[MASTER.]REPLACE_LONG(argument-1, argument-2, argument-3)
```

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in each value expression:
   - Character string literal, mixed character string literal, or national character string literal
   - Column specification
   - SQL variable or SQL parameter
   - Concatenation operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. Specify one of the following for the data type of *argument-1*, *argument-2*, and *argument-3*:
   - Character string type using any character set (CHAR, VARCHAR)
   - National character string type (NCHAR, NVARCHAR)

- Mixed character string type (MCHAR, MVARCHAR)

5. The following table indicates the maximum lengths of the values of *argument-2* and *argument-3*.

   Table 2-67: Maximum lengths of argument-2 and argument-3 of the REPLACE and REPLACE_LONG system-defined scalar functions

Data type of argument-1 (and of argument-2 and argument-3)		Maximum length of argument-2 and argument-3
CHAR or VARCHAR	Default character set or a character set other than UTF-16	255
	UTF-16 character set	510
MCHAR or MVARCHAR		255
NCHAR or NVARCHAR		127

6. If the data type of *argument-1*, *argument-2*, and *argument-3* are character string types (CHAR or VARCHAR), use the same character set for the value expressions specified in the arguments.
7. If *argument-3* is omitted, an empty character string is assumed, and the function deletes all substrings matching the character string specified in *argument-2* from the character string specified in *argument-1*.
8. The following table lists the data types of the results.

   Table 2-68: Data type of the result of the REPLACE system-defined scalar function

Data type of argument-1	Data type of result
CHAR(*n*) or VARCHAR(*n*)	VARCHAR(*n*)
MCHAR(*n*) or MVARCHAR(*n*)	MVARCHAR(*n*)
NCHAR(*n*) or NVARCHAR(*n*)	NVARCHAR(*n*)

   Table 2-69: Data type of the result of the REPLACE_LONG system-defined scalar function

Data type of argument-1	Data type of result
CHAR(*n*) or VARCHAR(*n*)	VARCHAR(32000)
MCHAR(*n*) or MVARCHAR(*n*)	MVARCHAR(32000)
NCHAR(*n*) or NVARCHAR(*n*)	NVARCHAR(16000)

9. If the data type of the result is a character string type (CHAR or VARCHAR), it uses the same character set as *argument-1*.
10. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of any argument is the null value, the result is also the null value.
11. The length of the result must not exceed the maximum length for the result data type. If the length of the result will exceed the length of the *argument-1* character string, we recommend that the REPLACE_LONG function be used.

(d) Examples

Examples of the REPLACE function are shown below:

```
REPLACE('a big dog and a small dog', 'dog', 'cat')
  ==> 'a big cat and small cat'
REPLACE('a big dog and a small dog', 'big ')
  ==> 'a dog and a small dog'
```

(44) REVERSESTR

(a) Function

Returns a character string that is the reverse of the character string specified in an argument (the returned character string reads from left to right the same as the specified character string reads from right to left).

(b) Format

[MASTER.]REVERSESTR(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Character string literal, mixed character string literal, or national character string literal
   - Column specification
   - SQL variable or SQL parameter
   - Concatenation operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression

- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. Specify one of the following for the data type of the argument:
   - Character string type using any character set (CHAR, VARCHAR)
   - National character string type (NCHAR, NVARCHAR)
   - Mixed character string type (MCHAR, MVARCHAR)
5. The following table lists the data types of the results.

   Table 2-70: Data type of the result of the REVERSESTR system-defined scalar function

Data type of argument	Data type of result
CHAR(*n*) or VARCHAR(*n*)	VARCHAR(*n*)
MCHAR(*n*) or MVARCHAR(*n*)	MVARCHAR(*n*)
NCHAR(*n*) or NVARCHAR(*n*)	NVARCHAR(*n*)

6. If the data type of the result is a character string type (CHAR or VARCHAR), it uses the same character set as the argument.
7. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.

(d) Examples

Examples of the REVERSESTR function are shown below:

```
REVERSESTR('data base')      ==> 'esab atad'
REVERSESTR('esab atad')      ==> 'data base'
REVERSESTR('   esab atad')   ==> 'data base   '
```

(45) RIGHTSTR

(a) Function

Returns from the end (the rightmost position) of the character string specified in *argument-1* the substring consisting of the number of characters specified in

argument-2.

(b) Format

```
[MASTER.]RIGHTSTR(argument-1, argument-2)
```

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression of *argument-1*:
   - Character string literal, mixed character string literal, or national character string literal
   - Column specification
   - SQL variable or SQL parameter
   - Concatenation operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. The following can be specified in the value expression of *argument-2*:
   - Integer literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter

- Scalar subquery

4. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
5. Specify one of the following for the data type of *argument-1*:
   - Character string type using any character set (CHAR, VARCHAR)
   - National character string type (NCHAR, NVARCHAR)
   - Mixed character string type (MCHAR, MVARCHAR)
6. The data type of *argument-2* must be INTEGER or SMALLINT.
7. The following table indicates the range of values that can be specified in *argument-2*.

   Table 2-71: Range of specifiable values in argument-2 of the RIGHTSTR system-defined scalar function

Data type of argument-1		Range of values in argument-2
CHAR(*n*) or VARCHAR(*n*)	Default character set or a character set other than UTF-16	0 to *n*
	UTF-16 character set	0 to *n* ÷ 2
MCHAR(*n*) or MVARCHAR(*n*)		0 to *n*
NCHAR(*n*) or NVARCHAR(*n*)		0 to *n*

8. The following table lists the data types of the results.

   Table 2-72: Data type of the result of the RIGHTSTR system-defined scalar function

Data type of argument-1	Data type of result
CHAR(*n*) or VARCHAR(*n*)	VARCHAR(*n*)
MCHAR(*n*) or MVARCHAR(*n*)	MVARCHAR(*n*)
NCHAR(*n*) or NVARCHAR(*n*)	NVARCHAR(*n*)

9. If the data type of the result is a character string type (CHAR or VARCHAR), it uses the same character set as the numerical value of *argument-1*.
10. The value of the result is not NOT NULL constrained (null values are allowed). If

the value expression of *argument-1* or *argument-2* is the null value, the result is also the null value.

11. If the number of characters in the *argument-1* character string is less than the number of characters specified in *argument-2*, the result will be the *argument-1* value.

(d) Examples

Examples of the RIGHTSTR function are shown below:

```
RIGHTSTR('data base system', 6) ==> 'system'
RIGHTSTR('data system', 0)      ==> ''
                                    (Character string of 0 length)
```

(46) ROUND

(a) Function

Rounds the value of *argument-1* following the n^{th} place after the decimal point (number of digits following the decimal point given by 10^{-n}), where *n* is the value of *argument-2*.

Specifying *argument-3* causes the function to round up if the value of *argument-1* at the rightmost digit position given by $1 \times 10^{(-n\_1)}$ is greater than or equal to the value of *argument-3*; otherwise, the function rounds down.

(b) Format

```
[MASTER.]ROUND(argument-1, [, argument-2[, argument-3]])
```

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in each value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression

- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. *argument-1* must be a numeric data type; the data types of *argument-2* and *argument-3* must be INTEGER or SMALLINT.
5. The default for *argument-2* is 0. The following table indicates the range of values that can be specified in *argument-2*.

   Table 2-73: Range of specifiable values in argument-2 of the ROUND system-defined scalar function

Data type of result	Range of values in argument-2
INTEGER	-9 to 0
DECIMAL(*p*, *s*)	-(*p* - *s* - 1) to *s*
FLOAT	HP-UX (32-bit mode): -307 to 307 HP-UX (64-bit mode), Solaris, AIX, Linux, and Windows: -307 to 323

6. The following table lists the data types of the results.

   Table 2-74: Data type of the result of the ROUND system-defined scalar function

Data type of argument-1	Data type of result
SMALLINT	INTEGER
INTEGER	INTEGER
DECIMAL(*p*, *s*)	DECIMAL(*p*, *s*)
SMALLFLT	FLOAT
FLOAT	FLOAT

7. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of any argument is the null value, the result is also the null value.

8. The default for *argument-3* is 5. The range of values that can be specified in *argument-3* is 1 to 9. If the value of *argument-1* at the rightmost digit position given by $1 \times 10^{(-n\_1)}$ is greater than or equal to the value of *argument-3*, the value of *argument-1* is rounded up; otherwise, it is rounded down.
9. If the ROUND function's result is a value that cannot be represented by the result data type, an overflow error will result. For details about the operational results when overflow error suppression is in effect, see *2.18 Operational results with overflow error suppression specified*.

(47) ROUNDMONTH

(a) Function

Rounds off the date specified in *argument-1* to the first day of either the specified date's month or the following month, using the day of the month specified in *argument-2* to determine the point for rounding up to the following month. This function can be used, for example, to change a date on or after the 20th of the month to the first of the next month.

(b) Format

```
[MASTER.]ROUNDMONTH(argument-1 [, argument-2])
```

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression for *argument-1*:
   - CURRENT_DATE
   - Column specification
   - SQL variable or SQL parameter
   - Date operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. The following can be specified in the value expression for *argument-2*:

- Integer literal
- Column specification
- SQL variable or SQL parameter
- Arithmetic operation
- Set function
- Scalar function
- Function call
- `CASE` expression
- `CAST` specification
- Embedded variable or `?` parameter
- Scalar subquery

4. When you specify only an embedded variable or a `?` parameter in a value expression, you must specify the `AS` data type. When the `AS` data type is specified, items other than an embedded variable or a `?` parameter cannot be specified. Specifying the `AS` data type causes the function that has a parameter with the data type specified in the `AS` clause to be called.
5. The data type of *argument-1* must be `DATE`; the data type of *argument-2* must be `INTEGER` or `SMALLINT`.
6. The data type of the result will be `DATE`.
7. The value of the result is not `NOT NULL` constrained (null values are allowed). If the value expression of *argument-1* or *argument-2* is the null value, the result is also the null value.
8. The default for *argument-2* is 16. The range of values that can be specified in *argument-2* is 1 to 32.
9. The result is determined according to the following rules:
   - If day of *argument-1* < *argument-2*:

     Year of result = year of *argument-1*

     Month of result = month of *argument-1*

     Day of result = 1
   - If day of *argument-1* ≥ *argument-2*:
     - If month of *argument-1* < 12:

     Year of result = year of *argument-1*

Month of result = month of *argument-1* + 1

Day of result = 1

- If month of *argument-1* = 12:

Year of result = year of *argument-1* + 1

Month of result = 1

Day of result = 1

10. If the result of the ROUNDMONTH function is not within the range 0001/01/01 - 9999/12/31, an overflow error will result. For details about the operational results when overflow error suppression is in effect, see *2.18 Operational results with overflow error suppression specified*.

(d) Examples

Examples of the ROUNDMONTH function are shown below:

```
ROUNDMONTH(DATE('1999-08-15'))       ==> '1999-08-01'
ROUNDMONTH(DATE('1999-08-16'))       ==> '1999-09-01'
ROUNDMONTH(DATE('1999-09-20'), 21)   ==> '1999-09-01'
ROUNDMONTH(DATE('1999-09-21'), 21)   ==> '1999-10-01'
ROUNDMONTH(DATE('1999-12-21'), 21)   ==> '2000-01-01'
ROUNDMONTH(DATE('1999-03-31'), 32)   ==> '1999-03-01'
ROUNDMONTH(DATE('1999-02-29'), 29)   ==> '2000-03-01'
ROUNDMONTH(DATE('1999-03-31'), 29)   ==> '2000-04-01'
ROUNDMONTH(DATE('9999-12-31'), 26)   ==> Overflow
```

(48) RTRIM

(a) Function

Beginning at the right end of the character string specified in *argument-1*, removes all instances of each of the characters in the character string specified in *argument-2* until a character not found in the *argument-2* character string is encountered.

(b) Format

```
[MASTER.]RTRIM(argument-1 [, argument-2])
```

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in each value expression:
   - Character string literal, mixed character string literal, or national character string literal
   - Column specification

- SQL variable or SQL parameter
- Concatenation operation
- Set function
- Scalar function
- Function call
- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. Specify one of the following for the data type of *argument-1* and *argument-2*:
   - Character string type using any character set (CHAR, VARCHAR)
   - National character string type (NCHAR, NVARCHAR)
   - Mixed character string type (MCHAR, MVARCHAR)
5. If the data type of both *argument-1* and *argument-2* is a character string type (CHAR or VARCHAR), use the same character set for the value expressions specified in the arguments.
6. If the data type is a character string type or mixed character string type, the length of the value of *argument-2* must be 30 bytes or less (60 bytes or less for a character string data type that uses the UTF-16 character set). If the data type is a national character string type, the length of the value of *argument-2* must be 30 characters or less.
7. The following table lists the data types of the results.

Table 2-75: Data type of the result of the RTRIM system-defined scalar function

Data type of argument-1 (and argument-2)	Data type of result
CHAR(*n*) or VARCHAR(*n*)	VARCHAR(*n*)
MCHAR(*n*) or MVARCHAR(*n*)	MVARCHAR(*n*)
NCHAR(*n*) or NVARCHAR(*n*)	NVARCHAR(*n*)

8. If the data type of the result is a character string type (CHAR or VARCHAR), it uses the same character set as *argument-1*.
9. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of *argument-1* or *argument-2* is the null value, the result is also the null value.
10. If *argument-2* is omitted, one space character in the data type of the result is assumed.

(d) Examples

Examples of the RTRIM function are shown below:

```
RTRIM('data basebcabcabc', 'abc')   ==> 'data base'
RTRIM('database    ')               ==> 'data base'
```

(49) RTRIMSTR

(a) Function

Beginning at the right end of the character string specified in *argument-1*, deletes each successive occurrence of the character string specified in *argument-2* until it does not find that character string.

(b) Format

```
[MASTER.]RTRIMSTR(argument-1, argument-2)
```

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in each value expression:
   - Character string literal, mixed character string literal, or national character string literal
   - Column specification
   - SQL variable or SQL parameter
   - Concatenation operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter

- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. Specify one of the following for the data type of *argument-1* and *argument-2*:
   - Character string type using any character set (CHAR, VARCHAR)
   - National character string type (NCHAR, NVARCHAR)
   - Mixed character string type (MCHAR, MVARCHAR)
5. If the data type of both *argument-1* and *argument-2* is a character string type (CHAR or VARCHAR), use the same character set for the value expressions specified in the arguments.
6. The following table indicates the maximum length of *argument-2*.

   Table 2-76: Maximum length of argument-2 of the RTRIMSTR system-defined scalar function

Data type of argument-1 (and argument-2)		Maximum length of argument-2
CHAR or VARCHAR	Default character set or a character set other than UTF-16	255
	UTF-16 character set	510
MCHAR or MVARCHAR		255
NCHAR or NVARCHAR		127

7. The following table lists the data types of the results.

   Table 2-77: Data type of the result of the RTRIMSTR system-defined scalar function

Data type of argument-1 (and argument-2)	Data type of result
CHAR(*n*) or VARCHAR(*n*)	VARCHAR(*n*)
MCHAR(*n*) or MVARCHAR(*n*)	MVARCHAR(*n*)
NCHAR(*n*) or NVARCHAR(*n*)	NVARCHAR(*n*)

8. If the data type of the result is a character string type (CHAR or VARCHAR), it uses the same character set as *argument-1*.

9. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of *argument-1* or *argument-2* is the null value, the result is also the null value.

(d) Example

An example of the RTRIMSTR function is shown below:

```
RTRIMSTR('data basebcabcabc', 'abc')   ==> 'data basebc'
```

(50) SIGN

(a) Function

Returns the sign of an argument (+1 for positive, -1 for negative, 0 for zero).

(b) Format

[MASTER.]SIGN(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a numeric data type.

5. The following table lists the data types of the results.

Table 2-78: Data type of the result of the SIGN system-defined scalar function

Data type of argument-1	Data type of result
SMALLINT	INTEGER
INTEGER	INTEGER
DECIMAL(*p*, *s*)	DECIMAL(1,0)
SMALLFLT	FLOAT
FLOAT	FLOAT

6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.

(51) SIN

(a) Function

Returns the sine (SIN trigonometric function) of an argument in which an angle is specified in radians.

(b) Format

[MASTER.]SIN(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification

- Embedded variable or ? parameter
- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a numeric data type.
5. The data type of the result will be FLOAT.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.

(52) SINH

(a) Function

Returns the hyperbolic sine of an argument.

(b) Format

[MASTER.]SINH(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery

3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a numeric data type.
5. The data type of the result will be FLOAT.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.
7. If the SINH function's result is a value that cannot be represented by the result data type, an overflow error will result. For details about the operational results when overflow error suppression is in effect, see *2.18 Operational results with overflow error suppression specified*.

(53) SQRT

(a) Function

Returns the square root of the value of an argument.

(b) Format

[MASTER.]SQRT(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter

- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a numeric data type.
5. The data type of the result will be FLOAT.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.
7. Specifying a negative value in the argument causes a definition area error (domain error occurs).

(54) STRTONUM

(a) Function

Converts the character string representation of the numeric value specified in *argument-1* into a numeric data type. Conversion is into INTEGER when the data type of *argument-2* is INTEGER or SMALLINT; into DECIMAL with the same precision and scaling when the data type of *argument-2* is DECIMAL; and into FLOAT when the data type of *argument-2* is FLOAT or SMALLFLT.

(b) Format

[MASTER.]STRTONUM(*argument-1*, *argument-2*)

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression for *argument-1*:
   - Character string literal
   - Column specification
   - SQL variable or SQL parameter
   - Concatenation operation
   - Set function
   - Scalar function
   - Function call

- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

3. The following can be specified in the value expression for *argument-2*:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
4. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called..
5. *argument-1* must be a character string data type (CHAR or VARCHAR). The length of the character string specified in *argument-1* must not exceed 255 bytes.
6. Use a character set other than UTF-16 for the data type of *argument-1*. If you specify a character string data type that uses the UTF-16 character set in *argument-1*, use the CAST specification to convert the character string data type to the default character set.
7. *argument-2* must be a numeric data type.
8. The following table lists the data types of the results.

Table 2-79: Data type of the result of the STRTONUM system-defined scalar function

Data type of argument-2	Data type of result
SMALLINT	INTEGER
INTEGER	INTEGER
DECIMAL(*p*, *s*)	DECIMAL(*p*, *s*)
SMALLFLT	FLOAT
FLOAT	FLOAT

9. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of *argument-1* is the null value, the result is also the null value. The value of *argument-2* does not affect the result.
10. Specify the value of the character string in *argument-1* in the following format:

    • Data type of *argument-2* is INTEGER:

```
[space-character...] [+|-] [space-character ...] numeric-character...
    [space-character ...]
```

    • Data type of *argument-2* is DECIMAL:

```
[space-character...] [+|-] [space-character...]
    |numeric-character...[. [numeric-character...]]
    | .numeric-character...| [space-character...]
```

    • Data type of *argument-2* is FLOAT:

```
[space-character...] [+|-] [space-character...]
    |numeric-character... [. [numeric-character...]]
    | .numeric-character...|
    [{E|e} [+|-] numeric-character...] [space-character...]
```

11. If the STRTONUM function's result is a value that cannot be represented by the result data type, an overflow error will result. For details about the operational results when overflow error suppression is in effect, see *2.18 Operational results with overflow error suppression specified*.
12. When the data type of *argument-2* is DECIMAL and low-order digits are truncated by the data type conversion, the result value is rounded off.

(d) Examples

Examples of the STRTONUM function are shown below:

■ When *argument-2* is INTEGER:

```
STRTONAM(' - 1234567 ', 0)
```

```
  ==> -1234567                    (INTEGER type)
STRTONUM('  + 1234567890123  ', 0)
  ==> Overflow
STRTONUM('  1234567.89  ', 9)
  ==> Invalid value in argument-1 in STRTONUM function
      during conversion to INTEGER
```

■ When *argument-2* is DECIMAL:

```
STRTONUM(' -1234567  ', 123456789012.
  ==> -1234567                  (DECIMAL(12, 0) type)
STRTONUM('  1234567.89  ', 9999999999.999)
  ==. 1234567.89                (DECIMAL(13, 3) type)
STRTONUM('  1234567.89  ', 99999.999)
 ==> Overflow
STRTONUM('  1.23456789E6 ', 9999999999.999)
 ==> Invalid value in argument-1 in STRTONUM function
     during conversion to DECIMAL
```

■ When *argument-2* is FLOAT:

```
STRTONUM('  1234567.89  ', 1e0)
  ==> 1.23456789e6                (FLOAT type)
STRTONUM('  -1234567890123  ', 0e0)
  ==> -1.234567890123E12          (FLOAT type)
STRTONUM('  1.23456789E6 ', 1E1)
  ==> 1.23456789E6                (FLOAT type)
STRTONUM('  1.0E310  ', 9E9)
  ==> Overflow
```

(55) TAN

(a) Function

Returns the tangent (TAN trigonometric function) of an argument in which an angle is specified in radians.

(b) Format

[MASTER.]TAN(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter

- Arithmetic operation
- Set function
- Scalar function
- Function call
- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a numeric data type.
5. The data type of the result will be FLOAT.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.
7. If the TAN function's result is a value that cannot be represented by the result data type, an overflow error will result. For details about the operational results when overflow error suppression is in effect, see *2.18 Operational results with overflow error suppression specified*.

(56) TANH

(a) Function

Returns the hyperbolic tangent of an argument.

(b) Format

[MASTER.]TANH(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - Numeric literal
   - Column specification

- SQL variable or SQL parameter
- Arithmetic operation
- Set function
- Scalar function
- Function call
- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in the value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The argument must be a numeric data type.
5. The data type of the result will be FLOAT.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.
7. If the TANH function's result is a value that cannot be represented by the result data type, an overflow error will result. For details about the operational results when overflow error suppression is in effect, see *2.18 Operational results with overflow error suppression specified*.

(57) TRANSL (TRANSL_LONG)

(a) Function

If any of the characters in the character string specified in *argument-2* is included in the character string specified in *argument-1*, returns a character string in which those characters are translated into the corresponding characters in the character string specified in *argument-3*, where the i^{th} character in the character string specified in *argument-2* corresponds to the i^{th} character in *argument-3*. If the number of characters in *argument-3* is less than the number of characters in the character string of *argument-2*, for the character string in *argument-3*, HiRDB assumes a character string that is obtained by repeatedly filling the character string in *argument-3* with the characters in *argument-4*.

(b) Format

```
[MASTER.]TRANSL(argument-1, argument-2, argument-3[, argument-4])
[MASTER.]TRANSL_LONG(argument-1, argument-2, argument-3
    [, argument-4])
```

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in each value expression:
   - Character string literal, mixed character string literal, or national character string literal (for TRANS_LONG, mixed character string literal only)
   - Column specification
   - SQL variable or SQL parameter
   - Concatenation operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. If there are fewer characters in the character string specified in *argument-3* than in the character string specified in *argument-2*, the character specified in *argument-4* is used to pad out the *argument-3* character string to the same length as the *argument-2* character string.
5. Only one character can be specified in *argument-4*. If *argument-4* is omitted, a space from the character set used for *argument-1* (double-byte spaces if the data type of *argument-1* is NCHAR or NVARCHAR) is assumed.
6. For TRANSL, specify one of the following for the data type of all arguments *argument-1*, *argument-2*, *argument-3*, and *argument-4*:

- Character string type using any character set (CHAR, VARCHAR)
- National character string type (NCHAR, NVARCHAR)
- Mixed character string type (MCHAR, MVARCHAR)

7. For the TRANSL_LONG function, the data type of all arguments must be MCHAR or MVARCHAR.
8. If the data type of all arguments *argument-1*, *argument-2*, *argument-3*, and *argument-4* for TRANSL is a character string type (CHAR or VARCHAR), use the same character set for the value expressions specified in the arguments.
9. The following table indicates the maximum lengths of the values of *argument-2* and *argument-3*.

   Table 2-80: Maximum lengths of argument-2 and argument-3 of the TRANSL and TRANSL_LONG system-defined scalar functions

Data type of argument-1		Maximum length of argument-2 and argument-3
CHAR or VARCHAR#	Default character set or a character set other than UTF-16	255
	UTF-16 character set	510
MCHAR or MVARCHAR		255
NCHAR or NVARCHAR#		127

   #: Not specifiable for TRANSL_LONG.

10. The following table lists the data types of the results.

    Table 2-81: Data type of the result of the TRANSL system-defined scalar function

Data type of argument-1	Data type of result
CHAR(*n*) or VARCHAR(*n*)	VARCHAR(*n*)
MCHAR(*n*) or MVARCHAR(*n*)	MVARCHAR(*n*)
NCHAR(*n*) or NVARCHAR(*n*)	NVARCHAR(*n*)

Table 2-82: Data type of the result of the TRASNSL_LONG system-defined scalar function

Data type of argument-1	Data type of result
CHAR(*n*) or VARCHAR(*n*)	Not specifiable
MCHAR(*n*) or MVARCHAR(*n*)	MVARCHAR(32000)
NCHAR(*n*) or NVARCHAR(*n*)	Not specifiable

11. If the data type of the result of TRANSL is a character string type (CHAR or VARCHAR), it uses the same character set as *argument-1*.
12. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of any argument is the null value, the result is also the null value.

(d) Examples

Examples of the TRANSL and TRANSL_LONG functions are shown below:

```
TRANSL('data base system', 'abcdefghijklmnopqrstuvwxyz',
       'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
  ==> 'DATA BASE SYSTEM'  (Translation of lowercase characters
                           into uppercase characters)
TRANSL('<data base> system', '[ ] { } ( ) < >', '')
  ==> ' data base system '
      (argument-3 is a character string with a length of 0, and all
      enclosure symbols (such as [, ], {, }, (, ), <, >) are translated into spaces)

TRANSL('+12345.678', '+-012345678', 'SS', '9')
  ==> 'S99999.999'
TRANSL (M' d a t a  b a s e  s y s t e m'
        M' 0 1 2 3 4 5 6 7 8 9 a b c d e f g h i j k l m n
           o p q r s t u v w x y z'
        M'0123456789abcdefghijklmnopqrstuvwxyz')
  ==> M'data base system' (Translation of double-byte characters
                           into single-byte characters)

TRANSL_LONG(M'data base system',
            M'0123456789abcdefghijklmnopqrstuvwxyz',
            M'0 1 2 3 4 5 6 7 8 9 a b c d e f g h i j k l m n
           o p q r s t u v w x y z')
  ==> M'd a t a  b a s e  s y s t e m'
      (Translation of single-byte characters into double-byte characters)

TRANSL('+12345.678', '+012345678', '', 'S')
  ==> 'SSSSSS.SSS'
TRANSL('2000-03-31 12:23:30', '-:', '/.')
```

```
==> '2000/03/31 12.23.30'
```

(58) TRUNC

(a) Function

Returns the value obtained by rounding off the value of *argument-1* at the n^{th} place after the decimal point (number of digits following the decimal point given by 10^{-n}), where *n* is the value specified in *argument-2*.

(b) Format

`[MASTER.]TRUNC(`*argument-1* `[,` *argument-2*`])`

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [`AS` *data-type*]
2. The following can be specified in each value expression:
   - Numeric literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - `CASE` expression
   - `CAST` specification
   - Embedded variable or `?` parameter
   - Scalar subquery
3. When you specify only an embedded variable or a `?` parameter in a value expression, you must specify the `AS` data type. When the `AS` data type is specified, items other than an embedded variable or a `?` parameter cannot be specified. Specifying the `AS` data type causes the function that has a parameter with the data type specified in the `AS` clause to be called.
4. *argument-1* must be a numeric data type; the data type of *argument-2* must be `INTEGER` or `SMALLINT`.
5. The default for *argument-2* is `0`. The following table indicates the range of values that can be specified in *argument-2*.

Table 2-83: Range of specifiable values in argument-2 of the TRUNC system-defined scalar function

Data type of result	Range of values in argument-2
INTEGER	-9 to 0
DECIMAL(*p*, *s*)	-(*p* - *s* - 1) to *s*
FLOAT	HP-UX (32-bit mode): -307 to 307 HP-UX (64-bit mode) Solaris, AIX, Linux, and Windows: -307 to 323

6. The following table lists the data types of the results.

   Table 2-84: Data type of the result of the TRUNC system-defined scalar function

Data type of argument-1	Data type of result
SMALLINT	INTEGER
INTEGER	INTEGER
DECIMAL(*p*, *s*)	DECIMAL(*p*, *s*)
SMALLFLT	FLOAT
FLOAT	FLOAT

7. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of *argument-1* or *argument-2* is the null value, the result is also the null value.

(59) TRUNCYEAR

(a) Function

Based on a month specified in *argument-2* and a day specified in *argument-3* as the beginning of the fiscal year, returns the first day of the fiscal year in which the date specified in *argument-1* falls. This function can be used to determine a fiscal year that ends, for example, on March 20 or on the last day of March.

(b) Format

```
[MASTER.]TRUNCYEAR(argument-1[, argument-2[, argument-3]])
```

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]

2. The following can be specified in the value expression of *argument-1*:
   - CURRENT_DATE
   - Column specification
   - SQL variable or SQL parameter
   - Date operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. The following can be specified in the value expressions of *argument-2* and *argument-3*:
   - Integer literal
   - Column specification
   - SQL variable or SQL parameter
   - Arithmetic operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
4. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
5. The data type of *argument-1* must be DATE.

6. The data types of *argument-2* and *argument-3* must be INTEGER or SMALLINT.
7. The data type of the result will be DATE.
8. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of any argument is the null value, the result is also the null value.
9. For *argument-2*, the default is 1 and the range of values that can be specified is 1 to 12.
10. For *argument-3*, the default is 1 and the range of values that can be specified is 1 to 29 when the value of *argument-2* is 2 and 1 to (number of days in the month specified in *argument-2*) in all other cases.
11. The result is determined according to the following rules:
    - If month of *argument-1* < *argument-2* or month of *argument-1* = *argument-2* and day of *argument-1* < *argument-3*:

      Year of result = year of *argument-1* - 1
    - If month of *argument-1* > *argument-2* or month of *argument-1* = *argument-2* and day of *argument-1* ≥ *argument-3*:

      Year of result = year of *argument-1*
    - If year of result is non-leap year and *argument-2* = 2 and *argument-3* = 29:

      Month of result = 3

      Day of result = 1
    - If the year of the result is a leap year and *argument-2* is not 2, or *argument-3* is not 29:

      Month of result = *argument-2*

      Day of result = *argument-3*
12. If the result of the TRUNCYEAR function is not within the range 0001/01/01 - 9999/12/31, an overflow error will result. For details about the operational results when overflow error suppression is in effect, see *2.18 Operational results with overflow error suppression specified*.

(d) Examples:

Examples of the TRUNCYEAR function are shown below:

```
TRUNCYEAR(DATE('1999-09-10'))          ==> '1999-01-01'
TRUNCYEAR(DATE('1999-09-11'), 4)       ==> '1999-04-01'
TRUNCYEAR(DATE('1999-03-31'), 4)       ==> '1999-04-01'
TRUNCYEAR(DATE('1999-08-11'), 3, 21)   ==> '1999-03-21'
TRUNCYEAR(DATE('1999-03-20'), 3, 21)   ==> '1999-03-21'
TRUNCYEAR(DATE('2000-02-28'), 2, 29)   ==> '1999-03-01'
```

(1999-02-29 is a nonexistent date.)

```
TRUNCYEAR(DATE('2000-03-20'), 2, 29)  ==> '2000-02-29'
TRUNCYEAR(DATE('0001-03-20'), 4)      ==> Overflow
```

(60) WEEK

(a) Function

Returns an integer, in the range 1 to 54, that represents the number of the week since the beginning of the year in which the date specified in an argument falls (assuming that each week begins on Sunday).

(b) Format

`[MASTER.]WEEK(`*argument*`)`

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [`AS` *data-type*]
2. The following can be specified in the value expression:
   - `CURRENT_DATE`
   - Column specification
   - SQL variable or SQL parameter
   - Date operation
   - Set function
   - Scalar function
   - Function call
   - `CASE` expression
   - `CAST` specification
   - Embedded variable or `?` parameter
   - Scalar subquery
3. When you specify only an embedded variable or a `?` parameter in the value expression, you must specify the `AS` data type. When the `AS` data type is specified, items other than an embedded variable or a `?` parameter cannot be specified. Specifying the `AS` data type causes the function that has a parameter with the data type specified in the `AS` clause to be called.
4. The data type of the argument must be `DATE`.
5. The data type of the result will be `INTEGER`.

6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.

(d) Examples

Examples of the WEEK function are shown below:

```
WEEK(DATE('2000-01-01'))    ==> 1
WEEK(DATE('2000-01-02'))    ==> 2
WEEK(DATE('2000-02-29'))    ==> 10
WEEK(DATE('2000-12-30'))    ==> 53
WEEK(DATE('2000-12-31'))    ==> 54
```

2000-01-01 is a Saturday, and 2000-12-31 is a Sunday.

(61) WEEKOFMONTH

(a) Function

Returns an integer, in the range 1 to 6, that represents the number of the week since the beginning of the month in which the date specified in an argument falls.

(b) Format

[MASTER.]WEEKOFMONTH(*argument*)

(c) Rules

1. The following can be specified in the argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in the value expression:
   - CURRENT_DATE
   - Column specification
   - SQL variable or SQL parameter
   - Date operation
   - Set function
   - Scalar function
   - Function call
   - CASE expression
   - CAST specification
   - Embedded variable or ? parameter
   - Scalar subquery
3. When you specify only an embedded variable or a ? parameter in the value

expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.

4. The data type of the argument must be DATE.
5. The data type of the result will be INTEGER.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of the argument is the null value, the result is also the null value.

(d) Examples

Examples of the WEEKOFMONTH function are shown below:

```
WEEKOFMONTH(DATE('2000-01-01'))    ==> 1
WEEKOFMONTH(DATE('2000-01-02'))    ==> 2
WEEKOFMONTH(DATE('2000-01-29'))    ==> 5
WEEKOFMONTH(DATE('2000-01-30'))    ==> 6
```

2000-01-01 is a Saturday, and 2000-01-31 is a Sunday.

(62) YEARS_BETWEEN

(a) Function

Returns as a real number (FLOAT type) the number of years between two dates specified in arguments.

(b) Format

[MASTER.]YEARS_BETWEEN(*argument-1*, *argument-2*)

(c) Rules

1. The following can be specified in each argument:
   - *value-expression* [AS *data-type*]
2. The following can be specified in each value expression:
   - CURRENT DATE
   - Column specification
   - SQL variable or SQL parameter
   - Date operation
   - Set function
   - Scalar function
   - Function call

- CASE expression
- CAST specification
- Embedded variable or ? parameter
- Scalar subquery

3. When you specify only an embedded variable or a ? parameter in a value expression, you must specify the AS data type. When the AS data type is specified, items other than an embedded variable or a ? parameter cannot be specified. Specifying the AS data type causes the function that has a parameter with the data type specified in the AS clause to be called.
4. The data types of *argument-2* and *argument-3* must be DATE.
5. The data type of the result will be FLOAT.
6. The value of the result is not NOT NULL constrained (null values are allowed). If the value expression of *argument-1* or *argument-2* is the null value, the result is also the null value.
7. The number of years which is the result of YEARS_BETWEEN is calculated according to the following rules:

   - If *date-1* < *date-2*:

     Result = -YEARS_BETWEEN (*date-2*, *date-1*)

   - If *date-1* ≥ *date-2*:

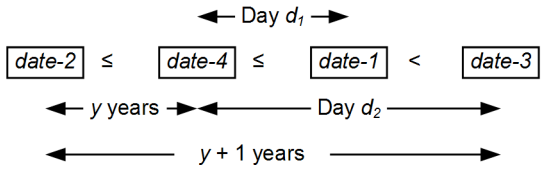
     Let y be an integer greater than or equal to 0, and let *date-4* and *date-3* be the date (*date-2* + y years) and (*date-2* + (y + 1)) years, respectively, such that *date-4* ≤ *date-1* < *date-3*

     Let d_1 and d_2 be the number of days between *date-4* and *date-1* (DAYS (*date-1*) - DAYS (*date-4*)) and the number of days between *date-4* and *date-3* (DAYS (*date-3*) - DAYS (*date-4*)), respectively (where d_2 is one of the following values: 365, 366)

     The number of years in the result will be ($y + d_1 \div d_2$) years.

   The following figure shows the relationships between the dates *date-1*, *date-2*, *date-3*, and *date-4*.

Figure 2-8: Relationships between the dates date-1, date-2, date-3, and date-4



(d) Examples

Examples of the YEARS_BETWEEN function are shown below:

```
YEARS_BETWEEN(DATE('2000-06-10'),DATE('1999-06-10'))==>    1
YEARS_BETWEEN(DATE('2000-06-11'),DATE('1999-06-10'))==>
1.002739...
YEARS_BETWEEN(DATE('1999-06-11'),DATE('1999-06-10'))==>
1.002732...
YEARS_BETWEEN(DATE('2014-09-09'),DATE('1999-06-10'))==>
15.249315...
YEARS_BETWEEN(DATE('1999-06-10'),DATE('1999-06-10'))==>
-15.249315...
```

2.16.3 Plug-in definition scalar functions

(1) XMLQUERY

A scalar function that can be used with HiRDB XML Extension.

For details, see *1.14.3(1) XMLQUERY*.

(2) XMLSERIALIZE

A scalar function that can be used with HiRDB XML Extension.

For details, see *1.14.3(2) XMLSERIALIZE*.

(3) XMLPARSE

A scalar function that can be used with HiRDB XML Extension.

For details, see *1.14.3(3) XMLPARSE*.

2.17 CASE expressions

(1) Function

A CASE expression specifies a value with a condition.

(2) Format

CASE-expression : : = {*searched-CASE-expression* | *simple-CASE-expression* | *CASE-abbreviation*}

searched-CASE-expression : : = CASE
{WHEN *search-condition* THEN {*value-expression* | NULL}}...
[ELSE {*value-expression* | NULL}]
END

simple-CASE-expression : : = CASE *value-expression*
{WHEN *value-condition* THEN {*value-expression* | NULL}}...
[ELSE {*value-expression* | NULL}]
END

CASE-abbreviation : : = {NULLIF (*value-expression*, *value-expression*)
| COALESCE (*value-expression*, [, *value-expression*]...)}

(3) Rules

1. A maximum of 255 WHEN statements can be specified in a single CASE expression.
2. If some of the search conditions in the CASE expression are TRUE, the result value of the first WHEN for which the search condition is TRUE is converted to the CASE expression data type and used as the value of the CASE expression.
3. If none of the search conditions in the CASE expression is TRUE, the result value of the ELSE that is assumed or specified is converted to the CASE expression data type and used as the value of the CASE expression.
4. For search conditions, see *2.7 Search conditions*.
5. The data type and data length of the result of a CASE expression are the same as for the set operation.
6. The data type of the result of a CASE expression is without the NOT NULL constraint (the null value is allowed).
7. The following table lists the character set used for the result if the data type of the result of a CASE expression is the character string data type.

Table 2-85: Character set of the result of a CASE expression

CASE expression		Character set of result
Search CASE expression		Character set of the value expression specified in the first THEN
Simple CASE expression		Character set of the value expression specified in the first THEN
CASE abbreviation	NULLIF	Character set of the first value expression
	COALESCE	Character set of the first value expression

8. For COALESCE, value expressions are evaluated from left to right and the first non-null value is used as the result.
9. A value expression must be specified in at least one THEN in a CASE expression.
10. The ? parameter or an embedded variable cannot be specified alone in the value expression in CASE, THEN, or ELSE (including specification in monomial operational expressions).
11. If ELSE is not specified, ELSE NULL is assumed.
12. Values with any of the following data types cannot be specified in a CASE value expression, WHEN value expression, or NULLIF and COALESCE value expressions:
    - BLOB
    - BINARY with a minimum defined length of 32,001 bytes
    - BOOLEAN
    - Abstract data type
13. Values with the following data type cannot be specified in THEN or ELSE value expressions:
    - BLOB
    - BINARY with a minimum defined length of 32,001 bytes
    - BOOLEAN
    - Abstract data type
14. If the value expression specified in THEN or ELSE is a character string data type, use the same character set for all value expressions. However, if the value expression specified in a second or later THEN or ELSE is the one listed below, the value expression is converted to the character set of the value expression specified in the first THEN.
    - Character string literal
15. Values with the following data type cannot be specified in THEN or ELSE value

expressions:

- BLOB
- BINARY with a minimum defined length of 32,001 bytes
- BOOLEAN
- Abstract data type

16. A simple CASE expression is the same as a searched CASE expression for which V2=V1 is specified as a search condition, where V1 is a value expression in WHEN and V2 is a value expression in CASE.
17. The ? parameter or an embedded variable cannot be specified alone in the value expression in the first WHEN in a simple CASE specification, the first value expression in COALESCE, or both value expressions in NULLIF (including specification in monomial operational expressions).
18. If the value expression in at least one WHEN in a simple CASE expression is the ? parameter or an embedded variable, the data type of the ? parameter or embedded variable is assumed to be the same as that of the value expression in the first WHEN.
19. If one of the value expressions in NULLIF is the ? parameter or an embedded variable, the data type of the ? parameter or embedded variable is assumed to be the same as that of the other value expression.
20. If at least one value expression in COALESCE is the ? parameter or an embedded variable, the data type of the ? parameter or embedded variable is assumed to be the same as that of the first value expression.
21. A maximum of 255 value expressions can be specified in COALESCE.
22. NULLIF(V1, V2) is the same as the following CASE expression:
```
CASE WHEN V1=V2 THEN NULL ELSE V1 END
```
23. COALESCE(V1, V2) is the same as the following scalar function VALUE or CASE expression:
```
VALUE(V1, V2)
CASE WHEN V1 IS NOT NULL THEN V1 ELSE V2 END
```
24. COALESCE(V1, V2, ..., V*n*), where *n* is at least 3, is the same as the following scalar function VALUE or CASE expression:
```
VALUE(V1, V2, ..., Vn)
CASE WHEN V1 IS NOT NULL THEN V1 ELSE COALESCE (V2, ..., Vn)
END
```
25. Comparable data types must be used for the THEN and ELSE value expressions in a searched CASE expression or a simple CASE expression. For the comparable data types, see *1.2 Data types*. The following data types cannot be compared:

- Date data and its character string representation
- Time data and its character string representation
- Time stamp data and the character string representation of time stamp data
- Date interval data and its decimal representation
- Time interval data and its decimal representation
- Binary data and hexadecimal character string literals

26. For a simple `CASE` expression, comparable data types must be used for the value expressions of `WHEN` and `CASE`. For the comparable data types, see *1.2 Data types*. The following data types cannot be compared:
    - Date data and its character string representation
    - Time data and its character string representation
    - Time stamp data and the character string representation of time stamp data
    - Date interval data and its decimal representation
    - Time interval data and its decimal representation
    - Binary data and hexadecimal character string literals
27. In a simple `CASE` expression, if the value expressions of `CASE` and `WHEN` are character string types, use the same character set for the value expressions of `CASE` and `WHEN`. However, if the value expression of `WHEN` is the value expression listed below, it is converted to the character set used for the `CASE` value expression before it is compared.
    - Character string literal
28. For `NULLIF` or `COALESCE`, comparable data types must be used for their value expressions. For the comparable data types, see *1.2 Data types*. The following data types cannot be compared:
    - Date data and its character string representation
    - Time data and its character string representation
    - Date interval data and its decimal representation
    - Time interval data and its decimal representation
29. If both value expressions in `NULLIF` are a character string data type, use the same character set for the two value expressions. However, if either of the value expressions is the one listed below, it is converted to the character set used for the other value expression:
    - Character string literal

30. If both value expressions in COALESCE are character string data types, use the same character set for the two value expressions. However, if the second value expression is the one listed below, it is converted to the character set used for the first value expression:
    - Character string literal
31. Specifying a subscript, repetition columns can be specified in a CASE expression. Also, ANY as a subscript can be specified for a repetition column as a search condition in a CASE expression. Unsubscripted repetition columns can be specified in the IS NULL predicate (at the same locations as in a search condition). ANY as a subscript cannot be specified for a repetition column in a CASE expression that is specified in a selection expression.
32. A window function cannot be specified.

(4) Examples

(a) Searched CASE

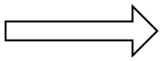
In table T1, change AA and BB in column C1 to AAA and BBB, respectively:

```
UPDATE T1 SET C1 =
   CASE WHEN C1='AA' THEN 'AAA' WHEN C1='BB' THEN 'BBB'
   ELSE C1
END
```

Table name: T1

Column C1

'AA'
'BB'
'BB'
'CC'



Execution result

Column C1

'AAA'
'BBB'
'BBB'
'CC'

(b) Simple CASE

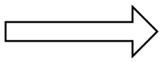
In table T1, change AA and BB in column C1 to AAA and BBB, respectively:

```
UPDATE T1 SET C1 =
   CASE C1 WHEN 'AA' THEN 'AAA' WHEN 'BB' THEN 'BBB'
   ELSE C1
END
```

Table name: T1

Column C1

C1
'AA'
'BB'
'BB'
'CC'



Execution result

Column C1

C1
'AAA'
'BBB'
'BBB'
'CC'

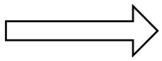
(c) CASE abbreviation (COALESCE)

Extract columns that do not contain all null values from table T1, in the order of columns C1, C2, and C3; if all columns contain only null values, set 0 as the result:

```
SELECT COALESCE(C1,C2,C3,0) FROM T1
```

Table name: T1

Column C1	Column C2	Column C3
*	20	*
*	*	*



Execution result

20
0

Note: * indicates the null value.

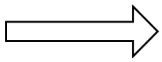
(d) CASE abbreviation (NULLIF)

If the values in columns C1 and C2 of table T1 are equal, the null value is returned as the result; if the values in columns C1 and C2 of table T1 are not equal, column C1 is extracted:

```
SELECT NULLIF(C1,C2) FROM T1
```

Table name: T1

Column C1	Column C2
10	20
20	20



Execution result

10
*

Note: * indicates the null value.

2.18 Operational results with overflow error suppression specified

When the overflow error suppression feature is set by the `pd_overflow_suppress` option of the system common definition, the following conditions do not generate an error (HiRDB treats the result of the operation that generates the overflow as the null value and continues processing):

- 0 is specified as the value of the second operand for division
- Overflow occurs during computation

The following types of overflows are subject to overflow error suppression:

- Numeric-type overflow
- Division-by-zero error
- Date data type overflow
- Time data type overflow
- Date interval data type overflow
- Time interval data type overflow
- Labeled duration overflow
- Overflow in the `SUM`, `COUNT`, `COUNT_FLOAT`, or `AVG` set function
- Window function overflow
- Scalar function overflow (the following table lists the scalar functions subject to overflow error suppression)

Table 2-86: Scalar functions subject to overflow error suppression

Classification	Scalar function	Scalar function type
Conversion function	`DEGREES`	System-defined scalar function
	`NUMEDIT`	
	`STRTONUM`	

Classification	Scalar function	Scalar function type
Mathematical function	ABS	System-built-in scalar function
	MOD	
	CEIL	System-defined scalar function
	COSH	
	EXP	
	FLOOR	
	POWER	
	ROUND	
	SINH	
	TAN	
	TANH	
	TRUNCYEAR	
Date manipulation	ADD_INTERVAL	
	NEXT_DAY	
	ROUNDMONTH	

An error does occur in the following cases even when the overflow error suppression feature is in effect:

- Attempt is made to receive data into a UAP without specifying an indicator variable
- Attempt is made to update a NOT NULL constrained column with the null value or to insert the null value into a NOT NULL constrained column when the null value is set by means of the overflow error suppression feature

When overflow occurs, a 'W' warning is set to SQLWARNB in the SQL Communications Area. Whether or not overflow has occurred can be determined by referencing SQLWARNB.

Shown below is an example of the processing that occurs when the overflow error suppression feature is in effect. The examples are based on the following stock control table (CONTROL):

Table name: Stock Control Table (CONTROL)

PCODE (Product Code)	PRICE (Unit price)	QUANTITY (Quantity)	TOTAL (Total)
A-200	200	1000	Null value
A-280	280	500	Null value
B-300	300	80000	Null value
B-350	350	60000	Null value
B-380	380	50000	Null value

Note: The "Unit price" (PRICE), "Quantity" (QUANTITY), and "Total" (TOTAL) columns must be of the INTEGER data type; the "Total" (TOTAL) column must be a NOT NULL constrained column.

2.18.1 Example of overflow in a search condition

Retrieve from the stock control table (CONTROL) the rows for which the product of the unit price (PRICE) and the quantity (QUANTITY) is greater than 15,000,000:

(1) SQL

```
SELECT PCODE, PRICE, QUANTITY
  FROM CONTROL
  WHERE PRICE * QUANTITY > 15000000
```

Table name: Stock Control Table (CONTROL)

PCODE (Product Code)	PRICE (Unit price)	QUANTITY (Quantity)	
A-200	200	1000	→ TRUE
A-280	280	500	→ FALSE
B-300	300	80000	→ Overflow
B-350	350	60000	→ TRUE
B-380	380	50000	→ TRUE

(2) Retrieval results

PCODE (Product Code)	PRICE (Unit price)	QUANTITY (Quantity)
A-200	200	1000
B-350	350	60000
B-380	380	50000

If the overflow error suppression feature is not in effect, overflow occurs in the product of the unit price and the quantity (PRICE * QUANTITY) for the row with B-300 as the product code (PCODE). Processing is cancelled when the overflow occurs.

Even with the overflow error suppression feature in effect, overflow occurs for the row with B-300 as the product code (PCODE), but processing continues.

The result of an operation in which overflow occurs is the null value. Because the null value does not satisfy the search condition, the row that contains the null value is not retrieved.

2.18.2 Example of overflow in an update value

Update in the stock control table (CONTROL), the total (TOTAL) column using the product of the unit price (PRICE) and the quantity (QUANTITY):

(1) SQL

```
UPDATE CONTROL
   SET TOTAL = PRICE * QUANTITY
```

(2) Update results (with overflow error suppression feature specified)

Table name: Stock Control Table (CONTROL)

PCODE (Product Code)	PRICE (Unit price)	QUANTITY (Quantity)	TOTAL (Total)
A-200	200	1000	200000
A-280	280	500	140000
B-300	300	80000	Null value
B-350	350	60000	21000000
B-380	380	50000	19000000

If the overflow error suppression feature is not in effect, overflow occurs in the product of the unit price and the quantity (PRICE * QUANTITY) for the row with B-300 as the product code (PCODE). Processing is cancelled when the overflow occurs.

Even with the overflow error suppression feature in effect, overflow occurs for the row

with B-300 as the product code (PCODE), but processing continues.

The result of an operation in which overflow occurs is the null value, so the total column (TOTAL) is updated with the null value. If a table is updated while the overflow error suppression feature is in effect, the table is updated with the null value. This fact should be taken into consideration when a table is to be updated.

2.19 Lock option

(1) Function

The lock option provides exclusive controls for retrieving data.

The lock option is specified in the cursor declaration, the cursor specification in a FOR statement, the single-row SELECT statement, or the dynamic SELECT statement.

(2) Format

lock-option::=[{WITH {SHARE|EXCLUSIVE} LOCK
|WITHOUT LOCK [{WAIT|NOWAIT}]}]
[{WITH ROLLBACK|NO WAIT}]

(3) Operands

[{WITH {SHARE|EXCLUSIVE} LOCK|WITHOUT LOCK [{WAIT|NOWAIT}]}]

However, the default for this operand is WITH EXCLUSIVE LOCK if a FOR UPDATE clause is specified in the cursor declaration or in the dynamic SELECT statement, or the row using the cursor is updated or deleted.

WITH SHARE LOCK

Specifies that the contents of retrieved data may be referenced but may not be updated by other users (shared mode) until the current transaction has terminated.

WITH EXCLUSIVE LOCK

Specifies that the contents of retrieved data are not to be referenced (except for referencing by WITHOUT LOCK NOWAIT) or updated (exclusive mode) until the current transaction has terminated.

The WITH EXCLUSIVE LOCK option is effective only on the table specified in the FROM clause of a query specification.

Even when the WITH EXCLUSIVE LOCK option is specified, the shared-mode exclusive use option is applied to the key values of an index only when the index is referenced, and the exclusive use option is reset upon completion of the referencing operation. This permits other users to access the same index. However, if the index that is used for retrieval purposes is updated by another user, the user who needs to use the index for retrieval purposes may have to wait. The LOCK statement ensures that any processing of the index by another user must wait until the current transaction has been completed.

WITHOUT LOCK [WAIT]

Specifies that the contents of retrieved data are not to be subject to exclusive control until the current transaction terminates. If a search is performed by an SQL statement that has WITHOUT LOCK [WAIT] specified, locking of the referenced locked resource (row or page) is canceled.

Specifying WITHOUT LOCK [WAIT] causes HiRDB to reset the lock upon completion of the retrieval of a row without waiting for termination of the current transaction, thus reducing lock resource requirements. This technique minimizes the resources required to perform exclusive control and improves HiRDB's ability to process transactions concurrently. It should be noted, however, that the WITHOUT LOCK [WAIT] option can create a situation in which the same row retrieved twice during the same transaction provides different results.

WITHOUT LOCK NOWAIT

Specifies that data being updated by another user (or data subject to the exclusive lock option) can be referenced before the updating operation is completed and that the contents of retrieved data are not to be subject to exclusive lock until the current transaction has been completed. However, if the table to be searched is a shared table and if the LOCK statement is being executed in the lock mode by another user, the table is lock-release pending, unavailable for referencing.

In a retrieval performed by an SQL statement with WITHOUT LOCK NOWAIT specified, resources can be referenced in the same manner as without a lock (except that a logical file cannot be referenced), even when other transactions are applying the lock option to the tables and rows in the EX mode. However, tables that are being accessed by the pdload or pdorg command cannot be referenced.

Note that when WITHOUT LOCK NOWAIT is specified, the same data cannot be received again when the same row is retrieved twice in the same transaction. Moreover, if a table updating transaction is cancelled by one user, another user may receive data that did not exist when the table was retrieved while it was being updated.

If the WITHOUT LOCK NOWAIT option is specified in a cursor declaration or in a dynamic SELECT statement, a FOR UPDATE clause cannot be specified in the cursor declaration or dynamic SELECT statement; similarly, the WITHOUT LOCK NOWAIT option specified in the cursor declaration conflicts with the UPDATE or DELETE operation that uses the cursor.

[{WITH ROLLBACK|NO WAIT}]

If this operand is omitted and the data to be searched is being used by another user, the data goes into the wait state (exclusive of WITHOUT LOCK NOWAIT) until the pending user transaction terminates.

WITH ROLLBACK

Specifies if the data to be searched is being used by another user and if the pending transaction is to be cancelled and nullified (any SQL statements that were executed before the error occurred will be rolled back).

NO WAIT

Specifies when the data to be searched is being used by another user and the search is to be flagged as an error (lock error) without canceling the transaction (SQL statements that were executed before the error occurred will not be rolled back).

In this case, any lock applied by the specified SQL statement is not unlocked. Because the cursor remains open, it should be closed by executing the CLOSE statement in the UAP.

However, if a lock error occurs under any of the following circumstances, the transaction will be canceled and invalidated even when the NO WAIT option is specified:

- With respect to a table that is subject to a retrieval by a subquery
- A derived table that is the object of search by the FROM clause
- With respect to a table that is subject to a retrieval by a query in which a value expression is specified in the GROUP BY clause
- See *2.21 Inner derived tables* for details about in-table derived tables that are subject to a retrieval by a query that satisfies one of the conditions that result in an "inner derived table."

(4) Notes

Because WITHOUT LOCK NOWAIT has a different meaning from WITHOUT LOCK NO WAIT, these options must be specified with care.

(5) Specification example (lock option is specified in the SELECT statement)

```
SELECT PCODE FROM STOCK
  WHERE PNAME = N'blouse'
  WITH SHARE LOCK
  WITH ROLLBACK
```

2.20 Function calls

(1) Function

A function call calls a specified function.

(2) Format

function-call::=[*authorization-identifier*.] *routine-identifier*
([*argument* [, *argument*]...])
argument::=*value-expression* [AS *data-type*]

(3) Operands

authorization-identifier

Specifies the authorization identifier of the function to be called.

When calling a public function, specify PUBLIC, in all caps and enclosed in double quotation marks ("), in the authorization identifier.

routine-identifier

Specifies the routine identifier of the function to be called.

argument::= *value-expression* [AS *data-type*]

value-expression

Specifies a value expression for a parameter of the function to be called.

AS *data-type*

Specifies the ? parameter or a predefined data type for the embedded variable for a parameter of the function to be called.

(4) Rules

1. Arguments are associated with parameters in the order in which they are specified.
2. The data type of an argument must be compatible with the data type of the corresponding parameter. For compatible data types, see *1.2.2 Data types that can be converted (assigned or compared)*. The following combinations of data types are incompatible:
   - Character data that uses a different character set
   - Character data and mixed character data
   - Date interval data and literals that express a date interval in decimal
   - Date interval data and embedded variables that are associated with

DECIMAL(8,0)

- Time interval data and literals that express a time interval in decimal
- Time interval data and embedded variables that are associated with DECIMAL(6,0)
- Date data and literals that express a date as a character string
- Date data and embedded variables that are associated with CHAR(10)
- Date data and embedded variables that are associated with CHAR(20) when the UTF-16 character set is used
- Date data and CHAR or VARCHAR embedded variables with a length of at least 10 bytes
- Date data and CHAR or VARCHAR embedded variables that use the UTF-16 character set with a length of at least 20 bytes and that are divisible by two
- Time data and literals that express a time as a character string
- Time data and embedded variables that are associated with DECIMAL(6,0)
- Time data and CHAR or VARCHAR embedded variables with a length of at least eight bytes
- Time data and CHAR or VARCHAR embedded variables that use the UTF-16 character set with a length of at least 16 bytes and that are divisible by 2
- Time stamp data and literals representing time stamps in character strings
- Embedded variable of CHAR or VARCHAR data type greater than or equal to 19 bytes or greater if the fractional second precision of the time stamp data is 0, or 22 bytes or greater if the latter is 2 or greater.
- Embedded variable of the CHAR or VARCHAR data type that use the UTF-16 character set, are greater than or equal to 38 bytes in length, and that are divisible by 2 if the fractional second precision of the time stamp data is 0, or that are 44 bytes or greater in length and that are divisible by 2 if the precision of the time stamp data is 2 or greater.
- BINARY type and hexadecimal character string literals

In addition, the data type of an argument must have a priority greater than or equal to the data type of the parameter. For the priorities of data types, see *Table 2-88* and *Table 2-89*.

3. When only the ? parameter or an embedded variable is specified in a value expression, the AS data type must be specified in order to determine the data type for the value expression.
4. When the AS data type is specified, items other than the ? parameter or an

embedded variable cannot be specified in a value expression.

5. Unary operations using either the ? parameter or an embedded variable cannot be specified in a value expression.
6. When a repetition column is specified in an argument, a subscript must be specified. However, ANY cannot be specified as the subscript.
7. If only the ? parameter or an embedded variable is specified in a value expression, the ? parameter or an embedded variable must be a simple structure.
8. The maximum number of nesting levels for a function call is 255. The nesting level for a function call is equal to the nesting level specified in the parentheses in "*routine-identifier* (".
9. In *argument*, the SUBSTR scalar function producing either BLOB as the data type of the result or the BINARY type with a minimum length of 32,001 bytes cannot be specified as a single value expression.
10. The window function cannot be specified.

(5) Notes

For the default values for an authorization identifier, see *1.1.9 Schema path*.

(6) Rules for determining the function to be called and the data type of the result

1. The specified function is called only if the number of authorization identifiers, the number of routine identifiers, and the number of arguments are all the same, the data types of the arguments do not include the abstract data type, and the data types of parameters are in complete agreement with the order in which arguments are listed. In such a case, the data type of the result of the function will be the data type of the RETURNS clause of the function being called.
2. If there is any disagreement in the numbers of authorization identifiers, routine identifiers, and arguments, the specified function is not called.
3. The following table lists the rules for determining the function to be called.

Table 2-87: Rules for determining the specified contents of the function call and the function to be called

<table>
<tr><th colspan="3">Specified contents of the function call</th><th>Rules for determining the function to be called</th></tr>
<tr><td rowspan="4">Number of authorization identifiers, routine identifiers, and arguments is the same.</td><td colspan="2">Number of arguments is 0.</td><td rowspan="2">Calls the function to make the data types the same for authorization identifiers, routine identifiers, and arguments.</td></tr>
<tr><td rowspan="2">Does not include abstract data types in the arguments.</td><td>Data types of the parameters are in complete agreement with the order in which arguments are listed.#</td></tr>
<tr><td>Data types of the parameters are not in agreement with the order in which arguments are listed.</td><td>Follows the rules described in When an abstract data type is not included, below.</td></tr>
<tr><td colspan="2">Includes abstract data type in the arguments.</td><td>Follows the rules described in When an abstract data type is included, below.</td></tr>
<tr><td colspan="3">Number of authorization identifiers, routine identifiers, and arguments is not the same.</td><td>No function can be called, so an error occurs when the SQL statement is analyzed.</td></tr>
</table>

#

If the data type of the arguments is a character string type, *data types are in agreement* means that the same character sets are being used.

- When an abstract data type is not included:

  Based upon the predefined data type of each argument as a standard, beginning with the leftmost argument, either the function whose priority is the same as the standard priority or the function whose parameter has a predefined data type that has the next highest data type priority is called. The table below indicates the priorities of predefined data types. Because the function to be called is determined uniquely during SQL analysis, the data type of the result of the function will be the data type of the `RETURNS` clause of the function being called.

Table 2-88: Priorities of predefined data types

Data type of argument	Priority order
Numeric data	`SMALLINT` → `INTEGER` → `DECIMAL` → `SMALLFLT` → `FLOAT`
Character data	`CHAR` → `VARCHAR`

Data type of argument	Priority order
National character data	NCHAR → NVARCHAR
Mixed character data	MCHAR → MVARCHAR
Large-object data and binary data	BINARY → BLOB

A → B: Priority of A is higher than priority of B.

- When an abstract data type is included:

  If the data types of arguments include the abstract data type, the function to be called is determined as follows:

4. Determine the base function.

   The base function is determined as follows: Based upon the predefined data type of each argument as a standard, beginning with the leftmost argument, either the function that has a priority equal to the standard priority or the function whose parameter has a predefined data type that has the next highest data type priority is designated as the base function. If the data type of a given function is a predefined data type, the priority listed in *Table 2-88* is used. If the data type is an abstract data type, the priority listed in the following table is used.

   Table 2-89: Priorities of abstract data types

Data type of argument	Priority order
Abstract data type	Same data type → Super type#

A → B: Priority of A is higher than priority of B.

#: The super type that is specified directly in the UNDER clause in the definition of an abstract data type has higher priority than any other super type.

5. Determine candidate functions.

   If the argument is an abstract data type, the data type of the actual value that the argument can take is either the data type of the abstract data type in which the argument is defined or its subtype. Therefore, in addition to the base function, all functions that have the same data type as the abstract data type of the argument or are associated with parameters with the abstract data type of the subtype will be candidate functions.

   If only one candidate function (that is, the base function) is found, that function

is called. The data type of the result of the function will be the data type of the RETURNS clause of the function being called.

6. Narrow candidate functions by using the data type of the RETURNS clause.

   If the argument includes an abstract data type, the data type of the RETURNS clause of the base function and the data type of the RETURNS clause of candidate functions other than the base function is checked for compatibility. Functions that have an incompatible RETURNS clause data type are eliminated as candidates. After the compatibility check, the data type of the result of the functions is determined based on the data type of the RETURNS clause of the remaining candidate functions.

   After the compatibility check, the data type of the result of the functions is determined based on the data type of the RETURNS clause of the remaining candidate functions.

   If, after the compatibility check, the data type of the RETURNS clause of the remaining candidate functions is an abstract data type, the abstract data type of the RETURNS clause of the base function is used as the data type of the result.

   If, after the compatibility check, the data type of the RETURNS clause in the remaining candidate functions includes a BINARY or BLOB type function that is 32,001 bytes or longer, the following rules apply.

   - The data length of the result is the longest data length.
   - If the BINARY and BLOB types occur on a mixed basis, the result is the BLOB type.

   Otherwise, the data type and length of the result are the same as the data type and length of the result of the set operation (UNION ALL or EXCEPT ALL). For details, see *2.2 Query expressions*.

7. Determine the function to be used during execution of an SQL statement.

   If steps 2 and 3 above fail to determine uniquely the function to be used, the function to be called from the candidate functions is determined uniquely according to the actual data type of the argument of an abstract data type during execution of an SQL statement. Beginning with the leftmost argument, if the actual value of an argument is NOT NULL, the data type of that value is used as a base. If it is NULL, the data type of that argument is used as a base, and the function that has as a parameter a data type that is equal in priority to the base data type or that is highest in priority among the candidate functions that are lower in priority than the base data type is chosen and designated as the function to be called.

Examples

Determine the function to be called when an abstract data type is included

Let A, B, and C denote abstract data types, where C is the super type of B and B is the

super type of A (priority of abstract data types: A → B → C).

Example 1:

Preconditions

Table definition:
```
CREATE TABLE T1(C1 C)
```

Function definition:
```
f(A), f(B), f(C)
```

SQL statement:
```
SELECT f(C1) FROM T1
```

Results

Base function:
```
f(C)
```

Candidate functions for which the function call is `f(C1)`:
```
f(A),f(B),f(C)
```

Function to be called

The following functions are called during execution of the SQL statement:

Actual value of T1.C1	Function to be called
Type A	`f(A)`
Type B	`f(B)`
Type C	`f(C)`
`NULL` value	`f(C)`

Example 2:

Preconditions

Table definition:
```
CREATE TABLE T1(C1 C,C2 B)
```

Function definition:
```
f(A,A), f(A,B), f(A,C), f(B,A), f(B,C), f(C,A), f(C,B),
f(C,C)
```

SQL statement:
```
SELECT f(C1,C2) FROM T1
```

Results

Base function:
`f(C,B)`

Candidate functions for which the function call is `f(C1,C2)`:
`f(A,A),f(A,B),f(A,C),f(B,A),f(B,C),f(C,A),f(C,B)`

Function to be called

The following functions are called during execution of the SQL statement:

Actual value of T1.C1	Actual value of T1.C2	Function to be called
Type A	Type A	`f(A,A)`
	Type B	`f(A,B)`
	`NULL` value	`f(A,B)`
Type B	Type A	`f(B,A)`
	Type B	`f(B,C)`
	`NULL` value	`f(B,C)`
Type C	Type A	`f(C,A)`
	Type B	`f(C,B)`
	`NULL` value	`f(C,B)`
`NULL` value	Type A	`f(C,A)`
	Type B	`f(C,B)`
	`NULL` value	`f(C,B)`

2.21 Inner derived tables

View tables and the query names of the WITH clause are called named derived tables.

When a named derived table is specified in a query specification, an internal work table may be created for the named derived table. A named derived table for which a work table is created is called an inner derived table.

2.21.1 Conditions for an inner derived table

If any of the named derived tables shown in (1)-(10) is specified in the FROM clause of a query specification, an inner derived table can be created under the following conditions:

(1) Named derived table that is derived by specifying SELECT DISTINCT

A named derived table is included in the subquery. Alternatively, the query specification, specifying a named derived table in a FROM clause, directly contains one of the following items:

- GROUP BY clause, HAVING clause, or a set function
- SELECT DISTINCT
- Table joining (including outer joins and inner joins)
- A value expression other than a column specification is specified in the selection expression.
- A scalar subquery is specified in the selection expression.
- The selection expression does not specify all the columns, column by column, in the view table specified in the FROM clause.
- ORDER BY clause
- NEXT VALUE expression

Examples

If named derived table V1 (a view table) and Q1 (the query name in the WITH clause) that are derived by SELECT DISTINCT C1,C2 FROM T1 are specified in the FROM clause, the following coding creates an internally derived table with respect to V1:

Example 1:

```
SELECT * FROM T2 WHERE EXISTS(SELECT * FROM V1)
```

Example 2:

```
SELECT VC1,VC2 FROM V1 GROUP BY VC1,VC2
```

Example 3:

```
WITH Q1(QC1,QC2) AS (SELECT DISTINCT C1,C2 FROM T1)
   SELECT DISTINCT * FROM V1
```

Example 4:

```
SELECT X.VC1,Y.C1 FROM V1 X,T2 Y WHERE X.VC1=Y.C2
```

Example 5:

```
SELECT V1.VC1,T2.C2 FROM V1 LEFT JOIN T2 ON T2.C2=V1.VC2
```

Example 6:

```
WITH Q1(QC1,QC2) AS (SELECT DISTINCT C1,C2 FROM T1)
   SELECT QC1+100,CURRENT_DATE FROM Q1
```

Example 7:

```
SELECT VC1,(SELECT C1 FROM T2) FROM V1
```

Example 8:

```
SELECT VC1 FROM V1
```

Example 9:

```
SELECT VC1,VC2 FROM V1 ORDER BY 1
```

Example 10:

```
INSERT INTO T2 SELECT NEXT VALUE FOR SEQ1,VC1,VC2 FROM V1
```

(2) Named derived table that is derived by specifying a GROUP BY clause, HAVING clause, or set function

The query specification specifying a named derived table in a FROM clause directly contains one of the following items:

- GROUP BY clause, HAVING clause, or a set function
- Table joining (including outer joins and inner joins)
- Window function
- NEXT VALUE expression

Examples

If named derived table V1 (a view table) and Q1 (the query name in the WITH clause) that are derived by SELECT C1,C2 FROM T1 GROUP BY C1,C2 and named derived table V2 (a view table) that is derived by SELECT MAX(C1),C2 FROM T1 GROUP BY C2 HAVING C2<100 are specified in the FROM clause, the following coding creates an internally derived table with respect to V1, V2 and Q2:

Example 1:

```
WITH Q1(QC1,QC2) AS (SELECT C1,C2 FROM T1
```

```
      GROUP BY C1,C2) SELECT AVG(QC1),QC2 FROM Q1
         GROUP BY QC2
```

Example 2:

```
SELECT V1.VC1,V2.VC1 FROM V1,V2 WHERE V1.VC1=V2.VC1
```

Example 3:

```
WITH Q1(QC1,QC2) AS (SELECT C1,C2 FROM T1
  GROUP BY C1,C2)
  SELECT Q1.QC1,V1.VC1 FROM Q1 INNER JOIN V1 ON
         V1.VC2=Q1.QC2
```

Example 4:

```
SELECT COUNT(*) OVER(),C1 FROM V1
```

Example 5:

```
INSERT INTO T2 SELECT NEXT VALUE FOR SEQ1,VC1,VC2 FROM V1
```

(3) Named derived table that is derived by specifying a value expression other than a column specification in a selection expression

A query specification specifying a named derived table in a FROM clause directly contains one of the following items:

- GROUP BY clause, HAVING clause, or a set function#
- Window function
- Outer or inner join

#

Excludes cases in which the rapid grouping facility is enabled. However, the rapid grouping facility does not apply to a named derived table that is derived by specifying a component specification in a selection expression. For details about the rapid grouping facility, see the manual *HiRDB Version 9 UAP Development Guide*.

Examples

If named derived table V1 (a view table) and Q1 (the query name in the WITH clause) that are derived by SELECT C1+100, C2||C2 FROM T1 are specified in the FROM clause, the following coding creates an internally derived table with respect to V1 and Q1:

Example 1:

```
SELECT AVG(VC1),VC2 FROM V1 GROUP BY VC2
```

Example 2:
```
SELECT * FROM V1 LEFT JOIN T2 ON T2.C2=V1.VC2
```

Example 3:
```
WITH Q1(QC1,QC2) AS (SELECT C1+100,C2| |C2 FROM T1)
    SELECT QC1,QC2 FROM Q1 GROUP BY QC1,QC2
           HAVING QC1<=100
```

Example 4:
```
SELECT COUNT(*) OVER(),VC1 FROM V1
```

(4) Named derived table that is derived by specifying a set function specifying a DISTINCT specification

The query specification, specifying a named derived table in a FROM clause, directly contains one of the following items:

- SELECT DISTINCT
- GROUP BY clause, HAVING clause, or set function
- Joined table (including inner or outer joins)
- Window function

Example

If named derived table V1 (a view table) that is derived by SELECT AVG (DISTINCT C1) FROM T1 is specified in the FROM clause, the following coding creates an internally derived table with respect to V1:
```
SELECT DISTINCT VC1 FROM V1
WITH Q1(C1) AS (SELECT AVG(DISTINCT C1) FROM T1)
  SELECT COUNT(*) OVER(),C1 FROM Q1
```

(5) Named derived table that is derived by specifying a joined table

A named derived table is specified in an inner or outer join table reference.

Examples

If named derived table V1 (a view table) and Q1 (the query name in the WITH clause) that are derived by SELECT T1.C1,T2.C1 FROM T1,T2 are specified in the FROM clause, the following coding creates an internally derived table with respect to V1 and Q1:

Example 1:
```
SELECT V1.* FROM V1 LEFT JOIN T3 ON T3.C1=V1.VC1
```

Example 2:
```
WITH Q1(QC1,QC2) AS (SELECT T1.C1,T2.C1 FROM T1,T2
```

```
SELECT * FROM Q1 INNER JOIN T3 ON Q1.QC1=T3.C1
```

(6) Named derived table that is derived by specifying an inner or outer join

The query specification specifying a named derived table in a FROM clause directly contains a table join.

Examples

In the specification of named derived tables V1 (a view table) and Q1 (a query name) derived by SELECT T1.C1,T2.C1 FROM T1 LEFT JOIN T2 ON T1.C2=T2.C2 in a FROM clause, the following coding creates an inner derived table with respect to V1 and Q1:

Example 1:

```
SELECT V1.VC1,T3.C1 FROM V1 LEFT JOIN T3 ON T3.C2=V1.VC2
```

Example 2:

```
WITH Q1(QC1,QC2) AS (SELECT T1.C1,T2.C1 FROM T1 LEFT JOIN T2
   ON T1.C2=T2.C2) SELECT Q1.QC1 FROM Q1 INNER JOIN T3 ON
T3.C2=Q1.QC2
```

(7) Named derived table that is derived by specifying a value expression, containing a subquery, in a selection expression

A query specification specifying a named derived table in a FROM clause directly contains one of the following items:

- SELECT DISTINCT
- GROUP BY clause, HAVING clause, or a set function
- Joined table (including inner or outer joins)
- A value expression, other than a column specification, in a selection expression
- A scalar subquery specification in a selection expression
- Same column derived from a value expression containing a subquery specified two or more times in the selection expression of a named derived table
- A column derived from a value expression containing a subquery specified in the selection expression of a named derived table as a column that references outside
- View table defined before version 07-02

Examples

In the specification of named derived tables V1 (a view table) and Q1 (a query name) derived by SELECT (SELECT C1 FROM T2),C1 FROM T1 in a FROM clause, the following coding creates an inner derived table with respect to V1 and

```
Q1:
```

Example 1:

```
SELECT VC1,VC2 FROM V1 WHERE VC1>0
```

Example 2:

```
WITH Q1(QC1,QC2) AS (SELECT (SELECT C1 FROM T2),C1 FROM T1)
  SELECT QC1,QC2 FROM Q1 WHERE QC1>0
```

(8) Named derived table that is derived by set operations

One of the following conditions must be satisfied:

1. One of the operands in the set operation directly contains one of the following items:
   - A query on an inner derived table
   - A query specifying a derived table
   - A scalar subquery in a selection expression
2. One of the operands in the set operation and a query on the named derived table satisfy one of the conditions listed in (1)-(8).
3. Condition (9) or (10) is satisfied depending on whether a specified set operation contains options other than `UNION ALL` or it contains the option `UNION ALL` only.

Example

In the specification of named derived tables `V1` (a view table) and `Q1` (a query name) derived by `SELECT (SELECT C1 FROM T2),C2 FROM T1 UNION SELECT C1,C2 FROM T3` in a `FROM` clause, the following coding creates an inner derived table with respect to `V1` and `Q1`:

Example 1:

```
SELECT * FROM V1
```

Example 2:

```
WITH Q1(QC1,QC2) AS (
  SELECT (SELECT C1 FROM T2),C2 FROM T1 UNION SELECT C1,C2
FROM T3)
  SELECT * FROM Q1
```

(9) Named derived table that is derived by a set operation containing options other than UNION ALL

One of the following conditions must be satisfied:

1. The query specification specifying a named derived table in the `FROM` clause directly contains one of the following items:

- GROUP BY clause, HAVING clause, or a set function
- SELECT DISTINCT
- Table joining (including inner and outer joins)
- WHERE clause
- Subquery
- A value expression, other than a column specification, in a selection expression
- Not all columns in the named derived table are specified once per selection expression
- NEXT VALUE expression

2. The condition *(8) Named derived table that is derived by set operations* is satisfied.

Examples

In the specification of named derived tables V1 (a view table) and Q1 (a query name) derived by SELECT C1,C2 FROM T1 UNION SELECT C1,C2 FROM T2 in a FROM clause, the following coding creates an inner derived table with respect to V1 and Q1:

Example 1:

```
SELECT C1,C2 FROM V1 GROUP BY C1,C2
```

Example 2:

```
WITH Q1(QC1,QC2) AS (SELECT C1,C2 FROM T1 UNION SELECT C1,C2
FROM T2)

SELECT QC1,QC2 FROM Q1,T3 WHERE QC1=T3.C1
```

Example 3:

```
INSERT INTO T2 SELECT NEXT VALUE FOR SEQ1, VC1, VC2 FROM V1
```

(10) Named derived table that is derived by a set operation specifying UNION ALL only

One of the following conditions must be satisfied:

1. The query specification specifying a named derived table in the FROM clause directly contains one of the following items:
   - GROUP BY clause, HAVING clause, set function
   - Window function
   - WHERE clause, subquery (provided that the query in (1) is included in one of

the following: a subquery, an operand in a set operation, or the query body of an INSERT statement)

- Function call
- System-defined scalar function
- Component specification
- WRITE specification
- GET_JAVA_STORED_ROUTINE_SOURCE specification
- A column not contained in a selection item is specified as a sort item
- A subquery specifying the named derived table meeting Condition (2) in a FROM clause
- A subquery specifying a derived table
- A subquery specifying a value expression, other than a column specification, in a GROUP BY clause
- One of the following data types is specified in an SQL variable, SQL parameter, or function call specified in the selection expression:

  BLOB type

  BINARY with a minimum defined length of 32,001 bytes

  Abstract data type

  BOOLEAN type

2. One of the following items is specified for the query specification specifying a named derived table for joining tables:
   - A named derived table specified for referencing a table other than a foreign table on the farthest left of an outer join
   - Comma join specified for a FROM clause in which a named derived table is specified (that is, another table reference is specified other than the joined table specifying the named derived table)
   - A subquery or derived table
   - Query specification contained in a subquery, or the operation term of a set operation
   - A set operation term for deriving a named derived table contains one of the following items:
     - Table joining
     - GROUP BY clause, HAVING clause, set function

- SELECT DISTINCT
- A value expression, other than a column specification, in a selection expression
- Query that generates an inner derived table
- Query specifying a derived table

- A named derived table derived by specifying a set operation specified in addition to another named derived table
- One of the following items is specified for a table reference for a joined table for which a named derived table is specified:
  - A named derived table derived by specifying table joining
  - A named derived table derived by specifying a GROUP BY clause, HAVING clause, or set function
  - A named derived table derived by specifying SELECT DISTINCT
  - A named derived table derived by specifying a value expression other than a column specification for the selection expression
  - A named derived table derived by specifying a query that generates an inner derived table
  - A named derived table derived by specifying a subquery
- The total number of tables obtained from the following expression exceeds 65:

  total-number-of-tables =
  (*aggregate-number-of-tables-from-which-named-derived-tables-are-derived*)
  + (*number-of-set-operations-for-deriving-named-derived-tables* + 1)
  x
  (*aggregate-number-of-tables-to-be-specified-on-the-right-side-of-outer-join*)
  +
  (*aggregate-number-of-tables-specified-for-query-if-query-is-specified-in-addition-to-query-specifying-named-derived-table*)

3. Satisfies the conditions described in *(8) Named derived table that is derived by set operations*.

Examples

In the specification of named derived tables V1 (a view table) and Q1 (a query name) derived by SELECT C1,C2 FROM T1 UNION ALL SELECT C1,C2 FROM

T2 in a FROM clause, the following coding creates an inner derived table with respect to V1 and Q1:

Example 1:

```
SELECT C1,C2 FROM V1 GROUP BY C1,C2
```

Example 2:

```
WITH Q1(QC1,QC2) AS (SELECT C1,C2 FROM T1 UNION ALL SELECT
C1,C2 FROM T2)

SELECT QC1,QC2 FROM Q1,T3 WHERE QC1=T3.C1
```

Example 3:

```
SELECT * FROM T1 WHERE EXISTS(SELECT * FROM V1 WHERE
V1.C1=T1.C1)
```

Example 4:

```
INSERT INTO T3 SELECT * FROM V1 WHERE C1>'C001'
```

2.22 WRITE specification

(1) Function

A WRITE specification outputs BLOB data to a file at the single server or front-end server unit and returns the IP address and the filename of the output unit.

(2) Format

WRITE (*output-BLOB-value*, *file-prefix*, *file-output-option*)

(3) Operands

- *output-BLOB-value*

Specifies any of the following (note that the data type of output-BLOB-value must be the BLOB type):

- Column specification
- Component specification
- Function call
- SUBSTR scalar function that produces a result that is the BLOB data type

The following rules apply to output-BLOB-value:

1. Embedded variables and ? parameters can be specified in the argument for the function call or in the second or third argument of the SUBSTR scalar function that is specified in *output-BLOB-value*.
2. If a BLOB column is specified in output-BLOB-value, the BLOB column cannot by itself be specified in a selection expression or in the output-BLOB-value for another WRITE specification.

   Examples of invalid specifications are shown below:
   ```
   SELECT WRITE(C1,***),C1 FROM ***
   SELECT WRITE(C1,***),WRITE(C1,***) *** FROM ***
   ```
3. If a FOR READ ONLY clause is specified, only a column specification can be specified in output-BLOB-value.
4. A subquery cannot be specified in an argument of a function call or in an argument of the scalar function SUBSTR specified as an output BLOB value.

- *file-prefix*

Specifies the prefix part of the file name that HiRDB will assemble. The data type of file-prefix must be VARCHAR and the length must not exceed 222 bytes.

The following can be specified in a file-prefix:

- Literal (character string)
- Embedded variable or ? parameter

The following rules apply to file-prefix:

1. Specify an absolute path name, including directory, that is valid at the unit for the single-server or front-end server to which HiRDB is connected. Additionally, the HiRDB administrator must grant the file user the privileges (access type: full control) for performing all types of operations on the directory. In the case of the UNIX edition, the HiRDB administrator must grant the file user the privileges to read, write, and search within the directory.
2. For the file separator to be specified as a file prefix, specify a forward slash (/) if the HiRDB server is UNIX, and specify a backslash (\) if the HiRDB server is Windows. For the characters that can be specified in a file name, the rules for the HiRDB server platform apply. However, if utf-8 or chinese-gb18030 is specified for the character code classification in the pdntenv command (for the UNIX edition, the pdsetup command), specify it using the ASCII code range.
3. If only an embedded variable or a ? parameter is specified in the file prefix, the embedded variable or the ? parameter must have a simple structure.
4. Use the default character set for the file prefix.

■ *file-output-option*

Specifies the file output mode as a numeric data type (INTEGER type is returned during execution of the DESCRIBE INPUT statement).

The following can be specified in file-output-option:

- Numeric literal
- Embedded variable or ? parameter

The following values can be specified:

Function	Value
Re-create (overwrites any existing file)	0
Append (adds to the end of an existing file)	1
Overwrite disable (flags an error if an existing file is encountered)	2
Asynchronous output (requests asynchronous output from the operating system)	4

Note

The asynchronous output option can be specified in combination with re-create, append, or overwrite disable. When asynchronous output is combined with

another option, specify the value that is the logical union of the asynchronous output option's value and the value for the other option. When the asynchronous output option is not specified, HiRDB requests synchronous output (immediate WRITE) from the operating system.

If only an embedded variable or a ? parameter is specified in the file prefix, the embedded variable or the ? parameter must have a simple structure.

(4) Rules on the results of a WRITE specification

1. The result of a WRITE specification uses a VARCHAR type without the NOT NULL constraint (allowing null values), a defined length of 255 bytes, and the default character set.
2. The result of a WRITE specification is in the following format:

   IP-address : *file-prefix* - *column-number* - *row-counter*

   <- - - - - *file-name* - - - - - ->

   Note: A colon (:) separates the IP address and the file prefix. The file prefix, column number, and row counter are delimited by the hyphen (-).

   Explanation

   IP-address

   Returns the IP address of the unit for the single-server or front-end server to which the client is connected. An IP address is in the format *XXX.XXX.XXX.XXX* and its length is 7-15 bytes (*XXX* is numeric characters in the range 0-255).

   file-prefix

   Returns the file prefix that was specified in the second argument of the WRITE specification.

   column-number

   Returns a number indicating the specified position for the derived table. A column number consists of five numeric digits; the number of the first column is 1 (preceded by leading zeros).

   row-counter

   Returns numbers in ascending order, corresponding to the number of rows that are retrieved. The row counter is a 10-digit numeric character string beginning with 1 (preceded by leading zeros). When the count goes past 2,147,483,646 rows, the counter resets itself to 1.
3. If a retrieval is performed from a client at which the WRITE specification cannot be used, no IP address is set; instead, a character string beginning with a colon is returned.

4. If any of output-BLOB-value, file-prefix, or file-output-option is the null value, the result is also the null value. If the default value setting facility for null values is used in an embedded variable, only an IP address is returned.
5. Similar to an ordinary character data retrieval, if the embedded variable receiving the result of a WRITE specification is shorter than the result, the excess part is truncated and the actual length is assigned to an indicator variable. The truncation process treats the length of the embedded variable as (embedded variable length - 15) bytes by excluding the maximum length of the IP address; if the length includes an IP address, truncation can occur even if the result of the WRITE specification can be stored in the embedded variable.
6. An error may result if the embedded variable receiving the results of the WRITE specification is shorter than 15 bytes.

(5) Rules on the file to which BLOB data is to be output

1. The BLOB data specified in *output-BLOB-value* is output to a file on a unit at the single-server or front-end server.
2. If the result of the WRITE specification is the null value, no file is created.
3. If the actual length of the BLOB data produced by a WRITE specification is 0 bytes, a file whose size is 0 bytes is created.
4. The format of the output file contains BLOB data only; it does not include information on the actual length of the data.
5. In the UNIX edition, the owner of the created file and the mode are as follows:

   Owner: HiRDB administrator

   Group: Group that includes the HiRDB administrator

   Mode: rw-rw-rw-
6. If fewer embedded variables receiving retrieval results are specified than the number of columns for retrieval results and if there is no embedded variable to receive the results of the WRITE specification, no file is created.

(6) Common rules

1. A WRITE specification can be specified by itself in a selection expression for the outermost query specification.
2. A WRITE specification cannot be specified in a sort key field when ORDER BY is specified.
3. In a derived query expression in a WITH clause, a WRITE specification cannot be specified in a selection expression.
4. In a set operation, a WRITE specification cannot be specified in a column of the derived table that is subject to the operation.

5. A WRITE specification cannot be specified in a selection expression for a subquery (including a derived table in a FROM clause).
6. A WRITE specification cannot be specified in a selection expression for a derived query expression for a view definition.
7. A WRITE specification cannot be specified in a selection expression with a query specification in an INSERT statement.
8. A WRITE specification cannot be specified in query specification in an SQL procedure statement in a routine.

(7) Notes

1. Any files that are created should be deleted by the user. If a file name is returned to the UAP as a retrieval result, HiRDB will not manipulate (read or write) the created file; however, caution should be exercised with respect to the following points:
   - If a file is deleted after a FETCH, and if the preceding FETCH result and the BLOB value for the same cursor retrieval are the same, the same file name may be returned but the file will not be re-created. In such a case, you should note the preceding file name and delete the file when its file name has changed.
   - The file can be deleted unconditionally after the cursor is closed.
   - The file can be deleted unconditionally after the transaction is resolved.
2. If the event of an error or rollback, HiRDB does not delete the created file.
3. Normally in the event of an SQL error, any file that is created by the affected SQL statement is deleted. However, if an error occurs after the file output processing within the HiRDB server is completed, files may not be deleted in some case; e.g., in the case of a communication error involving return of results from the HiRDB server to the HiRDB client.
4. If the FETCH facility is used with an array, each FETCH creates a file equal in size to the number of elements in the array, in which case it is important to monitor the available disk space.
5. If the block transfer facility is used, the first FETCH creates a file equal in size to the number of rows transferred per block. Subsequently during each FETCH following completion of the FETCH equal to the number of transferred rows per block, files are repeatedly created equal to the number of rows transferred per block, in which case it is important to monitor the available disk space.
6. Files that are not deleted can compete for OS resources, such as disk space; it is important to be aware of this possibility.
7. If file names conflict with other transactions or cursor retrievals, files can destroy

one another; it is important to be aware of this possibility. Hitachi recommends that duplicate file names be avoided, such as by using a different directory name in the file prefix or a different file name for each transaction and cursor.

8. If character string truncation occurs as a result of a `WRITE` specification, a complete file name cannot be acquired but a file is created. It is important to monitor any competition for disk space for such a possibility.

9. If asynchronous output is specified as the file output option, HiRDB will output `BLOB` data to a file without specifying synchronous output (immediate `WRITE`) to the OS. As a result, the file output processing by the operating system can remain incomplete due to a high load on output devices even if the file output processing within HiRDB is complete. Consequently, even if a file name is returned to the client, the file is not actually created or its creation remains pending under certain timing conditions.

   This situation can be avoided by not specifying asynchronous output as the file output option, at the expense of I/O overhead having a significant impact on response time.

(8) Example

Retrieval examples using the file output facility involving `BLOB` data are shown below:

(a) Retrieving BLOB columns

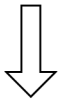
Retrieve columns `C1` and `C2` from Table `T1`, writing `BLOB` data from `C1` to a file and acquiring the file name:

Table name: T1

C1	C2
BLOB-value-1	10
BLOB-value-2	20
BLOB-value-3	30
BLOB-value-4	40

SQL statement

```
SELECT WRITE(C1,'c:\blob_files\t1',0),C2 FROM T1
```

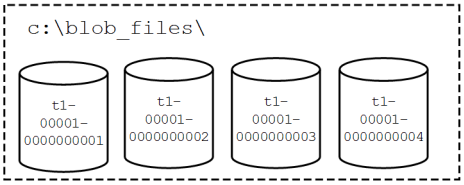


Execution result

C1	C2
172.16.202.5:c:\blob_files\t1-00001-0000000001	10
172.16.202.5:c:\blob_files\t1-00001-0000000002	20
172.16.202.5:c:\blob_files\t1-00001-0000000003	30
172.16.202.5:c:\blob_files\t1-00001-0000000004	40

BLOB data that is output to the server

- IP address: 172.16.202.5



(b) Retrieving abstract data types with the BLOB attribute

From table T1, retrieve column ADT1 such that CONTAINS() is TRUE, writing a BLOB value, which is the result of passing the column value to an argument of EXTRACTS(), to a file, and acquiring the file name (this example illustrates the case in which all columns are retrieved).

Table name: T2

ADT1
abstract-data-type-value-1
abstract-data-type-value-2
abstract-data-type-value-3
abstract-data-type-value-4

SQL statement

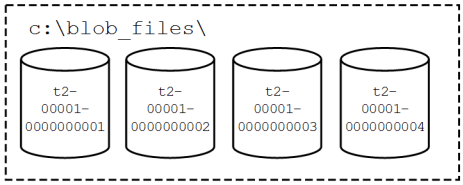
```
SELECT WRITE(EXTRACTS(ADT1,...),'c:\blob_files\t2',0) FROM T2
    WHERE CONTAINS(ADT1,...) IS TRUE
```

Execution result

ADT1
172.16.202.5:c:\blob_files\t2-00001-0000000001
172.16.202.5:c:\blob_files\t2-00001-0000000002
172.16.202.5:c:\blob_files\t2-00001-0000000003
172.16.202.5:c:\blob_files\t2-00001-0000000004

BLOB data that is output to the server

- IP address: 172.16.202.5



2.23 GET_JAVA_STORED_ROUTINE_SOURCE specification

(1) Function

A GET_JAVA_STORED_ROUTINE_SOURCE specification extracts a Java class source file from a JAR file.

Java routines can be used in the HP-UX, Solaris, AIX, Linux, and Windows editions of HiRDB. For HP-UX, Solaris, and AIX, Java routines cannot be used if a POSIX library version of HiRDB is not set up (by executing the pdsetup command) or if a POSIX library version of HiRDB has been reinstalled as a non-POSIX library version of HiRDB. For details of the pdsetup command, see the manual *HiRDB Version 9 Command Reference*.

(2) Format

```
GET_JAVA_STORED_ROUTINE_SOURCE (class-name, JAR-filename
        [, source-file-maximum-length])
```

(3) Operands

- *class-name*

Specifies a class name as a character string of no more than 255 bytes; following is the format:

'[*package-name*.] *class-identifier*'

In *class-name*, VARCHAR type value expressions with a maximum of 255 bytes can be specified.

If only a ? parameter or an embedded variable is specified in class-name, the ? parameter or the embedded variable must be a simple structure.

The default character set must be used.

- *JAR-filename*

Specifies the name of a JAR file as a character string of no more than 255 bytes.

In *JAR-filename*, VARCHAR type value expressions with a maximum of 255 bytes can be specified.

A JAR file name must not be specified as a path name.

If only a ? parameter or an embedded variable is specified in *JAR-filename*, the ? parameter or the embedded variable must be a simple structure.

The default character set must be used.

- *source-file-maximum-length*

Specifies as an integer literal the maximum length (in bytes) of the source file from which data is to be extracted.

The specifiable range is 1 to 2,147,483,647. The null value cannot be specified. The default is 2,147,483,647.

(4) Rules on GET_JAVA_STORED_ROUTINE_SOURCE specification results

1. The result of a GET_JAVA_STORED_ROUTINE_SOURCE specification will be the BLOB type without the NOT NULL constraint (null value will be permitted) and a defined length equal to the length specified in source-file-maximum-length.
2. The result of a GET_JAVA_STORED_ROUTINE_SOURCE specification is the contents of the file that is fetched from the JAR file (any portion that is in excess of the value specified in *source-file-maximum-length* is truncated).
3. The result of a GET_JAVA_STORED_ROUTINE_SOURCE specification is the null value in any of the following cases:
   - Any argument is the null value.
   - The specified JAR file has not been installed.
   - No source file associated with the specified class is found in the JAR file.

(5) Common rules

1. GET_JAVA_STORED_ROUTINE_SOURCE can be specified in the following places:
   - Singly in a selection expression in an outermost query specification
   - In an argument of the LENGTH scalar function in a selection expression in an outermost query specification
2. If the ORDER BY clause is specified, GET_JAVA_STORED_ROUTINE_SOURCE cannot be specified in a sort key field.
3. GET_JAVA_STORED_ROUTINE_SOURCE cannot be specified in a selection expression in a derived query expression in the WITH clause.
4. When a set operation is performed, GET_JAVA_STORED_ROUTINE_SOURCE cannot be specified in a column for the derived table to be subject to the operation.
5. A GET_JAVA_STORED_ROUTINE_SOURCE specification cannot be specified in a selection expression for a subquery (including a derived table in a FROM clause).
6. GET_JAVA_STORED_ROUTINE_SOURCE cannot be specified in a selection expression in a derived query expression in a view definition.
7. GET_JAVA_STORED_ROUTINE_SOURCE cannot be specified in a selection expression in a query specification in an INSERT statement.
8. GET_JAVA_STORED_ROUTINE_SOURCE only references a class identifier that is

obtained by removing the package name from the class name. Therefore, if there are multiple source files with the same class identifier in the JAR file, the result will be a linkage of the multiple source files.

9. GET_JAVA_STORED_ROUTINE_SOURCE regards character strings with 'java' suffixed to the specified class identifier as Java-class source files.
10. In order for a source file to be fetched by GET_JAVA_STORED_ROUTINE_SOURCE, it must be available in the JAR file. For details of creating JAR files, see the *HiRDB Version 9 UAP Development Guide*.

(6) Notes

1. GET_JAVA_STORED_ROUTINE_SOURCE operates on JAR files that are installed in HiRDB.
2. In order for SQL statements that include a GET_JAVA_STORED_ROUTINE_SOURCE specification to be executed, the Java environment must be installed.
3. Class names and JAR file names must not include space characters.

(7) Example

Extract the contents of the source file for a Java routine (JAVAROUTINE) registered in a schema (USER1):

```
SELECT GET_JAVA_STORED_ROUTINE_SOURCE(CLASS_NAME,JAR_NAME)
  FROM MASTER.SQL_ROUTINES
   WHERE ROUTINE_SCHEMA='USER1' AND ROUTINE_NAME='JAVAROUTINE'
```

2.24 SQL optimization specification

- Function

  This function allows you to specify optimization for improving the retrieval efficiency of an SQL statement in the SQL statement.

  SQL optimization can be specified for the following items:

  - Used indexes
  - Join methods
  - Subquery execution methods

- Common rules

  1. In some cases, SQL optimization, when specified, may not take effect. You can check whether the SQL optimization specification has taken effect by using the access path display utility. For details about this utility, see the manual *HiRDB Version 9 Command Reference*.
  2. You can enclose an SQL optimization specification by placing the `/*>>` and `<<*/` symbols before and after it. Each SQL optimization specification must be enclosed separately. An SQL optimization specification enclosed in `/*>>` and `<<*/` is not interpreted as a comment. Specifying an SQL optimization by enclosing it in `/*>>` and `<<*/` can be useful for ensuring compatibility with APs that are common to other DBMSs.
  3. The SQL optimization specification has priority over the SQL optimization option and SQL extension optimizing option. For details about the SQL optimization option and SQL extension optimizing option, see the *ALTER PROCEDURE (Re-create SQL object of procedure)*, *ALTER ROUTINE (Re-create SQL objects for functions, procedures, and triggers)*, *ALTER TRIGGER (Re-create a trigger SQL object)*, *CREATE [PUBLIC] PROCEDURE (Define procedure, define public procedure)*, or *CREATE TRIGGER (Define a trigger)* section in Chapter 3.

- Notes

  1. If `BY NEST` is specified in a join method SQL optimization specification, and if an index is specified that cannot be used in a nest-loop join as an SQL optimization specification in a used index in the joined inner table, or the suppression of the use of the index is specified, the SQL optimization specification for the used index is nullified.
  2. An SQL optimization specification enclosed in `/*>>` and `<<*/` cannot be enclosed again in `/*>>` and `<<*/`.

2.24.1 SQL optimization specification for a used index

(1) Function

In the SQL optimization specification for a used index, you can specify either an index to be used for retrieving a table or the suppression of use of an index (table scanning).

(2) Format

```
SQL-optimization-specification-for-a-used-index :: =
     [WITH INDEX (index-specification[, index-specification] ... |) | WITHOUT INDEX]

Index-specification :: = [ [authorization-identifier.]index-identifier |PRIMARY KEY
            |CLUSTER KEY |PRIMARY CLUSTER KEY]
```

(3) Operands

- WITH INDEX(*index-specification* [, *index-specification*] ...)

Specifies the index to be used. If multiple indexes are specified, the operation uses multiple indexes.

- WITHOUT INDEX

Suppresses the use of an index (table scanning).

- [[*authorization-identifier*.]*index-identifier* | PRIMARY KEY | CLUSTER KEY | PRIMARY CLUSTER KEY]

authorization-identifier

Specifies the authorization identifier of the user who owns the index. For default values, see *1.1.8 Qualifying a name*.

index-identifier

Specifies the name of the index to be used.

PRIMARY KEY

Specifies the option of when to use an index that is defined by specifying a primary key (or a primary cluster key) during table definition. For primary and cluster keys, see *CREATE TABLE*.

CLUSTER KEY

Specifies the option of when to use an index that is defined by specifying an index key (or a primary cluster key) during table definition.

PRIMARY CLUSTER KEY

Specifies the option of when to use an index that is defined by specifying a

primary cluster key during table definition.

(4) Rules

1. SQL optimization cannot be specified for a named derived table that functions as an inner derived table. For inner derived tables, see *2.21 Inner derived tables*.
2. SQL optimization cannot be specified for named derived tables that are derived from the joining of two or more tables or as a result of a set operation.
3. If additional SQL optimization for a used index is specified in a named derived table that is derived by specifying SQL optimization for a used index, the later specification takes effect, and the specification of the derived query expression is nullified.

(5) Notes

1. For retrievals using an index, see the retrieval methods described in the *HiRDB Version 9 UAP Development Guide*. Indexes that are used can be checked by using the access path display utility.

2.24.2 Join method SQL optimization specification

(1) Function

The join method SQL optimization specification allows you to specify a join method for a joined table.

(2) Format

Join-method-SQL-optimization-specification::= BY {NEST|HASH|MERGE}

(3) Operands

- {NEST|HASH|MERGE}

NEST

This option can be specified when the join method is a nested-loop join.

If the inner table is a base table or a named derived table based on a base table, the join method becomes a nested-loop join.

HASH

This option can be specified when making the join method into a hash-join.

MERGE

This option can be specified when making the join method into a merge-join.

(4) Rules

1. Specifying HASH requires the same preparations as specifying execution of a hash-join or a subquery hash for the SQL extension optimizing option. For preparations for applying either hash-join or subquery-hash-join, see the *HiRDB Version 9 UAP Development Guide*.

(5) Notes

1. For join methods, see the *HiRDB Version 9 UAP Development Guide*. Join methods can be checked by using the access path display utility.

2.24.3 Subquery execution method SQL optimization specification

(1) Function

The subquery execution method SQL optimization specification allows you to specify a subquery execution method for a subquery that occurs in a predicate.

(2) Format

Subquery-execution-method-SQL-optimization-specification::= {HASH|NO HASH}

(3) Operands

- {HASH|NO HASH}

HASH

Specifies the option for making the subquery execution method into a hash execution.

NO HASH

Specifies the option for making the subquery execution method into a non-hash execution.

(4) Rules

1. Subquery execution method SQL optimization specification must be specified in a subquery that occurs in a predicate.
2. Specifying HASH requires the same preparations as specifying execution of a hash-join or a subquery hash for the SQL extension optimizing option. For preparations for applying either hash-join or subquery-hash-execution, see the *HiRDB Version 9 UAP Development Guide*.

(5) Notes

1. For details about the subquery execution methods, see the subquery execution method without external reference or the subquery execution method with

external reference in the *HiRDB Version 9 UAP Development Guide*. Subquery execution methods can be checked by using the access path display utility.

2.24.4 Examples of SQL optimization specification

Examples of using an SQL optimization specification are given as follows:

1. In a SELECT statement, an SQL optimization specification is specified for a used index. In this case, an index (IDX1) is used for retrieving a stock table (STOCK):
```
SELECT PNAME FROM STOCK WITH INDEX (IDX1)
  WHERE PRICE <= 500
```

2. In a SELECT statement, an SQL optimization specification is specified for a used index. In this case, multiple indexes (IDX1, IDX2) are used for retrieving a stock table (STOCK):
```
SELECT PNAME FROM STOCK WITH INDEX (IDX1,IDX2)
  WHERE PRICE <= 500 OR SQUANTITY > 100
```

3. In a SELECT statement, specify an SQL optimization specification for a used index. In this case, for the definition of a stock table (STOCK), an index that was defined by specifying the PRIMARY KEY option is used:
```
SELECT PNAME FROM STOCK WITH INDEX (PRIMARY KEY)
  WHERE PRICE <= 500
```

4. In a SELECT statement, specify an SQL optimization specification for a used index. In this case, use of an index (table scanning) for the retrieval of the stock table (STOCK) is suppressed:
```
SELECT PNAME FROM STOCK WITHOUT INDEX
  WHERE PRICE <= 500
```

5. In a SELECT statement, a join method SQL optimization specification is specified. In this case, a nested-loop join is used as the join method for the inner join.
```
SELECT STOCK.PCODE,STOCK.PNAME,ORDER.CCODE
  FROM STOCK INNER JOIN BY NEST ORDER
    ON STOCK.PCODE = ORDER.PCODE
```

6. In a SELECT statement, a join method SQL optimization specification is specified. In this case, a hash-join is used as the join method for the outer join.
```
SELECT STOCK.PCODE,STOCK.PNAME,ORDER.CCODE
  FROM STOCK LEFT OUTER JOIN BY HASH ORDER
    ON STOCK.PCODE = ORDER.PCODE
```

7. In a SELECT statement, a join method SQL optimization specification is specified. In this case, a merge-join is used as the join method for the inner join (with INNER omitted).
```
SELECT STOCK.PCODE,STOCK.PNAME,ORDER.CCODE
  FROM STOCK JOIN BY MERGE ORDER
```

```
    ON STOCK.PCODE = ORDER.PCODE
```

8. In a SELECT statement, a subquery execution method SQL optimization specification is specified. In this case, hash execution is used as the subquery execution method.
```
SELECT PNAME FROM STOCK
  WHERE PCODE =ANY
    (HASH SELECT PCODE FROM ORDER
       WHERE CCODE = '302S')
```

9. In a SELECT statement, a subquery execution method SQL optimization specification is specified. In this case, non-hash execution is used as the subquery execution method (in this example, the option is either work table execution or work table ATS execution).
```
SELECT PNAME FROM STOCK
  WHERE PCODE =ANY
    (NO HASH SELECT PCODE FROM ORDER
      WHERE CCODE = '302S')
```

10. A used index SQL optimization specification is specified by enclosing it in /*>> and <<*/.
```
SELECT PNAME FROM STOCK /*>> WITH INDEX (IDX1) <<*/
  WHERE PRICE <= 500
```

11. A join method SQL optimization specification is specified by enclosing it in /*>> and <<*/.
```
SELECT STOCK.PCODE,STOCK.PNAME,ORDER.CCODE
  FROM STOCK INNER JOIN /*>> BY NEST <<*/ ORDER
    ON STOCK.PCODE = ORDER.PCODE
```

12. A subquery execution method SQL optimization specification is specified by enclosing it in /*>> and <<*/.
```
SELECT PNAME FROM STOCK
  WHERE PCODE =ANY
    (/*>> HASH <<*/ SELECT PCODE FROM ORDER
      WHERE CCODE = '302S')
```

13. In a SELECT statement, a join method SQL optimization specification and a used index SQL optimization specification are specified. In this case, an index (IDX3) is used for retrieving a stock table (STOCK) and a hash-join is used as the join method for the inner join (with INNER omitted).
```
SELECT STOCK.PCODE,STOCK.PNAME,ORDER.CCODE
  FROM STOCK WITH INDEX (IDX3) JOIN BY HASH ORDER
    ON STOCK.PCODE = ORDER.PCODE
```

14. In a SELECT statement, a join method SQL optimization specification and a used index SQL optimization specification are specified by enclosing each in /*>> and

<<*/. In this case, an index (IDX3) is used for retrieving a stock table (STOCK) and a hash-join is used as the join method for the inner join (with INNER omitted).

```
SELECT STOCK.PCODE,STOCK.PNAME,ORDER.CCODE
  FROM STOCK /*>> WITH INDEX (IDX3) <<*/ JOIN /*>> BY HASH
<<*/ ORDER
    ON STOCK.PCODE = ORDER.PCODE
```

2.25 CAST specification

(1) Function

A CAST specification converts the data in a value expression into a specified data type.

(2) Format

CAST-specification::= CAST ([*value-expression* | NULL] AS *data-type*)

(3) Rules

1. The following data types cannot be specified in *value-expression*:
   - BLOB
   - BINARY with a minimum defined length of 32,001 bytes
   - Abstract data type
2. The following data types cannot be specified in AS *data-type*:
   - BLOB

     CAST(NULL AS BLOB) can be specified.
   - BINARY with a minimum length of 32,001 bytes

     CAST(NULL AS BINARY(*n*)) can be specified, where *n* is a byte count with a minimum of 32,001 bytes.
   - BOOLEAN
   - Abstract data type
3. The value of the result is not NOT NULL constrained (the null value is allowed).
4. If NULL is specified in *value-expression* or the result of the value expression is the null value, the value of the result is the null value.
5. If a real length of 0 bytes or character data with a real length of 0 is specified in *value-expression*, conversions into a character type are performed according to the data conversion rules. Any conversion into a data type other than a character type may result in an error.
6. If an embedded variable or a ? parameter by itself is specified in *value-expression* for the data type of the embedded variable or the ? parameter, HiRDB assumes the data type that was specified in AS *data-type*.
7. When specifying a repetition column in *value-expression*, specify a subscript,

except in the option ANY, for which a subscript cannot be specified.

8. In AS *data-type*, specify a data type that can be converted into the data type that is specified in *value-expression*. The following table indicates the convertibility of each data type.

Table 2-90: Data type convertibility between the result of the value expression and AS data type (1/2)

Data type of result of value expression			AS data type						
			Numeric data		Character string data (any character set)			National character data	Mixed character data
			Exact numeric	Approximate numeric					
			INTEGER, SMALLINT, DECIMAL	FLOAT, SMALLFLT	CHAR, VARCHAR			NCHAR, NVARCHAR	MCHAR, MVARCHAR
					DF	EK	U16		
Numeric data	Exact numeric	INTEGER, SMALLINT, DECIMAL	Y	Y	Y	Y	Y	N	Y
	Approximate numeric	FLOAT, SMALLFLT	Y	Y	Y	Y	Y	N	Y

Data type of result of value expression			AS data type						
			Numeric data		Character string data (any character set)			National character data	Mixed character data
			Exact numeric	Approximate numeric					
			INTEGER, SMALLINT, DECIMAL	FLOAT, SMALLFLT	CHAR, VARCHAR			NCHAR, NVARCHAR	MCHAR, MVARCHAR
					DF	EK	U16		
Character string data (any character set)	CHAR, VARCHAR	DF	Y	Y	Y	Y #2	Y	N	Y
		EK	Y	Y	Y #2	Y	N	N	N
		U16	Y	Y	Y	N	Y	N	Y
National character data	NCHAR, NVARCHAR		N	N	N	N	N	Y	N
Mixed character data	MCHAR, MVARCHAR		Y	Y	Y	N	Y	N	Y
Boolean data	BOOLEAN		N	N	Y	Y	Y	N	Y
Date data	DATE		N	N	Y #1	Y #1	Y#1	N	Y#1
Time data	TIME		N	N	Y #1	Y #1	Y#1	N	Y#1
Time stamp data	TIMESTAMP		N	N	Y #1	Y #1	Y#1	N	Y#1
Date interval data	INTERVAL YEAR TO DAY		Y	N	N	N	N	N	N
Time interval data	INTERVAL HOUR TO SECOND		Y	N	N	N	N	N	N

Data type of result of value expression		AS data type						
		Numeric data		Character string data (any character set)			National character data	Mixed character data
		Exact numeric	Approximate numeric					
		INTEGER, SMALLINT, DECIMAL	FLOAT, SMALLFLT	CHAR, VARCHAR			NCHAR, NVARCHAR	MCHAR, MVARCHAR
				DF	EK	U16		
Binary data	BINARY	N	N	Y	Y	Y	N	N

Legend:

Y: Data can be converted.

N: Data cannot be converted.

DF: Default character set

EK: EBCDIK

U16: UTF16

#1

If the length specified in the AS data type is greater than or equal to the length listed below, conversion can be performed:

- When the character set of the result of the value expression is a character set other than UTF-16

DATE:

10 bytes

TIME:

8 bytes

TIMESTAMP(*p*):

If *p* = 0, 19 bytes (no periods)

If *p* > 0, 20 + ↓(*p* + 1) ÷ 2↓ x 2 (bytes)

- When the character set of the result of the value expression is UTF-16

For DATE:

- 20 bytes

For TIME:

- 16 bytes

For TIMESTAMP(*p*):

- If *p* = 0, 38 bytes (no periods)
- If *p* > 0, 40 + ↓(*p* + 1) ÷ 2↓ x 4 bytes

If the length specified in the AS data type is less than the length indicated above, conversion cannot be performed

When converting to a fixed-length character data type and the length specified in the AS data type is greater than the length indicated above, the data is left-justified and filled with trailing spaces of the character set being used.

#2

SJIS characters from the default character set are treated as single-byte JIS8 characters, and conversion is performed between JIS8 encoding and EBCDIK encoding.

For details about the character encoding conversion rules, see the *HiRDB Version 9 UAP Development Guide*.

Table 2-91: Data type convertibility between the result of the value expression and AS data type (2/2)

Data type of result of value expression				AS data type					
				Date data	Time data	Time stamp data	Date interval data	Time interval data	Binary data
				DATE	TIME	TIMESTAMP	INTERVAL YEAR TO DAY	INTERVAL HOUR TO SECOND	BINARY
Numeric data	Exact number	INTEGER, SMALLINT, DECIMAL		N	N	N	Y	Y	N
	Approximate number	FLOAT, SMALLFLT		N	N	N	N	N	N
Character string data		CHAR, VARCHAR	DF	Y	Y	Y	N	N	Y
			EK	Y	Y	Y	N	N	Y
			U16	Y	Y	Y	N	N	Y
National character data		NCHAR, NVARCHAR		N	N	N	N	N	N
Mixed character data		MCHAR, MVARCHAR		Y	Y	Y	N	N	N
Boolean data		BOOLEAN		N	N	N	N	N	N

Data type of result of value expression		AS data type					
		Date data	Time data	Time stamp data	Date interval data	Time interval data	Binary data
		DATE	TIME	TIMESTAMP	INTERVAL YEAR TO DAY	INTERVAL HOUR TO SECOND	BINARY
Date data	DATE	Y	N	Y	N	N	N
Time data	TIME	N	Y	Y	N	N	N
Time stamp data	TIMESTAMP	Y	Y	Y	N	N	N
Date interval data	INTERVAL YEAR TO DAY	N	N	N	Y	Y	N
Time interval data	INTERVAL HOUR TO SECOND	N	N	N	Y	Y	N
Binary data	BINARY	N	N	N	N	N	Y

Legend:

Y: Data can be converted.

N: Data cannot be converted.

DF: Default character set

EK: EBCDIK

U16: UTF16

9. The window function cannot be specified.

(4) Conversion rules specific to the data types of results

(a) Numeric data

- The result in the value expression is numeric data:

  Higher effective digits should not be lost in the data type to be converted.
- The result in the value expression is a character string or a mixed character string:

  The result obtained by removing leading and trailing spaces from the data must

be the character string representation of a numeric literal. After the character string representation of the numeric literal is converted into a numeric value, the rules for numeric data are applied to the result stored in the value expression.

- The result in the value expression is either date interval data or time interval data:

  If the precision and decimal scaling position of the data type (DECIMAL) of the result are as listed in the following table, they are converted to the decimal format of the date interval data or time interval data.

  Table 2-92: Rules for the conversion of date interval data and time interval data into numeric data

Data type of result in value expression	Length specified in AS data type
INTERVAL YEAR TO DAY	Precision 8, scaling 0
INTERVAL HOUR TO SECOND	Precision 6, scaling 0

(b) Character data and mixed character data

- The following table describes the rules for conversion into character data and mixed character data.

  Table 2-93: Rules for conversion into character data and mixed character data

Relationship between length of result of value expression and length of AS data type	Data type of result of value expression	
	Character data and mixed character data	Other than character data or mixed character data
length-of-value-expression-result < *length-specified-in-AS-data-type*	If the data type specified for the AS data type has a fixed length, the data is left-justified and filled with trailing spaces (of the character set of the data type of the result).	
length-of-value-expression-result = *length-specified-in-AS-data-type*	Normal conversion	
length-of-value-expression-result > *length-specified-in-AS-data-type*	The data is left-justified and the remainder is truncated. If the truncated data contains characters other than spaces (of the character set of the data type of the result), 'W' is set to the SQLWARN1 area.#	An error may occur.

#: If a truncation occurs in the middle of a multi-byte character, a part of the multi-byte character is returned as the value of the result.

- If the result in the value expression is an exact numeric (SMALLINT, INTEGER, or DECIMAL):

  The shortest character string (with the leading zeros removed) that results from

the conversion of data into a numeric literal is the result.

If the data is less than 0, the negative sign (-) is assigned at the beginning.

- If the result in the value expression is an approximate numeric (SMALLFLT or FLOAT):

  The shortest character string (whose leading digit is non-zero; if the data is 0, the leading digit is 0E0) that results from the conversion of data into a numeric literal is the result.

  If the data is less than 0, the negative sign (-) is assigned at the beginning.

- If the result in the value expression is a character string or a mixed character string:

  If the data length of the result in the value expression is greater than the length specified in AS data type, the data is left-justified and the remainder is truncated. If the truncated data contains characters other than spaces (of the character set of the data type of the result), 'W' is set to the SQLWARN1 area.

  In addition, if the result of the value expression is character string data that specifies a character set, all characters in the value expression are converted and listed in the sort order defined for that character set.

- If the result in the value expression is Boolean data:

  If the result in the value expression is true, the conversion result is 'TRUE'; if the result in the value expression is false, the conversion result is 'FALSE'. If the result in the value expression is unknown, the conversion result is the null value.

- If the result in the value expression is date data, time data, or time stamp data:

  The data is converted into the defined character string representation of date data, time data, or time stamp data.

- If the result in the value expression is binary data:

  If the data length of the result in the value expression is greater than the data length specified for the data type of the result, 'W' is set to the SQLWARN1 area.

(c) National character data

- If the result in the value expression is a national character string:

  If the data type of the result is of a fixed length, and the result of converting the data in the value expression is less than the length specified in AS *data-type*, the data is filled with double-byte spaces that correspond to the character code.

  If the result of converting the data in the value expression into a national character string is greater than the data length specified for the data type of the result, the data is left-justified and truncated on the right. If the part of the data to be truncated contains characters other than a double-byte space corresponding to the

character code, 'W' is set to the SQLWARN1 area.

(d) Date data, time data, and time stamp data

- If the result in the value expression is a character string or a mixed character string:

  The data can be converted if it is the defined character string representation of date data, time data, or time stamp data. However, if the spaces (of the character set of the result's data type) before and after the data in the value expression are removed and the fractional second precision of the result is not 0, 2, 4, or 6, an error occurs.

- If the result in the value expression is date data, time data, or time stamp data:

  Conversion can be performed for the combinations listed in the following table.

  Table 2-94: Conversion rules for date data, time data, and time stamp data

Data type of result of value expression	AS data type	Conversion rule
DATE	DATE	Not converted.
	TIMESTAMP(p_2)	The time part is converted as '00:00:00'. The fractional second part is zero-filled.
TIME	TIME	Not converted.
	TIMESTAMP(p_2)	The date part is converted into CURRENT_DATE. The fractional second part is zero-filled.
TIMESTAMP(p_1)	DATE	Extracts and converts the date part.
	TIME	Extracts and converts the time part.
	TIMESTAMP(p_2)	Not converted. If $p_1 > p_2$, the fractional second part is truncated, and if $p_1 < p_2$, it is zero-filled.

(e) Date interval data and time interval data

- If the result in the value expression is DECIMAL:

  If the precision and decimal scaling position are as listed in the following table, they can be converted to the corresponding date interval data or time interval data.

Table 2-95: Conversion rules for numeric data into date interval data or time interval data

Format of numeric data	Data type of result of value expression
Precision 8, scaling 0	`INTERVAL YEAR TO DAY`
Precision 6, scaling 0	`INTERVAL HOUR TO SECOND`

- If the result in the value expression is date interval data or time interval data:

  Conversion can be performed for the combinations listed in the following table.

  Table 2-96: Conversion rules for date interval data and time interval data

Data type of result of value expression	AS data type	Conversion rule
`INTERVAL YEAR TO DAY`	`INTERVAL YEAR TO DAY`	Not converted.
	`INTERVAL HOUR TO SECOND`	The date part is converted into a time part, and if the result does not exceed the range of values for `INTERVAL HOUR TO SECOND`, the data can be converted. If the result exceeds this range, an error may occur.
`INTERVAL HOUR TO SECOND`	`INTERVAL YEAR TO DAY`	If the time part is greater than 24 hours, it is carried to the date part. Any data less than 24 hours is truncated.
	`INTERVAL HOUR TO SECOND`	Not converted.

(f) Binary data

- If the result of the value expression is a character string:

  If the conversion of the data in the value expression into `BINARY` produces a result that exceeds the length specified in `AS` *data-type*, `'W'` is set to the `SQLWARN1` area.

2.26 Extended statement name

(1) Overview

The extended statement name, when specified in a PREPARE statement, identifies the SQL statement that is prepared by the PREPARE statement. When specified in any of the following SQL statements, the extended statement name permits operations on the SQL statement that is identified by it:

- DESCRIBE statement
- DESCRIBE TYPE statement
- ALLOCATE CURSOR statement
- EXECUTE statement
- DEALLOCATE PREPARE statement

(2) Format

extended-statement-name ::= *scope-option* : *embedded-variable*
scope-option ::= GLOBAL

(3) Explanation

■ *scope-option*

Specifies the scope for the extended statement name. The following table lists the available scope options:

Table 2-97: How to specify a scope option

Specification	Scope
GLOBAL	The extended statement name is enabled during the current SQL session (from the time HiRDB is connected until it is disconnected).

■ : *embedded-variable*

Specifies an embedded variable of the variable-length character string type that has an SQL statement identifier as a value.

Only the default character set can be specified.

(4) Rules

1. Values must be specified in *embedded-variable* according to the specification method applicable to SQL statement identifiers. For details about SQL statement identifier specification methods, see *1.1.7 Specification of names*.

2. Valid extended statement names having the same value are identified as the same extended statement name.
3. If the value of an extended statement name is the same as an SQL statement identifier directly specified in an SQL statement, both are distinguished as identifying different SQL statements.

2.27 Extended cursor name

(1) Overview

The extended cursor name, when specified in an ALLOCATE CURSOR statement, identifies the cursor that is allocated to a group of result sets that are returned by a dynamic SELECT statement or a procedure. When specified in one of the following SQL statements, the extended cursor name permits operations on the cursor identified by it:

- DESCRIBE CURSOR statement
- OPEN statement
- FETCH statement
- DELETE statement (exclusive of extended cursor names that identify a results set cursor)
- UPDATE statement (exclusive of extended cursor names that identify a results set cursor)
- CLOSE statement

In addition, when directly specified in any of the following SQL statements, the value of an extended cursor name permits operations on the cursor identified by the extended cursor name:

- Preparable dynamic DELETE statement: locating (exclusive of extended cursor name values that identify a results set cursor)
- Preparable dynamic UPDATE statement: locating (exclusive of extended cursor name values that identify a results set cursor)

(2) Format

```
extended-cursor-name ::= scope-option : embedded-variable
scope-option ::= GLOBAL
```

(3) Explanation

- *scope-option*

  Specifies the scope for the extended statement name. The following table lists the available scope options.

Table 2-98: How to specify a scope option

Specification	Scope
GLOBAL	The extended statement name is enabled during the current SQL session (from the time HiRDB is connected until it is disconnected).

- : *embedded-variable*

  Specifies an embedded variable of the variable-length character string type having a cursor name as a value.

  Only the default character set can be specified.

(4) Rules

1. Values must be specified in *embedded-variable* according to the specification method applicable to cursor names. For details about cursor name specification methods, see *1.1.7 Specification of names*.
2. Extended cursor names having the same value are identified as the same extended cursor name.
3. An error may result if an extended cursor name is specified in an ALLOCATE CURSOR statement and if another valid extended cursor name having the same name already identifies another cursor.
4. If the value of an extended cursor name is the same as the cursor name that is directly specified in an SQL statement, both are distinguished as identifying different cursors, with the exception that a cursor name directly specified in a *preparable-dynamic* UPDATE-*statement-locating* or *preparable-dynamic* DELETE-*statement-locating* identifies the cursor that is the same as the extended cursor name having that value.

2.28 NEXT VALUE expression

(1) Overview

A NEXT VALUE expression returns the value generated by a sequence generator.

(2) Privileges

By specifying FOR PUBLIC USAGE when you define a NEXT VALUE expression, all users can access a sequence generator. Only the owner can access any other sequence generator.

(3) Format

NEXT-VALUE-expression::= NEXT VALUE FOR [*authorization-identifier*.]*sequence-generator-identifier*

(4) Explanation

- *authorization-identifier*

  Specifies the authorization identifier of the sequence generator's owner.

- *sequence-generator-identifier*

  Specifies the identifier of the sequence generator to be used.

(5) Syntax rules

1. The data type of the result for a NEXT VALUE expression is the data type of the value generated by the specified sequence generator.
2. A NEXT VALUE expression can be specified without specifying a subquery in the following locations:

   - Selection expression of a query specification in an INSERT statement
   - Insertion value in an INSERT statement
   - Update value in an UPDATE statement

   Specification is not possible in the following locations:

   - CASE expression
   - In a VALUE scalar function
   - Query specification specifying a GROUP BY clause, HAVING clause, or set function
   - Query specification specifying a window function
   - Query specification that includes DISTINCT

- Query specification as an operand of a set operation other than UNION ALL

3. The result of the NEXT VALUE expression is without the NOT NULL constraint (the null value is allowed).

(6) Common rules

1. When the NEXT VALUE expression is executed the first time after defining a sequence generator, the start value is returned.
2. When an insertion value is specified, the value generated by the specified sequence generator is inserted in each row specified in the INSERT statement.
3. When an update value is specified, the value generated by the specified sequence generator is used to update each row specified in the UPDATE statement.
4. If a NEXT VALUE expression is specified more than once for the same sequence generator for the same row, all of the NEXT VALUE expressions return the same value.

(7) Notes

1. The value from a sequence generator will not return to its original value, even if a rollback occurs and the transaction is disabled.
2. The value from a sequence generator may be updated even if the result of executing the SQL statement specifying a NEXT VALUE expression returns an error.

(8) Examples

1. This example defines an inventory table (STOCK) that has a product ID (PID), product name (PNAME), and price per unit (PRICE).
```
CREATE TABLE STOCK(
PID INTEGER,
PNAME NCHAR(8),
PRICE INTEGER)
```

2. This example defines an SEQ1 sequence generator that numbers product IDs from 1000 to 9999.
```
CREATE SEQUENCE SEQ1
START WITH 1000
INCREMENT BY 1
MAXVALUE 9999

NO CYCLE
```

3. This example registers new products into an inventory table (STOCK).
```
INSERT INTO STOCK VALUES(NEXT VALUE FOR SEQ1, N'Pants', 1200)
INSERT INTO STOCK VALUES(NEXT VALUE FOR SEQ1, N'Shirt', 1000)
INSERT INTO STOCK VALUES(NEXT VALUE FOR SEQ1, N'Sweater',
```

```
1500)
```

Execution result

Inventory table (STOCK)

Product ID (PID)	Product name (PNAME)	Unit price (PRICE)
1000	Pants	12.00
1001	Shirt	10.00
1002	Sweater	15.00

4. This example updates a product ID (PID) of an inventory table (STOCK) from 1001 to a new product ID.

```
UPDATE STOCK SET PID = NEXT VALUE FOR SEQ1 WHERE PID = 1001
```

Execution result

Inventory table (STOCK)

Product ID (PID)	Product name (PNAME)	Unit price (PRICE)
1000	Pants	12.00
1003	Shirt	10.00
1002	Sweater	15.00

Chapter 3. Definition SQL

This chapter explains the syntax and structure of the definition SQL.

General rules

Types and functions of the definition SQL

The definition SQL enables users to define and modify schemas and to define and delete tables, indexes, and privileges.

The following table lists the types and functions of definition SQL.

Table 3-1: Types and functions of definition SQL

Type	Function
`ALTER INDEX` (Alter index definition)	Changes the index name.
`ALTER PROCEDURE` (Re-create SQL object of procedure)	Re-creates the SQL object of a procedure.
`ALTER ROUTINE` (Re-create SQL objects for functions, procedures, and triggers)	Re-creates SQL objects for functions, procedures, and triggers.
`ALTER TABLE` (Alter table definition)	• Adds a new column to the end of a base table. • Increases the maximum length of an existing column of the variable-length data type. • Changes data types. • Deletes a base table column that contains no data. • Changes the uniqueness constraint for cluster keys for a base table containing no data. • Renames tables and columns. • Assigns the updatable column attribute. • Changes the table to a falsification-prevented table. • Changes the partitioning storage condition for row partitioned tables and matrix partitioned tables.
`ALTER TRIGGER` (Re-create trigger SQL object)	Re-creates a trigger SQL object.
`COMMENT` (Comment)	Provides a comment in a table or column.
`CREATE AUDIT` (Define audit event)	Defines audit events and their targets to be recorded as an audit trail.
`CREATE CONNECTION SECURITY` (Definition of the connection security facility)	Defines security items related to the connection security facility.
`CREATE FUNCTION` (Define function)	Defines a function.
`CREATE PUBLIC FUNCTION` (Define public function)	Defines a public function.

Type	Function
CREATE INDEX (Define index)	Defines an index (in ascending or descending order) for columns in a base table.
CREATE PROCEDURE (Define procedure)	Defines a procedure.
CREATE PUBLIC PROCEDURE (Define public procedure)	Defines a public procedure.
CREATE SCHEMA (Define schema)	Defines a schema.
CREATE SEQUENCE (Define sequence generator)	Defines a sequence generator.
CREATE TABLE (Define table)	Defines a base table.
CREATE TRIGGER (Define trigger)	Defines a trigger.
CREATE TYPE (Define type)	Defines an abstract data type.
CREATE VIEW (Define view)	Defines a view table.
CREATE PUBLIC VIEW (Define public view)	Defines a public view.
DROP AUDIT (Delete audit event)	Removes definitions that match the target audit events defined in CREATE AUDIT from being audited.
DROP CONNECTION SECURITY (Delete connection security facility)	Deletes security items related to the connection security facility.
DROP DATA TYPE (Delete user-defined data type)	Deletes an abstract data type.
DROP FUNCTION (Delete function)	Deletes a function.
DROP PUBLIC FUNCTION (Delete public function)	Deletes a public function.
DROP INDEX (Delete index)	Deletes an index.
DROP PROCEDURE (Delete procedure)	Deletes a procedure.
DROP PUBLIC PROCEDURE (Delete public procedure)	Deletes a public procedure.
DROP SCHEMA (Delete schema)	Deletes a schema.
DROP SEQUENCE (Delete sequence generator)	Deletes a sequence generator.
DROP TABLE (Delete table)	Deletes a base table, as well as any indexes, comments, access privileges, and view tables associated with the base table.
DROP TRIGGER (Delete trigger)	Deletes a trigger.

Type	Function
DROP VIEW (Delete view table)	Deletes a view table.
DROP PUBLIC VIEW (Delete public view)	Deletes a public view.
GRANT CONNECT (Grant CONNECT privilege)	Grants the CONNECT privilege to users.
GRANT DBA (Grant DBA privilege)	Grants the DBA privilege to users.
GRANT RDAREA (Grant RDAREA usage privilege)	Grants the RDAREA usage privilege to users.
GRANT SCHEMA (Grant schema definition privilege)	Grants the schema definition privilege to users.
GRANT access privilege (Grant access privileges)	Grants access privileges to users.
GRANT AUDIT (Change auditor password)	Changes the auditor's password.
REVOKE CONNECT (Revoke CONNECT privilege)	Revokes previously granted CONNECT privileges.
REVOKE DBA (Revoke DBA privilege)	Revokes previously granted DBA privileges.
REVOKE RDAREA (Revoke RDAREA usage privilege)	Revokes previously granted RDAREA usage privileges.
REVOKE SCHEMA (Revoke schema definition privilege)	Revokes previously granted schema definition privileges.
REVOKE access privilege (Revoke access privileges)	Revokes previously granted access privileges.

Common rules

Upon normal execution of a definition SQL statement, a COMMIT is made simultaneously with the completion of the processing.

Notes

A definition SQL cannot be specified from an X/Open-compliant UAP running under OLTP.

ALTER INDEX (Alter index definition)

Function

ALTER INDEX changes the name of an index defined by CREATE INDEX.

Privileges

Owner of the index

A user can only specify an index owned by that user.

Format

The item numbers in the following formats correspond to the operand numbers below.

Item no.	Format
1	ALTER INDEX [*authorization-identifier*.]*index-identifier* *alter-index-definition-operation*
	alter-index-definition-operation::=*alter-index-name-definition*

- Item details

Item no.	Format
2	*alter-index-name-definition*::= RENAME INDEX TO *authorization-identifier* [WITH PROGRAM]

Operands

1) *authorization-identifier*.*index-identifier*

authorization-identifier

Specifies the authorization identifier of the owner of the index whose definition is to be changed.

index-identifier

Specifies the name of the index whose definition is to be changed.

2) *alter-index-name-definition*::=

RENAME INDEX TO *authorization-identifier* [WITH PROGRAM]

Specify this to change the index name.

index-identifier

Specifies the new name of the index.

The rules for the index identifier are as follows.

1. You can change the names of a B-tree index defined in CREATE INDEX, an index with a specified index format, and a partial structure index.
2. You cannot use a name more than once for an index in a schema, an index with a specified index format, or a partial structure index.

WITH PROGRAM

Specify to prevent the procedure for using the index or the valid SQL object of a trigger when changing the index name.

The index definition cannot be changed if there is a valid SQL object of a procedure, or trigger that uses the index, and WITH PROGRAM is omitted.

Notes

1. Although a definition SQL statement can be executed from a Java procedure, if the procedure that calls the Java procedure issues an SQL statement using an index, then issuing ALTER INDEX for the same index in the called Java procedure results in an error.
2. ALTER INDEX cannot be specified from an X/Open-compliant UAP running under OLTP.
3. To execute an SQL object of a procedure or trigger that has been disabled by specifying WITH PROGRAM, you must execute ALTER ROUTINE, ALTER PROCEDURE, or ALTER TRIGGER, and re-create the SQL object of the procedure or trigger.
4. When a valid SQL object of a procedure or trigger is disabled by specifying WITH PROGRAM, the columns related to the disabled procedure or trigger in the SQL_ROUTINE_RESOURCES, SQL_TRIGGER_USAGE, SQL_TRIGGER_COLUMNS, and SQL_ROUTINE_PARAMS tables are deleted.
5. If there is a procedure or trigger that accesses a table in which is defined an index being modified by ALTER INDEX, and that index is not used, executing ALTER INDEX renders the index information in the SQL object of the procedure or trigger unavailable for use. In this case, although the procedure that uses the unavailable index information can be executed, its performance is degraded because the index must be recompiled each time the procedure is executed. In addition, the procedure or trigger for calling the procedure that uses the disabled index information can no longer be executed. This means that ALTER ROUTINE, ALTER PROCEDURE, or ALTER TRIGGER must be executed for the procedure or trigger that uses the disabled index information and the SQL object must be re-created. To determine whether the index information is disabled, check the INDEX_VALID column in the SQL_ROUTINE dictionary table.

6. When executing the database load utility (pdload) or database reorganization utility (pdrorg) and creating an index using the index information output mode, specifying ALTER INDEX to change the index name before the index is created prevents it from being created from the completed index information file. If the index name was changed accidentally, use ALTER INDEX to change the name of the index back to its original name, and then change the name of the index again after it has been created.
7. When performing a delayed batch creation of a plug-in index, using ALTER INDEX to change the plug-in index name before the index is created prevents it from being created from the completed index information file. If the plug-in index name was changed accidentally, change the name of the plug-in index back to its original name, and then change the name of the plug-in index again after it has been created.
8. ALTER INDEX can be used to change the name of an unfinished index (index immediately after CREATE INDEX specifying EMPTY is executed).

Examples

Changes the index name from (IDX1) to (IDX2).

```
ALTER INDEX IDX1 RENAME INDEX TO IDX2
```

ALTER PROCEDURE (Re-create SQL object of procedure)

Function

ALTER PROCEDURE re-creates the SQL object for procedures or modifies the compile options for a Java procedure.

Privileges

Owner of the procedure

A user can recreate an SQL object of any procedure owned by that user (including a public procedure defined by the user).

- Only the user's own authorization identifier can be specified in the AUTHORIZATION clause.
- Only the user's own procedure can be specified in the routine identifier.
- If the AUTHORIZATION clause and the routine identifier are both omitted, an error results.

Users with the DBA privilege

A user can recreate an SQL object of any procedure owned by that user (including a public procedure defined by the user) and procedures owned by other users (including a public procedure defined by other users).

- Both the user's own authorization identifier and other users' authorization identifiers can be specified in the AUTHORIZATION clause.
- The routine identifiers of the user's own procedures and of other users' procedures can be specified.
- By omitting both the AUTHORIZATION clause and the routine identifier, all procedures in the system can be re-created.

Format

```
ALTER PROCEDURE
    [{ [authorization-identifier.]routine-identifier
     |   [AUTHORIZATION authorization-identifier]
         [ALL|INDEX USING [authorization-identifier.]
             table-identifier] }]
    [SQL-compile-option[SQL-compile-option]...
|SUBSTR LENGTH maximum-character-length ]

SQL-compile-option ::={ISOLATION data-guarantee-level [FOR UPDATE
EXCLUSIVE]
                       |OPTIMIZE LEVEL SQL-optimization-option
```

```
    [,SQL-optimization-option]
|ADD OPTIMIZE LEVEL SQL-extension-
 optimizing-options
    [,SQL-extension-optimizing-option]
```

Operands

- [*authorization-identifier*.]*routine-identifier*

Specifies a specific procedure whose SQL object is to be re-created.

The SQL object is re-created, regardless of the validity of the index information of the specified procedure or the validity of the SQL object.

This operand is used to change the SQL compile option.

authorization-identifier

Specifies the authorization identifier of the owner of the procedure whose SQL object is to be re-created.

When recreating an SQL object of a public procedure, specify `PUBLIC` in all caps enclosed in double quotation marks (") for the authorization identifier.

routine-identifier

Specifies the name of the procedure whose SQL object is to be re-created.

- [`authorization` *authorization-identifier*]

  [`ALL`|`INDEX USING` [*authorization-identifier*.]*table-identifier*]

Specifies procedures that are to be re-created in terms of the authorization identifier of the owner of the procedures and the procedures' status.

[`AUTHORIZATION` *authorization-identifier*]

Specifies the authorization identifier of the owner of a procedure and recreates the SQL objects of all procedures owned by the user (including public procedures defined by the user).

When this operand is omitted, the SQL objects of all procedures in the system are re-created.

However, whether or not all the SQL objects will actually be re-created is determined by the combination of this specification and specification of the `ALL` or `INDEX USING` clause.

authorization-identifier

Specifies the authorization identifier of the owner of the procedures whose SQL objects are to be re-created.

[`ALL`|`INDEX USING` [*authorization-identifier*.]*table-identifier*]

Specifies the status of the procedures whose SQL objects are to be re-created.

If neither the ALL nor the INDEX USING clause is specified, SQL objects are re-created for only those procedures whose SQL objects are inactive.

ALL

Specifies that all the SQL objects are to be re-created, regardless of the validity of the index information of each specified procedure or the validity of each SQL object.

INDEX USING [*authorization-identifier*.]*table-identifier*

Specifies that only the SQL objects of procedures whose index information is invalid are to be re-created.

When an index is added or deleted, the index information in the procedure's SQL object becomes invalid. Therefore, specifying the base table identifier of a table in which an index was added or deleted enables re-creation of the SQL objects of all procedures that use that table and thus have invalid index information.

A procedure can still be executed when only the index information in its SQL object is invalid. However, better performance is achieved when the index information is valid.

An SQL object is re-created also for a procedure that uses the specified base table or a view table defined using the view table as the base table, if its index information is invalid.

When the INDEX USING clause is specified, the SQL object is re-created for a procedure in which only its index information is invalid, but the SQL object is not re-created for a procedure whose SQL object is inactive. If it is necessary to re-create an SQL object for a procedure whose SQL object is inactive, either the INDEX USING clause must be omitted or ALTER PROCEDURE with ALL specified must be issued.

[*authorization-identifier*.]*table-identifier*

Specifies the authorization identifier and table identifier of a table or view table that is used by the procedures whose SQL objects are to be re-created.

If the authorization identifier is omitted, the authorization identifier of the executing user is assumed.

When specifying a public view in *table-identifier*, in *authorization-identifier* specify the word PUBLIC enclosed in double quotation marks (").

- *SQL-compile-option*::={ISOLATION *data-guarantee-level* [FOR UPDATE EXCLUSIVE]

  |OPTIMIZE LEVEL *SQL-optimization-option*

[,*SQL-optimization-option*]...

|ADD OPTIMIZE LEVEL *SQL-extension-optimizing-option*}

[,*SQL-extension-optimizing-option*

|SUBSTR LENGTH *maximum-character-length*]...

ISOLATION, OPTIMIZE LEVEL, ADD OPTIMIZE LEVEL, and SUBSTR LENGTH can each be specified only once in *SQL-compile-option*.

[ISOLATION *data-guarantee-level* [FOR UPDATE EXCLUSIVE]]

Specifies an SQL data integrity guarantee level.

data-guarantee-level

A data integrity guarantee level specifies the point to which the integrity of the transaction data must be guaranteed. The following data integrity guarantee levels can be specified:

- 0

  Do not guarantee data integrity. Specifying 0 for a set of data allows the user to reference the data even when it is being updated by another user. If the table to be referenced is a shared table, and if another use is executing the LOCK statement, a lock release wait is required.

- 1

  Guarantee the integrity of data until a retrieval process is completed. When level 1 is specified, data that has been retrieved cannot be updated by other users until the retrieval process is completed (until HiRDB finishes viewing the current page or row).

- 2

  Guarantee the integrity of data that has been retrieved until the transaction is completed. When level 2 is specified, data that has been retrieved cannot be updated by other users until the transaction is completed.

When this operand is omitted, the data guarantee level specified for the most recent creation of an SQL object (execution of a CREATE PROCEDURE, ALTER PROCEDURE, or ALTER ROUTINE statement) is assumed.

For data integrity assurance levels, see the *HiRDB Version 9 UAP Development Guide*.

[FOR UPDATE EXCLUSIVE]

Specify this option if WITH EXCLUSIVE LOCK is always assumed irrespective of the cursor in a procedure for which the FOR UPDATE clause

is specified or assumed, or for the data guarantee level on a query that is specified in *SQL-compile-option*. If level 2 is specified in *data-guarantee-level*, `WITH EXCLUSIVE LOCK` is assumed for the cursor for which the `FOR UPDATE` clause is specified or assumed, or for the query, and, therefore, `FOR UPDATE EXCLUSIVE` need not be specified. If a data guarantee level is specified in *SQL-compile-option*, and `FOR UPDATE EXCLUSIVE` is omitted, `FOR UPDATE EXCLUSIVE` need not be specified.

Relationship with client environment definition

Any specification of `PDISLLVL` or `PDFORUPDATEEXLOCK` with respect to `ALTER PROCEDURE` has no effect.

Relationship with SQL statements

If a lock option is specified in an SQL statement in a procedure, the lock option specified in the SQL statement takes precedence over any data guarantee level specified in *SQL-compile-option* or the lock option assumed because of `FOR UPDATE EXCLUSIVE`.

`OPTIMIZE LEVEL` *SQL-optimization-option*[`,` *SQL-optimization-option*]`...`

Specifies the optimization method for determining the most efficient access path by taking the database's status into consideration.

SQL optimization options can be specified with identifiers (character strings) or numeric values. Hitachi recommends that identifiers be used.

The default is the value that was adopted during the previous SQL object creation (`CREATE PROCEDURE`, `ALTER PROCEDURE`, or `ALTER ROUTINE`).

Specification with identifiers:

`OPTIMIZE LEVEL "`*identifier*`"[, "`*identifier*`"] ...`

Specification examples

- Apply the *Prioritized nest-loop-join* and the *Rapid grouping facility* options:

```
OPTIMIZE LEVEL "PRIOR_NEST_JOIN", "RAPID_GROUPING"
```

- Do not apply any optimization:

```
OPTIMIZE LEVEL "NONE"
```

Rules

1. At least one identifier must be specified.
2. When multiple identifiers are specified, separate them with the comma (`,`).
3. For details about the contents that can be specified in an identifier

(optimization methods), see *Table 3-2 SQL optimization option specification values (ALTER PROCEDURE)*.

4. If no optimization options are to be applied, specify `"NONE"` as the identifier. If `"NONE"` and some other identifier are both specified, the `"NONE"` specification is ignored.
5. The identifiers are case-sensitive.
6. If the same identifier is specified more than once, it is treated as if it was specified only once; however, where possible, precautions should be taken to avoid specifying a given identifier in duplicate.

Specification with numeric values:

```
OPTIMIZE LEVEL unsigned-integer[, unsigned-integer] ...
```

Specification examples (optimization values listed in Table 3-2)

- Apply the *2. Making multiple SQL objects*, the *8. Suppressing use of AND multiple indexes*, and the *13. Forcing use of multiple indexes* options:

  Specify unsigned integers separated by commas:

```
OPTIMIZE LEVEL 4,10,16
```

  Specify a sum of unsigned integers:

```
OPTIMIZE LEVEL 30
```

- Add the new value 16 to the previously specified value 14 (4 + 10):

```
OPTIMIZE LEVEL 14,16
```

- Do not apply any optimization:

```
OPTIMIZE LEVEL 0
```

Rules

1. When HiRDB is upgraded from a version earlier than Version 06-00 to a Version 06-00 or later, the total value specification in the earlier version also remains valid. If the optimization option does not need to be modified, the specification value for this operand need not be changed when HiRDB is upgraded to a Version 06-00 or later.
2. At least one integer must be specified.
3. When multiple integers are specified, separate them with the comma (,).
4. For details about the contents that can be specified in an unsigned integer (optimization methods), see *Table 3-2 SQL optimization option specification values (ALTER PROCEDURE)*.

5. If no optimization options are to be applied, specify `0` as the integer. If `0` and another integer are both specified, the `0` specification is ignored.
6. If the same integer is specified more than once, it will be treated as a single instance of the integer. However, multiple specifications of the same integer should be avoided.
7. When multiple optimization options are to be applied, you can specify the sum of the appropriate unsigned integers. However, the same optimization option value must not be added in more than once to avoid the possibility of the addition result being interpreted as a different set of optimization options.
8. Specifying multiple optimization options by adding their values can be ambiguous as to which optimization options are actually intended, so Hitachi recommends that the option values be specified individually separated by commas. If multiple optimization option have already been specified by the addition method and another optimization option is required, you can specify the new option's value following the previous summed value by separating them with a comma.

Relationships to system definition

1. When specified in `ALTER PROCEDURE`, the system-defined operand `pd_optimize_level` has no effect.
2. When the `pd_floatable_bes` operand or the `pd_non_floatable_bes` operand is specified, specification of the *Increasing the target floatable servers (back-end servers for fetching data)* option or the *Limiting the target floatable servers (back-end servers for fetching data)* option, respectively, is invalid.
3. When `KEY` is specified in the `pd_indexlock_mode` operand of the system definition (i.e., in the case of index key value lock), specification of the *Suppressing creation of update-SQL work tables* option is invalid.

Relationship to client environment definition

The specification of `PDSQLOPTLVL` has no applicability to `ALTER PROCEDURE`.

Relationship with SQL

If SQL optimization is specified in an SQL statement, the SQL optimization specification takes precedence over SQL optimization options. For SQL optimization specifications, see *2.24 SQL optimization specification*.

SQL optimization option specification values

The following table lists the SQL optimization option specification values. For details about optimization methods, see the *HiRDB Version 9 UAP*

Development Guide.

Table 3-2: SQL optimization option specification values (ALTER PROCEDURE)

No.	Optimization option	Specification values	
		Identifier	Unsigned integer
1	Forced nest-loop-join	"FORCE_NEST_JOIN"	4
2	Making multiple SQL objects	"SELECT_APSL"	10
3	Increasing the target floatable servers (back-end servers for fetching data)#1, #2	"FLTS_INC_DATA_BES"	16
4	Prioritized nest-loop-join	"PRIOR_NEST_JOIN"	32
5	Increasing the number of floatable server candidates#2	"FLTS_MAX_NUMBER"	64
6	Priority of OR multiple index use	"PRIOR_OR_INDEXES"	128
7	Group processing, ORDER BY processing, and DISTINCT set function processing at the local back-end server#2	"SORT_DATA_BES"	256
8	Suppressing use of AND multiple indexes	"DETER_AND_INDEXES"	512
9	Rapid grouping facility	"RAPID_GROUPING"	1024
10	Limiting the target floatable servers (back-end servers for fetching data)#1, #2	"FLTS_ONLY_DATA_BES"	2048
11	Separating data collecting servers#1, #2	"FLTS_SEPARATE_COLLECT_SVR"	2064
12	Suppressing index use (forced table scan)	"FORCE_TABLE_SCAN"	4096
13	Forcing use of multiple indexes	"FORCE_PLURAL_INDEXES"	32768
14	Suppressing creation of update-SQL work tables	"DETER_WORK_TABLE_FOR_UPDATE"	131072
15	Deriving conditions for rapid searches	"DERIVATIVE_COND"	262144
16	Applying key conditions including scalar operations	"APPLY_ENHANCED_KEY_COND"	524288
17	Facility for batch acquisition from functions provided by plug-ins	"PICKUP_MULTIPLE_ROWS_PLUGIN"	1048576
18	Facility for moving search conditions into derived table	"MOVE_UP_DERIVED_COND"	2097152

#1: If the *3. Increasing the target floatable servers (back-end servers for fetching data)* option and the *10. Limiting the target floatable servers (back-end servers for fetching data)* option are both specified, neither of these options will be applied; instead, the *11 Separating data collecting servers* option will be applied.

#2: This option is ignored if specified for a HiRDB/Single Server

ADD OPTIMIZE LEVEL
SQL-extension-optimizing-option[,*SQL-extension-optimizing-option*]...

Specifies optimizing options for determining the most efficient access path, taking into consideration the status of the database.

SQL extension optimizing options can be specified with identifiers (character strings) or numeric values. Hitachi recommends that identifiers be used.

The default is the value that was used during the previous SQL object creation (CREATE PROCEDURE, ALTER PROCEDURE, or ALTER ROUTINE).

Specification with identifiers:

ADD OPTIMIZE LEVEL *identifier*[, *identifier*] ...

Specification examples

- Apply the *Application of optimizing mode 2 based on cost* and *Hash join, subquery hash execution* options:

```
ADD OPTIMIZE LEVEL
        "COST_BASE_2","APPLY_HASH_JOIN"
```

- Do not apply any optimizing:

```
ADD OPTIMIZE LEVEL "NONE"
```

Rules

1. At least one identifier must be specified.
2. When multiple identifiers are specified, separate them with the comma (,).
3. For details about the contents that can be specified in an identifier (optimization methods), see *Table 3-3 SQL extension optimizing option specification values (ALTER PROCEDURE)*.
4. If no extension optimizing options are to be applied, specify "NONE" as the identifier.
5. The identifiers are case-sensitive.
6. If the same identifier is specified more than once, it is treated as if it was specified only once; however, where possible, precautions should be taken to avoid specifying a given identifier in duplicate.

Specification with numeric values:

```
ADD OPTIMIZE LEVEL unsigned-integer[, unsigned-integer] ...
```

Specification examples

- Apply the *Application of optimizing mode 2 based on cost* and *Hash join, subquery hash execution* options:

```
ADD OPTIMIZE LEVEL 1,2
```

- Do not apply any optimizing:

```
ADD OPTIMIZE LEVEL 0
```

Rules

1. At least one integer must be specified.
2. When multiple integers are specified, separate them with the comma (,).
3. For details about the contents that can be specified in an unsigned integer (optimization methods), see *Table 3-3 SQL extension optimizing option specification values (ALTER PROCEDURE)*.
4. If no extension optimizing options are to be applied, specify `0` as the integer.
5. If the same unsigned integer is specified more than once, it is treated as if it was specified only once; however, where possible, precautions should be taken to avoid specifying a given unsigned integer in duplicate.

Relationship to system definition

The system-defined `pd_optimize_level` operand, if specified in `ALTER PROCEDURE`, has no effect.

Relationship to client environment definition

The specification of `PDADDITIONALOPTLVL` has no applicability to `ALTER PROCEDURE`.

Relationship with SQL

If SQL optimization is specified in an SQL statement, the SQL optimization specification takes precedence over SQL optimization options. For SQL optimization specifications, see *2.24 SQL optimization specification*.

SQL extension optimizing option specification values

The following table lists the SQL extension optimizing option specification values. For details about optimization methods, see the *HiRDB Version 9 UAP Development Guide*.

Table 3-3: SQL extension optimizing option specification values (ALTER PROCEDURE)

No.	Optimizing option	Specification values	
		Identifier	Unsigned integer
1	Application of optimizing mode 2 based on cost	"COST_BASE_2"	1
2	Hash join, subquery hash execution	"APPLY_HASH_JOIN"	2
3	Facility for applying join conditions including value expression	"APPLY_JOIN_COND_FOR_VALUE_EXP"	32

Note

Items 2-3 take effect when *Application of optimizing mode 2 based on cost* is specified.

[SUBSTR LENGTH *maximum-character-length*]

Specifies the maximum number of bytes for representing a single character.

The value specified for the maximum character length must be in the range from 3 to 6.

This operand is valid only when utf-8 is specified for the character code type in the pdntenv command (pdsetup command for the UNIX edition); it affects the length of the result of the SUBSTR scalar function. For details about SUBSTR, see *2.16.1(20) SUBSTR*.

Rules

When HiRDB is upgraded from a version earlier than version 08-00 to version 08-00 or later, 3 is assumed. If there is no need to change the maximum character length, you do not need to specify this operand when upgrading to HiRDB version 08-00 or later.

Relationships to system definition

When SUBSTR LENGTH is specified in ALTER PROCEDURE, the pd_substr_length system definition operand has no effect. For details about the pd_substr_length operand, see the manual *HiRDB Version 9 System Definition*.

Relationship to client environmental definition

The specification of PDSUBSTREN has no applicability to ALTER PROCEDURE. For details about PDSUBSTRLEN, see the manual *HiRDB Version 9 UAP Development Guide*.

Relationship to the character code type specified in the pdntenv **or** pdsetup

command

This operand is valid only when utf-8 is specified for the character code type.

For all other character code types, only a syntax check is performed and the specification is ignored.

When this operand is omitted, the value specified during creation of the most recent SQL object (execution of a CREATE PROCEDURE, ALTER PROCEDURE, or ALTER ROUTINE statement) is assumed.

Common rules

1. When the SQL compile option is specified in ALTER PROCEDURE, the length of the SQL statement that is created by incorporating the SQL compile option in the source CREATE PROCEDURE for the procedure to be re-created must not exceed the maximum allowable length for SQL statements.
2. The ALTER PROCEDURE cannot be executed from a Java procedure when the SQL object being executed can be re-created.

Notes

1. The ALTER PROCEDURE statement cannot be specified from an X/Open-compliant UAP running under OLTP.
2. When SQL objects for multiple procedures are being re-created, a COMMIT or ROLLBACK statement is executed automatically for each procedure.
3. By executing a GET DIAGNOSTICS statement immediately following execution of an ALTER PROCEDURE statement, diagnostic information can be obtained for the ALTER PROCEDURE statement. The return code for the SQL object of a procedure whose re-creation terminated normally is 0.
4. The data guarantee level of the SQL statement in the procedure, the SQL optimization option, the SQL extension optimizing option, and the maximum character length are determined by what is specified when the routine is being defined or re-created, and are not affected by the system definition or client environment variable definition that is in effect when the procedure is called.
5. Because no SQL object is created, the Java procedure and the Java function cannot re-create the SQL object; they can only update the SQL compile option.
6. The identifier for a trigger action procedure cannot be specified in *routine-identifier*. When recreating a trigger SQL object, use either ALTER ROUTINE or ALTER TRIGGER.
7. Even if you re-create an SQL object of a public procedure defined by another user, the definer of the SQL object does not change.

Examples

1. Re-create with data guarantee level 1 a procedure (`PROC1`) belonging to a user (`USER1`):
```
ALTER PROCEDURE
   USER1.PROC1 ISOLATION 1
```

2. Of the active procedures of a user (`USER1`) that reference a table (`T1`) belonging to that user, re-create those procedures whose SQL objects contain invalid index information:
```
ALTER PROCEDURE
   AUTHORIZATION USER1 INDEX USING USER1.T1
```

3. Of all procedures, re-create those procedures that contain inactive SQL objects:
```
ALTER PROCEDURE
```

ALTER ROUTINE (Re-create SQL objects for functions, procedures, and triggers)

Function

ALTER ROUTINE re-creates the SQL objects for functions, procedures, and triggers, or modifies the compile option for a Java function or procedure.

Privileges

Owner of the functions, procedures, and triggers

A user can recreate a function, procedure (including public functions and public procedures defined by the user), or SQL object of a trigger owned by that user.

- Only the user's own authorization identifier can be specified in the AUTHORIZATION clause.
- If the AUTHORIZAT ION clause is omitted, an error results.

Users with the DBA privilege

A user can recreate a procedure (including public functions and public procedures defined by the user) or trigger owned by that user as well as a function, procedure (including public functions and public procedures defined by the user), or SQL object of a trigger owned by other users.

- Both the user's own authorization identifier and other users' authorization identifiers can be specified in the AUTHORIZATION clause.
- By omitting the AUTHORIZATION clause, SQL objects for all functions and procedures in the system are re-created.

Format

```
ALTER ROUTINE [[AUTHORIZATION authorization-identifier] [ALL]]
    [SQL-compile-option [SQL-compile-option]...]
SQL-compile-option ::={ISOLATION data-guarantee-level [FOR UPDATE
EXCLUSIVE]
        |OPTIMIZE LEVEL SQL-optimization-option
                        [,SQL-optimization-option]...
        |ADD OPTIMIZE LEVEL SQL-extension-optimizing-option
                        [,SQL-extension-optimizing-option]...
                  |SUBSTR LENGTH maximum-character-length }
```

Operands

- [AUTHORIZATION *authorization-identifier*] [ALL]

Specifies the function, procedure, and trigger to be re-created in terms of the owner's

authorization identifier and the status of the function, procedure, and trigger.

[AUTHORIZATION *authorization-identifier*]

Specifies the authorization identifier of the owner of a function, procedure, or trigger (including public functions and public procedures defined by the user), and recreates the SQL objects of all functions, procedures, and triggers owned by that user. The default for this operand is to re-create the SQL objects for all the functions, procedures, and triggers in the system. Whether an SQL object is actually re-created is determined by the particular combination of the operands with the ALL clause.

authorization-identifier

Specifies the authorization identifier of the owner of functions, procedures, and triggers that are to be re-created.

[ALL]

Specifies that the status of the SQL objects is to be taken into consideration in determining which SQL objects for functions, procedures, and triggers are to be re-created. When this operand is omitted, only functions, procedures, and triggers whose SQL object is disabled are re-created.

ALL

Specifies that the SQL objects for all functions, procedures, and triggers are to be re-created, regardless of whether the SQL objects are enabled or disabled.

- *SQL-compile-option* ::= {ISOLATION *data-guarantee-level* [FOR UPDATE EXCLUSIVE]

  |OPTIMIZE LEVEL *SQL-optimization-option*

  [, *SQL-optimization-option*] . . .

  |ADD OPTIMIZE LEVEL *SQL-extension-optimizing-option*

  [, *SQL-extension-optimizing-option*] . . .

  |SUBSTR LENGTH *maximum-character-length*}

ISOLATION, OPTIMIZE LEVEL, ADD OPTIMIZE LEVEL, and SUBSTR LENGTH can each be specified only once in *SQL-compile-option*.

[ISOLATION *data-guarantee-level* [FOR UPDATE EXCLUSIVE]]

Specifies an SQL data integrity guarantee level.

data-guarantee-level

A data integrity guarantee level specifies the point to which the integrity of the transaction data must be guaranteed. The following data integrity

guarantee levels can be specified:

- 0

  Do not guarantee data integrity. When 0 is specified for a set of data, the user can reference the data even while it is being updated by another user. If the table to be referenced is a shared table, and if another user is executing the LOCK statement, a lock release wait is required.

- 1

  Guarantee the integrity of data until a retrieval process is completed. When level 1 is specified, data that has been retrieved cannot be updated until the retrieval process is completed (until HiRDB finishes viewing the current page or row).

- 2

  Guarantees the integrity of data that has been retrieved until the transaction is completed. When level 2 is specified, data that has been retrieved cannot be updated by other users until the transaction has been completed.

[FOR UPDATE EXCLUSIVE]

Specify this option if WITH EXCLUSIVE LOCK is always assumed irrespective of the cursor in a procedure for which the FOR UPDATE clause is specified or assumed, or for the data guarantee level on a query that is specified in *SQL-compile-option*. If level 2 is specified in *data-guarantee-level*, WITH EXCLUSIVE LOCK is assumed for the cursor for which the FOR UPDATE clause is specified or assumed, or for the query, and, therefore, FOR UPDATE EXCLUSIVE need not be specified. If a data guarantee level is specified in *SQL-compile-option*, and FOR UPDATE EXCLUSIVE is omitted, FOR UPDATE EXCLUSIVE need not be specified.

Relationship with client environment definition

Any specification of PDISLLVL or PDFORUPDATEEXLOCK with respect to ALTER ROUTINE has no effect.

Relationship with SQL statements

If a lock option is specified in an SQL statement in a procedure, the lock option specified in the SQL statement takes precedence over any data guarantee level specified in *SQL-compile-option* or the lock option assumed because of FOR UPDATE EXCLUSIVE.

When this operand is omitted, the data guarantee level specified for the most recent SQL object creation (execution of a CREATE PROCEDURE, ALTER PROCEDURE, CREATE TYPE, ALTER ROUTINE, CREATE TRIGGER, or ALTER TRIGGER statement) is assumed.

For data guarantee levels, see the *HiRDB Version 9 UAP Development Guide*.

`OPTIMIZE LEVEL` *SQL-optimization-option*[`,` *SQL-optimization-option*]`...`

Specifies the optimization method for determining the most efficient access path, taking into account the database's status.

SQL optimization options can be specified with identifiers (character strings) or numeric values. Hitachi recommends that identifiers be used.

The default is the value that was used during the previous creation of an SQL object (`CREATE PROCEDURE`, `ALTER PROCEDURE`, `CREATE TYPE`, `ALTER ROUTINE`, `CREATE TRIGGER`, or `ALTER TRIGGER`).

Specification with identifiers:

`OPTIMIZE LEVEL "`*identifier*`"[, "`*identifier*`"] ...`

Specification examples

- Apply the *Prioritized nest-loop-join* and the *Rapid grouping facility* options:

```
OPTIMIZE LEVEL "PRIOR_NEST_JOIN","RAPID_GROUPING"
```

- Do not apply any optimization:

```
OPTIMIZE LEVEL "NONE"
```

Rules

1. At least one identifier must be specified.
2. When multiple identifiers are specified, separate them with the comma (`,`).
3. For details about the contents that can be specified in an identifier (optimization methods), see *Table 3-4 SQL optimization option specification values (ALTER ROUTINE)*.
4. If no optimization options are to be applied, specify `"NONE"` as the identifier. If `"NONE"` and some other identifier are both specified, the `"NONE"` specification is ignored.
5. The identifiers are case-sensitive.
6. If the same identifier is specified more than once, it is treated as if it was specified only once; however, where possible, precautions should be taken to avoid specifying a given identifier in duplicate.

Specification with numeric values:

`OPTIMIZE LEVEL` *unsigned-integer*`[,` *unsigned-integer*`] ...`

Specification examples

- Apply the *2. Making multiple SQL objects*, *8. Suppressing use of AND multiple indexes*, and *13. Forcing use of multiple indexes* options:

  Specify unsigned integers separated by commas:

```
OPTIMIZE LEVEL 4,10,16
```

Specify a sum of unsigned integers:

```
OPTIMIZE LEVEL 30
```

- Add the new value 16 to the previously specified value 14 (4 + 10):

```
OPTIMIZE LEVEL 14,16
```

- Do not apply any optimization:

```
OPTIMIZE LEVEL 0
```

Rules

1. When HiRDB is upgraded from a version earlier than Version 06-00 to a Version 06-00 or later, the total value specification in the earlier version also remains valid. If the optimization option does not need to be modified, the specification value for this operand need not be changed when HiRDB is upgraded to a Version 06-00 or later.
2. At least one integer must be specified.
3. When multiple integers are specified, separate them with the comma (,).
4. For details about the contents that can be specified in an unsigned integer (optimization methods), see *Table 3-4 SQL optimization option specification values (ALTER ROUTINE)*.
5. If no optimization options are to be applied, specify `0` as the integer. If `0` and another integer are both specified, the `0` specification is ignored.
6. If the same integer is specified more than once, it will be treated as a single instance of the integer. However, multiple specifications of the same integer should be avoided.
7. When multiple optimization options are to be applied, you can specify the sum of the appropriate unsigned integers. However, the same optimization option value must not be added in more than once to avoid the possibility of the addition result being interpreted as a different set of optimization options.
8. If multiple optimization methods are specified by adding values, which optimization methods are being specified may not be apparent. In this case, Hitachi recommends specifying values by delimiting them with commas. If multiple optimization methods are already specified by adding values, and a new optimization method must be specified, the

additional value can be appended by delimiting it with a comma.

Relationships to system definition

1. The system-defined pd_optimize_level operand, if specified in ALTER ROUTINE, has no effect.
2. When the pd_floatable_bes operand or the pd_non_floatable_bes operand is specified, specification of the *Increasing the target floatable servers (back-end servers for fetching data)* option or the *Limiting the target floatable servers (back-end servers for fetching data)* option, respectively, is invalid.
3. When KEY is specified in the pd_indexlock_mode operand of the system definition (i.e., in the case of index key value lock), specification of the *Suppressing creation of update-SQL work tables* option is invalid.

Relationship to client environment definition

The specification of PDSQLOPTLVL has no applicability to ALTER ROUTINE.

Relationship with SQL

If SQL optimization is specified in an SQL statement, the SQL optimization specification takes precedence over SQL optimization options. For SQL optimization specifications, see *2.24 SQL optimization specification*.

SQL optimization option specification values

The following table lists the SQL optimization option specification values. For details about the optimization option, see the manual *HiRDB Version 9 UAP Development Guide*.

Table 3-4: SQL optimization option specification values (ALTER ROUTINE)

No.	Optimization option	Specification values	
		Identifier	Unsigned integer
1	Forced nest-loop-join	FORCE_NEXT_JOIN	4
2	Making multiple SQL objects	SELECT_APSL	10
3	Increasing the target floatable servers (back-end servers for fetching data)[#1, #2]	FLTS_INC_DATA_BES	16
4	Prioritized nest-loop-join	PRIOR_NEST_JOIN	32
5	Increasing the number of floatable server candidates[#2]	FLTS_MAX_NUMBER	64
6	Priority of OR multiple index use	PRIOR_OR_INDEXES	128

No.	Optimization option	Specification values	
		Identifier	Unsigned integer
7	Group processing, ORDER BY processing, and DISTINCT set function processing at the local back-end server#2	SORT_DATA_BES	256
8	Suppressing use of AND multiple indexes	DETER_AND-INDEXES	512
9	Rapid grouping facility	RAPID_GROUPING	1024
10	Limiting the target floatable servers (back-end servers for fetching data)#1, #2	FLTS_ONLY_DATA_BES	2048
11	Separating data collecting servers#1, #2	FLTS_SEPARATE_COLLECT_SVR	2064
12	Suppressing index use (forced table scan)	FORCE_TABLE_SCAN	4096
13	Forcing use of multiple indexes	FORCE_PLURAL_INDEXES	32768
14	Suppressing creation of update-SQL work tables	DETER_WORK_TABLE_FOR_UPDATE	131072
15	Deriving rapid retrieval conditions	"DERIVATIVE_COND"	262144
16	Applying key conditions including scalar operations	"APPLY_ENHANCED_KEY_COND"	524288
17	Facility for batch acquisition from functions provided by plug-ins	"PICKUP_MULTIPLE_ROWS_PLUGIN"	1048576
18	Facility for moving search conditions into derived table	"MOVE_UP_DERIVED_COND"	2097152

#1: If the *3. Increasing the target floatable servers (back-end servers for fetching data)* option and the *10. Limiting the target floatable servers (back-end servers for fetching data)* option are both specified, neither of these options will be applied; instead, the *11. Separating data collecting servers* option will be applied.

#2: This option is ignored if specified for a HiRDB/Single Server.

ADD OPTIMIZE LEVEL *SQL-extension-optimizing-option*[, *SQL-extension-optimizing-option*]...

Specifies optimizing options for determining the most efficient access path, taking into consideration the status of the database.

SQL extension optimizing options can be specified with identifiers (character strings) or numeric values. Hitachi recommends that identifiers be used.

The default is to use value that was used during the previous creation of an SQL object (CREATE PROCEDURE, ALTER PROCEDURE, CREATE TYPE, ALTER ROUTINE, CREATE TRIGGER, or ALTER TRIGGER).

Specification with identifiers:

```
ADD OPTIMIZE LEVEL "identifier"[, "identifier"] ...
```

Specification examples

- Apply the *Application of optimizing mode 2 based on cost* and *Hash join, subquery hash execution* options:

```
ADD OPTIMIZE LEVEL
        "COST_BASE_2", "APPLY_HASH_JOIN"
```

- Do not apply any optimizing:

```
ADD OPTIMIZE LEVEL "NONE"
```

Rules

1. At least one identifier must be specified.
2. When multiple identifiers are specified, separate them with the comma (,).
3. For details about the contents that can be specified in an identifier (optimization methods), see *Table 3-5 SQL extension optimizing option specification values (ALTER ROUTINE)*.
4. If no extension optimizing options are to be applied, specify "NONE" as the identifier.
5. The identifiers are case-sensitive.
6. If the same identifier is specified more than once, it is treated as if it was specified only once; however, where possible, precautions should be taken to avoid specifying a given identifier in duplicate.

Specification with numeric values:

```
ADD OPTIMIZE LEVEL unsigned-integer[, unsigned-integer] ...
```

Specification examples

- Apply the *Application of optimizing mode 2 based on cost* and *Hash join, subquery hash execution* options:

```
ADD OPTIMIZE LEVEL 1,2
```

- Do not apply any optimizing:

```
ADD OPTIMIZE LEVEL 0
```

Rules

1. At least one integer must be specified.
2. When multiple integers are specified, separate them with the comma (,).
3. For details about the contents that can be specified in an unsigned integer (optimization methods), see *Table 3-5 SQL extension optimizing option specification values (ALTER ROUTINE)*.
4. If no extension optimizing options are to be applied, specify `0` as the integer.
5. If the same integer is specified more than once, it will be treated as a single instance of the integer. However, multiple specifications of the same integer should be avoided.

Relationship to system definition

The system-defined `pd_optimize_level` operand, if specified in `ALTER ROUTINE`, has no effect.

Relationship to client environment definition

The specification of `PDADDITIONALOPTLVL` has no applicability to `ALTER ROUTINE`.

Relationship with SQL

If SQL optimization is specified in an SQL statement, the SQL optimization specification takes precedence over SQL optimization options. For SQL optimization specifications, see *2.24 SQL optimization specification*.

SQL extension optimizing option specification values

The following table lists the SQL extension optimizing option specification values. For details about optimization methods, see the *HiRDB Version 9 UAP Development Guide*.

Table 3-5: SQL extension optimizing option specification values (ALTER ROUTINE)

No.	Optimizing option	Specification values	
		Identifier	Unsigned integer
1	Application of optimizing mode 2 based on cost	`"COST_BASE_2"`	`1`
2	Hash join, subquery hash execution	`"APPLY_HASH_JOIN"`	`2`
3	Facility for applying join conditions including value expression	`"APPLY_JOIN_COND_FOR_VALUE_EXP"`	`32`

Note

Items 2-3 take effect when *Application of optimizing mode 2 based on cost* is specified.

[SUBSTR LENGTH *maximum-character-length*]

Specifies the maximum number of bytes for representing a single character.

The value specified for the maximum character length must be in the range from 3 to 6.

This operand is valid only when utf-8 is specified for the character code type in the pdntenv command (pdsetup command for the UNIX edition), it affects the length of the result of the SUBSTR scalar function. For details about SUBSTR, see *2.16.1(20) SUBSTR*.

Rules

When HiRDB is upgraded from a version earlier than version 08-00 to version 08-00 or later, 3 is assumed. If there is no need to change the maximum character length, you do not need to specify this operand when upgrading to HiRDB version 08-00 or later.

Relationships to system definition

When SUBSTR LENGTH is specified in ALTER ROUTINE, the pd_substr_length system definition operand has no effect. For details about the pd_substr_length operand, see the manual *HiRDB Version 9 System Definition*.

Relationship to client environmental definition

The specification of PDSUBSTREN has no applicability to ALTER ROUTINE. For details about PDSUBSTRLEN, see the manual *HiRDB Version 9 UAP Development Guide*.

Relationship to the character code type specified in the pdntenv or pdsetup command

This operand is valid only when utf-8 is specified for the character code type.

For all other character code types, only a syntax check is performed and the specification is ignored.

When this operand is omitted, the value specified during creation of the most recent SQL object (execution of a CREATE PROCEDURE, ALTER PROCEDURE, CREATE FUNCTION, CREATE TYPE, ALTER ROUTINE, CREATE TRIGGER, or ALTER TRIGGER statement) is assumed.

Common rules

1. When an SQL compile option is specified in ALTER ROUTINE, the SQL statement

created by incorporating the SQL compile option into the original CREATE PROCEDURE, CREATE FUNCTION, CREATE TYPE, or CREATE TRIGGER statement of the routine to be re-created must be of a length that does not exceed the maximum allowable length for SQL statements.

2. Specification of the SQL compile option is valid only for procedures or triggers; it has no effect on functions.
3. The ALTER ROUTINE cannot be executed from a Java procedure when the SQL object being executed can be re-created.

Notes

1. ALTER ROUTINE cannot be specified from an X/Open-compliant UAP running under OLTP.
2. When SQL objects for multiple functions, procedures, and triggers are re-created, the COMMIT or ROLLBACK statement is executed for each function, procedure, and trigger.
3. By executing a GET DIAGNOSTICS statement immediately following execution of an ALTER ROUTINE statement, diagnostic information can be obtained for the ALTER ROUTINE statement. The return code for the SQL object of a function, procedure, or trigger whose re-creation terminated normally is 0.
4. The data guarantee level of the SQL optimization option, the SQL extension optimizing option, and the maximum character length are determined by what is specified when the routine or trigger is being defined or re-created, and are not affected by the system definition or client environment variable definition that is in effect when the procedure or function is called or when the trigger operation is executed.
5. If an index is added or deleted to or from a table (exclusive of the table in which the trigger is defined) that is used in a procedure or trigger, any index information in the SQL object for the procedure and the trigger is nullified, in which case the affected trigger cannot be executed. Because the procedure cannot be executed from another procedure or trigger, the SQL object needs to be re-created by specifying ALL.
6. Because no SQL object is created, the Java procedure or Java function cannot re-create the SQL object; it can only update the SQL compile option.
7. If triggers must be nested, care should be taken with the following items:
   - If some or all of the nested triggers are disabled, a single execution of ALTER ROUTINE may not be able to put all the triggers in effect (ALTER ROUTINE results in a KFPA11528-E error). In this case, execute ALTER ROUTINE repeatedly until ALTER ROUTINE terminates normally.
8. If there are triggers to be nested, and if a function is specified in a search condition

during the operation of the triggers, care should be taken with the following item:

- If this function is deleted, a KFPA11529-E error may occur during the execution of the trigger, even when ALTER ROUTINE terminates normally. In this case, re-execute the trigger or the routine by recreating it (ALTER TRIGGER or ALTER PROCEDURE). The trigger or routine that caused the runtime error is the calling trigger, or routine or the trigger from which the function was deleted.

9. If there are looping triggers, care should be taken with the following item:
   - If all looping triggers are disabled, they cannot be re-created by executing ALTER ROUTINE. In this case, delete all the looping triggers and the triggers that are in the table defining the looping triggers. You need to re-define them.
10. Even if you re-create an SQL object of a public function or public procedure defined by another user, the definer of the SQL object does not change.

Examples

1. Re-create among all functions, procedures, and triggers those functions and procedures for which the SQL object has been nullified:
```
ALTER ROUTINE
```

2. Re-create all the functions, procedures, and triggers belonging to a user (USER1):
```
ALTER ROUTINE
   AUTHORIZATION USER1 ALL
```

ALTER TABLE (Alter table definition)

Function

ALTER TABLE has the following functions:

- Appends a new column to a base table.
- Adds a table storage RDAREA to a base table that is row-partitioned by a hash function.
- Changes the attribute of a base table or column.
  - Increases the maximum length of the variable-length data type.
  - Changes character data to mixed character string data.
  - Increases the maximum number of elements in a repetition column.
  - Changes the method of storing variable-length character data.
  - Changes the column recovery restriction.
  - Sets up, changes, or deletes a predefined value for a column.
  - Changes a NOT NULL constrained column without a predefined value to a NOT NULL constrained column with a predefined value.
  - Changes the updatable column attribute.
  - Changes the uniqueness constraint of a cluster key of a base table in which no data is stored.
  - Changes the unit of the minimum locked resource of a base table.
  - Changes the hash function of a base table that is row-partitioned by a hash function.
  - Enables or disables application of the free space reusage facility.
  - Changes the segment count upper limit for the free space reusage facility.
  - Changes to a falsification prevented table.
- Deletes base table columns that are not storing any data.
- Changes the name of a base table or column.
- Changes the partitioning storage conditions for row partitioned base tables and matrix partitioned base tables.

Privileges

Owner of a base table

This user can specify this command only on his or her own tables.

Format

The item numbers in the following format correspond with the operand numbers:

No.	Format
1	ALTER TABLE [*authorization-identifier*.]*table-identifier*
	table-definition-change::={ *column-addition-definition* |*RDAREA-addition-definition* |*column-attribute-change-definition* |*column-deletion-definition* |*table-name-change-definition* |*column-name-change-definition* |*partitioning-storage-condition-change-definition* }

- Details of individual items

No.	Format
2	*column-addition-definition*::= ADD *column-name-data-type*[ARRAY [*maximum-number-of-elements*]] [NO SPLIT] [[*column-recovery-restriction-1*] {*LOB-column-storage-RDAREA-specification* |*matrix-partitioned-LOB-column-storage-RDAREA-specification* |*abstract-data-type-definition-LOB-storage-RDAREA-specification* [*plug-in-specification*] |*matrix-partitioned-LOB-attribute-storage-RDAREA-specification* [*plug-in-specification*]}] [DEFAULT *clause*] {[NULL|NOT NULL [WITH DEFAULT]]#1 |[[NOT NULL] WITH DEFAULT]#2} [*updatable-column-attribute*] [WITH PROGRAM]
	column-recovery-restriction-1::= RECOVERY[[ALL | PARTIAL | NO]
	LOB-column-storage-RDAREA-specification::= IN [*LOB-column-storage-RDAREA-name* |(*LOB-column-storage-RDAREA-name*) |((*LOB-column-storage-RDAREA-name*) [,(*LOB-column-storage-RDAREA-name*)]...)]
	matrix-partitioned-LOB-column-storage-RDAREA-specification::= *2-dimension-storage-RDAREA-specification* *matrix-partitioned-LOB-attribute-storage-RDAREA-specification*::= *2-dimension-storage-RDAREA-specification* *2-dimension-storage-RDAREA-specification*::=(*matrix-partitioning-RDAREA-list* [, *matrix-partitioning-RDAREA-list*]...) *matrix-partitioning-RDAREA-list*::= (*RDAREA-name*[,*RDAREA-name*]...)

No.	Format
	abstract-data-type-LOB-storage-RDAREA-specification::= ALLOCATE(*attribute-name*[... *attribute-name*]... IN [*LOB-attribute-storage-RDAREA-name* |(*LOB-attribute-storage-RDAREA-name*) | ((*LOB-attribute-storage-RDAREA-name*) [,(*LOB-attribute-storage-RDAREA-name*)]...)} [, *attribute-name* [.. *attribute-name*]... IN {*LOB-attribute-storage-RDAREA-name* | (*LOB-attribute-storage-RDAREA-name*) | ((*LOB-attribute-storage-RDAREA-name*) [, (*LOB-attribute-storage-RDAREA-name*)]...)}]...)
	DEFAULT *clause*::= DEFAULT [*default-value*] *default-value*::= {*literal*|USER | CURRENT_DATE | CURRENT DATE |CURRENT_TIME | CURRENT TIME | CURRENT_TIMESTAMP [(*fractional-second-precision*)] [USING BES] | CURRENT TIMESTAMP [(*fractional-second-precision*)] [USING BES] |NULL}
	updatable-column-attribute::= UPDATE [ONLY FROM NULL]
3	*RDAREA-addition-definition*::= ADD RDAREA *table-storage-RDAREA-name* [FOR COLUMN *column-name* {*LOB-column-storage-RDAREA-specification* |*abstract-date-type-definition-LOB-storage-RDAREA-specification*} [, *column-name* {*LOB-column-storage-RDAREA-specification* |*abstract-date-type-definition-LOB-storage-RDAREA-specification*}]...] [FOR INDEX *index-identifier index-storage-RDAREA-specification* [, *index-identifier index-storage-RDAREA-specification*]...] [FOR [PRIMARY] CLUSTER KEY *index-storage-RDAREA-specification*] [FOR PRIMARY KEY *index-storage-RDAREA-specification*] [WITH PROGRAM]
	LOB-column-storage-RDAREA-specification::= IN {*LOB-column-storage-RDAREA-name* | (*LOB-column-storage-RDAREA-name*) | ((*LOB-column-storage-RDAREA-name*) [, (*LOB-column-storage-RDAREA-name*)]...)}
	abstract-data-type-LOB-storage-RDAREA-specification::= ALLOCATE (*attribute-name*-[.. *attribute-name*]... IN {*LOB-attribute-storage-RDAREA-name* | (*LOB-attribute-storage-RDAREA-name*) | ((*LOB-attribute-storage-RDAREA-name*) [, (*LOB-attribute-storage-RDAREA-name*)]...)} [, *attribute-name* [,,, *attribute-name*]... IN {*LOB-attribute-storage-RDAREA-name* | (*LOB-attribute-storage-RDAREA-name*) | ((*LOB-attribute-storage-RDAREA-name*) [, (*LOB-attribute-storage-RDAREA-name*)]...)}]...)

No.	Format
	index-storage-RDAREA-specification::= IN {*index-storage-RDAREA-name* | (*index-storage-RDAREA-name*) | ((*index-storage-RDAREA-name*) [, (*index-storage-RDAREA-name*)]...)}
4	*column-attribute-change-definition*::= CHANGE{*column-name*{[{VARCHAR (*data-length*) |NVARCHAR (*data-length*) |MCHAR({*|*data-length*}) |MVARCHAR({*|*data-length*})} [ARRAY [{*|*maximum-number-of-elements*}]]] |[ARRAY [*maximum-number-of-elements*]] |BINARY(*data-length*)} [{NO SPLIT|SPLIT}]] [*column-recovery-restriction-2*] [{SET DEFAULT *clause*|DROP DEFAULT}] [WITH DEFAULT] [*updatable-column-attribute*] |CLUSTER KEY[UNIQUE] |LOCK{ROW|PAGE} |HASH *hash-function-name* |SEGMENT REUSE {[*number-of-segments*[{K|M|G}]]|NO}} | INSERT ONLY [WHILE {*date-interval-data*| *labeled-interval*} BY *column-name*]} [WITH PROGRAM]
	column-recovery-restriction-2::= RECOVERY{ALL|PARTIAL|NO}
5	*column-deletion-definition*::= DROP *column-name* [WITH PROGRAM]
6	*table-name-change-definition*::= RENAME TABLE TO *table-identifier* [WITH PROGRAM]
7	*column-name-change-definition*::= RENAME COLUMN FROM *pre-change-column-name* TO *post-change-column-name* [WITH PROGRAM]
8	*partitioning-storage-condition-change-definition*::= CHANGE RDAREA {*row-partitioned-table-change-specification*|*matrix-partitioned-table-change-specification*} [*LOB-column-storage-RDAREA-change-specification*] [*index-storage-RDAREA-change-specification* [*index-storage-RDAREA-change-specification*]...] [*cluster-key-storage-RDAREA-change-specification*] [*primary-key-storage-RDAREA-change-specification*] [WITHOUT PURGE] [WITH PROGRAM]

No.	Format
8-1	*row-partitioned-table-change-specification*::= {[PARTITIONED] *pre-change-boundary-value-list* INTO *post-change-boundary-value-partition-specification* |PARTITIONED CONDITION *pre-change-RDAREA-information-list* INTO *post-change-storage-condition-partition-specification*} *matrix-partitioned-table-change-specification*::= MULTIDIM(*change-target-column-name pre-change-boundary-value-list*) AT *post-change-boundary-value-list* INTO *matrix-partitioned-table-storage-RDAREA-change-specification*
8-2	*pre-change-boundary-value-list*::= *boundary-value-list* *post-change-boundary-value-list*::=*boundary-value-list* *boundary-value-list*::= (({*boundary-value*|MAX})[, ({*boundary-value*|MAX})]...)
8-3	*post-change-boundary-value-partitioning-specification*::= {*table-storage-RDAREA-name* | (*table-storage-RDAREA-name*) | ([(*table-storage-RDAREA-name*) *boundary-value*,] ...(*table-storage-RDAREA-name*))}
8-4	*pre-change-RDAREA-information-list*::= {*name-of-RDAREA-for-table* |(*name-of-RDAREA-for-table*) |((*name-of-RDAREA-for-table*) [,(*name-of-RDAREA-for-table*)]...[,OTHERS]) |OTHERS}
8-5	*pre-change-RDAREA-information-list*::= {*table-storage-RDAREA-name* |(*table-storage-RDAREA-name*) |((*table-storage-RDAREA-name*) [,(*table-storage-RDAREA-name*)]...[,OTHERS]) |OTHERS}
8-6	*index-storage-RDAREA-change-specification*::= FOR INDEX *index-name* INTO *post-change-index-storage-RDAREA-name-list* *post-change-index-storage-RDAREA-name-list*::= *RDAREA-name-list* *index-name*::= {*index-storage-RDAREA-name* | (*index-storage-RDAREA-name*) | ((*index-storage-RDAREA-name*) [, (*index-storage-RDAREA-name*)]...[,OTHERS]) |*2-dimension-storage-RDAREA-specification* |OTHERS}
8-7	*primary-key-storage-RDAREA-change-specification*::= FOR PRIMARY KEY INTO *post-change-index-storage-RDAREA-name-list*

No.	Format
8-8	*cluster-key-storage-RDAREA-change-specification*::= FOR [PRIMARY] CLUSTER KEY INTO *post-change-index-storage-RDAREA-name-list*
8-9	*LOB-column-storage-RDAREA-change-specification*::= FOR COLUMN *column-name* {*LOB-column-storage-RDAREA-change-specification* |INTO *matrix-partitioned-LOB-column-storage-RDAREA-change-specification*} [, *column-name* {*LOB-column-storage-RDAREA-change-list* |INTO *matrix-partitioned-LOB-column-storage-RDAREA-change-specification*}]]...
8-10	*LOB-column-storage-RDAREA-change-list*::= INTO {*LOB-column-storage-RDAREA-name* | (*LOB-column-storage-RDAREA-name*) | ((*LOB-column-storage-RDAREA-name*) [, (*LOB-column-storage-RDAREA-name*)]...[,OTHERS]) |OTHERS}
8-11	*matrix-partitioned-table-storage-RDAREA-change-specification*::=*2-dimension-storage-RDAREA-specification* *matrix-partitioned-LOB-column-storage-RDAREA-change-specification*::=*2-dimension-storage-RDAREA-specification* *2-dimension-storage-RDAREA-specification*::=(*matrix-partitioned-storage-RDAREA-area-list*) [, *matrix-partitioned-storage-RDAREA-area-list*]...) *matrix-partitioned-storage-RDAREA-area-list*::=(*RDAREA-name*[,*RDAREA-name*]...)

#1: Columns in a non-FIX table

#2: Columns in a FIX table

Operands

1) [*authorization-identifier*.]*table-identifier*

authorization-identifier

Specifies the authorization identifier of the owner of the base table to be redefined.

table-identifier

Specifies the name of the base table to be redefined.

2) *column-addition-definition*::=

ADD *column-name data-type* [ARRAY [*maximum-number-of-elements*]] [NO SPLIT]

[[*column-recovery-restriction-1*]

{*LOB-column-storage-RDAREA-specification*

|*matrix-partitioned-LOB-column-storage-RDAREA-specification*

|*abstract-date-type-definition-LOB-column-storage-RDAREA-specification*

[*plug-in-specification*]

|*matrix-partitioned-LOB-attribute-storage-RDAREA-specification*

[*plug-in-specification*]}]

[DEFAULT *clause*]

{[NULL|NOT NULL[WITH DEFAULT]]#1

|[[NOT NULL]WITH DEFAULT]#2}

[*updatable-column-attribute*]

[WITH PROGRAM]

#1: Columns in a non-FIX table

#2: Columns in a FIX table

column-name

Specifies the name of the column to be added to a base table.

The following rules apply to column names:

1. When a column is to be added to a base table, the new column must be distinct in name from any columns that are already in the table.
2. More than 30,000 columns cannot be added to a base table.
3. A column cannot be added to a FIX table containing data.
4. When a column is added, the null value is assigned to the added column in an existing row. For details about how to assign the null value, see the *UPDATE statement Format 1 (Update data)* for data manipulation SQL statements.

data-type

Specifies the data type of the column to be added to the base table.

The following rules apply to data types:

When specifying BLOB in *data-type*, specify a LOB storage RDAREA.

When specifying an abstract data type in *data-type*, specify a LOB attribute storage RDAREA. An abstract data type for which BLOB is defined in super-type cannot be specified.

If the authorization identifier for the abstract data type is omitted and an abstract data type for the default authorization identifier does not exist, and if the abstract data type of the same name is in the MASTER schema, HiRDB assumes that that abstract data type is specified.

ARRAY [*maximum-number-of-elements*]

Specifies the maximum number of elements for the repetition column to be added to the base table.

The following rules apply to ARRAY *maximum-number-of-elements*:

1. In *maximum-number-of-elements*, specify an unsigned integer in the 2 to 30,000 range.
2. Omitting ARRAY *maximum-number-of-elements* indicates that the column is not a repetition column.
3. The following data types cannot be specified for a repetition column:
   - BLOB
   - BINARY
   - Abstract data type
4. A repetition column cannot be specified in a FIX table.

NO SPLIT

This option is specified when storing one row per page if the actual data length of a variable-length character string is 256 bytes or greater.

The NO SPLIT option can reduce the size of the database. For details about the NO SPLIT option, see the *HiRDB Version 9 Installation and Design Guide*.

NO SPLIT can be specified only with variable-length character types (VARCHAR, NVARCHAR, and MVARCHAR).

column-recovery-restriction-1 ::=[RECOVERY[{ALL|PARTIAL|NO}]]

When adding a BLOB data-type column or an abstract data-type column with a BLOB attribute to a table, specify an update log acquisition mode for the LOB column storage RDAREA in which the column is to be stored or for the database for the LOB storage RDAREA in the abstract data type definition.

Column recovery restriction cannot be specified for columns other than a BLOB data type column or an abstract data type column with a BLOB attribute.

ALL

This option is specified when operating the LOB RDAREA in the log acquisition mode. When the RDAREA is operated in the log acquisition mode, HiRDB acquires the database update log necessary for rollback and

roll-forward operations.

PARTIAL

This option is specified when operating the LOB RDAREA in the pre-update log acquisition mode. When the RDAREA is operated in the pre-update log acquisition mode, HiRDB acquires the database update log necessary for rollback operations.

NO

This option is specified when operating the user LOB RDAREA in the no-log mode. When the RDAREA is operated in the no-log mode, HiRDB does not acquire a database update log.

Depending on the specific update log acquisition method specified for a database, HiRDB employs different UAP execution methods or user LOB RDAREA recovery methods in the event of an error. For details about no-log mode operations, see the *HiRDB Version 9 System Operation Guide*.

LOB-column-storage-RDAREA-specification::=

IN {*LOB-column-storage-RDAREA-name*

|(*LOB-column-storage-RDAREA-name*)

|((*LOB-column-storage-RDAREA-name*)

[, (*LOB-column-storage-RDAREA-name*)]...)}

When adding a BLOB-type column, specifies the name of the user LOB RDAREA in which BLOB column data is to be stored.

The following rules apply to LOB column storage RDAREAs:

1. If BLOB is specified as a data type for a column, always specify the name of the LOB column storage RDAREA. A name cannot be specified for a column for which a data type other than BLOB is specified.
2. When adding a column to a row-partitioned table, specify the same number of LOB RDAREAs as the user RDAREAs specified in the table definition, taking care that user RDAREAs and LOB RDAREAs on the same server are in the same order.
3. When adding a LOB column to a row-partitioned table, hash-partitioned table, or matrix-partitioned table of a boundary value specification, specify a LOB column storage RDAREA for each table storage RDAREA specified in the table definition. If there are duplicate RDAREAs in the table storage RDAREA specified in the table definition, specify duplicate LOB column storage RDAREAs in the same positions.

   In all other cases, LOB column storage RDAREAs cannot be specified or

duplicated.

matrix-partitioned-LOB-column-storage-RDAREA-specification::=*two-dimensional-storage-RDAREA-specification*

two-dimensional-storage-RDAREA-specification::=(*matrix-partitioning-RDAREA-list*

[, *matrix-partitioning-RDAREA-list*]...)

matrix-partitioning-RDAREA-list::=

(*RDAREA-name*[, *RDAREA-name*]...)

When adding a BLOB-type column to a matrix-partitioned table, specifies the name of the RDAREA in which the BLOB column is to be stored.

When adding a BLOB-type column to a matrix-partitioned table, specifies the RDAREA for storing matrix-partitioned LOB columns.

For RDAREA names, see the explanation in *LOB-column-storage-RDAREA-specification*.

abstract-date-type-definition-LOB-column-storage-RDAREA-specification::=

ALLOCATE(*attribute-name*[..*attribute-name*]...

IN {*LOB-attribute-storage-RDAREA-name*

|(*LOB-attribute-storage-RDAREA-name*)

|((*LOB-attribute-storage-RDAREA-name*)

[, (*LOB-attribute-storage-RDAREA-name*)]...)}

[, *attribute-name*[..*attribute-name*]...

IN {*LOB-attribute-storage-RDAREA-name*

|(*LOB-attribute-storage-RDAREA-name*)

|((*LOB-attribute-storage-RDAREA-name*)

[, (*LOB-attribute-storage-RDAREA-name*)]...)}]...)

This option is specified when adding a column of the abstract data type including the LOB attribute.

attribute-name[..*attribute-name*]

Specifies attribute names that comprise an abstract data type. If the attribute of an abstract data type is the abstract data type and if the attribute of the nested abstract data type has a LOB-type attribute, specify the attribute name of the LOB type.

Specify an attribute name in the following cases:

- Attribute of an abstract data type definition

  This operand is specified if the attribute of the abstract data type definition is the LOB type.

- Nest of an abstract data type definition

  If the attribute of an abstract data type is the abstract data type and if the attribute of a nested abstract data type is a LOB-type attribute, specify the attribute name of the LOB type.

LOB-attribute-storage-RDAREA-name

Specifies the name of the user LOB RDAREA that stores BLOB attribute data, located in any hierarchy of abstract data types.

The following rules apply to LOB attribute storage RDAREAs:

1. When an abstract data type including BLOB is specified as a data type, always specify the name of a user LOB RDAREA for each BLOB attribute. Such a name cannot be specified in attributes for which a non-BLOB data type is specified.
2. When adding a column to a row-partitioned table, specify the same number of LOB RDAREAs as the user RDAREAs specified in the table definition, taking care that user RDAREAs and LOB RDAREAs on the same server are in the same order.
3. When adding abstract data type columns that include LOB attributes to a row-partitioned table, hash-partitioned table, or matrix-partitioned table of a boundary value specification, specify a LOB attribute storage RDAREA for each table storage RDAREA specified in the table definition. If there are duplicate RDAREAs in the table storage RDAREA specified in the table definition, specify duplicate LOB column storage RDAREAs in the same positions.

   In all other cases, LOB attribute storage RDAREAs cannot be specified or duplicated.

plug-in-specification ::=PLUGIN *plug-in-option*

Specifies as a character string literal (of up to 255 bytes) parameter information to be passed to the plug-in facility for a column that is defined as an abstract data type for which the plug-in facility is implemented. Hexadecimal character string literals cannot be specified as parameter information.

For details about parameter information, see the respective plug-in manuals.

matrix-partitioned-LOB-attribute-storage-RDAREA-specification ::=*two-dimensional-storage-RDAREA-specification*

two-dimensional-storage-RDAREA-specification::=(*matrix-partitioning-RDA*

REA-list

[, *matrix-partitioning-RDAREA-list*]...)

matrix-partitioning-RDAREA-list ::=

(*RDAREA-name*[, *RDAREA-name*]...)

When adding a column of abstract data type including the LOB attribute to a matrix-partitioned table, specifies the name of the RDAREA in which the LOB attribute is to be stored.

When adding a column of abstract data type including the LOB attribute to a matrix-partitioned table, specifies the name of the RDAREA in which the matrix-partitioned LOB attribute is to be stored.

For details about RDAREA names, see the explanation in *LOB-attribute-storage-RDAREA-name*.

DEFAULT *clause* ::=DEFAULT[*predefined-value*]

This option is specified when setting a predefined value in the column being added.

The following rules apply to DEFAULT *clause*:

1. DEFAULT *clause* cannot be specified together with WITH DEFAULT in a single ALTER TABLE.
2. If data is stored in the base table, DEFAULT *clause* cannot be specified on the column being added.
3. The DEFAULT *clause* cannot be specified on a BLOB-type column, an abstract data-type column, or a BINARY-type column, with a defined length of 32,001 bytes or greater.
4. The DEFAULT *clause* cannot be specified on a repetition column.
5. The following items cannot be specified in a DEFAULT *clause* if the column to be added does not use the default character set:
   - CURRENT_DATE (CURRENT DATE)
   - CURRENT_TIME (CURRENT TIME)
   - CURRENT TIMESTAMP (CURRENT_TIMESTAMP)

NULL

The NULL option can be specified on the column specified in *column-name* if the column allows the null value.

The NULL option cannot be specified on a column in a FIX table.

NOT NULL

This option is specified if the column specified in *column-name* must be constrained (NOT NULL-constrained) so that it does not allow the null value.

The following rules apply to the NOT NULL option:

1. NOT NULL cannot be specified if data is already stored in the base table.
2. If NOT NULL is omitted, the added column allows the null value, and if the column was added to a row containing data, the null value is assigned to the column.
3. The NOT NULL option cannot be specified on repetition columns or columns of the abstract data type.

WITH DEFAULT

This option is specified when adding a NOT NULL-constrained column containing a predefined value.

The following rules apply to the WITH DEFAULT option:

1. Specify NOT NULL when specifying WITH DEFAULT for a non-FIX table.
2. WITH DEFAULT cannot be specified together with DEFAULT *clause* in a single ALTER TABLE.
3. WITH DEFAULT cannot be specified for a repetition column.
4. WITH DEFAULT cannot be specified for columns of the abstract data type.

updatable-column-attribute::= UPDATE [ONLY FROM NULL]

Specify this operand when adding an updatable column to a falsification-prevented table or adding an updatable column to a table that is to be changed to a falsification-prevented table.

The updatable column attribute is valid only when specified on a falsification-prevented table.

For details about how to change a given table to a falsification-prevented table, see the INSERT ONLY option in CHANGE.

UPDATE

Specify this operand when adding an updatable column to a falsification-prevented table.

UPDATE ONLY FROM NULL

Specify this operand when adding a column in which row values can be changed only once from the null value to a non-null value in a falsification-prevented table.

The following table summarizes the conditions under which the value of an

UPDATE ONLY FROM NULL-specified column can be changed in a falsification-prevented table.

Column value before change	Column value after change	Whether updatable
Null value	Null value	Y
Null value	Non-null-value	Y
Non-null-value	Null value	N
Non-null-value	Non-null-value#	N

Legend:

Y: Updatable

N: Not updatable

Note

For a repetition column, only updates by column from the null value (a value in which the number of elements is 0) to an unsubscripted specification can be executed.

#: Contains the same value as the pre-update value.

Specification of the UPDATE ONLY FROM NULL operand is subject to the following rules:

1. The operand cannot be specified for a NOT NULL-specified column.
2. The operand cannot be specified for a FIX table.
3. The operand cannot be specified for BLOB-type columns and for BINARY type columns with a defined length of 32,001 bytes or greater.

If the attribute is specified, the column value of the updatable column attribute can be updated under the following conditions:

Table type	UPDATE specification		UPDATE ONLY FROM NULL specification		No specification	
	Specifiable	Column value updatable	Specifiable	Column value updatable	Specifiable	Column value updatable
Non-falsification-prevented table	Y	Y	Y	Y	--	Y
Falsification-prevented table	Y	Y	Y	Y#	--	N

Legend:

Y: Updatable

N: Not updatable

--: Not applicable

#: Can be updated only once from the null value to a non-null value.

WITH PROGRAM

Specify WITH PROGRAM in any of the following operations when an SQL object with an effective function, procedure, or trigger with respect to the table is to be nullified:

- Adding a column with a DEFAULT clause to a table
- Adding a column with a NOT NULL specification to a table
- Adding a column of the BLOB or abstract data type to a table

3) *RDAREA-addition-definition*::=

ADD RDAREA *table-storage-RDAREA-name*

[FOR COLUMN *column-name*

{*LOB-column-storage-RDAREA-specification*

|*abstract-date-type-definition-LOB-column-storage-RDAREA-specification*}

[, *column-name* {*LOB-column-storage-RDAREA-specification*

|*abstract-date-type-definition-LOB-column-storage-RDAREA-specification*}]...]

[FOR INDEX *index-identifier index-storage-RDAREA-specification*

[, *index-identifier index-storage-RDAREA-specification*]...]

[FOR [PRIMARY] CLUSTER KEY *index-storage-RDAREA-specification*]

[FOR PRIMARY KEY *index-storage-RDAREA-specification*]

[WITH PROGRAM]

table-storage-RDAREA-name

This operand is specified when a user RDAREA is to be added to a row-partitioned table by a hash function.

The following rules apply to the names of table storage RDAREAs:

1. The addition of a user RDAREA to a table may fail if a unique-specification index is defined on the table. For details, see the explanation for *CREATE*

INDEX Format 1, [*UNIQUE*] .

2. If the rebalancing facility is not used, a user RDAREA cannot be added to a FIX hash-partitioned table in which data is stored. For details about the rebalancing facility, see the *HiRDB Version 9 System Operation Guide*.
3. Other tables and indexes cannot be added to the RDAREA that stores tables that use the rebalancing facility.
4. If an RDAREA is added to a FIX hash-partitioned table that uses the rebalancing facility, performance of the SQL statement that retrieves or updates the table may deteriorate until such time as the rebalancing utility is successfully executed.
5. If an index with a unique specification, a unique cluster key, or the primary key is defined for a FIX hash-partitioned table using the rebalancing facility, and if an RDAREA is added to that table, no data can be added to or updated in the table until such time as the rebalancing utility is successfully executed.
6. When adding an RDAREA to a table using the free space reusage facility, the free space reusage facility is also applied to the RDAREA that is added.
7. A shared RDAREA cannot be specified in a table storage RDAREA.
8. The maximum total number of split RDAREAs after a change in table definition, exclusive of duplicates, is 1,024.

column-name

This operand is specified when the table to which an RDAREA is being added contains a BLOB column or a column of the abstract data type including the BLOB attribute.

In *column-name*, specify either a BLOB-type column or a column defined in the abstract data type including the BLOB attribute.

All BLOB-type columns and columns of the abstract data type including the BLOB attribute need to be specified.

LOB-column-storage-RDAREA-specification ::=

```
IN {LOB-column-storage-RDAREA-name
   |(LOB-column-storage-RDAREA-name)
   |((LOB-column-storage-RDAREA-name)
   [, (LOB-column-storage-RDAREA-name)]...)}
```

Specifies the name of the user LOB RDAREA in which BLOB-column data is to be stored.

The name of the RDAREA to be specified should be the user LOB RDAREA that

is located on the same server as the RDAREA specified in *table-storage-RDAREA-name*. Specify an LOB column storage RDAREA for each table storage RDAREA specified in the table definition. If there are duplicate RDAREAs in the table storage RDAREA specified in the table definition, specify duplicate LOB column storage RDAREAs in the same positions. In all other cases, LOB column storage RDAREAs cannot be specified or duplicated.

abstract-data-type-definition-LOB-storage-RDAREA-specification::=

```
ALLOCATE(attribute-name[..attribute-name]...
  IN {LOB-attribute-storage-RDAREA-name
     |(LOB-attribute-storage-RDAREA-name)
     |((LOB-attribute-storage-RDAREA-name)
     [, (LOB-attribute-storage-RDAREA-name)]...)}
  [, attribute-name[..attribute-name]...
   IN {LOB-attribute-storage-RDAREA-name
     |(LOB-attribute-storage-RDAREA-name)
      |((LOB-attribute-storage-RDAREA-name)
     [, (LOB-attribute-storage-RDAREA-name)]...)}]...)
```

attribute-name[..*attribute-name*]

Specifies the name of the attribute that comprises the abstract data type. If the attribute of an abstract data type is the abstract data type and if the attribute of a nested abstract data type is a LOB-type attribute, specify the attribute name of the LOB type.

Specify *attribute-name* in the following cases:

- Attribute of an abstract data type definition

  Specify *attribute-name* if the data type of an attribute of an abstract data type definition is the LOB type.

- Nest of an abstract data type definition

  If the attribute of an abstract data type is the abstract data type and if the attribute of a nested abstract data type is a LOB-type attribute, specify the attribute name of the LOB type.

LOB-attribute-storage-RDAREA-name

Specifies the name of the user LOB RDAREA for the storage of BLOB attribute data, in any hierarchy of abstract data types.

The following rules apply to LOB attribute storage RDAREAs:

1. Specify a user LOB RDAREA for all BLOB attributes that are in the abstract data type.
2. For the RDAREA name to be specified, specify the user LOB RDAREA that is defined on the same server as the RDAREA that was defined in *table-storage-RDAREA-name*.
3. Specify an LOB attribute storage RDAREA for each table storage RDAREA specified in the table definition. If there are duplicate RDAREAs in the table storage RDAREA specified in the table definition, specify duplicate LOB column storage RDAREAs in the same positions.

   In all other cases, LOB attribute storage RDAREAs cannot be specified or duplicated.

FOR INDEX *index-identifier index-storage-RDAREA-specification*

[, *index-identifier index-storage-RDAREA-specification*]...]

If an index is defined for the table to which an RDAREA is being added, specifies the index storage RDAREA corresponding to the table storage RDAREA being added.

index-identifier

Specifies the index identifier of the index that is defined for the table to which the RDAREA is being added.

index-storage-RDAREA-specification

For index storage RDAREA specifications, see the explanation in *ADD RDAREA,index-storage-RDAREA-specification::=*.

FOR [PRIMARY] CLUSTER KEY *index-storage-RDAREA-specification*

If a cluster key is already defined for the table to which an RDAREA is to be added, specifies the cluster key storage RDAREA associated with the table storage RDAREA being added.

PRIMARY

This option is specified if the cluster key is defined as the primary key.

index-storage-RDAREA-specification

For index storage RDAREA specifications, see the explanation in *ADD RDAREA,index-storage-RDAREA-specification::=*.

FOR PRIMARY KEY *index-storage-RDAREA-specification*

If the primary key is already defined for the table to which an RDAREA is to be

added, specifies the primary key storage RDAREA associated with the table storage RDAREA being added.

If a table comprising a cluster key is defined as the primary key, specify `FOR PRIMARY CLUSTER KEY`.

For index storage RDAREA specifications, see the explanation in *ADD RDAREA,index-storage-RDAREA-specification::=*.

index-storage-RDAREA-specification`::=`

`IN{`*index-storage-RDAREA-name*

`|(`*index-storage-RDAREA-name*`)`

`|((`*index-storage-RDAREA-name*`)`

`[, (`*index-storage-RDAREA-name*`)]...)}`

Specifies the name of the RDAREA in which an index, a cluster key, or the primary key is to be stored.

Specifies the user RDAREA for an index defined using `CREATE INDEX` format 1 (for index column data type other than abstract data) and for an index (partitioned index), cluster key, or primary key defined using `CREATE INDEX` format 3. However, for HiRDB/Parallel Server, a shared RDAREA is specified for a shared table.

For the index (the data type of the index constituent column is the abstract data type) defined in `CREATE INDEX` format 2, specify a user LOB RDAREA.

The following rules apply to the specification of an index storage RDAREA:

1. For HiRDB/Parallel Server, the RDAREA for storing the table being added and the RDAREA that stores the associated index must be on the same back-end server.
2. The index that is row-partitioned on the server must have the same number of table-storage RDAREAs as index-storage RDAREAs.
3. Indexes on a server that are not row-partitioned must have the same number of index-storing RDAREAs as the back-end server on which tables are stored.
4. When adding a user RDAREA, you need to specify an RDAREA that satisfies the following formula:

   Key length ≤ (*page-size-of-index-storage-RDAREA* ÷ 2) - 1242
5. Specify an index storage RDAREA for each table storage RDAREA specified in the table definition. If there are duplicate RDAREAs in the table storage RDAREA, specify the index storage RDAREAs in the same positions.

6. The following table indicates whether an index storage RDAREA can be specified depending on the index type. For an index for which the specification of an index storage RDAREA is mandatory, an index storage RDAREA should be specified.

Table 3-6: Whether an index storage RDAREA can be specified depending on the index type

Index definition method	Index-partitioning method		Specifiability of index storage RDAREA
Index and primary key defined using `CREATE INDEX` format 1 or format 3	Row-partitioning on one server#1	On-server partitioning key index	Y
		On-server non-partitioning key index	
	Row-partitioning only between servers#2	On-server partitioning key index	
		On-server non-partitioning key index	N#5, #6
	Row partitioning on and between servers#3	On-server partitioning key index	Y
		On-server non-partitioning key index	
	Non-partitioning#4	Non-partitioning key index	N#5
Index and cluster key defined using `CREATE INDEX` format 2	Row-partitioning on one server#1		Y
	Row-partitioning only between servers#2		
	Row-partitioning on and between servers#3		

Legend:

Y: An index storage RDAREA needs to be specified.

N: An index storage RDAREA is not required.

#1: Refers to row-partitioning on a HiRDB/Single Server or row-partitioned that is closed to a single back-end server on a HiRDB/Parallel Server.

#2: Refers to row-partitioning extending across multiple back-end servers on a HiRDB/Parallel Server and not partitioning an index on the back-end server.

#3: Refers to row-partitioning extending across multiple back-end servers on a HiRDB/Parallel Server and partitioning an index on the back-end server.

#4: Refers to an index that is not row-partitioned.

#5: If the addition of RDAREAs increases the number of back-end servers on which tables are stored, you also need to specify index storage RDAREAs.

#6: If the addition of RDAREAs does not change the number of back-end servers on which tables are stored, you can also specify index storage RDAREAs. If an index storage RDAREA is specified, the index becomes a row-partitioned index on the server.

WITH PROGRAM

When adding an RDAREA to a hash-partitioned table, this option is specified to nullify the SQL object if there is an effective SQL object for a function, procedure, or trigger that uses that table.

4) *column-attribute-change-definition*::=

```
CHANGE{column-name{ [{VARCHAR(data-length)
          |NVARCHAR(data-length)
          |MCHAR({*|data-length})
          |MVARCHAR({*|data-length}) }
          [ARRAY [{*|maximum-number-of-elements}]]]
         |[ARRAY [maximum-number-of-elements]]
         |BINARY(data-length) }
         [{NO SPLIT|SPLIT}]
           [column-recovery-restriction-2]
         [{SET DEFAULT clause|DROP DEFAULT}]
         [WITH DEFAULT]
         [updatable-column-attribute]
        |CLUSTER KEY[UNIQUE]
        |LOCK{ROW|PAGE}
        |HASH hash-function-name
        |SEGMENT REUSE { [number-of-segments[{K|M|G}]]|NO}}
       |INSERT ONLY [WHILE {date-interval-data|labeled-interval} BY
    column-name]}
      [WITH PROGRAM]
```

column-name

When modifying the definition of a column, specifies the name of the column.

The following rules apply to column names:

1. Columns for which any of the following specifications is made during a view table definition cannot be redefined using `ALTER TABLE`:

   - A column specified as an argument in a scalar function in a selection expression or a search condition
   - A column specified as an argument in a function call in a selection expression or a search condition
   - A column specified as an operation term for a concatenation operation in a selection expression or a search condition
   - A column specified in a value expression in a `CASE` expression in a selection expression or a search condition
   - A column specified in a value expression in a `CAST` specification in a selection expression or a search condition

2. If any of the following items is specified in a view definition statement during the definition of a view table, the definition of columns in the table specified in the view table cannot be altered using `ALTER TABLE`.

   - An item containing a row subquery
   - An item containing a scalar subquery or a table subquery using set operations
   - An item containing a table subquery with a results column count of 2 or greater in a `FROM` clause derived table or in a predicate other than the `EXISTS` predicate
   - A selection expression containing a scalar subquery
   - A row value constructor on the left side of a comparison predicate, `IN` predicate, or quantified predicate containing a scalar subquery
   - A row value constructor, on the right side of a comparison predicate, containing a scalar subquery, with the number of row value constructor elements being 2 or greater
   - A list of row value constructors, on the right side of an `IN` predicate, containing a scalar subquery
   - Scalar subquery contained in a `BETWEEN` predicate, `LIKE` predicate, `XLIKE` predicate, or `SIMILAR` predicate

3. In *column-name*, specify the name of a column in the base specified in *table-identifier*.

4. After *column-name*, specify one or more operands that provide a description of the change in column definition.

{VARCHAR (*data-length)*)

|NVARCHAR(*data-length*)

|MCHAR({*|*data-length*})

|MVARCHAR({*|*data-length*})

|BINARY (*data-length*)}

Specifies *data-length* when changing the data length of variable-length data; it can also be specified when changing the data type from CHAR to MCHAR or from VARCHAR to MVARCHAR.

The following rules apply:

1. The maximum length of a column of the variable-length data type cannot be reduced.
2. For a variable-length data-type column for which an index is defined, its data length cannot be changed in a manner that violates the following formula:

   Key length ≤

   MIN ((*page-size-of-index-storage-RDAREA* ÷ 2) - 1242, 4036)
3. A data type should be omitted when you are not changing the maximum length.
4. Of the columns that are specified in a derived query expression in a view definition statement, columns that are subject to any of the following specifications cannot be changed to BINARY greater than or equal to 32,001 bytes:

   • Columns that are specified in a subquery in a comparison predicate, quantified predicate, or IN predicate

   • Columns specified for duplicate elimination

   • Columns specified for grouping or in a set function

   • Columns specified in a selection expression with a query specification that is subject to set operations

   • Columns that are specified during the definition of a view table that is expanded as an inner derived table in a query specification that satisfies one of the conditions for the creation of an inner derived table

   For conditions for creating an inner derived table, see *2.21 Inner derived tables*.
5. CHAR can be changed to MCHAR, but the data length cannot be changed. For data length, specify either the data length before the change or an asterisk (*).

6. VARCHAR can be changed to MVARCHAR while increasing the maximum length of the column. If the maximum length is not to be changed, for data length, specify either the maximum length before the change or an asterisk (*).
7. The data type cannot be changed for columns for which a character set was specified.
8. The following columns cannot be changed to a BINARY type of 32,001 bytes or greater:

   • Columns for which UPDATE ONLY FROM NULL of the updatable column attribute is specified

   • Columns to which UPDATE ONLY FROM NULL of the updatable column attribute is assigned at the same time as a change in the attribute's data length

ARRAY [{*|*maximum-number-of-elements*}]

This operand is specified when increasing the maximum number of elements for a repetition column.

The following rules apply:

1. In *maximum-number-of-elements*, specify an unsigned integer in the 2 to 30,000 range.
2. The default is that the column is not a repetition column.
3. When not changing the maximum number of elements, specify an asterisk (*).
4. The maximum number of elements cannot be reduced.

ARRAY [*maximum-number-of-elements*]

This operand is specified when increasing the maximum number of elements for a repetition column.

In *maximum-number-of-elements*, specify an unsigned integer in the 2 to 30,000 range.

The maximum number of elements cannot be reduced.

{NO SPLIT|SPLIT}

This operand is specified to reduce the size of the database when storing data of a variable-length character type (VARCHAR, NVARCHAR, or MVARCHAR).

For details about NO SPLIT and SPLIT (the NO SPLIT option), see the *HiRDB Version 9 Installation and Design Guide*.

NO SPLIT

This option is specified when the real data length is greater than or equal to 256 bytes.

- When defining a table or adding a column definition, specify this option for a variable-length character string type for which NO SPLIT is not specified.
- This option cannot be specified for a base table containing data.

SPLIT

This option is specified when the real data length is less than or equal to 255 bytes.

- When defining a table or adding a column definition, specify this option for a variable-length character type for which NO SPLIT is specified.
- This option cannot be specified for a base table containing data.

column-recovery-restriction-2 ::= RECOVERY{ALL|PARTIAL|NO}

When the column data type is BLOB or when the column is an abstract data type containing the BLOB attribute, this option is specified to change the method of acquiring the database update log corresponding to the column. The details of the ALL, PARTIAL, and NO options are the same as when *column-recovery-restriction-1* is specified for ALTER TABLE ADD.

{SET DEFAULT *clause*|DROP DEFAULT}

SET DEFAULT *clause*

DEFAULT *clause* ::=DEFAULT[*predefined-value*]

Specify this when setting or changing the default value. The following rules apply:

1. The column data type that can be specified is the same as the default clause of ALTER TABLE ADD.
2. Specifying the SET DEFAULT *clause* on a column for which WITH DEFAULT is defined nullifies the WITH DEFAULT specification and causes the SET DEFAULT *clause* specification to take effect.
3. This option cannot be specified with WITH DEFAULT in a single ALTER TABLE statement.
4. The following items cannot be specified in a DEFAULT *clause* if the column whose maximum length is to be changed does not use the default character set:
   - CURRENT_DATE (CURRENT DATE)
   - CURRENT_TIME (CURRENT TIME)

- CURRENT_TIMESTAMP (CURRENT TIMESTAMP)

5. This option cannot be specified for an insert history maintenance column with a specified deletion prevented duration in a falsification prevented table.

DROP DEFAULT

This option is specified when deleting a predefined value.

DROP DEFAULT should be specified for columns for which DEFAULT *clause* is specified.

WITH DEFAULT

This option is specified when changing a NOT NULL-constrained column without a predefined value into a NOT NULL-constrained column for which there is a predefined value.

The following rules apply to WITH DEFAULT:

1. This option cannot be specified on not NOT NULL-constrained columns.
2. WITH DEFAULT cannot be specified on NOT NULL-constrained columns for which there already is a predefined value.
3. WITH DEFAULT cannot be specified on columns for which DEFAULT *clause* is defined.
4. This option cannot be specified together with a SET DEFAULT or DROP DEFAULT *clause* in a single ALTER TABLE statement.
5. This option cannot be used to change a column with a predefined value into a column without a predefined value.
6. This option cannot be specified for a repetition column.
7. This option cannot be specified for an abstract data type column.

updatable-column-attribute::= UPDATE [ONLY FROM NULL]

Specify this operand if the updatable column attribute is to be changed before a table is changed into a falsification-prevented table.

The updatable column attribute is valid only with falsification-prevented tables.

For changing a given table into a falsification-prevented table, see the INSERT ONLY option.

The following rules apply to the updatable column attribute:

1. The attribute cannot be specified for a falsification-prevented table.
2. The attribute cannot be specified for a column for which SYSTEM GENERATED is specified.

3. The attribute cannot be specified for a column for which the updatable column attribute is already specified.
4. The attribute cannot be specified for any of the following non-updatable columns:
   - Cluster key constituent column
   - Partitioning key constituent column (not including partitioning key constituent columns for a flexible hash partitioning table)

`UPDATE`

With this option specified, the column is updatable after the table is changed into a falsification-prevented table.

`UPDATE ONLY FROM NULL`

After a change into a falsification-prevented table, the column value can be updated only once from the null value to a non-null value.

The following table indicates the updatability of the values of a column specified in a falsification-prevented table with the `UPDATE ONLY FROM NULL` option:

Column value before update	Column value after update	Updatable
Null value	Null value	Y
Null value	Non-null value	Y
Non-null value	Null value	N
Non-null value	Non-null value#	N

Legend:

Y: Updatable

N: Not updatable

Note

For a repetition column, only an update by column without a subscript specification can be executed from the null value (with an element count of 0).

#: Contains the same value as before the update.

The `UPDATE ONLY FROM NULL` specification is subject to the following rules:

- The operand cannot be specified for columns with a `NOT NULL`

specification.

- The operand cannot be specified for FIX tables.
- The operand cannot be specified for the primary key or a cluster key constituent column.
- The operand cannot be specified for a partitioning key constituent column.
- The operand cannot be specified for a BLOB-type column or a BINARY-type column with a defined length of 32,001 bytes or greater.

If the attribute is specified, the column value of the updatable column attribute can be updated under the following conditions:

Table type	UPDATE specification		UPDATE ONLY FROM NULL specification		No specification	
	Specifiable	Column value updatable	Specifiable	Column value updatable	Specifiable	Column value updatable
Non-falsification-prevented table	Y	Y	Y	Y	--	Y
Falsification-prevented table	N	Y	N	Y#	--	N

Legend:

Y: Updatable

N: Not updatable

--: Not applicable

#: Can be updated only once from the null value to a non-null value.

CLUSTER KEY UNIQUE

This option is specified when changing a cluster key without uniqueness constraint into a cluster key with uniqueness constraint.

The following rules apply to CLUSTER KEY UNIQUE:

1. The cluster key attribute of a base table cannot be changed if the table already contains data.
2. This option cannot be specified for a table if a uniqueness-constrained cluster key is defined in the table definition.
3. This option cannot be specified for flexible hash-partitioned tables.

CLUSTER KEY

This option is specified when changing a cluster key with uniqueness constraint into a cluster key without uniqueness constraint.

The cluster key attribute cannot be changed for a base table if the table already contains data.

This option cannot be specified for a table if the table definition defines a not uniqueness constrained cluster key.

LOCK{ROW|PAGE}

This option is specified when changing the minimum lock resource unit for a table.

Specify LOCK ROW when changing the minimum lock resource unit into units of rows; specify LOCK PAGE when changing it into units of pages.

HASH *hash-function-name*

For a table that is row-partitioned by a hash function, this option can be specified when changing hash functions. This option cannot change hash functions on FIX hash-partitioned tables containing data.

HASH1-HASH6 and HASH0 cannot be specified as a hash function for a table that uses the rebalancing facility. Similarly, HASHA-HASHF cannot be used as a hash function for a table that does not use the rebalancing facility.

SEGMENT REUSE { [*number-of-segments* [{K|M|G}]] |NO}

This option is specified when applying the free space reusage facility for a table for which the free space reusage facility is not being used.

Because of the overhead of reusing free space, use the free space reusage facility to prioritize storage efficiency over performance.

For details about the free space reusage facility, see the *HiRDB Version 9 Installation and Design Guide*.

number-of-segments [{K|M|G}]

When using the free space reusage facility for a table and assigning an upper limit on the number of segments for the table, this operand specifies an upper limit on the number of segments in an unsigned integer in the 1 to 268,435,440 range. As a unit, you can also specify K (kilo), M (mega), or G (giga).

When a table is subject to frequent insertion or deletion of rows, this operand can improve row insertion performance and the storage efficiency in a specified segment.

When *number-of-segments* is not specified

The operand *number-of-segments* can be omitted when the free space reusage facility is being used and no maximum number of segments is set for the table.

Use this operand when a table is subject to frequent row insertions or deletions and only the specified table is to be stored in the RDAREA. This operand also improves row insertion performance and the storage efficiency of free space in the RDAREA in which the specified table is to be stored.

NO

This operand is specified when not using the free space reusage facility.

Specify NO for tables that are not subject to frequent row insertions or deletions.

The following rules apply to SEGMENT REUSE:

1. The free space reusage facility has no effect on LOB columns, abstract data type columns of the LOB attribute, or indexes.
2. The free space reusage facility cannot be specified for rebalancing tables.

INSERT ONLY [WHILE {*date-interval-data* | *labeled-interval*} BY *column-name*]

Specify this operand when changing a given table into a falsification-prevented table. For details about falsification-prevented tables, see the *HiRDB Version 9 Installation and Design Guide*.

If a table is converted into a falsification-prevented table, its values cannot be updated; however, its updatable columns can be updated.

For the falsification-prevented table, a period during which deletion of rows is disabled (a deletion-prevented duration) can be specified. For specifying a deletion-prevented duration, use WHILE to specify a time period, and as a column name, specify an insert history maintenance column (a column of the DATE type and SYSTEM GENERATED). If a deletion-prevented duration is not specified, the column cannot be deleted at any time.

date-interval-data

Specifies a deletion-prevented duration in a decimal representation of date interval data. See *1.4.4 Decimal representation of date interval data*.

Date interval data can be specified in positive numbers only.

labeled-interval

Specifies a deletion-prevented duration as a labeled duration. See *2.11 Date operations* for labeled durations.

Only a positive integer literal not enclosed in parentheses can be specified in a value expression for a labeled duration.

column-name

Specify a DATE-type column that is SYSTEM GENERATED.

The deletion-prevented duration includes the day the row was inserted, and the calculation of a deletion-prevented duration is subject to the rules on adding or subtracting date data and date interval data in *2.11 Date operations*. The last deletion-prevented date and the deletion-enabled date result from the following operations:

- *last-deletion-prevented-date = row-insertion-date + deletion-prevented-duration - 1-day*
- *deletion-enabled-date = row-insertion-date + deletion-prevented-duration*

For the relationship between a last deletion-prevented date and a deletion enabled date at specified values of a row insertion date and deletion-prevented duration, see *Table 3-30 Relationship between last day of deletion prevention and the deletion-allowed data*.

Changing a given table to a falsification-prevented table is subject to the following restrictions:

1. A table that is already converted into a falsification-prevented table cannot be specified.
2. A table for which a foreign key is defined cannot be changed into a falsification-prevented table.
3. A table for which a check constraint is defined cannot be changed into a falsification-prevented table.
4. A table for which the updatable column attribute is specified for all columns cannot be changed into a falsification-prevented table.
5. If a given table is defined in an RDAREA to which the inner replica facility is applied, the table cannot be changed into a falsification-prevented table.
6. A table containing rows cannot be changed into a falsification-prevented table.

WITH PROGRAM

This option can be specified in the following cases:

- For disabling an enabled SQL statement for the function, procedure, or trigger for the table in question when changing column names, changing the default for the DEFAULT clause, changing the lock mode for a table, changing hash functions for a hash-partitioned table, changing SEGMENT REUSE, or changing falsification-prevented tables.

- Nullifying an SQL object that is effective for a function, procedure, or trigger on a cluster key when the uniqueness constraint attribute of the cluster key is to be changed.

5) *column-deletion-definition* : : =

```
DROP column-name [WITH PROGRAM]
```

column-name

This operand is specified when deleting a column.

The following rules apply to the *column-name* operand:

1. In *column-name*, specify the name of the column that is in the base table specified in *table-identifier*.
2. Columns in a base table cannot be deleted if the base table already contains data.
3. If a column is the only one left in a table, it cannot be deleted.
4. Deleting a column causes any index on the column, comments, and any view table (including public view tables) using the column also to be deleted.
5. Columns comprising an index for which a cluster key or the primary key is defined cannot be deleted.
6. Columns with a `BLOB` data type cannot be deleted.
7. Columns of a table containing abstract data-type columns cannot be specified.
8. Insert history maintenance columns for a falsification-prevented table cannot be deleted.
9. A column in a falsification-prevented table cannot be deleted if its deletion makes all columns in the table updatable.

`WITH PROGRAM`

This option is specified when deleting a column in a table and when SQL objects for which functions, procedures, and triggers using the table are to be nullified.

6) *table-name-change-definition* : : =

```
RENAME TABLE TO table-identifier
    [WITH PROGRAM]
```

This option is specified when renaming a table.

table-identifier

Specifies the name of the renamed base table.

The following rules apply to the *table-identifier* operand:

1. This operand can also be used to rename indexed tables.
2. This operand cannot be used to rename view tables.
3. The names of the following tables cannot be modified:
   - A table that becomes the base for a view table
   - A table specified in a `CREATE PROCEDURE` SQL procedure statement
   - A table specified in a `CREATE TRIGGER` SQL procedure statement and a table that defines a trigger
4. The name of a table cannot be changed to a name identical to a base table, a view table, that is in the schema.
5. Falsification-prevented tables cannot be renamed.

`WITH PROGRAM`

Specified to disable the procedure for using the table, or a valid SQL object of a trigger, when changing the table identifier name. Note that even when `WITH PROGRAM` is specified, the table name of the table for which the trigger is defined cannot be modified.

7) *column-name-change-definition* `:::=`

`RENAME COLUMN FROM` *pre-change-column-name* `TO` *post-change-column-name*

`[WITH PROGRAM]`

This option is specified to change the name of a column.

old-column-name`,` *new-column-name*

When renaming a column, specifies the old name and the new name.

The following rules apply to *old-column-name* and *new-column-name* operands:

1. Index key columns can also be renamed.
2. Columns in a view table cannot be renamed.
3. The names of the following columns cannot be altered:
   - Columns in a table that becomes the base for a view table
   - Columns in a table specified in a `CREATE PROCEDURE` SQL procedure statement
   - Columns in a table specified in a `CREATE TRIGGER` SQL procedure statement, trigger event columns, and columns specified in a trigger action condition

4. The name of a column cannot be changed to a column name that is already in the table.
5. A column cannot be renamed if its existing column name is not in the table.
6. Columns in a falsification-prevented table cannot be renamed.

WITH PROGRAM

When the name of a column is modified, this option is specified to invalidate the valid SQL object of a function, procedure, or trigger defined for the table containing the column. However, the WITH PROGRAM option cannot rename the table for which a trigger is defined or any of the following columns:

- Trigger event columns
- Columns that are referenced in a trigger action condition using an old or new values correlation name
- Columns that are referenced in a trigger SQL statement using an old or new values correlation name

8) *partitioning-storage-condition-change-definition*::=

CHANGE RDAREA
{*row-partitioned-table-change-specification*|*matrix-partitioned-table-change-specification*}

[*LOB-column-storage-RDAREA-change-specification*]

[*index-storage-RDAREA-change-specification*

[*index-storage-RDAREA-change-specification*]...]

[*cluster-key-storage-RDAREA-change-specification*]

[*primary-key-storage-RDAREA-change-specification*]

[WITHOUT PURGE]

[WITHOUT PROGRAM]

Specified when changing the partitioning storage conditions of a row-partitioned or matrix-partitioned table.

The partitioning storage condition for the following tables cannot be changed:

- Tables with abstract data type columns

In the following text, columns for which the BLOB type is specified are referred to as *LOB columns*.

In an ALTER TABLE statement, table partitioning storage conditions can be partitioned and combined in the following units:

Table type	Table partitioning method		Execution type	
			Partitioning	Combining
Row-partitioned table	Key range partitioning	Boundary value specification	Partitions an arbitrarily selected boundary value into 2 to 16 values.	Combines 2 to 16 arbitrarily selected boundary values into a single value.
		Storage condition specification	Partitions an arbitrarily selected RDAREA into 2 to 16 segments.	Combines 2 to 16 arbitrarily selected RDAREAs into a single RDAREA.
Matrix-partitioned table	Key range partitioning	Boundary value specification	Partitions the boundary value of a dimension into 2 to 16 segments. Partitions *n* boundary values into 2*n* to 16*n* boundary values as a result.	Combines 2 to 16 boundary values of a dimension into one segment. Combines 2*n* to 16*n* boundary values into *n* boundary value as a result.

Legend:

n: Number of partitions with separate dimensions corresponding to the partitioning storage conditions of the table to be partitioned or joined

For SQL examples of partitioning and joining partitioning storage conditions in a table, see *9. (examples of modifying a boundary value for a row- partitioned table with a boundary value specification)*, *10. (examples of modifying an RDAREA of a row-partitioned table with a storage condition specification)*, and *11. (examples of partitioning and combining of partitioning storage conditions for a matrix-partitioned table)* in *Examples*. If a partitioning storage condition is modified, the system deletes any data stored in the RDAREA that is subject to modification when the `ALTER TABLE` command is executed. (If the `WITHOUT PURGE` option is specified, the scope of deletion of modification-object data may change. For details, see the *WITHOUT PURGE option* for details.) For this reason, any data that is required after the execution of `ALTER TABLE` must be stored again, by performing operations such as unloading the data before the partitioning storage condition is changed and loading it after the partitioning storage condition is changed. For details about how to store again the data to be deleted, see *Changing the partitioning storage condition for a table* in the manual *HiRDB Version 9 System Operation Guide*.

Modification of table partitioning storage conditions is subject to the following rules:

Items to be checked before changing table partitioning storage conditions

Before changing table partitioning storage conditions, the following items must be checked:

1. Modifying a partitioning storage condition requires *HiRDB Advanced High Availability*.
2. Some table types and partitioning methods involved prohibit a modification of partitioning storage conditions, as summarized in the following table:

Table type	Partitioning method			Modifiability
Row-partitioned table	Key range partitioning	Storage condition specification	Table in which only = is used for the storage condition comparison operator	Y
			Table in which an operator other than = is used for the storage condition comparison operator	N
		Boundary value specification		Y
	Hash partitioning			N
Matrix-partitioned table	First dimension: Boundary value specification Second dimension: Boundary value specification			Y
	First dimension: Boundary value specification Second dimension: Hash partitioning			Y#2
Non-partitioned table#1	--			N

Legend:

Y: Modifiable

N: Modifiable

--: Not applicable

#1: A table without a partitioning storage condition specification.

#2: Only the dimension of the boundary value specification can be changed. The dimension of a hash partitioning cannot be modified.

3. Partitioning storage conditions cannot be modified if table storage RDAREAs, index storage RDAREAs, and LOB column storage RDAREAs are not in 1-to-1 correspondence.
4. Partitioning storage conditions cannot be modified if all of the following conditions are met for a row-partitioned table with a storage condition specification, or a matrix-partitioned table with a combination

of key range partitioning and of a boundary value specification:

- An index key, primary key, or cluster key is defined in the table that is to be modified.
- There is only a single table storage RDAREA in the table that is to be modified or in the table resulting from the modification.

5. Partitioning storage conditions on falsification-prevented tables cannot be modified.
6. The following items must be checked if the inner replica facility is being used:

   - RDAREAs storing tables, indexes, primary keys, cluster keys, and `LOB` columns must all have the same generation number.
   - If a referential constraint is defined for a table, RDAREAs that store the referencing table and the referenced table must all have the same generation number.

Rules for modifying a table partitioning storage condition

Modifying a table partitioning storage condition is subject to the following rules:

1. The partitioning and combining of partitioning storage conditions cannot be performed simultaneously in a single execution of `ALTER TABLE`. These actions should be performed in two separate executions of `ALTER TABLE`.
2. RDAREAs of the table to be modified (RDAREAs specified in the pre-change boundary value list for a row-partitioned or matrix-partitioned table modification specification, or RDAREAs identified by the pre-change RDAREA list), index storage RDAREAs, and LOB column storage RDAREAs must have a 1-to-1 correspondence.
3. Tables, indexes, primary keys, cluster keys, and `LOB` columns must maintain correspondence between the number of partitions and any duplication of RDAREAs based on a table. (For example, if the system consolidates boundary values into a single value before and after the modification of a table, and if the RDAREA to be specified stores data other than a boundary value to be modified, the same RDAREA must be used before and after modification, and the index, the primary key, the cluster key, and the `LOB` column must also specify, in an identical manner, the position in which the same RDAREA is used. For details, see *Modifying the table partitioning storage condition* in the manual *HiRDB Version 9 System Operation Guide*.)

4. The maximum total number of partitioned RDAREAs before and after a change in table definitions is 1,024, exclusive of duplicates.
5. For a row-partitioned table with a storage condition specification, the name of the table storage RDAREA after the table definition modification must be unique.
6. The total number of partitioned RDAREAs after a change in table definitions is a maximum of 3,000, including duplicates.
7. The maximum total number of storage condition specifications after the table definition modification (including RDAREAs without storage condition specification) is 15,000.
8. The data in the table in the RDAREA specified as an object of modification is deleted by the system during the execution of `ALTER TABLE` (the data in the index storage RDAREA and the data in the `LOB` column storage RDAREA associated with the RDAREA are also deleted by the system). If the inner replica facility is used, the data in the replica RDAREAs of all generations that store the table, index, primary key, cluster key, or `LOB` column (including the `LOB`-type attribute column) that are subject to modification is also deleted by the system during the execution of `ALTER TABLE` (see item 4 in the notes about modification).
9. When `USE` is specified for the `pd_check_pending` operand in the system definition and the modification-target table is a referenced table, the table information in dictionary tables and in the RDAREA is set to check pending status for the table that is referencing the referenced table. Additionally, if the inner replica facility is used, the table information in the RDAREA is set to check pending status for all generations.
10. When the partitioning storage condition of a table that is in check pending status is modified, the following rules apply to releasing the check pending status:

    <<Check pending status specified in the table information of an RDAREA>>

    The check pending status of the modification-target table is released only from the RDAREA from which data is to be deleted.

    <<Check pending status specified in dictionary tables>>

    The table below lists the dictionary tables from which the check pending status is released when `USE` is specified for the `pd_check_pending` operand in the system definition. When `NOUSE` is specified, there are no dictionary tables from which the check pending

status is released.

Table 3-7: Dictionary tables from which check pending status is released when USE is specified for the pd_check_pending operand in the system definition

Constraint defined in the modification-target table	Is there an RDAREA in which the check pending status is specified for the table information?#	Dictionary tables from which the check pending status is released	
Referential constraint	No	SQL_REFERENTIAL_CONSTRAINTS table	CHECK_PEND column
		SQL_TABLES table	CHECK_PEND column
Check constraint	No	SQL_CHECKS table	CHECK_PEND2 column
		SQL_TABLES table	CHECK_PEND2 column
Referential constraint and check constraint	Referential constraint: No Check constraint: No	SQL_REFERENTIAL_CONSTRAINTS table	CHECK_PEND column
		SQL_CHECKS table	CHECK_PEND2 column
		SQL_TABLES table	CHECK_PEND column, CHECK_PEND2 column
	Referential constraint: No Check constraint: Yes	SQL_REFERENTIAL_CONSTRAINTS table	CHECK_PEND column
		SQL_TABLES table	CHECK_PEND column
	Referential constraint: Yes Check constraint: No	SQL_CHECKS table	CHECK_PEND2 column
		SQL_TABLES table	CHECK_PEND2 column
No constraint defined	No	--	--

Legend:

--: Not applicable

When the inner replica facility is used, includes the table RDAREAs

of the modification-target table in all generations.

11. The data to be modified is deleted according to the scope described below (when the `WITHOUT PURGE` option is specified, the scope in which the data subject to modification is deleted may vary; for details, see *WITHOUT PURGE option*):

Row-partitioned tables with a boundary value specification:

- Table, index, primary key, cluster key, and `LOB` column data of the RDAREA identified by the boundary value specified in the pre-modification boundary value list
- Data of the boundary value other than the modification target when the RDAREA identified by the boundary value specified in the pre-modification boundary value list is the same as the RDAREA that stores the data of the boundary value other than the modification target
- When the inner replica facility is used, table, index, primary key, cluster key, and `LOB` column data of all generations of replica RDAREAs of the RDAREA identified by the boundary value specified in the pre-modification boundary value list

Row-patterned tables with a storage condition specification:

- Table, index, primary key, cluster key, and `LOB` column data of the RDAREA specified in the pre-modification RDAREA information list
- When the inner replica facility is used, table, index, primary key, cluster key, and `LOB` column data of all generations of replica RDAREAs of the RDAREA identified in the pre-modification RDAREA information list

Matrix-partitioned tables:

- Table, index, primary key, cluster key, and LOB column data in the multiple RDAREAs identified by the boundary values specified in the modification target column name and pre-change boundary value list
- Data of boundary values other than the modification target when the RDAREA identified by the boundary values specified in the pre-change boundary value list is the same as the RDAREA that stores the data of the boundary values other than the modification target
- When the inner replica facility is used, table, index, primary key, cluster key, and LOB column data of all generations of replica RDAREAs of the RDAREA identified by the boundary value specified in the pre-change boundary value list

1. For a row-partitioned table or matrix-partitioned table with a boundary value specification, if the RDAREA is a duplicate (a non-adjacent RDAREA that stores a boundary value is the same RDAREA) or

contiguous (the adjacent RDAREA that stores a boundary value is the same RDAREA), the row-partitioned table and matrix-partitioned table with a boundary value specification are as listed below. For details, see the *HiRDB Version 9 System Operation Guide*.

The following shows the case of a row-partitioned table with a boundary value specification.

Item		Row-partitioned table with a boundary value specification
Modified RDAREA specified in ALTER TABLE#	Duplicate	Duplication acceptable.
	Contiguous	Contiguous allocation should be avoided.
RDAREA of an entire table of the results of modification of a partitioning storage condition	Duplicate	Duplication acceptable.
	Contiguous	During partitioning, the system consolidates contiguous RDAREAs into one area.
		Cannot be specified for combining.

The following shows the case of a matrix-partitioned table.

Item		Row-partitioned table with a boundary value specification
Modified RDAREA specified in ALTER TABLE#	Duplicate	Duplication acceptable.
	Contiguous	Contiguous allocation acceptable.
RDAREA of an entire table of the results of modification of a partitioning storage condition	Duplicate	Duplication acceptable.
	Contiguous	During partitioning, the system consolidates contiguous RDAREAs into one area.

#: RDAREA specified in a post-change boundary value partitioning specification (8-3) or matrix-partitioned table storage RDAREA modification specification (8-11).

2. A specification is not allowed if it results in the following condition of the table storage RDAREA as a result of the combining:
   - The number of table partitions is 1.
   - The number of partitions of each dimension is 1.
3. A shared RDAREA cannot be specified for a modified RDAREA.
4. If a function, procedure, or trigger is defined for the table in which partitioning storage conditions are to be modified, the function, procedure, or trigger must be disabled by specifying WITH PROGRAM.

When modifying a partitioning storage condition, observe the following points:

Notes on modification

1. If a function, procedure, or trigger is defined for a table in which partitioning storage conditions are to be modified, and if the function, procedure, or trigger is disabled by specifying `WITH PROGRAM`, the definition-nullified function, procedure, or trigger must be re-created by using `ALTER ROUTINE`. (A procedure can be re-created using `ALTER PROCEDURE`, and a trigger, by `ALTER TRIGGER`).
2. If the RDAREA identified with the boundary value to be modified is the duplicate of an RDAREA storing boundary value data that is not be to be modified, the system, which deletes all data in the table in the RDAREA to be modified, also deletes boundary value data that is not being modified.
3. If data in the table to be modified is to be re-stored in the modified table, or if a data backup is to be obtained for error recovery, operations are required such as unloading the pre-modification data by using the database reorganization utility (`pdrorg`) and loading it on the modified table, or creating a backup of the RDAREA using the database copy utility (`pdcopy`). For details, see *Changing a table's partitioning storage condition* in the manual *HiRDB Version 9 System Operation Guide*.
4. If the table being modified is a referenced table and the data in the table is to be deleted, compatibility between the referencing and referenced tables may be lost. In this case, modify the partitioning storage condition for the tables, re-store data in the modified table, and then verify the compatibility between the referencing and referenced tables. For details about how to store the data in the modified table, see *3.* above; for the verification method, see *Changing a table's partitioning storage condition* in the manual *HiRDB Version 9 System Operation Guide*.
5. If the inner replica facility is used, compatibility between the referencing and referenced tables may be lost unless the RDAREAs that store the referencing and referenced tables all have the same generation number. If compatibility between referencing and referenced tables is lost, operations such as restoring data from a backup RDAREA are required. For the operation method, see *Changing the partitioning storage condition for a table* in the manual *HiRDB Version 9 System Operation Guide*.
6. When `USE` is specified in the `pd_check_pending` operand in the system definition, and the table referencing the table whose partitioning

storage condition is to be changed is being used by another user, the referencing is set to check pending status once the transaction terminates.

8-1) *row-partitioned-table-change-specification* : : =

{ [PARTITIONED] *pre-change-boundary-value-list* INTO *post-change-boundary-value-partition-specification*

| PARTITIONED CONDITION *pre-change-RDAREA-information-list*

INTO *post-change-storage-condition-partition-specification*}

Specify this operand when modifying the partitioning storage condition of a row-partitioned table with a boundary value or storage condition specification.

PARTITIONED

Specify this operand when modifying the partitioning storage condition for a row-partitioned table with a boundary value specification.

The following rules apply to row-partitioned table modification specifications:

Partitioning

1. A single SQL statement can be used to partition a boundary value and an RDAREA associated with the boundary value.
2. Boundary values specified in a pre-modification boundary value list are partitioned according to the row-partitioned table modification specification.

Combining

1. A single SQL statement can be used to combine consecutive, multiple boundary values and the RDAREAs associated with the boundary values.
2. The specification combines the boundary values specified in a pre-modification boundary value list into the largest boundary value specified in the pre-modification boundary value list and into the RDAREA specified in the post-change boundary value partitioning specification.

PARTITIONED CONDITION

Specify this operand when modifying the partitioning storage condition for a row-partitioned table with a storage condition specification. The following rules apply to row-partitioned table modification specifications:

Partitioning

1. A single SQL statement can be used to partition a single RDAREA that

satisfies the storage condition.

2. The RDAREA specified in the pre-change RDAREA information list is partitioned according to the post-change storage condition partitioning specification.

Combining

1. A single SQL statement can be used to combine multiple RDAREAs that satisfy the storage condition.
2. The RDAREAs specified in the pre-change RDAREA information list are combined into the RDAREA specified in the post-change storage condition partitioning specification.

matrix-partitioned-table-change-specification::=

MULTIDIM(*change-target-column-name pre-change-boundary-value-list*)

AT *post-change-boundary-value-list*

INTO *matrix-partitioned-table-storage-RDAREA-change-specification*

Specify this to modify the partitioning storage condition of a matrix-partitioned table.

The following rules apply to a matrix-partitioned table storage RDAREA modification specification:

Partitioning

1. A single SQL statement can be used to partition boundary values of any dimension and the RDAREAs associated with the boundary values.
2. Using the boundary values specified in the post-change boundary value list, this specification partitions the RDAREAs identified by the pre-change boundary value list of the dimension identified by the modification target column name into the RDAREAs specified in the matrix-partitioned storage RDAREA modification specification.

Combining

1. A single SQL statement can be used to combine contiguous, multiple boundary values of any dimension and multiple RDAREAs.
2. Using the boundary values specified in the post-change boundary value list, this specification combines the RDAREAs identified by the pre-change boundary value list of the dimension identified by the modification target column name into the RDAREAs specified in the matrix-partitioned storage RDAREA modification specification.

MULTIDIM(*change-target-column-name pre-change-boundary-value-list*)

Specifies the name of the partition column and the boundary values to be changed in a matrix-partitioned table.

change-target-column-name

Specifies the name of the partition column of the dimension to be changed.

pre-change-boundary-value-list::=((*{boundary-value*|MAX}) [,(*{boundary-value*|MAX})]...)

Specifies the boundary values to be changed.

AT *post-change-boundary-value-list*

INTO *matrix-partitioned-table-storage-RDAREA-change-specification*

Specifies the RDAREA for storing the tables and the post-change boundary values.

8-2) *pre-change-boundary-value-list*::= *boundary-value-list*

boundary-value-list::= ((*boundary-value* | MAX)[, (*boundary-value* | MAX)]...)

Specifies a pre-modification boundary value, and identifies the RDAREA to be modified based on the specified boundary value.

The following rules apply to row-partitioned table modification specifications:

Partitioning

1. One boundary value or one instance of MAX can be specified.
2. Boundary values not defined in a table cannot be specified.
3. For a partitioning key value greater than the maximum boundary value specified in the table, specify MAX instead of a boundary value.

Combining

1. The number of boundary values and instances of MAX that can be specified is 2 to 16.
2. Only one instance of MAX can be specified.
3. Boundary values not defined in a table cannot be specified.
4. For a partitioning key value greater than the maximum boundary value specified in the table, specify MAX instead of a boundary value.
5. Consecutive boundary values should be specified in ascending order.
6. A given boundary value cannot be specified multiple times.

boundary-value

Specifies the boundary value that is defined for the table before the table is

modified. This operand is specified to identify the boundary value and the RDAREA to be modified.

`MAX`

Specify this option when a range greater than the maximum boundary value defined in the pre-modification table is to be modified.

post-change-boundary-value-list`::=`*boundary-value-list*

boundary-value-list`::=((`{*boundary-value*|`MAX`}`)[,(`{*boundary-value*|`MAX`}`)]...)`

Specifies the post-change boundary value.

The following rules apply to the post-change boundary value list:

Partitioning

1. The number of boundary values and instances of `MAX` that can be specified is in the range from 1 to 16.
2. Only one instance of `MAX` can be specified.
3. Specify a literal for a boundary value.
4. Specifying consecutive boundary values in ascending order.
5. Specify the value specified in the pre-change boundary value list as the last boundary value specified.
6. Boundary values outside of the range of boundary values specified in the pre-change boundary value list cannot be specified.
7. For details about the values that can be specified as boundary values, see *CREATE TABLE (Define table)* in this chapter.

Combining

1. Only one boundary value or instance of MAX can be specified.
2. Specify the value that was specified last in the pre-change boundary value list for the specified boundary value.
3. The number of partitions of a dimension has to be greater than one in the table that results from the modification.

boundary-value

Specifies the post-change boundary value.

`MAX`

Specifies the range of the last boundary value in the table that results from the modification.

8-3) *post-change-boundary-value-partitioning-specification* : : =
{*table-storage-RDAREA-name*

| (*table-storage-RDAREA-name*)

| ([(*table-storage-RDAREA-name*) *boundary-value*,]

...(*table-storage-RDAREA-name*))}

This operand changes the boundary value and the RDAREA to be modified that were specified in the pre-modification boundary value list (8-2) to a specified boundary value and RDAREA. Any data that has the specified boundary value as a partitioning key is stored in the RDAREA specified immediately before this specification.

Partitioning

1. The number of table storage RDAREA names that can be specified is 2 to 16.
2. A boundary value cannot be specified for the last RDAREA.
3. The number of boundary values that can be specified is 1 to 15.
4. In *boundary-value*, specify a literal.
5. Boundary values should be specified in ascending order.
6. For an RDAREA that stores the last range in which the boundary value to be modified is partitioned, as specified in the pre-modification boundary value list, specify an RDAREA name in the format in which a boundary value is not specified after the RDAREA.
7. Boundary values greater than the range to be modified, as specified in the pre-modification boundary value list, cannot be specified.
8. The same RDAREA name can be specified multiple times in *table-storage-RDAREA-name*. However, identical RDAREA names cannot be specified consecutively.
9. For values that can be specified in *boundary-value*, see *CREATE TABLE (Define table)*.

Combining

1. Only one table storage RDAREA name can be specified.
2. Boundary values cannot be specified.
3. A specification cannot be made such that there is only one storage RDAREA in a modified table.

Notes

If, as a result of partitioning, the RDAREAs that store adjacent boundary values are the same RDAREA, the system consolidates the boundary values into a maximum boundary value.

table-storage-RDAREA-name

Specifies the RDAREA that stores data after modification.

boundary-value

During partitioning, specifies a modified boundary value; this operand cannot be specified during combining.

8-4) *pre-change-RDAREA-information-list*::=

{*table-storage-RDAREA-name*

|(*table-storage-RDAREA-name*)

|((*table-storage-RDAREA-name*)

[,(*table-storage-RDAREA-name*)]...[,OTHERS])

|OTHERS}

Specifies the name of the pre-change table storage RDAREA. The specified table storage RDAREA name is used to specify the storage condition for the change target.

The following rules apply to the pre-change RDAREA information list:

Partitioning

1. You can specify only a single table storage RDAREA name or OTHERS.
2. You cannot specify a table storage RDAREA name not specified in the table.
3. You cannot specify a table storage RDAREA name for which only a single literal is specified for the storage condition. However, you can specify a table storage RDAREA name for which no storage condition is specified.
4. If you specify a table storage RDAREA name for which a storage condition is specified, you cannot partition the storage condition in the RDAREA into other RDAREAs and add a new storage condition.
5. If a table storage RDAREA name for which no storage condition is specified is specified, you can add a new storage condition.
6. To add a storage condition if the table whose definition is to be changed does not include a table storage RDAREA for which no storage condition is specified, specify OTHERS in place of the table storage RDAREA name.
7. If a table storage RDAREA for which no storage condition is specified does exist, you cannot specify OTHERS.

Combining

1. You can specify 2 to 16 table storage RDAREA names or OTHERS.
2. You can specify OTHERS only once.
3. You cannot specify a table storage RDAREA name not specified in the table.
4. The same table storage RDAREA name cannot be specified more than once.
5. If you make a specification containing a table storage RDAREA name for which no storage condition is specified, you can delete the storage condition.
6. To delete a storage condition if there is no table storage RDAREA name for which no storage condition is specified, specify OTHERS in place of the table storage RDAREA name.
7. If the table does include a table storage RDAREA for which no storage condition is specified, you cannot specify OTHERS.

table-storage-RDAREA-name

Specifies the table storage RDAREA name specified in the pre-change table. You specify this name to specify the storage condition to be changed.

OTHERS

Specify this option to add or delete a storage condition when a table storage RDAREA for which no storage condition is specified does not exist in the table to be changed.

8-5) *pre-change-RDAREA-information-list* ::= {*table-storage-RDAREA-name*
|(*table-storage-RDAREA-name*)

|((*table-storage-RDAREA-name*) *storage-condition*

{,(*table-storage-RDAREA-name*) *storage-condition*

[[,(*table-storage-RDAREA-name*) *storage-condition*

...]

[{,(*table-storage-RDAREA-name*)

|,OTHERS}])

|,(*table-storage-RDAREA-name*))

|,OTHERS)}

|OTHERS}

storage-condition ::=*column-name*={*literal*|(*literal*[,*literal*...]) }

Changes the change-target RDAREAs specified in the pre-change RDAREA

information list to the specified storage conditions and RDAREAs.

The following rules apply to post-change storage condition partitioning specifications:

Partitioning

1. You can specify 2 to 16 table storage RDAREA names or OTHERS.
2. You can specify OTHERS only once.
3. You can specify only one table storage RDAREA name for which no storage condition is specified.

When an RDAREA for which a storage condition is specified is specified in the pre-change RDAREA list

1. You cannot specify an RDAREA for which a storage condition is not specified or OTHERS.

When an RDAREA for which no storage condition is specified is specified in the pre-change RDAREA list

1. Include the name of the RDAREA for which no storage condition is specified or OTHERS in the specification.
2. When deleting an RDAREA for which no storage condition is specified, specify OTHERS.

When OTHERS is specified in the pre-change RDAREA list

1. Include the name of the RDAREA for which no storage condition is specified or OTHERS in the specification.
2. When adding a storage condition, specify OTHERS.

Combining

1. You can specify only a single table storage RDAREA name, or OTHERS.
2. You cannot specify a storage condition.
3. When deleting a storage condition, specify OTHERS.
4. You cannot make a specification that would eliminate all storage conditions from the combined table.

When only an RDAREA name for which a storage condition is specified is specified in the pre-change RDAREA list

1. You cannot specify an RDAREA name for which a storage condition is not specified.

When an RDAREA name for which a storage condition is not specified is also specified in the pre-change RDAREA list

1. When deleting an RDAREA for which no storage condition is specified, specify OTHERS.

When OTHERS is also specified in the pre-change RDAREA list

1. You can add an RDAREA for which no storage condition is specified.

table-storage-RDAREA-name

Specifies the name of the RDAREA that stores data after the change.

storage-condition

Specifies the post-change storage condition during partitioning. This option cannot be specified during combining.

For the column name, specify the column name specified for the partitioning key.

For the storage condition, specify a literal. For details about the values that can be specified as a storage condition, see *CREATE TABLE (Define table)* in this chapter.

When a table storage RDAREA name for which a storage condition is specified is specified in the pre-change RDAREA information list

1. Specify all storage conditions specified for the change-target table storage RDAREA.
2. You cannot specify a storage condition that does not exist.
3. You cannot specify a storage condition in duplicate.

When a table storage RDAREA name for which no storage condition is specified is specified or OTHERS in the pre-change RDAREA information list

1. Specify a storage condition that does not exist in the table definition.
2. You cannot specify a storage condition that exists in the table definition.

OTHERS

Specify this option when there is no need for an RDAREA for storing data that does not satisfy the storage condition defined in the table as a result of the table definition change.

8-6) *index-storage-RDAREA-change-specification* : : =

FOR INDEX *index-name*

INTO *post-change-index-storage-RDAREA-name-list*

Specify this operand if an index is defined for the table for which partitioning storage conditions are to be modified.

The following rules apply to RDAREAs for index modification specifications:

1. A modification cannot be specified if it only modifies an RDAREA for index.
2. The index storage RDAREA can be used to modify an RDAREA with the same range or storage conditions as that of the boundary value specified in the row-partitioned table modification specification or matrix-partitioned table modification specification.
3. All indexes that are defined in the table must be specified.

index-name

If an index is defined on the table for which partitioning storage conditions are to be modified, in this operand specify the identifier for the index that is defined.

post-change-index-storage-RDAREA-name-list ::=

{*RDAREA-name-for-index*

| (*RDAREA-name-for-index*)

| ((*RDAREA-name-for-index*)

[, (*RDAREA-name-for-index*)]...[, OTHERS])

| *2-dimension-storage-RDAREA-specification*}

| OTHERS}

2-dimension-storage-RDAREA-specification ::= (*matrix-partitioned-RDAREA-area-list*)

[, *matrix-partitioned-RDAREA-area-list*] . . .)

matrix-partitioned-RDAREA-area-list ::=

(*index-storage-RDAREA-name*

[, *index-storage-RDAREA-name*] . . .)

Specifies the name of the RDAREA that stores an index, primary key, or cluster key.

The following rules apply to modified RDAREA for index name lists:

1. The number of RDAREAs specified in *post-change-index-storage-RDAREA-name-list* must be equal to the number of RDAREAs specified in the following location:
   - Post-change boundary value partitioning specification (for a row-partitioned table with a boundary value specification)
   - Post-change storage condition partitioning specification (for a row-partitioned table with a storage condition specification)

- Matrix-partitioned table storage RDAREA change specification (for a matrix-partitioned table)

1. If a duplicate RDAREA is specified in a post-change boundary value partitioning specification, the RDAREA specified in the post-change index storage RDAREA name list must also be duplicated and specified in the same way.
2. Tables, indexes, primary keys, and cluster keys must maintain duplication correspondence with the number of partitions and RDAREAs based on a table. (For example, if the system consolidates boundary values into a single value before and after the modification of a table, and if a specified RDAREA stores data other than the boundary value to be modified, the same RDAREA must be used, and a position that specifies the same RDAREA must similarly be specified in the index, the primary key, and the cluster key.)
3. If `OTHERS` is specified in the post-change storage condition specification, `OTHERS` must also be specified in *post-change-index-storage-RDAREA-name-list* that corresponds to the post-change storage condition specification.

2-dimension-storage-RDAREA-specification

Specifies the storage RDAREA of the matrix-partitioned table.

matrix-partitioned-RDAREA-area-list

Specifies the RDAREA of a dimension in a matrix-partitioned table.

RDAREA-name-for-index

Specifies the name of the RDAREA in which modified data is to be stored.

`OTHERS`

Specify this option when `OTHERS` is specified in the post-change storage condition specification.

`OTHERS` cannot be specified if `OTHERS` is not specified in the post-change storage condition specification.

8-7) *primary-key-storage-RDAREA-change-specification* `::=`

`FOR PRIMARY KEY`

`INTO` *post-change-index-storage-RDAREA-name-list*

Specify this operand if the primary key is defined for the table for which partitioning storage conditions are to be modified.

The following rules apply to primary key storage RDAREA modification specifications:

1. Only one primary key storage RDAREA modification specification can be specified.
2. A modification that only modifies the RDAREA that stores the primary key cannot be specified.

`FOR PRIMARY KEY`

Specify this option if the primary key is defined for a table.

post-change-index-storage-RDAREA-name-list

Specifies the RDAREA that stores a modified primary key. For details, see *post-change-index-storage-RDAREA-name-list* in *8-4*.

8-8) *cluster-key-storage-RDAREA-change-specification* : : =

`FOR` [`PRIMARY`] `CLUSTER KEY`

`INTO` *post-change-index-storage-RDAREA-name-list*

Specify this operand if a cluster key is defined for the table for which partitioning storage conditions are to be modified.

The following rules apply to cluster key storage RDAREA modification specifications:

1. A modification that only modifies the RDAREA that stores a cluster key cannot be specified.
2. Only one cluster key storage RDAREA modification specification can be specified.

`FOR` [`PRIMARY`] `CLUSTER KEY`

Specify this option when modifying the RDAREA for index that stores a cluster key is to be modified.

`PRIMARY`

Specify this option if a cluster key is defined as the primary key.

post-change-index-storage-RDAREA-name-list

Specifies the name of the RDAREA that stores a modified cluster key. For details, see *post-change-index-storage-RDAREA-name-list* in *8-6*.

8-9) *LOB-column-storage-RDAREA-change-specification* : : =

`FOR COLUMN` *column-name*

{*LOB-column-storage-RDAREA-change-list*

| `INTO`
matrix-partitioned-LOB-column-storage-RDAREA-change-specification}

[, *column-name* {*LOB-column-storage-RDAREA-name-specification*

| INTO
matrix-partitioned-LOB-column-storage-RDAREA-change-specification}

]...

Specify this operand if a LOB column is defined for the table for which partitioning storage conditions are to be changed.

The following rules apply to LOB column storage RDAREA modification specifications:

1. A modification that only modifies a LOB column storage RDAREA cannot be specified.
2. In *column-name*, specify a LOB column.
3. All LOB columns that are defined for the table must be specified.

column-name

Specifies the name of the LOB column defined on the table.

8-10) *LOB-column-storage-RDAREA-change-list* ::=

INTO {*LOB-column-storage-RDAREA-name*

| (*LOB-column-storage-RDAREA-name*)

| ((*LOB-column-storage-RDAREA-name*)

[, (*LOB-column-storage-RDAREA-name*)]...[, OTHERS])

|OTHERS}

Specifies the name of the user LOB RDAREA that stores LOB column data for a row-partitioned table.

The following rules apply to LOB column storage RDAREAs:

1. For a column for which the BLOB data type is specified, a LOB column storage RDAREA name must always be specified. A LOB column storage RDAREA name cannot be specified for columns for which a non-BLOB data type is specified.
2. If an RDAREA is specified in duplicate in the post-change boundary value partitioning specification, the same duplicated RDAREA must also be specified in *LOB-column-storage-RDAREA-name*.
3. Tables and LOB columns must maintain duplication correspondence with the number of partitions and RDAREAs based on a table. (For example, if the system consolidates boundary values into a single value before and after the modification of a table, and if a specified RDAREA stores data other than the

boundary value to be modified, the same RDAREA must be used, and a position that specifies the same RDAREA must similarly be specified in the LOB column.)

4. If OTHERS is specified in the post-change storage condition specification, OTHERS must also be specified in *LOB-column-storage-RDAREA-change-list* that corresponds to the post-change storage condition specification.

LOB-column-storage-RDAREA-name

Specifies the name of the RDAREA in which the LOB column is to be stored.

OTHERS

Specify this when OTHERS is specified in a post-change storage condition partition specification.

OTHERS cannot be specified if OTHERS is not specified in a post-change storage condition partition specification.

8-11) *matrix-partitioned-table-storage-RDAREA-change-specification* ::=

2-dimension-storage-RDAREA-specification

2-dimension-storage-RDAREA-specification ::=

(*matrix-partitioned-RDAREA-area-list*

[, *matrix-partitioned-RDAREA-area-list*]...)

matrix-partitioned-RDAREA-area-list ::=

(*RDAREA-name*[,*RDAREA-name*]...)

Specifies the post-change RDAREA of a matrix-partitioned table.

The following rules apply when specifying a matrix-partitioned table storage RDAREA modification:

Partitioning (partitioned by a first-dimension partition column)

1. The number of matrix-partitioned RDAREA lists is the same as the number of boundary values specified in the post-modified boundary value list.
2. The number of RDAREAs specified in a matrix-partitioned RDAREA list is the same as the number of second-dimension partitions.
3. For an SQL example, see *Example 1* under *10* in *Examples*.

Partitioning (partitioned by a second-dimension partition column)

1. The number of matrix-partitioned RDAREA lists is the same as the number of first-dimension partitions.

2. The number of RDAREAs specified in a matrix-partitioned RDAREA list is the same as the number of boundary values specified in the post-modified boundary value list.
3. For an SQL example, see *Example 3* under *10* in *Examples*.

Combining (combining by a first-dimension partition column)

1. The number of matrix-partitioned RDAREA lists is 1.
2. The number of RDAREAs specified in a matrix-partitioned RDAREA list is the same as the number of second-dimension partitions.
3. For an SQL example, see *Example 2* under *10* in *Examples*.

Combining (combining by a second-dimension partition column)

1. The number of matrix-partitioned RDAREA lists is the same as the number of first-dimension partitions.
2. The number of RDAREAs specified in a matrix-partitioned RDAREA list is 1.
3. For an SQL example, see *Example 4* under *10* in *Examples*.

Note

The following notes apply to specifying a matrix-partitioned table storage RDAREA modification specification:

1. The system does not combine RDAREAs even when RDAREAs storing adjacent boundary values in a table that is the result of a modification are the same (the system combines RDAREAs when a row-partitioned table with a boundary value specification is modified).

2-dimension-storage-RDAREA-specification

Specifies the storage RDAREA of the matrix-partitioned table.

matrix-partitioned-RDAREA-area-list

Specifies the RDAREA of a dimension in a matrix-partitioned table.

RDAREA-name

Specifies the name of the RDAREA that stores the post-change data.

matrix-partitioned-LOB-column-storage-RDAREA-change-specification ::=

2-dimension-storage-RDAREA-specification

2-dimension-storage-RDAREA-specification ::=

(*matrix-partitioned-RDAREA-area-list*

[, *matrix-partitioned-RDAREA-area-list*]...)

matrix-partitioned-RDAREA-area-list::=

(*RDAREA-name*[,*RDAREA-name*]...)

Specifies the RDAREA name to be used for storage when an LOB column is defined in a matrix-partitioned table.

The following rules apply to a matrix-partitioned table storage RDAREA modification specification:

1. Specify a LOB column storage RDAREA name for the column with a data type of BLOB. This cannot be specified for a column whose data type is not BLOB.
2. If a duplicate RDAREA is specified in a matrix-partitioned table modification specification, the LOB column storage RDAREA must also be specified in duplicate.
3. Tables and LOB columns must maintain correspondence between the number of partitions and any duplicate RDAREA based on a table. For example, if the system consolidates boundary values into a single value before and after a table is modified, and if the RDAREA being specified stores data other than the boundary values being modified, the same RDAREA must be used before and after the modification, and the LOB column must specify, in an identical manner, the position and other information for that RDAREA.

2-dimension-storage-RDAREA-specification

Specifies storage RDAREA for the matrix-partitioned table being described.

matrix-partitioned-RDAREA-area-list

Lists the RDAREA of the dimensions in a matrix-partitioned table.

RDAREA-name

Specifies the name of the RDAREA that stores the post-change data.

WITHOUT PURGE

If the data in a post-change table does not fall within the boundary value range of the partitioning storage condition or match the storage condition specification, HiRDB deletes that data during execution of ALTER TABLE. However, specify this so that the same RDAREA table data is not deleted by HiRDB when executing ALTER TABLE if the same RDAREA is specified before and after modification (when the modification target RDAREA is specified in a post-change boundary value partition specification, post-change storage condition partition specification, or matrix-partitioned table modification specification). Even after the partitioning storage condition is changed, this specification allows you to use the table data as it existed before the change, which helps reduce table data unload and load operations.

However, the table data in the RDAREA that is not used after modification, even with WITHOUT PURGE specification, loses association with the table to be modified, and can no longer maintain compatibility. In this case, the system deletes the data.

WITHOUT PURGE can be specified when the conditions below are satisfied (for a matrix-partitioned table, when the conditions below are satisfied in each dimension used for partitioning and combining RDAREAs). Specifying WITHOUT PURGE in all other cases results in an error. (for details, see *Changing the partitioning storage condition for a table* in the manual *HiRDB Version 9 System Operation Guide*):

Row-partitioned table with a boundary value specification

The modification target RDAREA must be included in the post-change RDAREA.

Row-partitioned table with a storage condition specification

The modification target RDAREA must be included in the post-change RDAREA.

Matrix-partitioned tables

1. The modification target RDAREAs must be included in the post-change RDAREAs.
2. If the order of specification of the RDAREAs is the same

Notes

When specifying the WITHOUT PURGE option, observe the following points:

1. When partitioning a partitioning storage condition, you must verify that all data in the change-target RDAREA matches the data in the boundary value range to be allocated after partitioning, or the storage condition to be allocated after partitioning. For details, see *Changing the partitioning storage condition for a table* in the manual *HiRDB Version 9 System Operation Guide*.
2. When combining partitioning storage conditions, if the data in the RDAREA to be deleted must be reregistered, the data must be unloaded before the partitioning storage condition is modified, and it must be loaded to the table that is the result of the modification of the partitioning storage conditions. However, if data that is not to be deleted with WITHOUT PURGE is loaded after modifying the partitioning storage condition, duplicate data will be registered. Therefore, caution must be taken with regards to the unloading and loading of data to the RDAREA. For details, see *Changing the partitioning storage condition for a table* in the manual *HiRDB Version 9 System Operation Guide*.

3. For a row-partitioned table with a storage condition specification, you cannot specify WITHOUT PURGE if OTHERS is specified in the pre-change RDAREA information list during partitioning of the partitioning storage condition.

WITH PROGRAM

Specify this operand when the SQL object in which functions, procedures, and triggers that use the table being modified are in effect is to be nullified.

Common rules

1. Changing a column in a base table using the CHANGE clause also changes the columns in the view table.
2. ALTER TABLE cannot be specified for a column for which a storage condition is specified.
3. If WITH PROGRAM is omitted, the table definition can be changed if there is a table to be defined or an SQL object with an effective function, procedure or trigger that uses the view table defined by referencing that table.
4. A non-repetition column cannot be changed into a repetition column. Conversely, a repetition column cannot be changed into a non-repetition column.
5. When adding abstract data type columns that include a BLOB column or a BLOB attribute in a row-partitioned table, hash-partitioned table, or matrix-partitioned table of a boundary value specification, each LOB column storage RDAREA name or LOB attribute storage RDAREA name must be specified to correspond with the table storage RDAREA name that is specified in the table definition. Therefore, if there is an RDAREA name duplicate for the table storage RDAREA name that is specified in the table definition, the LOB column storage RDAREA name or the LOB attribute storage RDAREA name must also be specified in duplicate.
6. The table storage RDAREA, the LOB column storage RDAREA, the LOB attribute storage RDAREA, and the RDAREA for index that are specified in ADD COLUMN, ADD RDAREA, or CHANGE RDAREA should be created in advance using the database initialization utility, or must be added using the database structure modification utility.
7. LOB column storage RDAREAs, LOB attribute storage RDAREAs, and index storage RDAREAs using a user LOB RDAREA cannot be specified in duplicate. For each, a different user LOB RDAREA must be specified.
8. User LOB RDAREAs that are assigned to other BLOB columns, the BLOB attribute, or an index cannot be specified.
9. If the SQL object is nullified during the execution of this command, ALTER TABLE cannot be executed from within a Java procedure.
10. RDAREAs using the inner replica facility and those not using the facility cannot

be specified on a mixed basis in a table storage RDAREA, LOB column storage RDAREA, LOB attribute storage RDAREA, or RDAREA for index that is specified in ADD COLUMN, ADD RDAREA, or CHANGE RDAREA. When specifying an RDAREA to which the inner replica facility is applied, specify the name of the original RDAREA.

11. For execution conditions for ALTER TABLE using the inner replica facility, see the manual *HiRDB Version 9 Staticizer Option Description and User's Guide*.
12. ALTER TABLE cannot be executed for an audit trail table.
13. A table definition cannot be modified using the DROP clause on referenced and referencing tables.
14. Referenced and referencing tables cannot be renamed using the RENAME clause.
15. Modifying the definition of the primary key constituent column or foreign key constituent column for a referenced table is subject to the following constraints:
    - The data type or data length cannot be modified using the CHANGE clause.
    - Columns cannot be renamed using the RENAME clause.
16. The definition of a table for which a check constraint is defined cannot be modified using the DROP clause.
17. Tables for which a check constraint is defined cannot be renamed using the RENAME clause.
18. Any change of definition of columns for which a check constraint is defined is subject to the following constraints:
    - The data type or data length cannot be modified using the CHANGE clause.
    - SPLIT cannot be modified using the CHANGE clause.
    - Default values cannot be assigned or released using the CHANGE clause.
    - The WITH DEFAULT option cannot be set using the CHANGE clause.
    - Columns cannot be renamed using the RENAME clause.

Notes

1. If data is stored in the base table for which FIX was specified during the table definition, the following items can be specified in ALTER TABLE:
    - Adding a table storage RDAREA
    - Changing hash functions
    - Changing the column attribute from CHAR to MCHAR
    - Renaming tables and columns

- Changing units of minimum locking resources
- Changing the partitioning storage condition
- Assigning an updatable column attribute (UPDATE specification only)
- Changing a table to a falsification-prevented table

2. WITH DEFAULT cannot add columns to a table already containing data.
3. When passing or receiving date data in CHAR(10) using the row-unit interface, specify a column in CHAR(10) instead of a date data type.
4. When passing or receiving time data in CHAR(8) using the row-unit interface, specify a column in CHAR(8) instead of a time data type.
5. When passing or receiving time stamp data in 19, 22, 24, or 26-byte CHAR using the row-unit interface, specify a column in 19, 22, 24, or 26-byte CHAR instead of a time stamp data type.
6. ALTER TABLE cannot be specified from an X-Open compliant UAP running under OLTP.
7. If a table or column is renamed, when using a utility or an operating command, the new name should be specified.
8. If a table or column is renamed, any of the following files that were created before the renaming action cannot be used:
   - Unload data files for the database reorganization utility
   - Index information files for the database load utility or the database reorganization utility
   - Export files for the dictionary import/export utility
9. When changing a hash function or adding a table storage RDAREA, data need not be reloaded. However, because data is not stored in the added table storage RDAREA until the INSERT statement is executed, data is not stored when the table storage RDAREA is added.
10. If an SQL object for which functions, procedures, or triggers are in effect is nullified by specifying WITH PROGRAM, any rows associated with the nullified functions, procedures, or triggers in the SQL_ROUTINE_RESOURCES dictionary table are deleted.
11. Before executing the SQL object associated with the function, procedure, or trigger that was nullified by specifying WITH PROGRAM, you need to re-create the function, procedure, or trigger by executing ALTER ROUTINE, ALTER PROCEDURE, or ALTER TRIGGER.
12. Column data suppression cannot be specified for columns that are added by using the ADD option of ALTER.

13. CHANGE LOCK PAGE cannot be specified for tables for which the WITHOUT ROLLBACK option is specified.
14. Columns can be added to a table for which a trigger is defined. If a trigger event column is omitted from the trigger, any added columns are also subject to the execution of the trigger. If a trigger event column is specified, those columns are not subject to the trigger execution. Adding a column does not nullify the SQL object.
15. Trigger event columns can be changed in terms of definition, or they can be deleted. Deleting all trigger event columns associated with a trigger causes the trigger to be deleted. Adding a column with the same name as a deleted column does not make the new column a target of the execution of the trigger. Changing the definition of a trigger event column or deleting the column does not nullify the SQL object for the trigger.
16. If, after deleting a trigger event column, a different operation is used to nullify the SQL object for the trigger, one of the following operations must be performed before the nullified SQL object can be executed:
    - Reset the column definition, and execute either ALTER TRIGGER or ALTER ROUTINE to re-create the SQL object for the trigger.
    - Delete the nullified trigger by using DROP TRIGGER, and then redefine the trigger by using CREATE TRIGGER so that the deleted column is not reused. If triggers satisfying all of the following conditions exist, use DROP TRIGGER to delete them all, and redefine the triggers using CREATE TRIGGER in the order in which they were defined so that there is no change in sequence of execution of trigger actions.

      Conditions:

      - The defined trigger is later than the nullified trigger.
      - The nullified trigger is the same as the defined table.
      - The nullified trigger is the same as the trigger event (INSERT, UPDATE, or DELETE) (for UPDATE, the nullified trigger is considered to be the same as the trigger event regardless of whether a trigger event column is specified or the contents of the specification).
      - The nullified trigger has the same trigger action timing (BEFORE or AFTER) as the trigger event.
      - The nullified trigger has the same trigger action units (units of rows or statements) as the trigger event.
17. Adding a NOT NULL column to the table being referenced from a trigger SQL statement by specifying WITH PROGRAM nullifies the SQL object associated with the trigger. Before executing the nullified SQL object, you need to re-create the

trigger SQL object by executing either ALTER TRIGGER or ALTER ROUTINE.

18. Any of the following operations performed by specifying WITH PROGRAM nullifies the SQL object for the trigger:

    - Redefining a column deleting a column, renaming a column or renaming the table with respect to the table that is referenced from the trigger SQL statement.
    - Redefining or deleting the column being referenced from a trigger SQL statement by using an old or new-values correlation name

    To execute the SQL object for the nullified trigger, you need to perform one of the following operations:

    - Reset the column definition, column name, or table name, and then execute either ALTER TRIGGER or ALTER ROUTINE to re-create the SQL object for the trigger.
    - Use CREATE TRIGGER to redefine the trigger so that, after DROP TRIGGER is used to delete the nullified trigger, the column on which column definition change or column deletion was performed is not used, or so that the old column name or old table name is not used. If there are triggers that satisfy all of the following conditions, delete them all by using DROP TRIGGER and redefine them by using CREATE TRIGGER in the order in which they were originally defined, so that the order in which trigger actions are executed does not change.

      Conditions:

      - The defined trigger is later than the nullified trigger.
      - The nullified trigger is the same as the defined table.
      - The nullified trigger is the same as the trigger event (INSERT, UPDATE, or DELETE) (for UPDATE, the nullified trigger is considered to be the same as the trigger event regardless of whether a trigger event column is specified or the contents of the specification).
      - The nullified trigger has the same trigger action timing (BEFORE or AFTER) as the trigger event.
      - The nullified trigger has the same trigger action units (units of rows or statements) as the trigger event.

19. Changing its definition or deleting the column being referenced by using an old or new-values correlation name in a trigger action condition by specifying WITH PROGRAM causes the SQL object for the trigger to be nullified. Similarly, pre-processing a trigger-inducing SQL statement also causes an error. To execute the SQL object for a nullified trigger or perform pre-processing on a trigger-inducing SQL statement, you need to perform one of the following

operations:

- Reset the column definition, and execute either ALTER TRIGGER or ALTER ROUTINE to re-create the SQL object.
- Use CREATE TRIGGER to redefine the trigger so that, after DROP TRIGGER is used to delete the nullified trigger, the column on which column definition change or column deletion was performed is not used. If there are triggers that satisfy all of the following conditions, delete them all by using DROP TRIGGER and redefine them by using CREATE TRIGGER in the order in which they were originally defined, so that the order in which trigger actions are executed does not change.

  Conditions:

  - The defined trigger is later than the nullified trigger.
  - The nullified trigger is the same as the defined table.
  - The nullified trigger is the same as the trigger event (INSERT, UPDATE, or DELETE) (for UPDATE, the nullified trigger is considered to be the same as the trigger event regardless of whether a trigger event column is specified or the contents of the specification).
  - The nullified trigger has the same trigger action timing (BEFORE or AFTER) as the trigger event.
  - The nullified trigger has the same trigger action units (units of rows or statements) as the trigger event.

20. The table on which the trigger is defined or any of the following columns cannot be renamed:
    - Trigger event columns
    - Columns that are referenced in a trigger action condition, using an old or new-values correlation name
    - Columns that are referenced in a trigger SQL statement, using an old or new-values correlation name

Examples

1. Add a warehouse address (PADRS) to an inventory table (STOCK):
```
ALTER TABLE STOCK
    ADD PADRS VARCHAR(40)
```

2. Add a warehouse address (PADRS) in a NOT NULL constrained column with a predefined value to an inventory table (STOCK):
```
ALTER TABLE STOCK
    ADD PADRS VARCHAR(40)
    NOT NULL WITH DEFAULT
```

3. In the inventory table (STOCK), change the maximum length of the column of a variable-length data type warehouse address (PADRS) to 60:
```
ALTER TABLE STOCK
    CHANGE PADRS VARCHAR(60)
```

4. In the inventory table (STOCK), change the attribute of the cluster key that is assigned to the product code (PCODE) column from non-uniqueness-constrained to uniqueness-constrained:
```
ALTER TABLE STOCK
    CHANGE CLUSTER KEY UNIQUE
```

5. From the inventory table (STOCK), delete the warehouse address (PADRS) column:
```
ALTER TABLE STOCK
    DROP PADRS
```

6. Nullify the effective object of a procedure on the warehouse address (PADRS) in the inventory table (STOCK), and delete the warehouse address (PADRS) column:
```
ALTER TABLE STOCK
    DROP PADRS WITH PROGRAM
```

7. Add a warehouse map (PMAP) to the inventory table (STOCK):
```
ALTER TABLE STOCK
    ADD PMAP BLOB(1M) IN (RMAPLOB1)
```

8. Add an RDAREA in which a new hash-partitioned inventory table (NSTOCK) is to be stored. Also, add an index (IPCODE) and an RDAREA in which a column of a BLOB-type warehouse map (PMAP) is to be stored:
```
ALTER TABLE NSTOCK
    ADD RDAREA RDA3
      FOR COLUMN PMAP IN (RMAPLOB3)
      FOR INDEX IPCODE IN (RDA4)
```

9. The following are examples of modifying a boundary value for a row- partitioned table with a boundary value specification:

   Example 1: Partitioning and combining boundary values

   Before

```
CREATE FIX TABLE "T1"("C1" INT,"C2" INT) PARTITIONED BY "C1"

IN(("TA1")100,("TA2")200,("TA3")400,("TA4")500,("TA5")600,(
"TA6"))
CREATE INDEX "I1" ON "T1"("C1")
 IN(("IA1"),("IA2"),("IA3"),("IA4"),("IA5"),("IA6"))
```

State 1

Boundary value	100	200		400	500	600
RDAREA	TA1	TA2	TA3	TA4	TA5	TA6
	IA1	IA2	IA3	IA4	IA5	IA6

1. Combine (state 1 to state 2)

```
ALTER TABLE "T1" CHANGE RDAREA
  ((100),(200)) INTO "TA11"
  FOR INDEX "I1" INTO "IA11"
```

State 2

Boundary value	200		400	500	600
RDAREA	TA11	TA3	TA4	TA5	TA6
	IA11	IA3	IA4	IA5	IA6

2. Partition (state 2 to state 3)

```
ALTER TABLE "T1" CHANGE RDAREA
  ((400)) INTO (("TA12")300,("TA13"))
  FOR INDEX "I1" INTO (("IA12"),("IA13"))
```

State 3

Boundary value	200	300	400	500	600
RDAREA	TA11	TA12	TA13	TA4	TA5	TA6
	IA11	IA12	IA13	IA4	IA5	IA6

3. Combine (state 3 to state 4)

```
ALTER TABLE "T1" CHANGE RDAREA
  ((600),(MAX)) INTO "TA11"
  FOR INDEX "I1" INTO "IA11"
```

State 4

Boundary value	200	300	400	500	
RDAREA	TA11	TA12	TA13	TA4	TA11
	IA11	IA12	IA13	IA4	IA11

A#

#: States 1, 2, and 3 are required to set to state A.

Example 2: The system combines adjacent boundary values (a special case of partitioning)

Before

```
CREATE FIX TABLE "T1"("C1" INT,"C2" INT) PARTITIONED BY "C1"
  IN(("TA1")100,("TA2")200,("TA3")400,
     ("TA4")500,("TA5")600,("TA6"))
CREATE INDEX "I1" ON "T1"("C1")
  IN(("IA1"),("IA2"),("IA3"),
     ("IA4"),("IA5"),("IA6"))
```

Boundary value	100	200	400	500	600	
RDAREA	TA1	TA2	TA3	TA4	TA5	TA6
	IA1	IA2	IA3	IA4	IA5	IA6

Changing boundary values

```
ALTER TABLE "T1" CHANGE RDAREA
  ((400)) INTO (("TA3")300,("TA4"))
FOR INDEX "I1" INTO (("IA3"),("IA4"))
```

Boundary value	100	200	300	500	600	
RDAREA	TA1	TA2	TA3	TA4	TA5	TA6
	IA1	IA2	IA3	IA4	IA5	IA6

10. The following are examples of modifying an RDAREA of a row-partitioned table with a storage condition specification:

Before change (state 1)

```
CREATE FIX TABLE "T1"("C1" CHAR(3),"C2" INT)
  IN(("TA1")"C1"='001',("TA2")"C1"='002',
  ("TA3")"C1"='003',("TA4")"C1"=('004','005'),("TA5"))
```

```
CREATE INDEX "I1" ON "T1"("C1")
  IN(("IA1"),("IA2"),("IA3"),("IA4"),("IA5"))
```

State 1

Storage condition	='001'	='002'	='003'	=('004','005')	No specification
RDAREA	TA1	TA2	TA3	TA4	TA5
	IA1	IA2	IA3	IA4	IA5

Example 1: Adding a storage condition

1. Partitioning (state 1 to state 2)

```
ALTER TABLE "T1" CHANGE RDAREA PARTITIONED CONDITION
  (("TA5")) INTO (("TA6")"C1"='006',("TA5"))
   FOR INDEX "I1" INTO (("IA6"),("IA5"))
```

State 2

Storage condition	='001'	='002'	='003'	=('004','005')	='006'	No specification
RDAREA	TA1	TA2	TA3	TA4	TA6	TA5
	IA1	IA2	IA3	IA4	IA6	IA5

Example 2: Deleting a storage condition

2. Combining (state 1 to state 3)

```
ALTER TABLE "T1" CHANGE RDAREA PARTITIONED CONDITION
  (("TA1"),("TA5")) INTO "TA5"
   FOR INDEX "I1" INTO "IA5"
```

State 3

Storage condition	='002'	='003'	=('004','005')	No specification
RDAREA	TA2	TA3	TA4	TA5
	IA2	IA3	IA4	IA5

Example 3: Partitioning an RDAREA

3. Partitioning (state 1 to state 4)

```
ALTER TABLE "T1" CHANGE RDAREA PARTITIONED CONDITION
  (("TA4")) INTO (("TA4")"C1"='004',("TA7")"C1"='005')
   FOR INDEX "I1" INTO (("IA4"),("IA7"))
```

State 4

Storage condition	='001'	='002'	='003'	='004'	='005'	No specification
RDAREA	TA1	TA2	TA3	TA4	TA7	TA5
	IA1	IA2	IA3	IA4	IA7	IA5

Example 4: Combining RDAREAs

4. Combining (state 1 to state 5)

```
ALTER TABLE "T1" CHANGE RDAREA PARTITIONED CONDITION
  (("TA1"),("TA2")) INTO "TA2"
   FOR INDEX "I1" INTO "IA2"
```

State 5

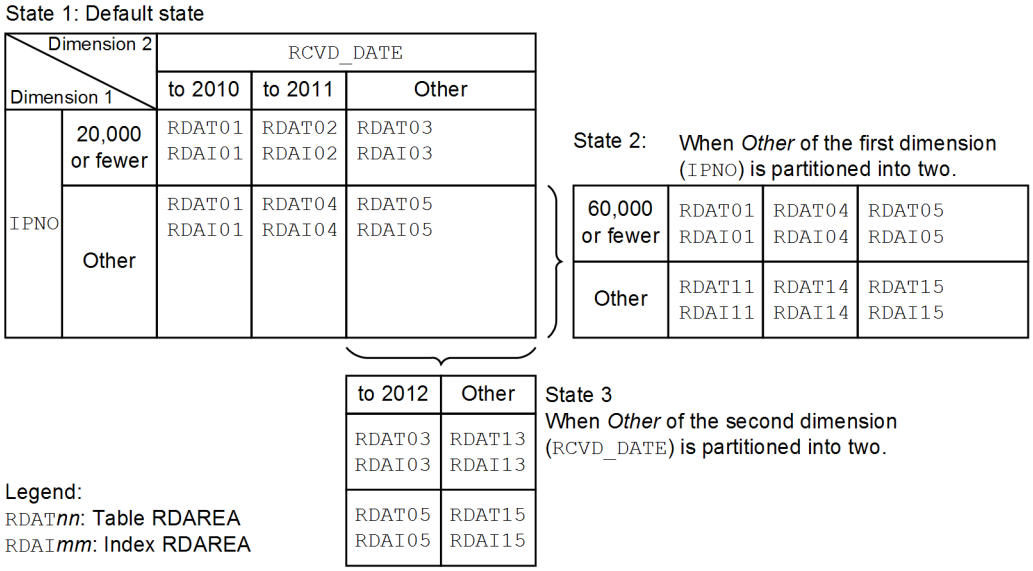
Storage condition	=('001','002')	='003'	=('004','005')	No specification
RDAREA	TA2	TA3	TA4	TA5
	IA2	IA3	IA4	IA5

11. The following are examples of partitioning and combining of partitioning storage conditions for a matrix-partitioned table.

Before

Define an inventory table (STOCK). Boundary values are set to each product code (IPNO) and inventory date (RCVD_DATE) at this time, and data for each is stored in the appropriate RDAREA. This will be referred to as *state 1*.

```
CREATE FIX TABLE "USERA"."STOCK"
("IPNO" CHAR(5),"IPNAME" CHAR(8),"PLANNED" CHAR(3),"PRICE"
INTEGER,"RCVD_DATE" CHAR(10))
  PARTITIONED BY MULTIDIM
    ("IPNO"      (('20000')),
     "RCVD_DATE" (('2010-12-31'),('2011-12-31')))
    IN (("RDAT01","RDAT02","RDAT03"),
        ("RDAT01","RDAT04","RDAT05"))
  CLUSTER KEY ("IPNO","RCVD_DATE")
    IN (("RDAI01","RDAI02","RDAI03"),
        ("RDAI01","RDAI04","RDAI05"))
```



Example 1: Partitioning RDAREAs in a first-dimension column (state 1 ➔ state 2)

```
ALTER TABLE "USERA"."STOCK" CHANGE RDAREA
MULTIDIM ("IPNO" ((MAX))) AT (('60000'),(MAX))
  INTO (("RDAT01","RDAT04","RDAT05"),
        ("RDAT11","RDAT14","RDAT15"))
FOR CLUSTER KEY
  INTO (("RDAI01","RDAI04","RDAI05"),
        ("RDAI11","RDAI14","RDAI15"))
```

Note

The RDAT01, RDAT04, and RDAT05 data and the RDAI01, RDAI04, and RDAI05 keys will be deleted.

Example 2: Combining RDAREAs in a first-dimension column (state 2 ➔ state 1)

```
ALTER TABLE "USERA"."STOCK" CHANGE RDAREA
MULTIDIM ("IPNO" (('60000'),(MAX))) AT ((MAX))
  INTO (("RDAT01","RDAT04","RDAT05"))
FOR CLUSTER KEY
  INTO (("RDAI01","RDAI04","RDAI05"))
```

Note

The RDAT01, RDAT11, RDAT04, RDAT14, RDAT05, and RDAT15 data and the RDAI01, RDAI11, RDAI04, RDAI14, RDAI05, and RDAI15 keys will be deleted.

Example 3: Partitioning RDAREAs in a first-dimension column (state 1 ➜ state 3)

```
ALTER TABLE "USERA"."STOCK" CHANGE RDAREA
MULTIDIM ("RCVD_DATE" ((MAX))) AT (('2012-12-31'),(MAX))
  INTO (("RDAT03","RDAT13"),
        ("RDAT05","RDAT15"))
FOR CLUSTER KEY
  INTO (("RDAI03","RDAI13"),
        ("RDAI05","RDAI15"))
```

Note

The RDAT03 and RDAT05 data and the RDAI03 and RDAI05 keys will be deleted.

Example 4: Combining RDAREAs in a first-dimension column (state 3 ➜ state 1)

```
ALTER TABLE "USERA"."STOCK" CHANGE RDAREA
MULTIDIM ("RCVD_DATE" (('2012-12-31'),(MAX))) AT ((MAX))
  INTO (("RDAT03"),
        ("RDAT05"))
FOR CLUSTER KEY
  INTO (("RDAI03"),
        ("RDAI05"))
```

Note

The RDAT03, RDAT13, RDAT05 and RDAT15 data and the RDAI03, RDAI13, RDAI05 and RDAI15 keys will be deleted.

12. Change a non-falsification-prevented order table (ORDER) to a falsification-prevented table with the following conditions:

    Table definition conditions

    - In advance, define an OINSDATE column as an insert history maintenance column.
    - Define a UACOL column as an updatable column after the table is changed to a falsification-prevented table.

```
CREATE TABLE ORDER
    (FNO CHAR(6),TCODE CHAR(5),PCODE CHAR(4),
     UACOL CHAR(60) UPDATE,
     OQTY INTEGER,ODATE DATE,OTIME TIME,
    OINSDATE DATE NOT NULL WITH DEFAULT SYSTEM GENERATED)
```

Condition for changing the table definition

- Set a deletion-prevented duration of 10 years.

```
ALTER TABLE ORDER
    CHANGE INSERT ONLY WHILE 10 YEARS BY OINSDATE
```

ALTER TRIGGER (Re-create a trigger SQL object)

Function

ALTER TRIGGER re-creates a trigger SQL object.

Privileges

Owner of a trigger

This user can re-create SQL objects for his or her own trigger.

DBA privilege users

These users can re-create their own triggers and SQL objects that are owned by other users.

Format

```
ALTER TRIGGER [authorization-identifier.]trigger-identifier
    {CHANGE|ALTER} ROUTINE OBJECT
    [SQL compile-option[SQL compile-option]...]

SQL compile-option ::={ISOLATION data-guarantee-level [FOR UPDATE EXCLUSIVE]
          |OPTIMIZE LEVEL SQL-optimization-option
                [,SQL-optimization-option]...
                 |ADD OPTIMIZE LEVEL SQL-extension-optimizing-option
                [,SQL-extension-optimizing-option]...
|SUBSTR LENGTH maximum-character-length}
```

Operands

- [*authorization-identifier*.]*trigger-identifier*

authorization-identifier

Specifies the authorization identifier of the owner of the trigger for which an SQL object is to be re-created. The default is the authorization identifier of the user who executes ALTER TRIGGER.

trigger-identifier

Specifies the identifier for the trigger for which an SQL object is to be re-created.

- {CHANGE|ALTER} ROUTINE OBJECT

This indicates that a trigger is to be re-created. Either CHANGE or ALTER can be specified without a change in meaning.

- SQL *compile-option*::={ISOLATION *data-guarantee-level* [FOR UPDATE

```
EXCLUSIVE]
      |OPTIMIZE LEVEL SQL-optimization-option
       [, SQL-optimization-option]...
      |ADD OPTIMIZE LEVEL SQL-extension-optimizing-option
       [, SQL-extension-optimizing-option]...
   |SUBSTR LENGTH maximum-character-length}
```

In `SQL` *compile-option*, `ISOLATION`, `OPTIMIZE LEVEL`, `ADD OPTIMIZE`, and `SUBSTR LENGTH` can each be specified only once.

`[ISOLATION` *data-guarantee-level* `[FOR UPDATE EXCLUSIVE]]`

Specifies an SQL data integrity guarantee level.

data-guarantee-level

A data integrity guarantee level specifies the point to which the integrity of the transaction data must be guaranteed. The following data integrity guarantee levels can be specified:

- `0`

  This option is specified when the integrity of data is not to be guaranteed.

  Level 0 permits the referencing of data even when the data is being updated by another user, without waiting for completion of the update process. However, if the table to be referenced is a shared table and another user is executing the `LOCK` statement in the lock mode, the system waits until the lock condition is released.

- `1`

  This option is specified when the integrity of data is to be guaranteed until the end of the retrieval process.

  Level 1 prevents other users from updating retrieved data until the retrieval process is completed (until HiRDB finishes viewing the page or row).

- `2`

  This option is specified when the integrity of retrieved data is to be guaranteed until the end of a transaction.

  Level 2 prevents other users from updating retrieved data until termination of the transaction.

`[FOR UPDATE EXCLUSIVE]`

Specify this operand if WITH EXCLUSIVE LOCK is always to be assumed, irrespective of the data guarantee level specified in SQL *compile-option* for a cursor or query in a procedure for which the FOR UPDATE clause is specified or assumed. If level 2 is specified in *data-guarantee-level*, WITH EXCLUSIVE LOCK is assumed for the cursor or query in a procedure for which the FOR UPDATE clause is specified or assumed, in which case it is not necessary to specify FOR UPDATE EXCLUSIVE. If a data guarantee level is specified in SQL *compile-option* and FOR UPDATE EXCLUSIVE is omitted, it is assumed that FOR UPDATE EXCLUSIVE is not specified.

Relationship to client environment definition

Any specification of PDISLLVL or PDFORUPDATEEXLOCK with respect to ALTER TRIGGER has no effect.

Relationship to SQL statements

If the lock option is specified in an SQL statement in a procedure, the lock option specified in the SQL statement takes precedence over the data guarantee level specified in SQL *compile-option* or the lock option assumed from FOR UPDATE EXCLUSIVE.

The default for this operand is the value that was specified during the previous SQL object creation (during the execution of CREATE TRIGGER, ALTER TRIGGER, or ALTER ROUTINE).

For data guarantee levels, see the *HiRDB Version 9 UAP Development Guide*.

[OPTIMIZE LEVEL *SQL-optimization-option*[, *SQL-optimization-option*]...]

Specifies an optimization method for determining the most efficient access path by taking the condition of the database into consideration.

An SQL optimization option can be specified using either an identifier (character string) or a numeric value. For most cases, Hitachi recommends the use of an identifier.

The default for this operand is the value that was specified during the previous SQL object creation (CREATE TRIGGER, ALTER TRIGGER, or ALTER ROUTINE).

Specifying with an identifier

OPTIMIZE LEVEL "*identifier*"[, "*identifier*"]...

Specification examples

- Applying prioritized nest-loop-join and rapid grouping processing:

  OPTIMIZE LEVEL "PRIOR_NEST_JOIN", "RAPID_GROUPING"

- Applying no optimization:

  ```
  OPTIMIZE LEVEL "NONE"
  ```

Rules

1. Specify one or more identifiers.
2. When specifying two or more identifiers, delimit them with commas (,).
3. For details about what can be specified in *identifier* (optimization methods), see *Table 3-8 SQL optimization option specification values* (ALTER TRIGGER).
4. If no optimization is to be applied, specify `NONE` in *identifier*. If an identifier other than `NONE` is specified at the same time, `NONE` is nullified.
5. Identifiers can be specified in both lower case and upper case characters.
6. If the same identifier is specified more than once, it is treated as if it was specified only once; however, where possible, precautions should be taken to avoid specifying a given identifier in duplicate.

Specifying a numeric value

```
OPTIMIZE LEVEL unsigned-integer[, unsigned-integer]...
```

Specification examples

- Creating multiple SQL objects, suppressing the use of AND multiple indexes, and forcing the use of multiple indexes

  For specifying unsigned integers by delimiting them with commas:

  ```
  OPTIMIZE LEVEL 4, 10, 16
  ```

  For specifying the sum of unsigned integers:

  ```
  OPTIMIZE LEVEL 30
  ```

- Adding 16 when 14 (4 + 10) is already specified:

  ```
  OPTIMIZE LEVEL 14, 16
  ```

- Applying no optimization:

  ```
  OPTIMIZE LEVEL 0
  ```

Rules

1. When HiRDB is upgraded from a version earlier than Version 06-00 to a Version 06-00 or later, the total value specification in the earlier version also remains valid. If the optimization option does not need to be modified, the specification value for this operand need not be changed when HiRDB is upgraded to a Version 06-00 or later.
2. Specify one or more unsigned integers.
3. When specifying two or more unsigned integers, separate them with commas (,).
4. For details about what can be specified in an unsigned integer (optimization method), see *Table 3-8 SQL optimization option specification values* (ALTER TRIGGER).
5. When not applying any optimization, specify `0` in *unsigned-integer*. If non-zero identifiers are specified at the same time, the specification of 0 is nullified.
6. If the same unsigned integer is specified more than once, it is treated as if it was specified only once; however, where possible, precautions should be taken to avoid specifying a given unsigned integer in duplicate.
7. When specifying multiple optimization methods, you can specify the sum of their unsigned integers. However, care should be taken not to add the value of the same optimization method multiple times (to prevent the possibility of the resulting sum from being interpreted as a separate optimization method).
8. To specify multiple optimization methods by adding their values, Hitachi recommends to separate each optimization method specification with a comma to avoid ambiguities regarding which optimization method is being specified. If a new optimization method needs to be specified after multiple optimization methods have been specified by adding their values, you can specify the new value by appending it, separated with a comma.

Relationship to system definitions

1. The system-defined `pd_optimize_level` operand, if specified for `ALTER TRIGGER`, has no effect.
2. If the system-defined `pd_floatable_bes` or `pd_non_floatable_bes` operand is specified, any specification of *Increasing the target floatable servers (back-end servers for fetching data)* or *Limiting the target floatable servers (back-end servers for fetching data)* has no effect.
3. If `KEY` is specified for the system-defined `pd_indexlock_mode`

operand, (for index key value-locking), any specification of *Suppressing creation of update-SQL work tables* has no effect.

Relationship to client definitions

PDSQLOPTLVL, if specified, has no effect on ALTER TRIGGER.

Relationship to SQL

If SQL optimization is specified in an SQL statement, the SQL optimization specification takes precedence over SQL optimization options. For SQL optimization specifications, see *2.24 SQL optimization specification*.

SQL optimization option specification values

The following table lists the SQL optimization option specification values. For details about optimization methods, see the *HiRDB Version 9 UAP Development Guide*.

Table 3-8: SQL optimization option specification values (ALTER TRIGGER)

No.	Optimization method	Specification value	
		Identifier	Unsigned integer
1	Forced nest-loop-join	"FORCE_NEST_JOIN"	4
2	Making multiple SQL objects	"SELECT_APSL"	10
3	Increasing the target floatable servers (back-end servers for fetching data)#1, #2	"FLTS_INC_DATA_BES"	16
4	Prioritized nest-loop-join	"PRIOR_NEST_JOIN"	32
5	Increasing the number of floatable server candidates#2	"FLTS_MAX_NUMBER"	64
6	Priority of OR multiple index use	"PRIOR_OR_INDEXES"	128
7	Group processing, ORDER BY processing, and DISTINCT set function processing at the local back-end server#2	"SORT_DATA_BES"	256
8	Suppressing the use of AND multiple indexes	"DETER_AND_INDEXES"	512
9	Rapid grouping processing	"RAPID_GROUPING"	1024
10	Limiting the target floatable servers (back-end servers for fetching data)#1, #2	"FLTS_ONLY_DATA_BES"	2048
11	Separating data collecting servers#1, #2	"FLTS_SEPARATE_COLLECT_SVR"	2064

No.	Optimization method	Specification value	
		Identifier	Unsigned integer
12	Suppressing index use (forced table scan)	"FORCE_TABLE_SCAN"	4096
13	Forcing use of multiple indexes	"FORCE_PLURAL_INDEXES"	32768
14	Suppressing creation of update-SQL work tables	"DETER_WORK_TABLE_FOR_UPDATE"	131072
15	Derivation of rapid search conditions	"DERIVATIVE_COND"	262144
16	Applying key conditions including scalar operations	"APPLY_ENHANCED_KEY_COND"	524288
17	Facility for batch acquisition from functions provided by plug-ins	"PICKUP_MULTIPLE_ROWS_PLUGIN"	1048576
18	Facility for moving search conditions into derived table	"MOVE_UP_DERIVED_COND"	2097152

#1: If both *Increasing the target floatable servers (back-end servers for fetching data)* and *Limiting the target floatable servers (back-end servers for fetching data)* are specified together, the respective optimization method does not take effect; instead, the specification operates as a separating data collecting server.

#2: When specified on a HiRDB/Single Server, this option has no effect.

[ADD OPTIMIZE LEVEL *SQL-extension-optimizing-option*[, *SQL-extension-optimizing-option*]...]

Specifies an optimization method for determining the most efficient access path by taking the condition of the database into consideration.

An SQL optimization option can be specified using either an identifier (character string) or a numeric value. For most cases, Hitachi recommends the use of an identifier.

The default for this operand is the value that was specified during the previous SQL object creation (CREATE TRIGGER, ALTER TRIGGER, or ALTER ROUTINE).

Specifying with an identifier

```
ADD OPTIMIZE LEVEL "identifier"[, "identifier"]...
```

Specification examples

- Applying *Optimizing mode 2 based on cost* and *Hash join, subquery hash execution*:

```
ADD OPTIMIZE LEVEL "COST_BASE_2","APPLY_HASH_JOIN"
```

- Applying no optimization:

```
ADD OPTIMIZE LEVEL "NONE"
```

Rules

1. Specify one or more identifiers.
2. When specifying two or more identifiers, delimit them with commas (,).
3. For details about what can be specified in *identifier* (optimization methods), see *Table 3-9 SQL extension optimizing option specification values* (ALTER TRIGGER).
4. If no optimization is to be applied, specify `NONE` in *identifier*.
5. Identifiers can be specified in both lower case and upper case characters.
6. If the same identifier is specified more than once, it is treated as if it was specified only once; however, where possible, precautions should be taken to avoid specifying a given identifier in duplicate.

Specifying a numeric value

```
ADD OPTIMIZE LEVEL unsigned-integer[, unsigned-integer]...
```

Specification examples

- Applying *Optimizing mode 2 based on cost* and *Hash join, subquery hash execution*:

```
ADD OPTIMIZE LEVEL 1,2
```

- Applying no optimization:

```
ADD OPTIMIZE LEVEL 0
```

Rules

1. Specify one or more unsigned integers.
2. When specifying two or more unsigned integers, separate them with commas (,).
3. For details about what can be specified in an unsigned integer (optimization method), see *Table 3-9 SQL extension optimizing option specification values* (ALTER TRIGGER).
4. When not applying any optimization, specify `0` in *unsigned-integer*.

5. If the same unsigned integer is specified more than once, it is treated as if it was specified only once; however, where possible, precautions should be taken to avoid specifying a given unsigned integer in duplicate.

Relationship to system definitions

The system-defined `pd_optimize_level` operand, if specified for `ALTER TRIGGER`, has no effect.

Relationship to client environment definition

`PDADDITIONALOPTLVL`, when specified, has no effect for `ALTER TRIGGER`.

Relationship with SQL

If SQL optimization is specified in an SQL statement, the SQL optimization specification takes precedence over SQL optimization options. For SQL optimization specifications, see *2.24 SQL optimization specification*.

SQL extension optimizing option specification values

The following table lists the SQL extension optimizing option specification values. For details about optimization methods, see the *HiRDB Version 9 UAP Development Guide*.

Table 3-9: SQL extension optimizing option specification values (ALTER TRIGGER)

No.	Optimization method	Specification value	
		Identifier	Unsigned integer
1	Application of optimizing mode 2 based on cost	`"COST_BASE_2"`	`1`
2	Hash join, subquery hash execution	`"APPLY_HASH_JOIN"`	`2`
3	Facility for applying join conditions including value expression	`"APPLY_JOIN_COND_FOR_VALUE_EXP"`	`32`

Note

Items 2 and 3 take effect when *Application of optimizing mode 2 based on cost* is specified.

`[SUBSTR LENGTH` *maximum-character-length*`]`

Specifies the maximum number of bytes for representing a single character.

The value specified for the maximum character length must be in the range from 3 to 6.

This operand is valid only when `utf-8` is specified for the character code type in the `pdntenv` command (`pdsetup` command for the UNIX edition); it affects the length of the result of the `SUBSTR` scalar function. For details about `SUBSTR`, see *2.16.1(20) SUBSTR*.

Rules

When HiRDB is upgraded from a version earlier than version 08-00 to version 08-00 or later, 3 is assumed. If there is no need to change the maximum character length, you do not need to specify this operand when upgrading to HiRDB of version 08-00 or later.

Relationships to system definition

When `SUBSTR LENGTH` is specified in `ALTER TRIGGER`, the `pd_substr_length` system definition operand has no effect. For details about the `pd_substr_length` operand, see the manual *HiRDB Version 9 System Definition*.

Relationship to client environmental definition

The specification of `PDSUBSTREN` has no applicability to `ALTER TRIGGER`. For details about `PDSUBSTRLEN`, see the manual *HiRDB Version 9 UAP Development Guide*.

Relationship to the character code type specified in the `pdntenv` or `pdsetup` command

This operand is valid only when `utf-8` is specified for the character code type.

For all other character code types, only a syntax check is performed and the specification is ignored.

When this operand is omitted, the value specified during creation of the most recent SQL object (execution of a `CREATE TRIGGER`, `ALTER TRIGGER`, or `ALTER ROUTINE` statement) is assumed.

Common rules

1. Upon normal termination of the execution of `ALTER TRIGGER` for a trigger for which an SQL object is not in effect, the SQL object for that trigger takes effect.
2. Upon normal termination of the execution of `ALTER TRIGGER` for a trigger for which the SQL object is in an index-disabled state, the trigger's SQL object is released from the index-disabled state. Triggers for which the SQL object is in an index-disabled state may result in a runtime error.
3. When specifying an SQL compile option in `ALTER TRIGGER`, make sure that the

SQL statement that is generated as a result of the SQL compile option with respect to CREATE TRIGGER for the source trigger to be re-created does not exceed the maximum allowable length for an SQL statement.

4. Under the following condition, ALTER TRIGGER cannot be executed from a Java procedure:
   - The SQL object being executed is re-created.

Notes

1. ALTER TRIGGER cannot be specified from an X-Open compliant UAP running under OLTP.
2. Executing a GET DIAGNOSTICS statement immediately after the execution of ALTER TRIGGER enables you to acquire diagnostic information on ALTER TRIGGER. In this case, the trigger for which the re-creation process terminated normally produces an SQL code of 0.
3. The data guarantee level of the trigger SQL statement in the trigger, the SQL optimization option, the SQL extension optimizing option, and the maximum character length are determined by what is specified when the trigger is being defined or modified, and are not affected by the system definition or client environment variable definition that is in effect when the trigger action is executed.

Example

1. Re-create an SQL object for a trigger (TRIG1) that has been nullified:

```
ALTER TRIGGER TRIG1
    CHANGE ROUTINE OBJECT
```

COMMENT (Comment)

Function

COMMENT inserts a comment into a table or a column or changes an existing comment.

Privileges

Owner of the base table

A user can write comments into base tables owned by that user.

Format

```
COMMENT ON {TABLE [authorization-identifier.]table-identifier
            |COLUMN [authorization-identifier.]table-identifier.column-
             name}
           IS 'character-string'
```

Operands

- COMMENT ON

Specifies that a comment is to be processed for a base table or column owned by the user.

- {TABLE [*authorization-identifier*.]*table-identifier*

  |COLUMN [*authorization-identifier*.]*table-identifier.column- name*}

Specifies TABLE to write a comment into a table; specifies COLUMN to write a comment into a column.

When specifying a public view in *table-identifier*, in *authorization-identifier* specify the word PUBLIC, all in uppercase, enclosed in double quotation marks (").

- *character-string*

Specifies a comment in a character string. The length of a character string that can be specified is 0 to 255 bytes.

When specifying a national character string literal as a character string, you need not specify 'N'.

The following character string cannot be specified:

- Hexadecimal character string literal

Notes

1. An assigned comment can be referenced by retrieving the SQL_TABLES table or the SQL_COLUMNS table of the data dictionary table.

2. When specified for an existing comment, the COMMENT function deletes the existing comment and sets the new comment.
3. A comment cannot be inserted into a data dictionary table.
4. The COMMENT statement cannot be specified from an X/Open-compliant UAP running under OLTP.

Examples

1. Insert a comment into a stock table (STOCK):
```
COMMENT ON TABLE STOCK IS 'CREATED JULY 1995'
```

2. Insert a comment into the unit price column (PRICE) of a stock table (STOCK):
```
COMMENT ON COLUMN STOCK.PRICE IS 'REVISED JULY 1995'
```

CREATE AUDIT (Define the target audit event)

Function

CREATE AUDIT defines the target audit event to be recorded as an audit trail, and its target.

Privileges

Audit-privilege users

These users can execute CREATE AUDIT definition statements.

Format

No.	Format
1	CREATE AUDIT [AUDITTYPE {PRIVILEGE | EVENT | ANY}]
2	FOR *operation-type*
9	[*selection-option*]
3	[WHENEVER {SUCCESSFUL|UNSUCCESSFUL|ANY}]

Details about items

No.	Format
2	○*operation-type*::= {ANY | SESSION [{*session-type* | ANY}] | PRIVILEGE [{*privilege-operation-type* | ANY}] | DEFINITION [{*object-definition-event-type* | ANY}] | ACCESS [{*object-operation-event-type* | ANY}] | UTILITY [{*utility-event-type* | ANY}]}
9	*selection-option*::=ON *object-name*

No.	Format
	object-name::= {FUNCTION *authorization-identifier*.*routine-identifier* |INDEX *authorization-identifier*.*index-identifier* |LIST *authorization-identifier*.*table-identifier* |PROCEDURE *authorization-identifier*.*routine-identifier* |RDAREA RDAREA-name |SCHEMA *authorization-identifier* |TABLE [*authorization-identifier*.]*table-identifier* |TRIGGER *authorization-identifier*.*trigger-identifier* |TYPE *authorization-identifier*.*data-type-identifier* |VIEW *authorization-identifier*.*table-identifier*} |SEQUENCE *authorization-identifier*.*sequence-generator-identifier*}
4	*session-type*::= {CONNECT | DISCONNECT | AUTHORIZATION}
5	*privilege-operation-type*::= {GRANT | REVOKE}
6	*object-definition-event-type*::= {CREATE |DROP | ALTER}
7	*object-operation-event-type*::= {SELECT | INSERT | UPDATE | DELETE | PURGE | ASSIGN | CALL | LOCK | NEXT VALUE}
8	*utility-event-type*::= {PDLOAD | PDRORG | PDEXP | PDCONSTCK}

Operands

1) [AUDITTYPE {PRIVILEGE | EVENT | ANY}]

Specifies whether an audit trail during a privilege check is to be acquired or an audit trail on the final results of an event is to be acquired.

PRIVILEGE

Acquires an audit trail during a privilege check.

EVENT

Acquires an audit trail on the final results of an event.

ANY

Acquires an audit trail on any of the above types.

The PRIVILEGE, EVENT, and ANY operands can be defined and deleted individually. For example, if only ANY is deleted using DROP AUDIT on a given audit event for which PRIVILEGE, EVENT, and ANY are all defined, both the

PRIVILEGE and EVENT definitions remain intact (remain subject to auditing).

2) *operation-type* : : = ANY

| SESSION [{*session-type* | ANY}]

| PRIVILEGE [{*privilege-operation-type* | ANY}]

| DEFINITION [{*object-definition-event-type* | ANY}]

| ACCESS [{*object-operation-event-type* | ANY}]

| UTILITY [{*utility-event-type* | ANY}]

Specifies the type of operation to be audited. The individual operation types and ANY are individually defined and deleted. For example, if only ANY is deleted using DROP AUDIT for a given condition for which SESSION, PRIVILEGE, and ANY are all defined, both the SESSION and PRIVILEGE definitions remain intact (remain subject to auditing).

ANY

Specifies all operation types as being subject to auditing.

SESSION [{*session-type* | ANY}]

Specify this operand when session security events are to be made subject to auditing.

ANY makes any session security event subject to auditing. The individual session types and ANY are individually defined and deleted. For example, if only ANY is deleted using DROP AUDIT on a given condition for which CONNECT, AUTHORIZATION, and ANY are defined, both the CONNECT and AUTHORIZATION definitions remain intact (remain subject to auditing).

PRIVILEGE [{*privilege-operation-type* | ANY}]

Specify this operand when privilege management events are to be made subject to auditing. ANY makes any privilege management event subject to auditing. The individual privilege operation types and ANY are individually defined and deleted. For example, if only ANY is deleted using DROP AUDIT for a given condition for which GRANT, REVOKE, and ANY are defined, both the GRANT and REVOKE definitions remain intact (remain subject to auditing).

DEFINITION [{*object-definition-event-type* | ANY}]

Specify this operand when definition SQL events are to be made subject to auditing. ANY makes any definition SQL event subject to auditing. The individual object definition event types and ANY are individually defined and deleted. For example, if only ANY is deleted using DROP AUDIT for a given condition for which CREATE, DROP, and ANY are defined, both the CREATE

and DROP definitions remain intact (remain subject to auditing).

ACCESS [{*object-operation-event-type* | ANY}]

Specify this operand when manipulation SQL events are to be made subject to auditing. ANY makes any manipulation SQL event subject to auditing. The individual object operation types and ANY are individually defined and deleted. For example, if only ANY is deleted using DROP AUDIT for a given condition for which SELECT, INSERT, and ANY are defined, both the SELECT and INSERT definitions remain intact (remain subject to auditing).

UTILITY [{*utility-event-type* | ANY}]

Specify this operand when utility events are to be made subject to auditing. ANY makes any utility event subject to auditing. The individual utility event types and ANY are individually defined and deleted. For example, if only ANY is deleted using DROP AUDIT for a given condition for which PDLOAD, PDRORG, and ANY are defined, both the PDLOAD and PDRORG definitions remain intact (remain subject to auditing).

3) [WHENEVER {SUCCESSFUL | UNSUCCESSFUL | ANY}]

Specifies whether a given audit event is to be audited, depending on whether the file result of the audit event or a privilege check is successful.

The following table describes the audit trail that is acquired based on the specification in WHENEVER.

Table 3-10: Audit trail that is acquired based on the specification in WHENEVER

WHENEVER specification	When PRIVILEGE or ANY is specified in AUDITTYPE	When EVENT or ANY is specified in AUDITTYPE
SUCCESSFUL	Audit trail during a privilege check is collected only when the privilege check is successful.	Audit trail of the final result of an audit event is collected only when the audit event is successful.
UNSUCCESSFUL	Audit trail during a privilege check is collected only when the privilege check is unsuccessful.	Audit trail of the final result of an audit event is collected only when the audit event is unsuccessful.
ANY	Audit trail during a privilege check is collected regardless of whether or not the privilege check is successful.	Audit trail of the final result of an audit event is collected regardless of whether or not the audit event is successful.

Some final results of an event can be partially unsuccessful. For a partially unsuccessful event, an audit trail is output regardless of whether SUCCESSFUL, UNSUCCESSFUL, or ANY is specified.

SUCCESSFUL, UNSUCCESSFUL, and ANY are individually defined and deleted.

For example, if only ANY is deleted using DROP AUDIT for a given audit event for which SUCCESSFUL, UNSUCCESSFUL, and ANY are defined, both the SUCCESSFUL and UNSUCCESSFUL definitions remain intact (remain subject to auditing).

4) *session-type*: := {CONNECT | DISCONNECT | AUTHORIZATION}

Specifies the type of HiRDB session as either the object of the audit, user modification while connected, or the disconnection operation. The following table lists session types and the operations that are generated by the associated audit events.

Table 3-11: Session types and operations generated by the associated audit events

Session type	Operation generated by privilege check audit event (PRIVILEGE specified in AUDITTYPE)	Operation generated by audit event acquiring audit trail on final results of an event (EVENT specified in AUDITTYPE)
CONNECT	Connection to HiRDB	Same as indicated at left
DISCONNECT	None	Disconnection from HiRDB
AUTHORIZATION	Execution of SET SESSION AUTHORIZATION statement	Same as indicated at left

5) *privilege-operation-type*: := {GRANT | REVOKE}

Specify this operand when operations related to privileges are to be subject to auditing. The following table lists privilege operation types and the operations that are generated by the associated audit events.

Table 3-12: Privilege operation types and operations generated by the associated audit events

Privilege operation type	Operation generated by privilege check audit event (PRIVILEGE specified in AUDITTYPE)	Operation generated by audit event acquiring audit trail on final results of an event (EVENT specified in AUDITTYPE)
GRANT	Execution of GRANT	Same as indicated at left
REVOKE	Execution of REVOKE	Same as indicated at left

6) *object-definition-event-type*: :={CREATE | DROP | ALTER}

Specifies creation, deletion, or definition change operations on the object to be monitored. The following table lists object definition event types and the operations that are generated by the associated audit events.

Table 3-13: Object definition event types and operations generated by the associated audit events

Object definition event type	Operation generated by privilege check audit event (PRIVILEGE specified in AUDITTYPE)	Operation generated by audit event acquiring audit trail on final results of an event (EVENT specified in AUDITTYPE)
CREATE	Execution of the following SQL statements: • ALTER PROCEDURE[#1] • ALTER ROUTINE[#1] • ALTER TRIGGER[#1] • ASSIGN LIST statement • CREATE CONNECTION SECURITY • CREATE FUNCTION • CREATE INDEX • CREATE PROCEDURE • CREATE PUBLIC FUNCTION • CREATE PUBLIC PROCEDURE • CREATE SCHEMA • CREATE SEQUENCE • CREATE TABLE • CREATE TRIGGER • CREATE TYPE • CREATE VIEW • CREATE PUBLIC VIEW • CALL statement from a UAP[#2]	Same as indicated at left
DROP	Execution of the following SQL statements: • DROP CONNECTION SECURITY • DROP DATA TYPE • DROP FUNCTION • DROP INDEX • DROP LIST statement • DROP PROCEDURE • DROP PUBLIC FUNCTION • DROP PUBLIC PROCEDURE • DROP SCHEMA • DROP SEQUENCE • DROP TABLE • DROP TRIGGER • DROP VIEW • DROP PUBLIC VIEW • REVOKE[#3]	Same as indicated at left

Object definition event type	Operation generated by privilege check audit event (PRIVILEGE specified in AUDITTYPE)	Operation generated by audit event acquiring audit trail on final results of an event (EVENT specified in AUDITTYPE)
ALTER	Execution of the following SQL statements: • ALTER INDEX • ALTER PROCEDURE • ALTER ROUTINE • ALTER TABLE • ALTER TRIGGER • COMMENT	Same as indicated at left

#1: Internally, CREATE PROCEDURE is executed.

#2: If index information on the procedure being called is invalid, CREATE PROCEDURE is internally executed on each call. In this case, the execution of CREATE PROCEDURE on every call can be suppressed by recreating the SQL object for the procedure that is called by using either ALTER PROCEDURE or ALTER ROUTINE.

#3: If the SELECT privilege from a view base table is deleted, DROP VIEW is internally executed to delete the view table.

The following privilege checks are performed in an object definition event:

- Schema definition privilege check in an audit event
- Access privilege check on data manipulation SQL and control SQL statements in an SQL procedure statement either during the creation of a stored procedure or during the re-creation of an SQL object for a stored procedure
- Access privilege check on data manipulation SQL and control SQL statements in an SQL procedure statement either during the definition of a user-defined type in which a member includes a procedure, or during the re-creation of an SQL object for a user-defined type member
- Access privilege check on data manipulation SQL and control SQL statements in an SQL procedure statement either during the definition of a trigger or during the re-creation of an SQL object for a trigger
- Access privilege check on a base table during the definition of a view

7) *object-operation-event-type* ::=

{SELECT|INSERT|UPDATE|DELETE|PURGE|ASSIGN|CALL|LOCK | NEXT VALUE}

This operand specifies an operation on the object to be audited. If an audit target

definition for an object operation event is specified, object operations in a procedure and in a trigger SQL statement are also subject to auditing. The following table lists object operation event types and the operations generated by the associated audit events.

Table 3-14: Object operation event types and operations generated by the associated audit events

Object operation event type	Operation generated by privilege check audit event (PRIVILEGE specified in AUDITTYPE)	Operation generated by audit event acquiring audit trail on final results of an event (EVENT specified in AUDITTYPE)
SELECT	• Execution of single-row SELECT statement# • Execution of INSERT statement with query specification# • Execution of UPDATE statement with subquery specification in a search condition# • Execution of DELETE statement with subquery specification in a search condition# • Execution of a query on a list#	• Same as indicated at left
INSERT	• Execution of INSERT statement#	• Same as indicated at left
UPDATE	• Execution of UPDATE statement#	• Same as indicated at left
DELETE	• Execution of DELETE statement#	• Same as indicated at left
PURGE	• Execution of PURGE TABLE statement#	• Same as indicated at left
CALL	• No privilege check events	• Execution of a procedure by CALL statement#
LOCK	• Execution of LOCK statement#	• Same as indicated at left
ASSIGN	• Execution of ASSIGN LIST statement Format 1	• Execution of ASSIGN LIST statement Format 1 • Execution of ASSIGN LIST statement Format 2
NEXT VALUE	• Execution of NEXT VALUE expression	• Same as indicated at left

#: Includes dynamic SELECT statements.

Queries in the following SQL statements produce an audit trail if an object operation event type is defined in SELECT:

- Query specification specified in an INSERT statement
- Subquery specified in a search condition in an UPDATE or DELETE statement
- Queries on a list

8) *utility-event-type*::=

{PDLOAD | PDRORG | PDEXP | PDCONSTCK}

This operand defines a utility event as being subject to auditing. The following table lists utility event types and the operations generated by the associated audit events.

Table 3-15: Utility event types and operations generated by the associated audit events

Utility event type	Operation generated by privilege check audit event (PRIVILEGE specified in AUDITTYPE)	Operation generated by audit event acquiring audit trail on final results of an event (EVENT specified in AUDITTYPE)
PDLOAD	Execution of pdload	Same as indicated at left
PDRORG	Execution of pdrorg	Same as indicated at left
PDEXP	Execution of pdexp or pddefrev	Same as indicated at left
PDCONSTCK	Execution of pdconstck	Same as indicated at left

9) *selection-option*::=ON *object-name*

Specify this operand to select an object from which to collect an audit trail. In *object-name*, specify the object from which you want to collect the audit trail. For details about the naming rules for object names, see *1.1.7 Specification of names*.

If a public view, public procedure, or public function is specified in an object, specify PUBLIC for the authorization identifier.

To specify a dictionary table, specify TABLE for the object name, and specify only the table identifier without the authorization identifier. In this case, the object owner column of the dictionary table SQL_AUDITS stores '(Data dictionary)'.

Rules

1. For details about the security audit facility, see the *HiRDB Version 9 System Operation Guide*.
2. Actual recording of an audit trail requires either the setting of the pd_audit operand of the system definition or the execution of the pdaudbegin command.
3. When the security audit facility is enabled, audit trails from the execution of

CREATE AUDIT or DROP AUDIT are always recorded.

Notes

1. CREATE AUDIT cannot be specified from an X/Open-compliant UAP running under OLTP.
2. A trail may not always be output when an audit-target event is defined, depending on the combination of the operation type and other operands. When such a definition is specified, the KFPA19680-E message is output. The following table provides details about specifiable combinations.

Table 3-16: Event type, event subtype, and specifiability of AUDITTYPE

Event type	Event subtype	AUDITTYPE specifiability		
		PRIVILEGE	EVENT	ANY
ANY	--	C#1, #2	Y	C#1, #2
SESSION	DISCONNECT	N	Y	C#2
	ANY	C#2	Y	C#2
	Any subtype other than the above	Y	Y	Y
PRIVILEGE	Any subtype	Y	Y	Y
DEFINITION	Any subtype	Y	Y	Y
ACCESS	CALL	N	Y	C#1
	ANY	C#1	Y	C#1
	Any subtype other than the above	Y	Y	Y
UTILITY	Any subtype	Y	Y	Y

Legend:

--: Not applicable

Y: Specifiable

N: Not specifiable (KFPA19680-E message generated)

C: Specifiable with or without audit trail output

#1: Does not output an audit trail for event CALL privilege checks.

#2: Does not output an audit trail for event DISCONNECT privilege checks.

Table 3-17: Event type, event subtype, and specifiability of object name (1/2)

Event type	Event subtype	FCTN	INDEX	LIST	PRCDR	RD	SCHM
ANY	--	C	C	C	C	C	C
SESSION	All	N	N	N	N	N	N
PRIVILEGE	GRANT	N	N	N	N	N	N
	REVOKE	N	N	N	N	N	N
	ANY	N	N	N	N	N	N
DEFINITION	CREATE	Y	Y	N	Y	Y	Y
	DROP	Y	Y	N	Y	N	Y
	ALTER	Y	Y	N	Y	Y	N
	ANY	Y	Y	N	Y	C	C
ACCESS	SELECT	N	N	Y	N	N	N
	INSERT	N	N	N	N	N	N
	UPDATE	N	N	N	N	N	N
	DELETE	N	N	N	N	N	N
	PURGE	N	N	N	N	N	N
	ASSIGN	N	N	Y	N	N	N
	CALL	N	N	N	Y	N	N
	LOCK	N	N	N	N	N	N
	NEXT VALUE	N	N	N	N	N	N
	ANY	N	N	C	C	N	N
UTILITY	PDLOAD	N	N	N	N	N	N
	PDRORG	N	N	N	N	N	Y
	PDEXP	N	N	N	Y	N	N
	PDCONSTCK	N	N	N	N	N	N
	ANY	N	N	N	C	N	C

Legend:

FCTN: FUNCTION

PRCDR: PROCEDURE

RD: RDAREA

SCHM: SCHEMA

--: Not applicable

Y: Specifiable

N: Not specifiable (KFPA19680-E message generated)

C: Specifiable with or without audit trail output

Table 3-18: Event type, event subtype, and specifiability of object name (2/2)

Event type	Event subtype	SVR	TBL	TRGR	TYP	VIEW	SEQ
ANY	--	C	C	C	C	C	C
SESSION	All	N	N	N	N	N	N
PRIVILEGE	GRANT	N	Y	N	N	Y	N
	REVOKE	N	Y	N	N	Y	N
	ANY	N	Y	N	N	Y	N
DEFINITION	CREATE	Y	Y	Y	Y	Y	Y
	DROP	Y	Y	Y	Y	Y	Y
	ALTER	N	Y	Y	N	Y	N
	ANY	C	Y	Y	C	Y	C

Event type	Event subtype	SVR	TBL	TRGR	TYP	VIEW	SEQ
ACCESS	SELECT	N	Y	N	N	Y	N
	INSERT	N	Y	N	N	Y	N
	UPDATE	N	Y	N	N	Y	N
	DELETE	N	Y	N	N	Y	N
	PURGE	N	Y	N	N	N	N
	ASSIGN	N	Y	N	N	N	N
	CALL	N	N	N	N	N	N
	LOCK	N	Y	N	N	Y	N
	NEXT VALUE	N	N	N	N	N	Y
	ANY	N	C	N	N	C	C
UTILITY	PDLOAD	N	Y	N	N	N	Y
	PDRORG	N	Y	N	N	N	N
	PDEXP	N	Y	Y	N	Y	N
	PDCONSTCK	N	Y	N	N	N	N
	ANY	N	Y	C	N	C	C

Legend:

SVR: SERVER

TBL: TABLE

TRGR: TRIGGER

TYP: TYPE

SEQ: SEQUENCE

--: Not applicable

Y: Specifiable

N: Not specifiable (KFPA19680-E message generated)

C: Specifiable with or without audit trail output

1. When the HiRDB version is upgraded with ANY specified in the AUDITTYPE

clause, in FOR <*operation-type*> and each type, or in the WHENEVER clause, if the number of individual types increases as a result, all of these types are included as audit targets.

For example, if the version is upgraded with CREATE AUDIT FOR ANY defined and the number of operation types increases as a result, the increased operation types also become audit targets.

2. You cannot execute the definition of an already defined audit event in a CREATE AUDIT statement. If you attempt to do so, the KFPA11908-E message is output.

Examples

1. Define privilege checks on all audit events as being subject to auditing.

```
CREATE AUDIT FOR ANY WHENEVER ANY
```

2. Define privilege checks on connection to HiRDB as being subject to auditing.

```
CREATE AUDIT FOR SESSION CONNECT
```

3. Define privilege checks on the execution of the GRANT statement as being subject to auditing.

```
CREATE AUDIT FOR PRIVILEGE GRANT
```

4. Define privilege checks on the creation of an object as being subject to auditing.

```
CREATE AUDIT FOR DEFINITION CREATE
```

5. Define privilege checks on INSERT as being subject to auditing.

```
CREATE AUDIT FOR ACCSESS INSERT
```

6. Define all audit events as being subject to auditing.

```
CREATE AUDIT AUDITTYPE ANY FOR ANY
```

7. Define the termination of any audit event as being subject to auditing.

```
CREATE AUDIT AUDITTYPE EVENT FOR ANY
```

8. Specify the object from which to acquire an audit trail as table `USER1.T1`.

```
CREATE AUDIT AUDITTYPE EVENT FOR ANY ON TABLE "USER1"."T1"
```

CREATE CONNECTION SECURITY (Define the connection security facility)

Function

CREATE CONNECTION SECURITY defines security items related to the connection security facility.

Privilege

Users who have DBA privilege

Users who have the DBA privilege can execute definition statements related to CREATE CONNECTION SECURITY.

Format

```
CREATE CONNECTION SECURITY FOR security-object[, security-object]

security-object::= {CONNECT [PERMISSION COUNT literal
   [LOCK {literal DAY[S]|literal HOUR[S]
   | literal MINUTE[S]|UNLIMITED}]]
  | PASSWORD [TEST] password-character-limit-definition}

password-character-limit-definition::= [MIN LENGTH literal]
   [USER IDENTIFIER {RESTRICT|UNRESTRICT}]
   [SIMILAR {RESTRICT|UNRESTRICT}]
```

Operands

- *security-object*::=

  {CONNECT [PERMISSION COUNT *literal*

  [LOCK {*literal* DAY [S] | *literal* HOUR [S]

  | *literal* MINUTE [S] | UNLIMITED}]]

  | PASSWORD [TEST] *password-character-limit-definition*}

  In *security-object*, CONNECT and PASSWORD can each be specified only once.

  If *security-object* is omitted, the omitted security object is not defined. For each *security-object* specification, you can omit either CONNECT or PASSWORD, but not both.

  If only CONNECT or PASSWORD is specified in *security-object* and all operands after CONNECT or PASSWORD are omitted, the default values for the omitted operands are assigned.

CONNECT [PERMISSION COUNT *literal*

[LOCK {*literal* DAY [S] | *literal* HOUR [S] | *literal* MINUTE [S]

| UNLIMITED}]]

Specifies the default for a consecutive certification failure limit.

PERMISSION COUNT *literal*

Specifies a permission count for the permitted number of consecutive certification failures until a consecutive certification failure account lock state occurs. A consecutive certification failure account lock state occurs when the permitted number of consecutive certification failures exceeds a specified value.

The default for PERMISSION COUNT is 2. If a PERMISSION COUNT specification is omitted, the LOCK option cannot be specified.

literal

Specifies a permission count for the permitted number of consecutive certification failures until a consecutive certification failure account lock state occurs.

The minimum is 1 (time); the maximum is 10 (times).

In *literal*, specify an unsigned integer.

LOCK {*literal* DAY [S] | *literal* HOUR [S] | *literal* MINUTE [S] | UNLIMITED}

Specifies the duration over which the consecutive certification failure account lock state is to be continued.

The default for LOCK is LOCK 1440 MINUTE(LOCK 1 DAY,LOCK 24 HOUR).

In *literal*, specify an unsigned integer.

literal DAY[S]

Specifies the period over which the consecutive certification failure account lock state is to be continued by day.

The minimum is 1 (day); the maximum is 31 (days).

literal HOUR[S]

Specifies the period over which the consecutive certification failure account lock state is to be continued by the hour.

The minimum is 1 (hour); the maximum is 744 (hours).

literal MINUTE[S]

Specifies the period over which the consecutive certification failure account

lock state is to be continued by the minute.

The minimum is 10 (minutes); the maximum is 44640 (minutes).

UNLIMITED

Specifies that the consecutive certification failure account lock state is to be continued indefinitely.

- PASSWORD [TEST] *password-character-limit-definition*

Specifies the default for strengthening the password character limit.

TEST

Before defining a password character limit, specify this operand to check in advance an authorization identifier that has a password unsuitable for the limit to be changed.

The TEST option determines whether the current password is compatible with the character limit specified in *password-character-limit-definition*.

If TEST is specified, the settings in *password-character-limit-definition* are not defined.

- *password-character-limit-definition* : : =

[MIN LENGTH *literal*]

[USER IDENTIFIER {RESTRICT | UNRESTRICT}]

[SIMILAR {RESTRICT | UNRESTRICT}]

MIN LENGTH *literal*

Specifies the minimum required length for passwords in bytes.

Passwords less than a specified literal in bytes are prohibited.

The default for MIN LENGTH is 8.

The minimum is 6; the maximum is 15.

In *literal*, specify an unsigned integer.

USER IDENTIFIER {RESTRICT | UNRESTRICT}

Specifies whether passwords containing an authorization identifier are to be prohibited. The default for USER IDENTIFIER is RESTRICT.

RESTRICT

Specify this operand if passwords containing an authorization identifier are to be prohibited.

UNRESTRICT

Specify this operand if passwords containing an authorization identifier are to be allowed.

SIMILAR {RESTRICT | UNRESTRICT}

Specifies whether all characters composing a password must be restricted to a single character type.

The default for SIMILAR is RESTRICT.

Specifies whether the type of password given in the following example is to be prohibited:

Examples:

FDBGLAOT (uppercase alphabetic characters only)

24681357 (numerics only)

RESTRICT

Specify this operand when prohibiting the restriction of all characters composing a password to a single character type.

UNRESTRICT

Specify this operand when not prohibiting the restriction of all characters composing a password to a single character type.

Common rules

1. If a specified security object is already defined, the same security object cannot be defined in duplicate.
2. When modifying the definition of an item related to the connection security facility, delete the definition of the item related to the connection security facility, and then redefine the item related to the connection security facility.
3. If any DBA privilege holder or auditor is in violation of a specified password character limit definition, the CREATE CONNECTION SECURITY FOR PASSWORD flags an error. The TEST option does not generate an error.

Notes

1. The default for the single-character type limit is RESTRICT, which puts the limit in effect. If no restriction is intended, the UNRESTRICT option should be specified.
2. Even when a password character limit is defined, you can check passwords in advance by specifying the TEST operand.

Examples

1. Define security parameters for the connection security facility by specifying the

following settings:

Definition for consecutive certification failure limit

- Permitted number of consecutive certification failures: 5
- Lock duration: 7 days

Password character limit definition

- Minimum password length in bytes: 10 characters
- Prohibit passwords containing an authorization identifier
- Prohibit single-character type passwords

```
CREATE CONNECTION SECURITY FOR
  CONNECT PERMISSION COUNT 5
          LOCK 7 DAY,
  PASSWORD MIN LENGTH 10
           USER IDENTIFIER RESTRICT
           SIMILAR RESTRICT
```

2. Define security parameters for the connection security facility by specifying the following settings:

   Definition for consecutive certification failure limit

   - Permitted number of consecutive certification failures: 5
   - Lock duration: 15 hours

   Password character limit definition

   - Assign a default value

```
CREATE CONNECTION SECURITY FOR
  CONNECT PERMISSION COUNT 5
          LOCK 15 HOUR,
  PASSWORD
```

3. Define security parameters for the connection security facility by specifying the following settings:

   Definition for consecutive certification failure limit

   - Not defined

   Password character limit definition

   - Assign a default value

```
CREATE CONNECTION SECURITY FOR PASSWORD
```

CREATE [PUBLIC] FUNCTION (Define function, define public function)

Function

CREATE FUNCTION defines a function.

CREATE FUNCTION can also be used to define *public* functions, which are available to all users without them having to qualify the routine identifier with an authorization identifier.

(1) CREATE FUNCTION (Define function)

Privileges

Owner of a schema

A user can define functions that will be owned by that user.

Format

```
CREATE function-body
function-body::=FUNCTION [authorization-identifier.]routine-identifier
                        ([SQL-parameter-name data-type
                            [, SQL-parameter-name data-type] ...])
                      RETURNS data-type
                        [LANGUAGE {SQL|JAVA|C}]
[SQL-compilation-option]
                          {SQL-procedure-statement
                           |external-routine-specification}
SQL-compilation-option::=SUBSTR LENGTH maximum-character-length
external-routine-specification ::= EXTERNAL NAME
{external-Java-routine-name|external-C-stored-routine-name}
                                  PARAMETER STYLE parameter-style
parameter-style ::= {JAVA|RDSQL}
```

Operands

- [*authorization-identifier*.]*routine-identifier*

authorization-identifier

Specifies the authorization identifier of the owner of the function that is being defined.

routine-identifier

Specifies a routine identifier for the function being defined. The same routine identifier can be used in all the owner's routines.

- ([*SQL-parameter-name data-type* [, *SQL-parameter-name data-type*] ...])

SQL-parameter-name

Specifies the name of a parameter for the function. The same SQL parameter name cannot be specified more than once for the same function.

data-type

Specifies the data type of the paired parameter for the function. The BOOLEAN data type cannot be specified.

If the specified data type is an abstract data type, no authorization identifier is specified, and the default authorization identifier does not have an abstract data type of the same name, and if there is an abstract data type of the same name in the 'MASTER' authorization identifier, that abstract data type is assumed to have been specified.

The following data types cannot be specified:

- Abstract data type if JAVA or C is specified in the LANGUAGE clause
- BINARY or BLOB type if C is specified in the LANGUAGE clause

For details about the data types that can be specified, see *1.10.1(2) Type mapping* or *Table 3-21 Correspondence between data type of SQL parameter and data type of parameter passed to C function*.

- RETURNS *data-type*

data-type

Specifies the data type for return values of the function.

The following data types cannot be specified:

- BOOLEAN if a data type other than C is specified in the LANGUAGE clause
- Abstract data type if JAVA or C is specified in the LANGUAGE clause
- BINARY or BLOB type if C is specified in the LANGUAGE clause

For details about the data types that can be specified, see *1.10.1(2) Type mapping* or *Table 3-21 Correspondence between data type of SQL parameter and data type of parameter passed to C function*.

If the data type being specified is the abstract data type and the authorization identifier is omitted, and the default authorization identifier does not have an abstract data type of the same name, the specified abstract data type is assumed, provided that the authorization identifier MASTER has an identically named abstract data type.

- LANGUAGE {SQL |JAVA |C}

Specifies the language used to write the function.

For an external routine, specify JAVA or C.

SQL

Specifies that the processing part of the function is made up of SQL statements.

JAVA

Specifies that the processing part of the function is specified as an external routine and the function is to be implemented as a Java class method.

C

Specifies that the processing part of the function is specified as an external routine specification and the function is implemented in the C language. If C is specified, the maximum number of SQL parameters is limited to 128.

Whether other operands needs to be specified depends on how this operand was specified. The following table indicates whether other operands need to be specified depending on the LANGUAGE clause specification.

Table 3-19: Whether other operands needs to be specified depending on the LANGUAGE clause specification of CREATE FUNCTION

Other operand	LANGUAGE clause specification		
	SQL	JAVA	C
EXTERNAL NAME	N	Y	Y
PARAMETER STYLE	N	JAVA	RDSQL
SQL procedure statement	Y	N	N

Legend:

Y: Specify.

N: Do not specify.

JAVA: Specify JAVA.

RDSQL: Specify RDSQL.

- *SQL-compilation-option*::=SUBSTR LENGTH *maximum-character-length*

[SUBSTR LENGTH *maximum-character-length*]

Specifies the maximum number of bytes for representing a single character.

The value specified for the maximum character length must be in the range from 3 to 6.

This operand is valid only when utf-8 is specified for the character code type in the pdntenv command (pdsetup command for the UNIX edition); it affects the length of the result of the SUBSTR scalar function. For details about SUBSTR, see *2.16.1(20) SUBSTR*.

Relationships to system definition

When SUBSTR LENGTH is omitted, the value specified in the pd_substr_length operand in the system definition is assumed. For details about the pd_substr_length operand, see the manual *HiRDB Version 9 System Definition*.

Relationship to client environmental definition

The specification of PDSUBSTREN has no applicability to CREATE FUNCTION. For details about PDSUBSTRLEN, see the manual *HiRDB Version 9 UAP Development Guide*.

Relationship to the character code type specified in the pdntenv or pdsetup command

This operand is valid only when utf-8 is specified for the character code type.

For all other character code types, only a syntax check is performed and the specification is ignored.

- *SQL-procedure-statements*

Specifies the SQL procedure statements to be executed by the SQL function (for details of SQL procedure statements, see the *General rules* section in *7. Routine Control SQL*). Only compound statements can be specified in an SQL procedure statement. The last SQL procedure statement executed in an SQL function must be the RETURN statement.

- EXTERNAL NAME {*external-Java-routine-name*|*external-C-stored-routine-name*}

external-Java-routine-name

Specifies the Java method implemented using the Java language as an external routine. For details about how to specify an external Java routine name, see *1.10.1(1) External Java routine names*.

external-C-stored-routine-name

Specifies the Java method implemented using the C language as an external routine. For details about how to specify an external Java routine name, see *1.10.2(1) External C stored routine names*.

When specifying C in the LANGUAGE clause, see *Table 3-21 Correspondence between data type of SQL parameter and data type of parameter passed to C function*.

■ PARAMETER STYLE *parameter-style*

Specifies items to be passed as parameters when an external routine is called.

JAVA *parameter-style*

Parameters for an external Java function, defined in an SQL data type, are passed as parameters to the Java method in a Java data type corresponding to the SQL data type.

The return value of the Java method, defined in a Java data type, is returned as a return value of the external Java function in an SQL data type corresponding to the Java data type.

RDSQL *parameter-style*

If the number of SQL parameters is *n*, they are passed to the external C function as a parameter of the C function, as described in the following table.

Table 3-20: Contents passed as parameters of the C function

Parameter no. (n is number of SQL parameters)	Contents	Description of contents	Data type
1 to *n*	Data part of SQL parameter	Input parameter corresponding to data part of SQL parameter	Data type corresponding to data type of SQL parameter[#1]
n + 1	Return value data	Output parameter used by the C function that implemented the external C stored routine to set the return value.	Data type corresponding to SQL data type set in RETURNS[#1]
n + 2 to 2*n* + 1	SQL parameter indicator part	Input parameter corresponding to indicator part of SQL parameter. If the data is a null value, a negative value is set and passed to the C function.	short*
2*n* + 2	Return value indicator part	Output parameter used by the external C stored routine for setting the return value indicator part. 0 is set as the default value. Set the indicator part in the C function that implemented the external C stored routine, according to the following description. • If output value is null value Setting: -1 • If output value is non-null value Setting: 0	short*

Parameter no. (n is number of SQL parameters)	Contents	Description of contents	Data type
2*n* + 3	SQLSTATE	Output parameter used by the C function that implemented the external C stored routine to set the SQLSTATE value. The area length is 6 bytes. The SQLSTATE value is set in bytes 1 to 5. Set the SQLSTATE value in the C function that implemented the external C stored routine, according to the following description. • If the C function terminates normally Setting: '00000' SQL execution result: Terminate normally • If the C function terminates abnormally Setting: Format of value '38*XYY*' where *X* and *Y* are values in the following ranges: *X*: I to Z *Y*: 0 to 9 or A to Z Examples: '38I01', '38ZCD' SQL execution result: SQL error occurs. • If the C function ends in an undetermined state Setting: Value other than '00000', and not having the format '38*XYY*' SQL execution result: SQL error occurs.	char*
2*n* + 4	Routine name	Input parameter specifying the routine name	struct{ short *variable-name-1*; char *variable-name-2*[30]; }*[#2]
2*n* + 5	ID name	Input parameter specifying the ID name used to identify the function	struct{ short *variable-name-1*; char *variable-name-2*[30]; }*[#2]

Parameter no. (n is number of SQL parameters)	Contents	Description of contents	Data type

2*n* + 6	Message text	Output parameter for setting the detailed reason for an error if an error occurs in the C function that implemented an external C stored routine.
		If the value of class 38 is set to SQLSTATE when an external C stored routine finishes execution, an error message with embedded message text is output.
		The maximum length of the message text that can be set is 80 bytes.
			`struct{`
			`  short` *variable-name-1* `;`
			`  char` *variable-name-2* `[80];`
			`}*`#2

#1

For details about the correspondence between the data type of an SQL parameter that can be set when defining an external C function and the data type of a parameter passed to the C function that implements an external C stored routine, see *Table 3-21 Correspondence between data type of SQL parameter and data type of parameter passed to C function*.

#2

Sets the character string length (in bytes) to *variable-name-1* and the character string indicating the contents to *variable-name-2*.

Table 3-21: Correspondence between data type of SQL parameter and data type of parameter passed to C function

Data type of SQL parameter	Data type of parameter passed to C function	Length (bytes)	Comments
`INT[EGER]`	int*	4	--
`SMALLINT`	short*	2	--
`[LARGE] DEC[IMAL][(`*p*`[,`*s*`])]`, `[LARGE] NUMERIC[(`*p*`[,`*s*`])],`	char* #2	$\downarrow p \div 2 \downarrow + 1$	$1 \leq p \leq 38, 0 \leq s \leq p$
`FLOAT`, `DOUBLE PRECISION`	double*	8	--
`SMALLFLT`, `REAL`	float*	4	--

Data type of SQL parameter	Data type of parameter passed to C function	Length (bytes)	Comments
CHAR[ACTER][(*n*)]	char* #1	*n* + 1	1 ≤ *n* ≤ 30,000
CHAR[ACTER][(*n*)] CHARACTER SET [MASTER.]EBCDIK			
CHAR[ACTER][(*n*)] CHARACTER SET [MASTER.]UTF16	char* #1	*n* + 2	2 ≤ *n* ≤ 30,000
VARCHAR(*n*)	char* #1	*n* + 1	1 ≤ *n* ≤ 32,000
VARCHAR(*n*) CHARACTER SET [MASTER.]EBCDIK			
CHAR[ACTER] VARYING(*n*)	char* #1	*n* + 1	1 ≤ *n* ≤ 32,000
CHAR[ACTER] VARYING(*n*) CHARACTER SET [MASTER.]EBCDIK			
VARCHAR(*n*) CHARACTER SET [MASTER.]UTF16	char* #1	*n* + 2	2 ≤ *n* ≤ 32,000
CHAR[ACTER] VARYING(*n*) CHARACTER SET [MASTER.]UTF16	char* #1	*n* + 2	2 ≤ *n* ≤ 32,000
NCHAR[(*n*)], NATIONAL CHAR[ACTER][(*n*)]	char* #1	2*n* + 1	1 ≤ *n* ≤ 15,000
NVARCHAR(*n*), NATIONAL CHAR[ACTER] VARYING(*n*), NCHAR VARYING(*n*)	char* #1	2*n* + 1	1 ≤ *n* ≤ 16,000
MCHAR[(*n*)],	char* #1	*n* + 1	1 ≤ *n* ≤ 30,000
MVARCHAR(*n*)	char* #1	*n* + 1	1 ≤ *n* ≤ 32,000
DATE	char*#4	4	--

Data type of SQL parameter	Data type of parameter passed to C function	Length (bytes)	Comments
TIME	char*#4	3	--
INTERVAL YEAR TO DAY	char* #3	5	--
INTERVAL HOUR TO SECOND	char* #3	4	--
TIMESTAMP[(*p*)]	char* #4	*n*	Area size n is determined as follows, according to the value of p. • If *p* = 0, *n* = 7 • If *p* = 2, *n* = 8 • If *p* = 4, *n* = 9 • If *p* = 6, *n* = 10
BOOLEAN	int* #5	4	0: False 1: True
Indicator variable	short*	2	--

Legend:

--: No comment

Note

If the system does not code int type data into a 4-byte binary format, the int data type passed to the C function as a parameter will be converted to a data type that indicates 4-byte binary format data.

#1

The rules for conversion between the SQL data type whose UTF-16 character set specification does not specify (CHAR(*n*), VARCHAR(*n*), NCHAR(*n*), NVARCHAR(*n*), MCHAR(*n*), MVARCHAR(*n*)) and the C language data type (char[*n* + 1], char[*n* + 1], char[2*n* + 1], char[2*n* + 1], char[*n* + 1], char[*n* + 1]) are shown below.

Conversion from an SQL data type to a C language data type (input parameter and input/output parameter for C stored procedure, and input parameter for C stored function):

- Conversion from CHAR(*n*) to char[*n* + 1]
- Conversion from VARCHAR(*n*) to char[*n* + 1]
- Conversion from NCHAR(*n*) to char[2*n* + 1]
- Conversion from NVARCHAR(*n*) to char[2*n* + 1]

- Conversion from MCHAR(*n*) to char[*n* + 1]
- Conversion from MVARCHAR(*n*) to char[*n* + 1]

A null character is added to the end of the character string.

Conversion from a C language data type to an SQL data type (output parameter and input/output parameter for C stored procedure, and return value for C stored function):

- Conversion from char[*n* + 1] to CHAR(*n*)
- Conversion from char[*n* + 1] to VARCHAR(*n*)
- Conversion from char[2*n* + 1] to NCHAR(*n*)
- Conversion from char[2*n* + 1] to NVARCHAR(*n*)
- Conversion from char[*n* + 1] to MCHAR(*n*)
- Conversion from char[*n* + 1] to MVARCHAR(*n*)

The length of the character string HiRDB constructs from the C language character string that it receives is the length from the start to the character before the null character. If the SQL data type is CHAR(*n*), NCHAR(*n*), or MCHAR(*n*) and the length of the character string constructed by HiRDB is not the same as the defined length of the SQL data type, it is filled with spaces to reach the defined length. If no null character is detected within *n* + 1 sequence elements, an abnormal termination occurs.

The rules for conversion between the SQL data type whose UTF-16 character set specification does specified (CHAR(*n*), VARCHAR(*n*)) and the C language data type (char[*n* + 2], char[*n* + 2]) are shown below.

Conversion from an SQL data type to a C language data type (input parameter and input/output parameter for C stored procedure, and input parameter for C stored function):

- Conversion from CHAR(*n*) to char[*n* + 2]
- Conversion from VARCHAR(*n*) to char[*n* + 2]

A two-byte null character is added to the end of the character string.

Conversion from a C language data type to an SQL data type (output parameter and input/output parameter for C stored procedure, and return value for C stored function):

- Conversion from char[*n* + 2] to CHAR(*n*)
- Conversion from char[*n* + 2] to VARCHAR(*n*)

The length of the character string HiRDB constructs from the C language character string it receives is the length from the start to character before the

null character. If the SQL data type is CHAR(*n*) and the length of the character string constructed by HiRDB is not the same as the defined length of the SQL data type, it is filled with spaces to reach the defined length. If no two-byte null character is detected within *n* + 2 sequence elements, an abnormal termination occurs.

#2

DECIMAL type stores data in packed decimal format. For details about DECIMAL type, see *1.2 Data types*.

Examples of setting the DECIMAL type value in the C language are shown below.

- For 123.4567 (odd digit)

  unsigned char ex1[4]={0x12,0x34,0x56,0x7c};

- For -123.456 (even digit)

  unsigned char ex2[4]={0x01,0x23,0x45,0x6d};

- For 0 (odd digit)

  unsigned char ex3[1]={0x0c};

#3

INTERVAL YEAR TO DAY and INTERVAL HOUR TO SECOND types store data in packed decimal format. For details about these data types, see *1.2 Data types*.

#4

DATE, TIME, and TIMESTAMP types store data in unsigned packed decimal format. For details about these data types, see *1.2 Data types*.

#5

BOOLEAN type can only be used as a return value of a C stored function.

Common rules

1. An SQL parameter cannot be specified in the target of an assignment statement.
2. The number of function parameters must not exceed 30,000 (128 if C is specified in the LANGUAGE clause). If anything other than SQL is specified in the LANGUAGE clause, an error may result at the time of execution due to external language specification limitations even though the number of specified function parameters is less than 30,000.
3. SQL parameters become input SQL parameters.
4. Only predefined functions can be specified in the SQL procedure statements.
5. When a function is defined, the HiRDB system defines a special name that identifies that function uniquely. Following are the conventions for determining

special names:

special-name`::=F` *function-name object-ID*

- First byte is the constant `'F'`.
- Function name begins in byte 2 (if the name of the function is longer than 19 bytes, only the first 19 bytes are used).
- Object ID occupies 10 bytes following the function name (left-justified and padded with trailing zeros).

6. A function that is defined must not be the same as any system-provided function. Any function that satisfies either of the following sets of conditions cannot be defined:

   (a)

   - The function is associated with two SQL parameters.
   - The data type of the first SQL parameter is an abstract data type.
   - An attribute with the same name as the function name is defined as an attribute in the abstract data type specified by the first SQL parameter.
   - The data type of the second SQL parameter is identical to the data type of an attribute with the same name as the function name.

   (b)

   - The function is associated with one SQL parameter.
   - The data type of the SQL parameter is an abstract data type.
   - An attribute with the same name as the function name is defined as an attribute in the abstract data type specified by the SQL parameter.

7. If the `LANGUAGE` clause is omitted or SQL is specified in the `LANGUAGE` clause, an external routine cannot be specified.
8. If anything other than SQL is specified in the `LANGUAGE` clause, an SQL procedure statement cannot be specified.
9. If the name of a function call specified in the function body is qualified with *authorization-identifier*, you cannot specify a function that has the same authorization identifier, routine name, and number of arguments as the function that you are currently defining.

   If the name of a function call specified in the function body is not qualified with *authorization-identifier*, you cannot specify a function that has the same routine name and number of arguments as the function that you are currently defining.
10. A `CREATE FUNCTION` statement cannot be executed if a function that satisfies all the following conditions is already defined:

- Has the same authorization identifier as the function being defined.
- Has the same routine identifier (function name) as the function being defined.
- Has the same number of parameters as the function being defined.
- Has the same parameter data type as the function being defined.#

#

If the character set specified for a fixed-length character data type and a variable-length character data type are different, they will be treated as different data types.

11. When the SQL compile option is specified in `ALTER ROUTINE`, the length of the SQL statement that is created by incorporating the SQL compile option in the source `CREATE FUNCTION` for the procedure to be re-created must not exceed the maximum allowable length for SQL statements.
12. If the SQL object being executed is invalidated, `CREATE FUNCTION` cannot be executed from within an external Java procedure.

Notes

1. The `CREATE FUNCTION` statement cannot be specified from an X/Open-compliant UAP running under OLTP.
2. SQL parameters can have the NULL value.
3. To execute multiple SQL statements in an SQL procedure statement, use a routine control SQL statement, such as a compound statement.
4. Defining an SQL function results in creation of an SQL object that specifies access procedures for execution of functions.
5. The following shows the invalidation conditions for the SQL object associated with the function definition:
   - If an SQL object for which functions, procedures, or triggers are in effect exists when the function is defined, and if a function with the same number of owners, routine identifiers, and SQL parameters already exist, the SQL object becomes invalid.
   - If there is a function that specifies `MASTER` as its authorization identifer that has the same number of routine identifiers and SQL parameters as the function being defined, any valid SQL object that belongs to the authorization identifier of the function being defined becomes invalid if it is in a function, procedure, or trigger that uses that `MASTER` authorization identifier function. In addition, if a valid SQL object is in a public function or a public procedure that has the same authorization identifier as that of the

function being defined, the SQL object in that public function or public procedure becomes invalid.

- If there is a public function that has the same number of routine identifiers and SQL parameters as the function being defined, any valid SQL object that belongs to the authorization identifier of the function being defined becomes invalid if it is in a function, procedure, or trigger that uses that public function. In addition, if a valid SQL object is in a public function or a public procedure that has the same authorization identifier as that of the function being defined, the SQL object in that public function or public procedure becomes invalid.

For notes about defining or deleting a stored function, see *HiRDB Version 9 UAP Development Guide*.

6. Of the invalidated functions indicated in note 5, if a function satisfying either of the following conditions is used in the view definition, the function definition results in an error:
   - Abstract data type is used in the data type of an argument.
   - Abstract data type is used in the data type of a return value.
7. If an SQL object with an effective function, procedure, or trigger is nullified, any rows containing the function, procedure, and trigger that have been nullified in the `SQL_ROUTINE_RESOURCES` dictionary table are deleted.
8. Before executing the SQL object with a nullified function, procedure, and trigger, you need to execute `ALTER ROUTINE`, `ALTER PROCEDURE`, or `ALTER TRIGGER` to re-create the SQL object with the function, procedure, or trigger.
9. Before using a view table that references an invalid function associated with the function definition, `ALTER ROUTINE` must be executed and the SQL object of the function recreated.
10. Stacks in the operating system may overflow if routine calls are repeated extensively or infinitely in a routine.
11. `SUBSTR LENGTH` of the SQL compile option is determined by what is specified when the function is defined or modified; it is not affected by the system definition or client environment variable definition that is in effect when the function is called.

(2) CREATE PUBLIC FUNCTION (Define public function)

Privileges

Users who own a schema

These users can define their own public functions.

Format

```
CREATE PUBLIC function-body
```

Operands

- PUBLIC

Specifies the function being defined as a public function.

When a function is created as a public function, multiple users can reference the function simply by specifying a routine identifier without each user having to define a function with the same contents.

- *function-body*

For descriptions of operands other than [*authorization-identifier*.]*routine-identifier*, see *(1) CREATE FUNCTION (Define function)* under *CREATE [PUBLIC] FUNCTION (Define function, define public function)* in this chapter.

[*authorization-identifier*.]*routine-identifier*

authorization-identifier

The authorization identifier cannot be specified because this is a public function.

routine-identifier

Specifies the routine identifier of the public function being defined. This same routine identifier can be used in the public function.

Common rules

1. You cannot specify the authorization identifier of [*authorization-identifier*.]*routine-identifier*] in the function body.
2. If the name of a function call specified in the function body is qualified with PUBLIC, you cannot specify a public function that has the same routine identifier and number of arguments as the public function that you are currently defining.

   If the name of a function call specified in the function body is not qualified with *authorization-identifier*, you cannot specify a function that has the same routine identifier and number of arguments as the public function that you are currently defining.
3. A CREATE PUBLIC FUNCTION statement cannot be executed if a function that satisfies all of the following conditions is already defined:
   - Has the same routine identifier (function name) as the public function being defined.

- Has the same number of parameters as the public function being defined.
- Has the same parameter data types as the public function being defined.#

#

If the character set specified for a fixed-length character data type and for a variable-length character data type are different, they will be treated as different data types.

4. If SQL compile options are specified in `ALTER ROUTINE`, the length of the SQL statement that is created as a result of incorporating the SQL compile option in the `CREATE PUBLIC FUNCTION` specification for the procedure being recreated must not exceed the maximum allowable length for SQL statements.
5. If the SQL object being executed is invalidated, `CREATE PUBLIC FUNCTION` cannot be executed from within an external Java procedure.
6. For details about other rules, see *(1) CREATE FUNCTION (Define function)* under *CREATE [PUBLIC] FUNCTION (Define function, define public function)* in this chapter.

Notes

1. The `CREATE PUBLIC FUNCTION` statement cannot be specified from an X/Open-compliant UAP running under OLTP.
2. The following shows the invalidation conditions for an SQL object associated with the public function definition:
   - If an SQL object for which functions, procedures, or triggers are in effect already exists when the public function is defined and a public function with the same number of routine identifiers and SQL parameters already exist, the SQL object becomes invalid.
   - If an SQL object for which functions, procedures, or triggers that specify `MASTER` for the authorization identifier are in effect already exists when the public function is defined and a function with the same number of routine identifiers and SQL parameters, and for which `MASTER` is specified as the authorization identifier, already exist, the SQL object becomes invalid.

   For notes about defining or deleting an invalidated function, see *HiRDB Version 9 UAP Development Guide*.
3. Among the invalidated functions indicated in note 2, if a function satisfying either of the following conditions is used in the view definition, the public function definition results in an error:
   - Abstract data type is used in the data type of an argument.
   - Abstract data type is used in the data type of a return value.

4. To use a view table that uses an invalid function, you must execute `ALTER ROUTINE` and re-create the SQL object of the function.
5. `PUBLIC` is set in the column that stores the owner of the dictionary table (such as `ROUTINE_SCHEMA` in the `SQL_ROUTINES` table). Also, the authorization identifier that defines the public function is stored in the `ROUTINE_CREATOR` column of the `SQL_ROUTINES` table.
6. Use `DROP PUBLIC FUNCTION` to delete public functions.
7. For other notes, see *(1) CREATE FUNCTION (Define function)* under *CREATE [PUBLIC] FUNCTION (Define function, define public function)* in this chapter.

CREATE INDEX Format 1 (Define index)

Function

CREATE INDEX (format 1) defines an index for one or more columns of a base table.

Privileges

Owner of the table

A user can define indexes in public user RDAREAs for tables owned by that user.

Table owner who has the usage privilege for private user RDAREAs

A user can define indexes for tables owned by that user in private user RDAREAs for which the user has the usage privilege.

Format 1 (Define Index)

```
CREATE [UNIQUE] INDEX [authorization-identifier.]index-Identifier
      ON [authorization-identifier.]table-identifier(column-name [{ASC|DESC}]
                                          [, column-name [{ASC|DESC}]]...)
       [IN {RDAREA-name
            |(RDAREA-name|
            |((RDAREA-name)[, (RDAREA- name)]...)
            |matrix-partitioned-index-storage-RDAREA-specification}]
       [index-option]...
matrix-partitioned-index-storage-RDAREA-specification::=
      two-dimensional-storage-RDAREA-specification
two-dimensional-storage-RDAREA-specification::=
       (matrix-partitioning-RDAREA-list
         [, matrix-partitioning-RDAREA-list]...)
matrix-partitioning-RDAREA-list::=
       (RDAREA-name[, RDAREA-name]...)
>index-option ::={PCTFREE =unused-space-percentage
                 |UNBALANCED SPLIT
                 |EMPTY
                 |exception-value-specification}
exception-value-specification ::=EXCEPT VALUES (NULL[, NULL]...)
```

Operands

- [UNIQUE]

Specifies that the key values (all the values in the column or columns defined for the index) must be different (the value in each row must be unique).

If duplicate key values are detected during creation or updating of an index that has the UNIQUE attribute, HiRDB returns an error. However, a null value may be duplicated.

When the UNIQUE option is specified, the considerations discussed below should be noted.

1. When table row-partitioning is performed

The following table indicates the specifiability of UNIQUE in conjunction with row-partitioning.

Table 3-22: Specifiability of UNIQUE in conjunction with row-partitioning of a table

Table partitioning method[#1]		Index constituent column for which UNIQUE is specified[#2]		Index partitioning method[#4]	UNIQUE specifiability
Row-partitioning within a server	Key range partitioning (not matrix partitioning) and FIX hash partitioning	Partitioning key index		Matches the number of table partitions.	Y
				Does not match the number of table partitions.	--
		Non-partitioning key index	Including partitioning keys (in any order)	Matches the number of table partitions.	Y
				Does not match the number of table partitions (no row-partitioning).	Y
			Not including partitioning keys (in any order)	Matches the number of table partitions.	N
				Does not match the number of table partitions (no row-partitioning).	Y[#3]
	Flexible hash partitioning	Not applicable		Not applicable	N
	Matrix partitioning	Partitioning key index		Matches the number of table partitions.	Y
				Does not match the number of table partitions.	--
		Non-partitioning key index	Including all partitioning keys (in any order)	Matches the number of table partitions.	Y
				Does not match the number of table partitions.	N

Table partitioning method[#1]		Index constituent column for which UNIQUE is specified[#2]		Index partitioning method[#4]	UNIQUE specifi-ability
			Not including any partitioning key (in any order)	Matches the number of table partitions.	N
				Does not match the number of table partitions (no row-partitioning).	--
Row-partitioning among servers (no partitioning within a server)	Key range partitioning (not matrix partitioning) and FIX hash partitioning	Partitioning key index		Matches the number of table partitions.	Y
				Does not match the number of table partitions.	--
		Non-partitioning key index	Including all partitioning keys (in any order)	Matches the number of table partitions.	Y
				Does not match the number of table partitions.	--
			Not including any partitioning key (in any order)	Matches the number of table partitions.	N
				Does not match the number of table partitions.	--
	Flexible hash partitioning	Not applicable		Not applicable	N
	Matrix partitioning	Partitioning key index		Matches the number of table partitions.	Y
				Does not match the number of table partitions.	--
		Non-partitioning key index	Including all partitioning keys (in any order)	Matches the number of table partitions.	Y
				Does not match the number of table partitions.	--

Table partitioning method[#1]		Index constituent column for which UNIQUE is specified[#2]		Index partitioning method[#4]	UNIQUE specifi-ability
			Not including any partitioning key (in any order)	Matches the number of table partitions.	N
				Does not match the number of table partitions.	--
Row-partitioning among servers (with partitioning within a server)	Key range partitioning (not matrix partitioning) and FIX hash partitioning	Partitioning key index		Matches the number of table partitions.	Y
				Does not match the number of table partitions.	--
		Non-partitioning key index	Including all partitioning keys (in any order)	Matches the number of table partitions (also partitioned on the server)	Y
				Does not match the number of table partitions (not partitioned on the server)	Y
			Not including any partitioning key (in any order)	Matches the number of table partitions (also partitioned on the server)	N
				Does not match the number of table partitions (not partitioned on the server)	N
	Flexible hash partitioning	Not applicable		Not applicable	N
	Matrix partitioning	Partitioning key index		Matches the number of table partitions.	Y
				Does not match the number of table partitions.	--

Table partitioning method#1		Index constituent column for which UNIQUE is specified#2		Index partitioning method#4	UNIQUE specifi-ability
		Non-partitioning key index	Including all partitioning keys (in any order)	Matches the number of table partitions.	Y
				Does not match the number of table partitions.	--
			Not including any partitioning key (in any order)	Matches the number of table partitions.	N
				Does not match the number of table partitions.	--

Legend:

Y: `UNIQUE` can be specified.

N: `UNIQUE` cannot be specified.

--: An index cannot be defined.

#1: *Row-partitioning within a server* refers to either row-partitioning on a HiRDB/Single Server or row-partitioning that is closed on a back-end server on a HiRDB/Parallel Server. *Row-partitioning among servers* refers to row-partitioning that extends across multiple servers on a HiRDB/Parallel Server. If a server contains a mix of tables, some whose table partitioning method is row-partitioning within a server and some whose table partitioning method is row-partitioning among servers, that server is classified as row-partitioning among servers. For details about row-partitioning of tables, see the *HiRDB Version 9 Installation and Design Guide*.

#2: *Partitioning key index* refers to an index in which the column (partitioning key) in which a storage condition is specified for the row-partitioning of a table is the first constituent column, and such indexes include the following:

Single-column partitioning:

A single-column index that is created in a partitioning key, or a multiple-column index that is created in multiple columns for which the partitioning key is the first column

Multiple-column partitioning:

A multiple-column index that is created in multiple columns and that contain all of the columns specified for partitioning from the beginning in the same order

Indexes that are not partitioning key indexes are referred to as *non-partitioning*

key indexes.

#3: The *N* mark indicates the use of the rebalancing facility for hash-partitioned tables (due to the fact that UNIQUE cannot be specified if an RDAREA is added to change the partitioning to row-partitioning among servers).

#4: The method by which an index is partitioned depends on how the index storage RDAREA is specified. With the exception of flexible hash partitioning. omitting the IN operand may prevent specification of UNIQUE, due to the index partitioning method. Notice that explicitly specifying an index storage RDAREA in the IN operand may prevent the specification of UNIQUE.

2. When a repetition column is used

A repetition column cannot be specified as a constituent column of an index for which UNIQUE is specified.

- [*authorization-identifier*.]*index-identifier*

Specifies the authorization identifier of the user who will own the created index and a name for the index.

The index name cannot be the same as the table name.

- [*authorization-identifier*.]*table-identifier*

Specifies the authorization identifier of the user who owns the table that is to be indexed and the name of the base table for which the index is being created.

- (*column-name* [{ASC|DESC}][,*column-name*[{ASC|DESC}]]...)

column-name

Specifies the name of a column for which the index is being defined.

A maximum of 16 column names can be specified.

When multiple column names are specified, each column name must be unique.

ASC

Specifies that the index is to be organized in ascending order of the key values.

DESC

Specifies that the index is to be organized in descending order of the key values.

- IN{*RDAREA-name*

  |(*RDAREA-name*)

  |((*RDAREA-name*)[, (*RDAREA-name*)]...)

  |*matrix-partitioned-index-storage-RDAREA-specification*}

  |*matrix-partitioned-index-storage-RDAREA-specification*::=

|*two-dimensional-storage-RDAREA-specification*

two-dimensional-storage-RDAREA-specification::=

(*matrix-partitioning-RDAREA-list*

[, *matrix-partitioning-RDAREA-list*]...)

matrix-partitioning-RDAREA-list::=

(*RDAREA-name*[, *RDAREA-name*]...)

Specifies the names of the RDAREAs in which the index is to be stored.

Specify *matrix-partitioned-index-storage-RDAREA-specification* when defining an index for a matrix-partitioned table. RDAREA names are subject to the same rules as non-matrix-partitioned tables.

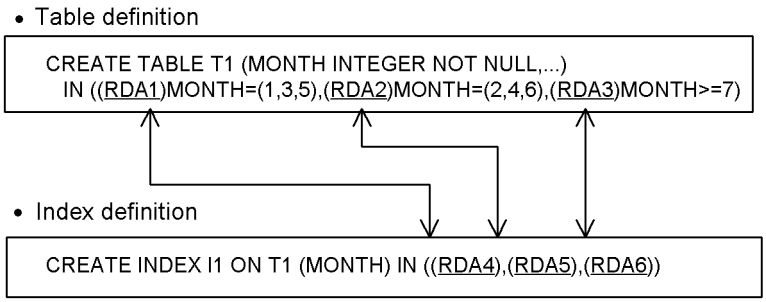
The following rules apply to RDAREAs that store indexes:

1. In addition, the following restrictions apply to HiRDB/Parallel Servers: If the table specified in *table-identifier* is a shared table, the RDAREA name must be a shared RDAREA. If the table specified in *table-identifier* is not a shared table, a shared RDAREA cannot be specified in the RDAREA name.
2. The specified RDAREAs must have already been created by the database initialization utility or added by the database structure modification utility.
3. If no RDAREA names are specified, the index is stored in the RDAREAs in which the table specified by the table identifier is stored. However, if the table is partitioned in a HiRDB/Single Server or is partitioned in the same back-end server in a HiRDB/Parallel Server, an index in which a column other than the partitioning key is specified at the beginning of the index is stored in the first table-storage RDAREA for which a partitioning condition was specified. For a matrix-partitioned table, a two-dimensional RDAREA specification cannot be omitted unless all the columns specified in a partitioned key are specified in the same order from the beginning of index constituent columns.
4. The same RDAREA name cannot be used when specifying multiple RDAREAs. However, duplicate storage RDAREA names can be specified in row-partitioned tables, hash-partitioned tables, and matrix-partitioned tables that specify a boundary value.
5. If a table is partitioned and stored in multiple RDAREAs, the index storage RDAREA is specified as follows:
   - For a partitioning key index, specify the same number of RDAREA names as the number of RDAREAs containing the table. In this case, the index will be stored in the order in which the table storage RDAREAs are specified in `CREATE TABLE`.
   - If a table storage RDAREA name is duplicated in a row-partitioned table,

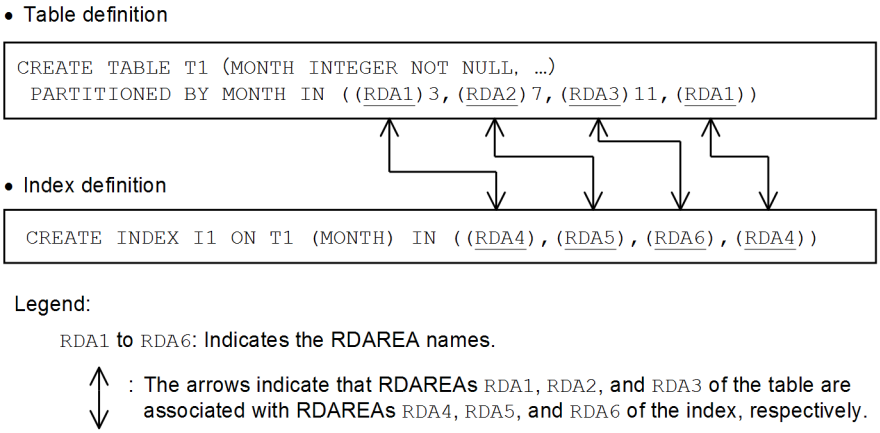
hash-partitioned table, or matrix-partitioned table that specifies a boundary value, specify the index storage RDAREA name so that it corresponds.

Definition examples are as follows:

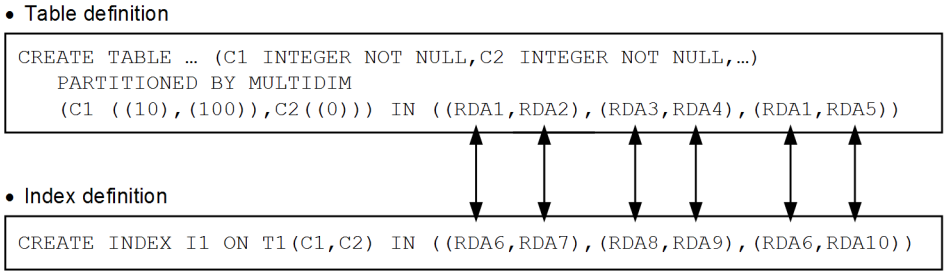
Example 1: Partitioning by rows based on storage conditions:



Example 2: Partitioning by rows based on a boundary value:



Example 3: Matrix partitioning



Legend:
RDA1 to RDA10: Indicates the RDAREA names.
: The arrows indicate that RDAREAs RDA1, RDA2, RDA3, RDA4, and RDA5 of the table are associated with RDAREAs RDA6, RDA7, RDA8, RDA9, and RDA10 of the index, respectively.

6. For a HiRDB/Parallel Server, the RDAREAs that store the table and the RDAREAs that store the associated index must be located in the same back-end server.
7. For a non-partitioning key index, specify RDAREAs as follows:
   - If an index is not row-partitioned within a server, specify RDAREA names in a number equal to the servers on which partitioned tables are stored. For a HiRDB/Parallel Server, specify an RDAREA for each back-end server containing a table. For a HiRDB/Single Server, only one RDAREA can be specified. Matrix-partitioned tables are treated in the same way as a partitioned key index.
   - If an index is to be row-partitioned within a server, specify RDAREA names in a number equal to the RDAREAs storing the table. In this case, the targets of index storage correspond to the order in which table storage RDAREAs were specified in CREATE TABLE. Also, if a table storage RDAREA name is duplicated in a row-partitioned table, hash-partitioned table, or matrix-partitioned table that specifies a boundary value, specify the index storage RDAREA name so that it corresponds.

   The RDAREAs can be specified in any order; the indexes associated with the table storage RDAREAs at the same server will be stored respectively.
8. User RDAREAs storing rebalancing tables cannot be specified.
9. When an index for a rebalancing table is defined, an RDAREA name must be specified.

■ PCTFREE=*unused-space-percentage*

Specifies the percentage of unused space to be left in each index page when the index is created. The range of specifiable values is 0 to 99, and the default is 30.

When created in batch by the database load utility and the database reorganization utility, indexes are created in a percentage equal to the percentage of unused space. In other addition or update operations by INSERT or UPDATE statements, the default PCTFREE=0 is assumed.

If rows will be added frequently after the index has been created, a high percentage of unused space should be specified.

- UNBALANCED SPLIT

Specifies that the key values are to be allocated unevenly among the pages when a page is split.

If the location where a key value is to be inserted is in the first half of the page to be split, more empty space is allocated to the left-side page after the split. If the key value insertion location is in the second half of the page, more empty space is allocated to the right-side page after the split. This is called an unbalanced index split.

For details of unbalanced index splits, see the *HiRDB Version 9 System Operation Guide*.

- EMPTY

Specifies that an unfinished index is to be created.

The EMPTY option improves the capacity for concurrent execution of index definitions. This option is also effective when a table contains a large amount of data and the definitions of multiple indexes must be executed concurrently. The option is not effective for tables that do not contain data.

For details of using the EMPTY option, see the *HiRDB Version 9 System Operation Guide*.

- *exception-value-specification*

Specifies that when the index is created, key values that are composed solely of null values are to be excluded.

The exception values option cannot be specified for an index that contains NOT NULL constraint columns. When the exception values option is specified for an index, indexes cannot be unloaded in index order by the database reorganization utility.

Constituent columns of an index for which the exception values option is specified cannot include repetition columns.

Common rules

1. A maximum of 255 indexes can be created for a table.
2. An index can be defined for columns that contain the null value and for columns that do not have any rows.
3. An index cannot be defined for columns of the following data types:

- BLOB
- BINARY
- Abstract data type

4. If an index is composed of multiple columns, columns of the following data types, in addition to item 3, cannot be specified:
   - FLOAT
   - SMALLFLT
5. The total length of columns comprising an index must satisfy the following formula:

   Total length of columns ≤

   MIN((*page size of index storage RDAREAs* ÷ 2) - 1242, 4036)

   The following table lists the lengths of the columns that comprise an index.

   Table 3-23: Lengths of multicolumn index columns

Data type	When the combined length of the columns does not exceed 255 bytes			When the combined length of the columns exceeds 255 bytes		
	Columns comprising a single column index	Columns comprising a multicolumn index		Columns comprising a single column index	Columns comprising a multicolumn index	
		Fixed length columns only	Variable length columns are also included		Fixed length columns only	Variable length columns are also included
INTEGER	4	5	6	--	5	7
SMALLINT	2	3	4	--	3	5
DECIMAL[(*m*[,*n*])]	↓*m* ÷ 2↓ + 1	↓*m* ÷ 2↓ + 2	↓*m* ÷ 2↓ + 3	--	↓*m* ÷ 2↓ + 2	↓*m* ÷ 2↓ + 4
FLOAT	8	--	--	--	--	--
SMALLFLT	4	--	--	--	--	--
CHAR(*n*), MCHAR(*n*)	*n*1	*n*1 + 1	*n*1 + 2	*n*1	*n*1 + 1	*n*1 + 3
NCHAR(*n*)	2 x *n*2	2 x *n*2 + 1	2 x *n*2 + 2	2 x *n*2	2 x *n*2 + 1	2 x *n*2 + 3
DATE	4	5	6	--	5	7
TIME	3	4	5	--	4	6

Data type	When the combined length of the columns does not exceed 255 bytes			When the combined length of the columns exceeds 255 bytes		
	Columns comprising a single column index	Columns comprising a multicolumn index		Columns comprising a single column index	Columns comprising a multicolumn index	
		Fixed length columns only	Variable length columns are also included		Fixed length columns only	Variable length columns are also included
TIMESTAMP	7 + *p* ÷ 2	8 + *p* ÷ 2	9 + *p* ÷ 2	--	8 + *p* ÷ 2	10 + *p* ÷ 2
INTERVAL YEAR TO DAY	5	6	7	--	6	8
INTERVAL HOUR TO SECOND	4	5	6	--	5	7
VARCHAR, MVARCHAR	*n*1 + 1	--	*n*1 + 2	*n*1 + 2	--	*n*1 + 3
NVARCHAR	2 x *n*2 + 1	--	2 x *n*2 + 2	2 x *n*2 + 2	--	2 x *n*2 + 3

m, *n*: Positive integers

*n*1: Actual data length

*n*2: Number of characters

p: Fractional second precision

--: Not applicable

6. Only one index can be defined for a single column, regardless of whether it is sorted in ascending order or descending order. For a multicolumn index, two indexes are considered to be identical if all their member columns sorted in ascending order are exact inverses of their descending-order counterparts.
7. If a procedure and a trigger are already defined for the table for which an index is to be defined, index information in the SQL object is nullified, and the trigger cannot be executed. Because the affected procedure or the trigger cannot be executed from another procedure, the SQL object needs to be re-created.
8. The same index option cannot be specified more than once.
9. If an index composed of multiple repetition columns is defined, the repetition columns must have the same number of current elements.
10. ALTER TABLE cannot be executed from a Java procedure if the execution result invalidates the SQL object being executed.

11. RDAREAs using the inner replica facility and those not using the facility cannot be specified on a mixed basis in the index storage RDAREAs. When specifying an RDAREA to which the inner replica facility is applied, specify the name of the original RDAREA.
12. For execution conditions for `CREATE INDEX` using the inner replica facility, see the manual *HiRDB Staticizer Option Version 9*.
13. A maximum of 500 indexes can be stored per RDAREA.

Notes

1. When a value in an indexed column is updated, the associated index is also updated.
2. When a multicolumn index is defined, the order in which the columns are specified is the order in which the key values are created.
3. A multicolumn index can include a column for which a single-column index is defined.
4. The `CREATE INDEX` statement cannot be specified from an X/Open-compliant UAP running under OLTP.
5. When an index is defined with the `EMPTY` option specified, it must be re-created with the index re-creation function of the database reorganization utility; for details, see the manual *HiRDB Version 9 Command Reference*.
6. For rules on defining an index for a table with a `WITHOUT ROLLBACK` specification, see the rules on `WITHOUT ROLLBACK` in *CREATE TABLE (Define table)*.
7. An index for which the exception values option is specified cannot be used in an SQL statement that is subject to selection of exception value rows.
8. When using a `CREATE INDEX` statement to create an index in a table that contains a large amount of data, a large amount of processing time may be required. Before executing the `CREATE INDEX` statement, set the timer monitoring specification as follows:
   - system-defined `pd_watch_time`

     Specify `0` or omit the specification.
   - Client environment definition `PDCWAITTIME`

     If the `CREATE INDEX` execution time can be estimated from past experience and the volume of data, specify the estimated value.

     If the execution time cannot be estimated, specify `0` or omit the specification.
9. If the table storage RDAREA or specified index storage RDAREA is in a blocked state when defining an index in a row-partitioned table defined across multiple

servers on a HiRDB/Parallel Server, the RDAREA will be in a lock wait state. In this case, an error may not be returned immediately when the specified value of the system-defined pd_lck_wait_timeout operand is reached and the process times out. To prevent this, cancel the blocked state of the table storage RDAREA or specified index storage RDAREA before executing the CREATE INDEX statement.

Examples

1. Define an ascending-order index (IDX1) for the product code (PCODE) column of a stock table (STOCK); assume that rows will be added frequently after the index has been defined (specify 50 as the percentage of unused space to be left in the index pages):

```
CREATE INDEX IDX1
   ON STOCK(PCODE ASC)
   PCTFREE = 50
```

2. Define a multicolumn index (IDX2) for the product name (PNAME) and color (COLOR) columns of a stock table (STOCK), and store the index in a user RDAREA (RDA1):

```
CREATE INDEX IDX2
   ON STOCK(PNAME,COLOR)
   IN RDA1
```

3. Define an ascending-order index (IDX3) for the product code (PCODE) column of a stock table (STOCK); partition the index and store it in RDAREAs RDA1, RDA2, and RDA3; assume that the stock table is stored on a split basis in three RDAREAs, using the product code as the partitioning key:

```
CREATE INDEX IDX3
   ON STOCK(PCODE)
   IN ((RDA1),(RDA2),(RDA3))
```

CREATE INDEX Format 2 (Define index)

Function

CREATE INDEX (format 2) defines an index of a specified index type.

Privileges

Owner of the table

A user can define indexes in public user RDAREAs for tables owned by that user.

Table owner who has the usage privilege for private user RDAREAs

A user can define indexes for tables owned by that user in private user RDAREAs for which the user has the privilege.

Format 2 (Define index of a specified index type)

```
CREATE INDEX [authorization-identifier.]index-identifier
    USING TYPE [authorization-identifier.]index-type-identifier
    ON [authorization-identifier.]table-identifier (column-name)
    IN {RDAREA-name
        |(RDAREA-name)
        |((RDAREA-name)[, (RDAREA-name)]...)
        |matrix-partitioned-index-storage-RDAREA-specification}
      [index-option]...
      [PLUGIN plug-in-option]
matrix-partitioned-index-storage-RDAREA-specification::=
    two-dimensional-storage-RDAREA-specification
two-dimensional-storage-RDAREA-specification::=
    (matrix-partitioning-RDAREA-list
      [, matrix-partitioning-RDAREA-list]...)
matrix-partitioning-RDAREA-list::=
    (RDAREA-name[, RDAREA-name]...)
index-option ::= EMPTY
```

Operands

- [*authorization-identifier*.]*index-identifier*

Specifies the authorization identifier of the user who will own the created index and specifies a name for the index.

The index name cannot be the same as the table name.

- USING TYPE [*authorization-identifier*.]*index-identifier*

authorization-identifier

Specifies the authorization identifier of the owner of the index type.

When the authorization identifier is omitted, the default authorization identifier does not include an index type of the same name, and there is an index type of the same name in the MASTER authorization identifier, that index type is assumed to have been specified.

index-type-identifier

Specifies the index type. For the index type identifiers, see the manual for the appropriate plug-in.

- [*authorization-identifier*.]*table-identifier* (*column-name*)

authorization-identifier

Specifies the authorization identifier of the user who will own the created table.

table-identifier

Specifies the name of the base table on which the created index is to be based.

FIX tables cannot be specified.

column-name

Specifies the name of the column for which the index is being defined. The only permissible data type for the column is an abstract data type.

- IN{*RDAREA-name*
  |(*RDAREA-name*)
  |((*RDAREA-name*)[, (*RDAREA-name*)]...)
  |*matrix-partitioned-index-storage-RDAREA-specification*}

  matrix-partitioned-index-storage-RDAREA-specification::=
  two-dimensional-storage-RDAREA-specification

  two-dimensional-storage-RDAREA-specification::=
  (*matrix-partitioning-RDAREA-list*
  [, *matrix-partitioning-RDAREA-list*]...)

  matrix-partitioning-RDAREA-list::=
  (*RDAREA-name*[, *RDAREA-name*]...)

Specifies the name of the RDAREA for storing indexes.

When defining an index for a matrix-partitioned table, specify a matrix-partitioned index storage RDAREA.

The following rules apply to RDAREAs that store an index:

1. The RDAREA name must be a user LOB RDAREA.
2. The RDAREA must be created or added in advance by using either the database initialization utility or the database structure modification utility, respectively.
3. The same RDAREA name cannot be used when specifying multiple RDAREAs. However, duplicate table storage RDAREA names can be specified in row-partitioned tables, hash-partitioned tables, and matrix-partitioned tables that specify a boundary value.
4. If a table is stored in multiple RDAREAs on a partitioned basis, index storage RDAREAs can be specified as follows:
   - Specify RDAREA names in a number equal to the RDAREAs storing tables. In this case, the targets of index storage correspond to the order in which table storage RDAREAs are specified in `CREATE TABLE`.
   - If a table storage RDAREA name is duplicated in a row-partitioned table, hash-partitioned table, or matrix-partitioned table that specifies a boundary value, specify the index storage RDAREA name so that it corresponds.

- `EMPTY`

Specifies that an empty plug-in index is being created.

The `EMPTY` option improve the capacity for concurrent execution of plug-in index definitions. This option is also effective when a table contains a large amount of data and the definitions of multiple indexes must be executed concurrently. This option is not effective for tables that do not contain data.

For details of using the `EMPTY` option, see the *HiRDB Version 9 System Operation Guide*.

- `PLUGIN` *plug-in-option*

Specifies as a character string literal (of up to 255 characters) parameter information for the plug-in index. Hexadecimal character string literals cannot be specified as parameter information. For details about the parameter information, see the documentation for the appropriate plug-in.

Common rules

1. A maximum of 255 indexes can be defined per table.
2. An index can be defined for a column that contains NULL values as well as for a column that does not contain any rows.
3. RDAREAs using the inner replica facility and those not using the facility cannot be specified on a mixed basis in the index storage RDAREAs. When specifying an RDAREA to which the inner replica facility is applied, specify the name of the original RDAREA.

4. For execution conditions for CREATE INDEX using the inner replica facility, see the manual *HiRDB Version 9 Staticizer Option*.
5. A maximum of 500 indexes can be stored per RDAREA.

Notes

1. When a value in an indexed column is updated by a user, the index is also updated.
2. The CREATE INDEX statement cannot be specified from an X/Open-compliant UAP running under OLTP.
3. A plug-in index that is defined with the EMPTY option specified must be re-created by the database reorganization utility. For details of re-creating an index, see the manual *HiRDB Version 9 Command Reference*.
4. When using a CREATE INDEX statement to create an index in a table that contains a large amount of data, a large amount of processing time may be required. Before executing the CREATE INDEX statement, set the timer monitoring specification as follows:
   - system-defined pd_watch_time

     Specify 0 or omit the specification.
   - Client environment definition PDCWAITTIME

     If the CREATE INDEX execution time can be estimated from past experience and the volume of data, specify the estimated value.

     If the execution time cannot be estimated, specify 0 or omit the specification.
5. If the table storage RDAREA or specified index storage RDAREA is in a blocked state when defining an index in a row-partitioned table defined across multiple servers on a HiRDB/Parallel Server, the RDAREA will be in a lock wait state. In this case, an error may not be returned immediately when the specified value of the system-defined pd_lck_wait_timeout operand is reached and the process times out. To prevent this, cancel the blocked state of the table storage RDAREA or specified index storage RDAREA before executing the CREATE INDEX statement.

CREATE INDEX Format 3 (Define substructure index)

Function

CREATE INDEX format 3 defines an index with an identified partial structure of a value in an XML type column as the key.

For details about the privileges, format, and rules, see *1.14.6(1) CREATE INDEX Format 3 (Define substructure index)*.

CREATE [PUBLIC] PROCEDURE (Define procedure, define public procedure)

Function

CREATE PROCEDURE defines a procedure.

CREATE PROCEDURE can also be used to define *public* procedures, which are available to all users without them having to qualify the routine identifier with an authorization identifier.

(1) CREATE PROCEDURE (Define procedure)

Privileges

Users who own a schema

These users can define their own procedures.

Format

```
CREATE procedure-body
    procedure-body::=PROCEDURE [authorization-identifier.]routine-identifier
                        ([{IN|OUT|INOUT} SQL-parameter-name data-type
                            [, {IN|OUT|INOUT}
                               SQL-parameter-name data-type]...])
                    [DYNAMIC RESULT SETS number-of-results-sets]
                    [LANGUAGE clause]
                    [SQL-compile-option[SQL-compile-option]...]
                    {SQL-procedure-statements |external-routine-specification}

LANGUAGE clause ::= LANGUAGE {SQL|JAVA|C}
SQL-compile-option ::= {ISOLATION data-guarantee-level [FOR UPDATE
EXCLUSIVE]
    |OPTIMIZE LEVEL SQL-optimization-option
                        [,SQL-optimization-option]...
    |ADD OPTIMIZE LEVEL SQL-extension-optimizing-option
                        [,SQL-extension-optimizing-option]...
|SUBSTR LENGTH maximum-character-length}
external-routine-specification ::= EXTERNAL NAME
{external-Java-routine-name|external-C-stored-routine-name}
                                    PARAMETER STYLE parameter-style
parameter-style ::= {JAVA|RDSQL}
```

Operands

- [*authorization-identifier*.]*routine-identifier*

authorization-identifier

Specifies the authorization identifier of the user who owns the procedure being defined.

routine-identifier

Specifies the name of a routine in the procedure being defined.

- ([{IN|OUT|INOUT}*SQL-parameter-name data-type*[, {IN|OUT|INOUT}*SQL-parameter-name data-type*]...])

{IN|OUT|INOUT}

I/O mode (parameter mode) must be specified for procedure parameters.

IN

This keyword specifies that the parameter is an input parameter.

OUT

This keyword specifies that the parameter is an output parameter.

INOUT

This keyword specifies that the parameter is an input/output parameter.

SQL-parameter-name

Specifies the name of the parameter in the procedure. SQL parameter names must be unique within a procedure.

data-type

Specifies the data type of the parameter in the procedure.

If the specified data type is an abstract data type, no authorization identifier is specified, and the default authorization identifier does not have an abstract data type of the same name, and if there is an abstract data type of the same name in the MASTER authorization identifier, that abstract data type is assumed to have been specified.

The following data types cannot be specified:

- Abstract data type when JAVA or C is specified in the LANGUAGE clause
- BINARY or BLOB type when C is specified in the LANGUAGE clause

For details about the data types that can be specified, see *1.10.1(2) Type mapping* or *CREATE [PUBLIC] FUNCTION (Define function, define public function)* in this chapter.

- DYNAMIC RESULT SETS *number-of-results-sets*

Specifies as an integer the maximum number of results sets that can be returned by the

procedure that is being defined.

The specifiable range of values is 0 to 1023. When 0 is specified or this operand is omitted, the default is that the procedure does not return a results set.

However, if C is specified in the LANGUAGE clause, the value specified in this operand is ignored.

- LANGUAGE {SQL |JAVA |C}

Specifies the language used to write the function.

For an external routine, SQL must not be specified.

SQL

Specifies that the processing part of the function is made up of SQL statements.

JAVA

Specifies that the processing part of the function is specified as an external routine and the function is to be implemented as a Java class method.

C

Specifies that the processing part of the procedure is specified as an external routine and the procedure is to be implemented as a C module. When C is specified, the maximum number of SQL parameters is limited to 128.

Whether other operands need to be specified depends on what is specified for this operand. The following table indicates whether other operands need to be specified depending on the LANGUAGE clause specification.

Table 3-24: Whether other operands need to be specified depending on the LANGUAGE clause specified in CREATE PROCEDURE

Other operand	LANGUAGE clause specification		
	SQL	JAVA	C
EXTERNAL NAME	N	Y	Y
PARAMETER STYLE	N	JAVA	RDSQL
SQL-procedure-statement	Y	N	N

Legend:

Y: Must be specified.

N: Cannot be specified.

JAVA: Specify JAVA.

RDSQL: Specify RDSQL.

- *SQL-compile-option*::={ISOLATION *data-guarantee-level* [FOR UPDATE EXCLUSIVE]

  |OPTIMIZE LEVEL *SQL-optimization-option*

  [, *SQL-optimization-option*]...

  |ADD OPTIMIZE LEVEL *SQL-extension-optimizing-option*

  [, *SQL-extension-optimizing-option*]...

  |SUBSTR LENGTH *maximum-character-length*}

In *SQL-compile-option*, ISOLATION, OPTIMIZE LEVEL, ADD OPTIMIZE LEVEL, and SUBSTR LENGTH can each be specified only once.

ISOLATION *data-guarantee-level* [FOR UPDATE EXCLUSIVE]

Specifies an SQL data integrity guarantee level.

data-guarantee-level

A data integrity guarantee level specifies the point to which the integrity of the transaction data must be guaranteed. The following data integrity guarantee levels can be specified:

- 0

  Do not guarantee data integrity. Specifying 0 for a set of data allows the user to reference the data even when it is being updated by another user. However, if the table to be referenced is a shared table, and another user is executing the LOCK statement in the lock mode, the system waits until the lock condition is released.

- 1

  Guarantee the integrity of data until a retrieval process is completed. When level 1 is specified, data that has been retrieved cannot be updated by other users until the retrieval process is completed (until HiRDB finishes viewing the current page or row).

- 2

  Guarantee the integrity of data that has been retrieved until the transaction is completed. When level 2 is specified, data that has been retrieved cannot be updated by other users until the transaction is completed.

[FOR UPDATE EXCLUSIVE]

Specify this operand if WITH EXCLUSIVE LOCK is always to be assumed, irrespective of the data guarantee level specified in *SQL-compile-option* for a

cursor or query in a procedure for which the FOR UPDATE clause is specified or assumed. If level 2 is specified in *data-guarantee-level*, WITH EXCLUSIVE LOCK is assumed for the cursor or query in a procedure for which the FOR UPDATE clause is specified or assumed, in which case it is not necessary to specify FOR UPDATE EXCLUSIVE.

Relationship with client environment definition

Any specification of PDISLLVL or PDFORUPDATEEXLOCK with respect to CREATE PROCEDURE has no effect.

Relationship to SQL statements

If a lock option is specified in an SQL statement in a procedure, the lock option specified in the SQL statement takes precedence over any data guarantee level specified in *SQL-compile-option* or the lock option assumed because of FOR UPDATE EXCLUSIVE.

The default for *data-guarantee-level* is 2.

For data guarantee levels, see the *HiRDB Version 9 UAP Development Guide*.

OPTIMIZE LEVEL *SQL-optimization-option*[, *SQL-optimization-option*]...

Specifies the optimization method for determining the most efficient access path by taking the database's status into consideration.

SQL optimization options can be specified with identifiers (character strings) or numeric values. Hitachi recommends that identifiers be used.

Specification with identifiers:

```
OPTIMIZE LEVEL "identifier"[, "identifier"] ...
```

Specification examples

- Apply the *Prioritized nest-loop-join* and the *Rapid grouping facility* options:

```
OPTIMIZE LEVEL "PRIOR_NEST_JOIN",
"RAPID_GROUPING"
```

- Do not apply any optimization:

```
OPTIMIZE LEVEL "NONE"
```

Rules

1. At least one identifier must be specified.
2. When multiple identifiers are specified, separate them with the comma (,).
3. For details about the contents that can be specified in an identifier (optimization methods), see *Table 3-25 SQL optimization option*

specification values (CREATE PROCEDURE).

4. If no optimization options are to be applied, specify `"NONE"` as the identifier. If `"NONE"` and some other identifier are both specified, the `"NONE"` specification is ignored.
5. The identifiers are case-sensitive.
6. If the same identifier is specified more than once, it is treated as if it was specified only once; however, where possible, precautions should be taken to avoid specifying a given identifier in duplicate.

Specification with numeric values:

`OPTIMIZE LEVEL` *unsigned-integer*`[,` *unsigned-integer*`] ...`

Specification examples

- Apply the *2. Making multiple SQL objects*, *8. Suppressing use of AND multiple indexes*, and *13. Forcing use of multiple indexes* options:

Specify unsigned integers separated by commas:

```
OPTIMIZE LEVEL 4,10,16
```

Specify a sum of unsigned integers:

```
OPTIMIZE LEVEL 30
```

- Add the new value 16 to the previously specified value 14 (4 + 10):

```
OPTIMIZE LEVEL 14,16
```

- Do not apply any optimization:

```
OPTIMIZE LEVEL 0
```

Rules

1. When HiRDB is upgraded from a version earlier than Version 06-00 to a Version 06-00 or later, the total value specification in the earlier version also remains valid. If the optimization option does not need to be modified, the specification value for this operand need not be changed when HiRDB is upgraded to a Version 06-00 or later.
2. At least one integer must be specified.
3. When multiple integers are specified, separate them with the comma (`,`).
4. For details about the contents that can be specified in an unsigned integer (optimization methods), see *Table 3-25 SQL optimization option specification values (CREATE PROCEDURE)*.
5. If no optimization options are to be applied, specify `0` as the integer. If `0` and another integer are both specified, the `0` specification is ignored.

6. If the same integer is specified more than once, it will be treated as a single instance of the integer. However, avoid multiple specifications of the same integer
7. When multiple optimization options are to be applied, you can specify the sum of the appropriate unsigned integers. However, the same optimization option value must not be added in more than once to avoid the possibility of the addition result being interpreted as a different set of optimization options.
8. Specifying multiple optimization options by adding their values can be ambiguous as to which optimization options are actually intended, so Hitachi recommends that the option values be specified individually separated by commas. If multiple optimization option have already been specified by the addition method and another optimization option is required, you can specify the new option's value following the previous summed value by separating them with a comma.

Relationships to system definition

1. If no SQL optimization option values are specified, the values specified in the `pd_optimize_level` operand of the system definition are assumed as the default. For details of the `pd_optimize_level` operand, see the manual *HiRDB Version 9 System Definition*.
2. When the `pd_floatable_bes` operand or the `pd_non_floatable_bes` operand is specified, specification of the *Increasing the target floatable servers (back-end servers for fetching data)* option or the *Limiting the target floatable servers (back-end servers for fetching data)* option, respectively, is invalid.
3. When `KEY` is specified in the `pd_indexlock_mode` operand of the system definition (i.e., in the case of index key value lock), specification of the *Suppressing creation of update-SQL work tables* option is invalid.

Relationship to client environment definition

The specification of `PDSQLOPTLVL` has no applicability to `CREATE PROCEDURE`.

Relationship with SQL

If SQL optimization is specified in an SQL statement, the SQL optimization specification takes precedence over SQL optimization options. For SQL optimization specifications, see *2.24 SQL optimization specification*.

SQL optimization option specification values

The following table lists the SQL optimization option specification values. For details about optimization methods, see the *HiRDB Version 9 UAP*

Development Guide.

Table 3-25: SQL optimization option specification values (CREATE PROCEDURE)

No.	Optimization option	Specification values	
		Identifier	Unsigned integer
1	Forced nest-loop-join	FORCE_NEXT_JOIN	4
2	Making multiple SQL objects	SELECT_APSL	10
3	Increasing the target floatable servers (back-end servers for fetching data)#1, #2	FLTS_INC_DATA_BES	16
4	Prioritized nest-loop-join	PRIOR_NEST_JOIN	32
5	Increasing the number of floatable server candidates#2	FLTS_MAX_NUMBER	64
6	Priority of OR multiple index use	PRIOR_OR_INDEXES	128
7	Group processing, ORDER BY processing, and DISTINCT set function processing at the local back-end server#2	SORT_DATA_BES	256
8	Suppressing use of AND multiple indexes	DETER_AND-INDEXES	512
9	Rapid grouping facility	RAPID_GROUPING	1024
10	Limiting the target floatable servers (back-end servers for fetching data)#1, #2	FLTS_ONLY_DATA_BES	2048
11	Separating data collecting servers#1,#2	FLTS_SEPARATE_COLLECT_SVR	2064
12	Suppressing index use (forced table scan)	FORCE_TABLE_SCAN	4096
13	Forcing use of multiple indexes	FORCE_PLURAL_INDEXES	32768
14	Suppressing creation of update-SQL work tables	DETER_WORK_TABLE_FOR_UPDATE	131072
15	Deriving rapid search conditions	"DERIVATIVE_COND"	262144
16	Applying key conditions, including scalar operations	"APPLY_ENHANCED_KEY_COND"	524288
17	Facility for batch acquisition from functions provided by plug-ins	"PICKUP_MULTIPLE_ROWS_PLUGIN"	1048576

No.	Optimization option	Specification values	
		Identifier	Unsigned integer
18	Facility for moving search conditions into derived table	"MOVE_UP_DERIVED_COND"	2097152

#1: If the *3. Increasing the target floatable servers (back-end servers for fetching data)* option and the *10. Limiting the target floatable servers (back-end servers for fetching data)* option are both specified, neither of these options will be applied; instead, the *11. Separating data collecting servers* option will be applied.

#2: This option is ignored if specified for a HiRDB/Single Server.

ADD OPTIMIZE LEVEL
SQL-extension-optimizing-option[, *SQL-extension-optimizing-option*]...

Specifies optimizing options for determining the most efficient access path, taking into consideration the status of the database.

SQL extension optimizing options can be specified with identifiers (character strings) or numeric values. Hitachi recommends that identifiers be used.

Specification with identifiers:

```
ADD OPTIMIZE LEVEL "identifier"[, "identifier"] ...
```

Specification examples

- Apply the *Application of optimizing mode 2 based on cost* and *Hash join, subquery hash execution* options:

```
ADD OPTIMIZE LEVEL
        "COST_BASE_2","APPLY_HASH_JOIN"
```

- Do not apply any optimizing:

```
ADD OPTIMIZE LEVEL "NONE"
```

Rules

1. At least one identifier must be specified.
2. When multiple identifiers are specified, separate them with the comma (,).
3. For details about the contents that can be specified in an identifier (optimization methods), see *Table 3-26 SQL extension optimizing option specification values (CREATE PROCEDURE)*.
4. If no extension optimizing options are to be applied, specify "NONE" as the identifier.

5. The identifiers are case-sensitive.
6. If the same identifier is specified more than once, it is treated as if it was specified only once; however, where possible, precautions should be taken to avoid specifying a given identifier in duplicate.

Specification with numeric values:

```
ADD OPTIMIZE LEVEL unsigned-integer[, unsigned-integer] ...
```

Specification examples

- Apply the *Application of optimizing mode 2 based on cost* and *Hash join, subquery hash execution* options:

```
ADD OPTIMIZE LEVEL 1,2
```

- Do not apply any optimizing:

```
ADD OPTIMIZE LEVEL 0
```

Rules

1. At least one integer must be specified.
2. When multiple integers are specified, separate them with the comma (,).
3. For details about the contents that can be specified in an unsigned integer (optimization methods), see *Table 3-26 SQL extension optimizing option specification values (CREATE PROCEDURE)*.
4. If no extension optimizing options are to be applied, specify `0` as the integer.
5. If the same integer is specified more than once, it will be treated as a single instance of the integer. However, multiple specifications of the same integer should be avoided.

Relationship to the system definition

If no SQL extension optimizing option values are specified, the values specified in the `pd_additional_optimize_level` operand of the system definition are assumed as the default. For details of the `pd_additional_optimize_level` operand, see the manual *HiRDB Version 9 System Definition*.

Relationship to the client environment definition

The specification of `PDADDITIONALOPTLVL` has no applicability to `CREATE PROCEDURE`.

Relationship with SQL

If SQL optimization is specified in an SQL statement, the SQL optimization

specification takes precedence over SQL optimization options. For SQL optimization specifications, see *2.24 SQL optimization specification*.

SQL extension optimizing option specification values

The following table lists the SQL optimization option specification values. For details about optimization methods, see the *HiRDB Version 9 UAP Development Guide*.

Table 3-26: SQL extension optimizing option specification values (CREATE PROCEDURE)

No.	Optimizing option	Specification values	
		Identifier	Unsigned integer
1	Application of optimizing mode 2 based on cost	"COST_BASE_2"	1
2	Hash join, subquery hash execution	"APPLY_HASH_JOIN"	2
3	Facility for applying join conditions including value expression	"APPLY_JOIN_COND_FOR_VALUE_EXP"	32

Note

Items 2 and 3 take effect when *Application of optimizing mode 2 based on cost* is specified.

[SUBSTR LENGTH *maximum-character-length*]

Specifies the maximum number of bytes for representing a single character.

The value specified for the maximum character length must be in the range from 3 to 6.

This operand is valid only when utf-8 is specified for the character code type in the pdntenv command (pdsetup command for the UNIX edition); it affects the length of the result of the SUBSTR scalar function. For details about SUBSTR, see *2.16.1(20) SUBSTR*.

Relationships to system definition

When SUBSTR LENGTH is omitted, the value specified in the pd_substr_length operand in the system definition is assumed. For details about the pd_substr_length operand, see the manual *HiRDB Version 9 System Definition*.

Relationship to client environmental definition

The specification of PDSUBSTREN has no applicability to CREATE PROCEDURE. For details about PDSUBSTRLEN, see the manual *HiRDB Version 9 UAP Development Guide*.

Relationship to the character code type specified in the `pdntenv` or `pdsetup` command

This operand is valid only when `utf-8` is specified for the character code type.

For all other character code types, only a syntax check is performed and the specification is ignored.

- *SQL-procedure-statements*

Specifies the SQL procedure statements to be executed in the procedure. For details about SQL procedure statements, see *General rules* in *7. Routine Control SQL*.

- `EXTERNAL NAME`
  {*external-Java-routine-name* | *external-C-stored-routine-name*}

external-Java-routine-name

Specifies the external routine, written in Java, that constitutes a Java method. For details about how to specify an external Java routine name, see *1.10.1(1) External Java routine names*.

external-C-stored-routine-name

Specifies a Java method written in the C language as an external routine. For details about how to specify an external Java routine name, see *1.10.2(1) External C stored routine names*.

- `PARAMETER STYLE` *parameter-style*

Specifies items to be passed as parameters when an external routine is called.

`JAVA` *parameter-style*

Parameters to be passed to an external Java procedure, defined using SQL data types, are passed to the Java method using the Java data type that corresponds to the SQL data type.

The `OUT` and `INOUT` parameters of an external Java procedure defined which SQL data types correspond which Java data types. Parameters are passed to a Java method is an array of one element. An array written by the Java method after its completion is treated as the output parameter of the external Java procedure.

`RDSQL` *parameter-style*

If the number of SQL parameters is *n*, the SQL parameter passed to the external C function is as described in the following table.

Table 3-27: Contents of the parameter passed to the C function

Parameter no. (n is number of SQL parameters)	Contents	Description of contents	Data type
1 to *n*	Data part of SQL parameter	Parameters corresponding to the data part of the SQL parameter. The input/output mode of each parameter is the same as that of the corresponding SQL parameter.	Data type corresponding to data type of SQL parameter[#1]
n + 1 to 2*n*	Indicator part of SQL parameter	Parameters corresponding to the indicator part of the SQL parameter. The input/output mode of each parameter is the same as that of the corresponding SQL parameter. If it is an input parameter and the data is a null value, a negative value is set and passed to the C function. If it is an output parameter, it is set to the indicator part in the C function that implemented the external C stored routine, according to the following description. • If the output value is a null value Setting: `-1` • If the output value is not a null value Setting: `0`	`short*`
2*n* + 1	`SQLSTATE`	Output parameter used by the C function that implemented the external C stored routine to set the `SQLSTATE` value. The area is 6 bytes in length. The `SQLSTATE` value is saved in the first 5 bytes. The `SQLSTATE` value is set in the C function that implemented the external C stored routine according to the following description. • If the C function terminates normally Setting: `'00000'` SQL execution result: Terminate normally • If the C function terminates abnormally Setting: Value in the format `'38`*XYY*`'` *X* and *Y* are values in the following range. *X*: `I` to `Z` *Y*: `0` to `9` or `A` to `Z` Examples: `'38I01'`, `'38ZCD'` SQL execution result: SQL error occurs. • If the C function ends in an indeterminate state Setting: Value other than `'00000'` and not in the format `'38`*XYY*`'` SQL execution result: SQL error occurs.	`char*`

Parameter no. (n is number of SQL parameters)	Contents	Description of contents	Data type
2*n* + 2	Routine name	Input parameter indicating the routine name	`struct{ short` *variable-name-1*`; char` *variable-name-2*`[30]; }*`#2
2*n* + 3	ID name	Input parameter indicating the ID name used to identify the procedure	`struct{ short` *variable-name-1*`; char` *variable-name-2*`[30]; }*`#2
2*n* + 4	Message text	Output parameter for setting the detailed reason for an error if an error occurs in the C function that implemented an external C stored routine. If the value of class 38 is set to `SQLSTATE` when an external C stored routine finishes execution, an error message with embedded message text is output. The maximum length of the message text that can be set is 80 bytes.	`struct{ short` *variable-name-1*`; char` *variable-name-2*`[80]; }*`#2

#1

The correspondence between the data type of an SQL parameter that is specified when defining an external C procedure and the data type of the parameter passed to the C function that implements an external C stored routine is the same as the correspondence when defining an external C function. For details, see *CREATE [PUBLIC] FUNCTION (Define function, define public function)* in this chapter.

#2

Sets the character string length (in bytes) in *variable-name-1* and the character string indicating the contents in *variable-name-2*.

Common rules

1. The same identifier used in the following procedures cannot be specified for the routine identifier:
   - Procedure of an owner

- Procedure of a public procedure

2. An input parameter cannot contain an I/O parameter for a routine that is subject to the CALL statement and defined with the OUT or INOUT option; also, it cannot be specified in a FETCH statement, in the INTO clause of a single-row SELECT statement, or in the target of an assignment statement.
3. Output parameters cannot be specified in any location with the exception of the following: input/output parameters for the arguments for the routines, subject to the CALL statement, defined in either OUT or INOUT; the INTO clause of the FETCH statement; the INTO clause of the single-row SELECT statement; the target of an assignment statement; or a value expression in the WRITE LINE statement.
4. The number of procedure parameters must not exceed 30,000 (128 if C is specified in the LANGUAGE clause). However, if Java is specified in the LANGUAGE clause, an error may result at execution time due to specification limitations depending on the JavaVM in use even though the number of specified procedure parameters does not exceed 30,000.
5. The designated name for designating a procedure is the same as [*authorization-identifier*.]*routine-identifier*.
6. The BOOLEAN data type cannot be specified in an input or output parameter.
7. If the name of a CALL statement specified in the procedure body is qualified with *authorization-identifier*, you cannot specify a procedure that has the same authorization identifier and routine identifier as the procedure that you are currently defining.

   If the name of a CALL statement specified in the procedure body is not qualified with *authorization-identifier*, you cannot specify a procedure that has the same routine identifier as the procedure that you are currently defining.
8. When the SQL compile option is specified in an ALTER PROCEDURE or ALTER ROUTINE statement, the length of the SQL statement that is created by incorporating the SQL compile option into the source CREATE PROCEDURE statement of the procedure being re-created must not exceed the maximum permissible length for SQL statements.
9. A WRITE specification cannot be specified in a query specification in an SQL procedure statement.
10. If C is specified in the LANGUAGE clause, no result set can be returned.

Notes

1. The CREATE PROCEDURE statement cannot be specified from an X/Open-compliant UAP running under OLTP.
2. The SQL parameter can assume the null value.

3. When executing multiple SQL statements in a procedure, routine control SQL, such as compound statements must be used.
4. When a procedure is defined, an SQL object is created that codes an access procedure for executing the procedure. When an external Java procedure is defined, no SQL object is created that codes an access procedure for executing the procedure.
5. The data type data guarantee level of the SQL statement in the procedure, the SQL optimization option, the SQL extension optimizing option, and the maximum character length are determined by what is specified when the procedure is being defined or modified, and are not affected by the system definition or client environment variable definition that is in effect when the procedure is called.
6. Stacks in the operating system may overflow if routine calls are repeated extensively or infinitely in a routine.
7. The following SQL statements cannot be executed from an external Java procedure:
   - Control SQL statements other than the `COMMIT`, `LOCK`, and `ROLLBACK` statements
   - Routine control SQL statements
8. A results set cannot be received in an SQL procedure.
9. A result set (`ResultSet`) returned by a method of the `DatabaseMetaData` class acquired within the external Java procedure cannot be returned as a dynamic result set. Use the metadata from the call source connection to the external Java procedure to acquire the information.

(2) CREATE PUBLIC PROCEDURE (Define public procedure)

Privileges

Users who own a schema

These users can define their own public procedures.

Format

```
CREATE PUBLIC procedure-body
```

Operands

- `PUBLIC`

Specifies the procedure that is to be defined as a public procedure.

When a procedure becomes a public procedure, it can be used by multiple users simply by specifying the routine identifier without each user having to define a procedure with the same contents.

- *procedure-body*

For descriptions of operands other than [*authorization-identifier*.]*routine-identifier*, see *(1) CREATE PROCEDURE (Define procedure)* under *CREATE [PUBLIC] PROCEDURE (Define procedure, define public procedure)* in this chapter.

[*authorization-identifier*.]*routine-identifier*

authorization-identifier

An authorization identifier cannot be specified because this is a public procedure.

routine-identifier

Specifies the name of the public procedure to be defined.

Common rules

1. You cannot specify the authorization identifier of [*authorization-identifier*.]*routine-identifier* in the procedure body.
2. The same routine identifier cannot be used in more than one procedure in a system.
3. The name used when identifying a procedure is the same as "PUBLIC".*routine-identifier*.
4. If the name of a CALL statement specified in the procedure body is qualified with PUBLIC, you cannot specify a procedure that has the same authorization identifier and routine identifier as the procedure that you are currently defining.

   If the name of a CALL statement specified in the procedure body is not qualified with *authorization-identifier*, you cannot specify a procedure that has the same routine identifier as the procedure that you are currently defining.
5. If SQL compile options are specified in ALTER PROCEDURE or ALTER ROUTINE, the length of the SQL statement that is created by incorporating the SQL compile options in the source of the procedure being created by the CREATE PUBLIC PROCEDURE must not exceed the maximum allowable length for an SQL statement.
6. For details about other rules, see under *(1) CREATE PROCEDURE (Define procedure) CREATE [PUBLIC] PROCEDURE (Define procedure, define public procedure)* in this chapter.

Notes

1. CREATE PUBLIC PROCEDURE cannot be specified from an X/Open-compliant

UAP running under OLTP.

2. PUBLIC is set in the column that stores the owner of the dictionary table (such as ROUTINE_SCHEMA in the SQL_ROUTINES table). Also, the authorization identifier that was used to define the public procedure is stored in the ROUTINE_CREATOR column of the SQL_ROUTINES table.
3. Use DROP PUBLIC FUNCTION to delete public functions.
4. For other notes, see under *(1) CREATE PROCEDURE (Define procedure) CREATE [PUBLIC] PROCEDURE (Define procedure, define public procedure)* in this chapter.

CREATE SCHEMA (Define schema)

Function

CREATE SCHEMA defines a schema.

Privileges

Users with the schema definition privilege

These users can define schemas for themselves only.

Users with the DBA privilege

Can define schemas of other users who have the CONNECT or DBA privilege.

Format

```
CREATE SCHEMA schema-name-clause
schema-name-clause::=[AUTHORIZATION authorization-identifier]
```

Operands

- [AUTHORIZATION *authorization-identifier*]

Specifies the authorization identifier of the user for whom a schema is to be defined. The default is the executing user's own authorization identifier.

Notes

1. Once a schema has been defined, tables, indexes, abstract data types, index types, functions, procedures, triggers, and access privileges can be defined.
2. Only one schema can be defined for each user.
3. The authorization identifier specified as the owner of a table or an index must be the same authorization identifier specified in the schema definition for that user.
4. A user who receives a schema defined for that user by a user with the DBA privilege is granted the schema definition privilege.
5. The CREATE SCHEMA statement cannot be specified from an X/Open-compliant UAP running under OLTP.

Example

Define a schema for a user (USER1):

```
CREATE SCHEMA
  AUTHORIZATION USER1
```

CREATE SEQUENCE (Define sequence generator)

Function

CREATE SEQUENCE defines a sequence generator.

Privileges

Users who own a schema

These users can define a sequence generator in a private-use RDAREA with RDAREA usage privileges or in a public RDAREA.

Format

```
CREATE SEQUENCE [authorization-identifier.]sequence-generator-identifier
                [FOR PUBLIC USAGE]
                [sequence-generator-option-column]
                [IN sequence-generator-storage-RDAREA-name]

sequence-generator-option-column ::=sequence-generator-option [sequence-generator-option]
sequence-generator-option ::={sequence-generator-option-data-type-option
                |common-sequence-generator-option-column}
common-sequence-generator-option-column ::=common-sequence-generator-option
                [common-sequence-generator-option] . . .
common-sequence-generator-option ::={sequence-generator-start-option
                |common-sequence-generator-option}
basic-sequence-generator-option ::={sequence-generator-increment-option
                |sequence-generator-maxvalue-option
                |sequence-generator-minvalue-option
                |sequence-generator-cycle-option
                |sequence-generator-log-output-interval-option

sequence-generator-data-type-option ::=AS data-type
sequence-generator-start-option ::=START WITH start-value
sequence-generator-increment-option ::=INCREMENT BY increment
sequence-generator-maxvalue-option ::={MAXVALUE maxvalue|NO MAXVALUE}
sequence-generator-minvalue-option ::={MINVALUE minvalue|NO MINVALUE}
sequence-generator-cycle-option ::={CYCLE|NO CYCLE}
sequence-generator-log-output-interval-option ::=LOG INTERVAL output-interval
```

Operands

- [*authorization-identifier*.]*sequence-generator-identifier*

authorization-identifier

Specifies the authorization identifier of the user who is the owner of the sequence generator being defined.

If this is omitted, the authorization identifier of the executing user is assumed.

sequence-generator-identifier

Specifies the identifier of the sequence generator.

- FOR PUBLIC USAGE

Specifies that the sequence generator being generated is to be usable by all users.

If this is omitted, the sequence generator can only be used by the owner.

- IN *sequence-generator-storage-RDAREA-name*

Specifies the name of the user RDAREA used to store the sequence generator.

An RDAREA using the inner replica facility cannot be specified for the RDAREA name.

If the RDAREA name is omitted, the RDAREA used to store the sequence generator is determined according to the following priority:

1. Public RDAREA with the smallest total number of defined tables and sequence generators from among the RDAREAs not using the inner replica facility.
2. If there is more than one RDAREAs meeting condition 1, the first RDAREA found by HiRDB.
3. An error occurs if an RDAREA cannot be determined using the above criteria.

- *sequence-generator-option-column*

An option cannot be repeatedly specified in *sequence-generator-option-column*.

- *sequence-generator-data-type-option* ::=AS *data-type*

Specifies the data type of the value generated by a sequence generator.

INTEGER, SMALLINT, or DECIMAL of decimal scaling position 0 can be specified in the data type.

If this option is omitted, INTEGER is assumed.

- *sequence-generator-start-option* ::=START WITH *start-value*

Specifies an integer value to be used as the starting value for the sequence generator.

When the first NEXT VALUE expression is executed after a sequence generator is defined, the sequence generator returns the starting value.

Specify an integer value that is between the minimum and maximum values specified for the sequence generator in the sequence generator start option.

If this option is omitted, the minimum value specified for the sequence generator is assumed for an ascending sequence generator, and the maximum value specified for the sequence generator is assumed for a descending sequence generator.

For details about ascending and descending sequence generators, see *sequence-generator-increment-option*.

- *sequence-generator-increment-option* ::=INCREMENT BY *increment*

Specifies the value to be added when updating the value generated by the sequence generator (current value).

If the increment is positive, this is an ascending sequence generator, and if the increment is negative, this is a descending sequence generator.

Specify an integer other than 0 that satisfies the following conditions in *sequence-generator-increment-option*:

- *Increment* ≤ *maximum value of sequence generator data type*
- *Increment* ≥ *minimum value of sequence generator data type*

If this option is omitted, 1 is assumed.

- *sequence-generator-increment-option*::={MAXVALUE *maxvalue*|NO MAXVALUE}

Specifies the maximum value that can be generated by a sequence generator (sequence number).

When specifying the maximum value in *sequence-generator-maxvalue-option*, the following conditions must be satisfied:

- Sequence generator maximum value ≤ maximum value of sequence generator data type
- Sequence generator maximum value > sequence generator minimum value
- Sequence generator maximum value ≥ sequence generator start value

If this option is omitted or NO MAXVALUE is specified, the maximum value of the sequence generator data type is assumed.

- *sequence-generator-minvalue-option* ::={MINVALUE *minvalue*|NO MINVALUE}

Specifies the minimum value that can be generated by the sequence number.

When specifying the minimum value in *sequence-generator-minvalue-option*, the following conditions must be satisfied:

- *Sequence generator minimum value* ≥ *minimum value of sequence generator data type*
- *Sequence generator minimum value* < *sequence generator maximum value*
- *Sequence generator minimum value* ≤ *sequence generator start value*

If this option is omitted or NO MINVALUE is specified, the minimum value of the

sequence generator data type is assumed.

■ *sequence-generator-cycle-option* ::={CYCLE|NO CYCLE}

Specifies whether to cycle the sequence numbers when the next value that would be generated by the sequence generator exceeds the maximum or minimum value. The operation when CYCLE is specified in *sequence-generator-cycle-option* are as follows:

- For ascending sequence generator

  If the sequence generator maximum value is exceeded, the sequence generator returns the sequence generator minimum value as the next sequence number.

- For descending sequence generator

  If the sequence generator minimum value is exceeded, the sequence generator returns the sequence generator maximum value as the next sequence number.

Note that once the sequence generator is cycled, duplicate numbers will be generated.

If this option is omitted or NO CYCLE is specified, the operation is as follows:

- For ascending sequence generator

  An error occurs when the sequence generator maximum value is exceeded.

- For descending sequence generator

  An error occurs when the sequence generator minimum value is exceeded.

■ *sequence-generator-log-output-interval-option* ::={LOG INTERVAL *output-interval*}

Specifies the interval for outputting sequence generator logs.

Specifying a large output interval reduces the number of logs output, thereby increasing the processing performance. However, if a system failure occurs and 2 or more is specified for the output interval, there may be missing numbers in the sequence of numbers for up to the number of output intervals specified here. No missing numbers will occur if 1 is specified for the output interval or if this option is omitted.

The output interval has a range of 1 to 2^{31}-1 and must satisfy the following condition:

Output interval ≤ *absolute value of* (↓(*sequence generator maximum value* - *sequence generator minimum value*) ÷ *increment*↓)

If this option is omitted, 1 is assumed.

For details about the sequence generator log output interval, see *HiRDB Version 9 UAP Development Guide*.

Common rules

1. Up to 500 sequence generators can be defined together with a table in an

RDAREA.

2. The rules for generating sequence numbers are as follows:

- If the specified sequence generator has never been used, the start value of the sequence generator is returned, and the current value is set to the start value.
- If the specified sequence generator has been used one or more times, the current value is returned, and then the current value is incremented.

Notes

1. `CREATE SEQUENCE` cannot be specified from an X/Open-compliant UAP running under OLTP.

Examples

A sequence generator (`SEQ1`) is defined using the following conditions:

- Stored in RDAREA `RDA1`
- `INTEGER` type
- Start value of `1`
- Increment of `1`
- Maximum value of `999`
- Minimum value of `1`
- No cycle
- Log output interval of `20`

```
CREATE SEQUENCE USER1.SEQ1
AS INTEGER
START WITH 1
INCREMENT BY 1
MAXVALUE 999
MINVALUE 1
NO CYCLE
LOG INTERVAL 20
IN RDA1
```

CREATE TABLE (Define table)

Function

`CREATE TABLE` defines a table.

Privileges

Owner of a schema

The owner of a schema can define tables in either private user RDAREAs or public user RDAREAs for which the owner has the RDAREA usage privilege.

Format

The numbers in the left column below correspond to the section numbers on the following pages where the operands are explained.

No.	Format
1	`CREATE [[SHARE] FIX] TABLE`
2	`[`*authorization-identifier*`.]`*table-identifier*
3	`(`*table-element*`[,` *table-element*`]...)`

4	`[{IN {`*table-storage-RDAREA-name*
	`|(`*table-storage-RDAREA-name*`)`
	`|([(`*table-storage-RDAREA-name*`)` *storage-condition*`,]...`
	`(`*table-storage-RDAREA-name*`)` `[`*storage-condition*`])}`
	`|PARTITIONED BY` *column-name*
	`IN([(`*table-storage-RDAREA-name*`)` *boundary-value*`,]...`
	`(`*table-storage-RDAREA-name*`)` *boundary-value*`,`
	`(`*table-storage-RDAREA-name*`))`
	`|PARTITIONED BY MULTIDIM`
	`(`*first-dimension-column-name first-dimension-boundary-value-list*
	`,` {*second-dimension-column-name second-dimension-boundary-value-list*
	`| [FIX] HASH [`*hash-function-name*`]`
	`BY` *second-dimension-column-name* `[,`*second-dimension-column-name*`]...})`
	`IN` *matrix-partitioned-table-storage-RDAREA-specification*
	`|[FIX] HASH [`*hash-function-name*`] BY` *column-name*
	`[,` *column-name*`]...`
	`IN(`*table-storage-RDAREA-name*`,` *table-storage-RDAREA-name*`,...)}]`

5	`[`*table-option*`]...`
6	`[`*table-restriction-definition*`]...`
27	`[WITH PROGRAM]`

■ Details of format

No.	Format
3	*table-element* ::= {*column-definition*|*table-restriction-definition*}
7	*column-definition* ::= *column-name data-type* [ARRAY [*maximum-number-of-elements*]] [NO SPLIT] [{*column-data-suppression-specification* | [*column-recovery-restriction*] {IN {*LOB-column-storage-RDAREA-name* | (*LOB-column-storage-RDAREA-name*) | ((*LOB-column-storage-RDAREA-name*) [,(*LOB-column-storage-RDAREA-name*)]...) | *matrix-partitioned-LOB-column-storage-RDAREA-specification*} | *abstract-data-type-LOB-column-storage-RDAREA-specification*}}] [*plug-in-specification*] [DEFAULT *clause*] [*column-restriction*]... [*updatable-column-attribute*]
8	*column-data-suppression-specification* ::= SUPPRESS
9	*column-recovery-restriction* ::= RECOVERY [{ALL|PARTIAL|NO}]
10	*abstract-data-type-LOB-column-storage-RDAREA-specification* ::= ALLOCATE (*attribute-name*[..*attribute-name*]... IN {*LOB-attribute-storage-RDAREA-name* | (*LOB-attribute-storage-RDAREA-name*) | ((*LOB-attribute-storage-RDAREA-name*) [,(*LOB-attribute-storage-RDAREA-name*)]) | *matrix-partitioned-LOB-attribute-storage-RDAREA-specification*} [,*attribute-name*[..*attribute-name*]... IN {*LOB-attribute-storage-RDAREA-name* | (*LOB-attribute-storage-RDAREA-name*) | ((*LOB-attribute-storage-RDAREA-name*) [,(*LOB-attribute-storage-RDAREA-name*)]) | *matrix-partitioned-LOB-attribute-storage-RDAREA-specification*}]...)
11	*plug-in-specification* ::= PLUGIN *plug-in-option*
12	DEFAULT *clause*::=DEFAULT [*predefined-value*] *predefined-value*::={*literal*|USER|CURRENT_DATE|CURRENT DATE |CURRENT_TIME|CURRENT TIME |CURRENT_TIMESTAMP[(*fractional-second-precision*)] [USING BES] |CURRENT TIMESTAMP[(*fractional-second-precision*)] [USING BES] |NULL}

No.	Format
13	*column-restriction* ::= {*NOT-NULL-constraint-specification* |*single-column-uniqueness-constraint-definition* [*index-option* [*index-option*]] | {*single-column-check-constraint-definition* [*constraint-name-definition*] | [*constraint-name-definition*] *single-column-check-constraint-definition*}#3 | {*single-column-referential-constraint-definition* [*constraint-name-definition*] | [*constraint-name-definition*] *single-column-referential-constraint-definition*}}#3
14	*updatable-column-attribute*::= UPDATE [ONLY FROM NULL]
15	*NOT-NULL-constraint-specification* ::= {[NULL|NOT NULL[WITH DEFAULT[SYSTEM GENERATED]]]#2 |[[NOT NULL] WITH DEFAULT[SYSTEM GENERATED]]}#1
16	*single-column-uniqueness-constraint-definition* ::= {[{UNIQUE|PRIMARY}] CLUSTER KEY [{ASC|DESC}] [IN {*index-storage-RDAREA-name* |(*index-storage-RDAREA-name*) |((*index-storage-RDAREA-name*) [,(*index-storage-RDAREA-name*)]...)}] |PRIMARY KEY [{ASC|DESC}] [IN {*index-storage-RDAREA-name* |(*index-storage-RDAREA-name*) |((*index-storage-RDAREA-name*) [,(*index-storage-RDAREA-name*)]...)}]}
17	*index-option* ::= {PCTFREE = *percentage-of-free-area* |UNBALANCED SPLIT}
18	*single-column-check-constraint-definition*::= CHECK (*search-condition*)
19	*single-column-referential-constraint-definition*::= *reference-specification*
6	*table-constraint-definition*::= {*multicolumn-uniqueness-constraint-definition* [*index-option* [*index-option*]] | {*multicolumn-check-constraint-definition* [*constraint-name-definition*] | [*constraint-name-definition*] *multicolumn-check-constraint-definition*}#3 | {*multicolumn-referential-constraint-definition* [*constraint-name-definition*] | [*constraint-name-definition*] *multicolumn-referential-constraint-definition*}}#3

No.	Format
20	*multicolumn-uniqueness-constraint-definition* ::= {[{UNIQUE|PRIMARY}] CLUSTER KEY (*column-name* [{ASC|DESC}][,*column-name* [{ASC|DESC}]]...) [IN {*index-storage-RDAREA-name* |(*index-storage-RDAREA-name*) |((*index-storage-RDAREA-name*) [,(*index-storage-RDAREA-name*)]...) |*matrix-partitioned-index-storage-RDAREA-specification*}] |PRIMARY KEY (*column-name* [{ASC|DESC}] [,*column-name* [{ASC|DESC}]]...) [IN {*index-storage-RDAREA-name* |(*index-storage-RDAREA-name*) |((*index-storage-RDAREA-name*) [,(*index-storage-RDAREA-name*)]...) |*matrix-partitioned-index-storage-RDAREA-specification*}]}
21	*multicolumn-check-constraint-definition*::= CHECK (*search-condition*)
22	*multicolumn-referential-constraint-definition*::= FOREIGN KEY (*column-name*[, *column-name*]...) *reference-specification*
23	*storage-condition* ::= *column-name* {=| < > | ^=|!=| <| < =| > | > =} {*literal*| (*literal*[,*literal*]...)
-	*first-dimension-column-name*::=*column-name* *second-dimension-column-name*::=*column-name* *first-dimension-boundary-value-list*::=*boundary-value-list* *second-dimension-boundary-value-list*::=*boundary-value-list* *boundary-value-list*::=((*boundary-value*)[,(*boundary-value*)]...) *matrix-partitioned-table-storage-RDAREA-specification*::=*two-dimensional-storage-RDAREA-specification* *matrix-partitioned-index-storage-RDAREA-specification*::=*two-dimensional-storage-RDAREA-specification* *matrix-partitioned-LOB-column-storage-RDAREA-specification*::=*two-dimensional-storage-RDAREA-specification* *matrix-partitioned-LOB-attribute-storage-RDAREA-specification*::=*two-dimensional-storage-RDAREA-specification* *two-dimensional-storage-RDAREA-specification*::=(*matrix-partitioning-RDAREA-list* [,*matrix-partitioning-RDAREA-list*]...) *matrix-partitioning-RDAREA-list*::=(*RDAREA-name*[,*RDAREA-name*]...)
24	*hash-function-name* ::= {HASH1|HASH2|HASH3|HASH4|HASH5|HASH6|HASH0 |HASHA|HASB|HASHC|HASHD|HASHE HASHF}

No.	Format
25	*reference-specification*::= REFERENCES *referenced-table* [*referential-constraint-operation-specification*] *referenced-table*::= *table-name* *referential-constraint-operation-specification*::= {*delete-operation* [*update-operation*] | *update-operation* [*delete-operation*]} *delete-operation*::= ON DELETE *reference-operation* *update-operation*::= ON UPDATE *reference-operation* *reference-operation*::= {CASCADE | RESTRICT}
26	*constraint-name-definition*::= CONSTRAINT *constraint-name*
5	*table-option* ::= {PCTFREE = {*percentage-of-free-area* |([*percentage-of-free-area*],*percentage-of-free-pages-in-segment*)} |{LOCK ROW|LOCK PAGE} |SUPPRESS [DEC [IMAL]] |WITHOUT ROLLBACK |INDEX LOCK {NONE|PAGE} |SEGMENT REUSE {[*number-of-segments*[{K|M}]]|NO} |INSERT ONLY [WHILE {*date-interval-data*|*labeled-duration*} BY *column-name*]}

Legend:

--: See applicable items.

#1: For a column in a FIX table or a column that is a member of a cluster key or primary key

#2: For a column other than the above

#3: The position of a restriction definition is determined by the specification value in the system common definition, pd_constraint_name operand, or that in the client environment variable PDCNSTRNTNAME. For details, see *6) Table-restriction-definition*.

Operands

1) [SHARE]FIX

Specifies that the table is to have a fixed row length. To store table data in a shared RDAREA and make it a shared table, specify SHARE.

However, on a HiRDB/Single Server that has no shared RDAREAs, a shared table can be defined by specifying SHARE to maintain SQL compatibility with HiRDB/Parallel Server. In this case, table data is stored in the regular user RDAREA.

When the FIX option is specified, database operations that change the row length cannot be performed; however, the efficiency of row storage is enhanced. When used in conjunction with the row-unit interface, the FIX option can improve access efficiency for a table that contains a large number of columns.

The following rules apply to the FIX option:

1. The FIX option is incompatible with the following data types:
   - VARCHAR
   - NVARCHAR
   - MVARCHAR
   - BLOB
   - Abstract data type
2. A repetition column for which FIX has been specified cannot be specified.
3. The FIX option can be specified only if the row length does not exceed the following value:

   ↓Page length of RDAREA in which rows are stored |1000 ↓ x1000
4. The NOT NULL option is assumed for all columns of a table for which the FIX option is defined.
5. If both SHARE and FIX are specified, table data cannot be saved on a split basis in multiple RDAREAs.

2) [*authorization-identifier* .]*table-identifier*

authorization-identifier

Specifies the authorization identifier of the owner of the base table being defined.

table-identifier

Specifies a name for the base table being defined. Each table identifier must be unique among the tables of the specified owner.

3) *table-element* : : = {*column-definition* ||*table-restriction-definition*}

column-definition

Defines a column (column name, data type, etc.) that is to compose the table. The NOT NULL constraint, uniqueness constraint, check constraint, and referential constraint items can be specified for each column.

4) {IN {*table-storage-RDAREA-name*

|(*table-storage-RDAREA-name*)

|([(*table-storage-RDAREA-name*) *storage-condition* ,] ...

(*table-storage-RDAREA-name*) [*storage-condition*])}

|PARTITIONED BY *column-name*

```
    IN ([(table-storage-RDAREA-name) boundary-value , ] ...
        (table-storage-RDAREA-name) boundary-value ,
          (table-storage-RDAREA-name))
  |PARTITIONED BY MULTIDIM
        (first-dimension-column-name  first-dimension-boundary-value-list
         {second-dimension-column-name second-dimension-boundary-value-list
          | [FIX] HASH [hash-function-name]
            BY second-dimension-column-name[ ,
second-dimension-column-name]...})
    IN  matrix-partitioned-table-storage-RDAREA-specification
  |[FIX] HASH  [hash-function-name] BY column-name[ ,  column-name] ...
    IN (table-storage-RDAREA-name ,  table-storage-RDAREA-name , ...)}
```

IN

Specifies the RDAREAs in which table rows are to be stored.

table-storage-RDAREA-name

Specifies the name of a user RDAREA in which rows of the table are to be stored.

However, HiRDB/Parallel Server is subject to the following restrictions. If SHARE is specified, the name of a shared RDAREA must be specified. Conversely, if SHARE is not specified, the name of a shared RDAREA cannot be specified.

If an RDAREA name is omitted, the RDAREA for storing the table is determined as described below. In addition, if SHARE is specified on HiRDB/Parallel Server, a shared RDAREA is identified as a candidate storage area. Conversely, if SHARE is not specified, a shared RDAREA is not identified as a candidate storage area.

A user RDAREA that stores rebalancing tables cannot be specified.

Table 3-28: How a table storage RDAREA is determined by default

Primary key or cluster key specified	Index storage RDAREA specified#	Table storage RDAREA determination method
No	--	A table storage RDAREA is determined according to the following priority: 1. Public RDAREA with the smallest total number of defined tables and sequence generators from among the RDAREAs not using the inner replica facility 2. If there is more than one RDAREA meeting condition 1, the first RDAREA found by HiRDB 3. If the RDAREA cannot be determined by condition 1 or 2, the public RDAREA with the smallest total number of defined tables and sequence generators from among the RDAREAs (original RDAREAs) using the inner replica facility 4. If there is more than one RDAREA meeting condition 3, the first RDAREA found by HiRDB
Yes	No	A table storage RDAREA is determined according to the following priority: 1. Public RDAREA with the smallest total number of defined tables and sequence generators from among the RDAREAs not using the inner replica facility 2. If there is more than one RDAREA meeting condition 1, the public RDAREA having the fewest index definitions 3. If there is more than one RDAREA meeting conditions 1 and 2, the first RDAREA found by HiRDB 4. If the RDAREA cannot be determined by condition 1, 2, or 3, the public RDAREA with the smallest total number of defined tables and sequence generators from among the RDAREAs (original RDAREAs) using the inner replica facility 5. If there is more than one RDAREA meeting condition 4, the public RDAREA having the fewest index definitions 6. If there is more than one RDAREA meeting conditions 4 and 5, the first RDAREA found by HiRDB
	RDAREA not subject to inner replica facility	1. Public RDAREA with the smallest total number of defined tables and sequence generators from among the RDAREAs not using the inner replica facility 2. If there is more than one RDAREA meeting condition 1, the first RDAREA found by HiRDB
	RDAREA (original RDAREA) subject to inner replica facility	1. Of the RDAREAs subject to the inner replica facility, the original and public RDAREAs having equal numbers of replica RDAREAs 2. If there is more than one RDAREA meeting condition 1, the first RDAREA found by HiRDB

Legend:

--: Not applicable

#: On HiRDB/Parallel Server, if an RDAREA for index is specified, the RDAREA residing on the same server as the RDAREA for index is subject to table-storage RDAREA selection.

storage-condition

Specifies conditions for storing the table in multiple RDAREAs on a split basis (row-partitioning of the table). If a storage condition is specified, single-column partitioning will be used in partitioning the table.

Multiple literals can be specified as the storage condition for a column such as the following:

- Column whose values cannot be grouped for specification of ranges (e.g., store numbers, organization codes)
- Column consisting of noncontiguous values, such as character strings

The following rules apply to *storage-condition*:

1. When multiple storage conditions are specified, the same column name must be specified in all of them.
2. When multiple storage conditions are specified, HiRDB evaluates them in the order in which they are specified; rows are stored in the RDAREA associated with the first storage condition that tests `TRUE`. Rows for which no condition is `TRUE` are stored in an RDAREA for which no storage conditions are specified. If there is no RDAREA for which no storage conditions are specified, rows cannot be stored.
3. If there is an RDAREA in which no rows are stored as a result of HiRDB's evaluation of storage conditions, the table is not defined.
4. For data insertion, data cannot be inserted if there is no RDAREA corresponding to the data.
5. For data updating, you cannot update column values that are specified in storage conditions.
6. Each storage condition is associated with one RDAREA. A maximum of 1,024 RDAREAs can be specified. The same RDAREA cannot be specified for more than one storage condition.

`PARTITIONED BY` *column-name*

Specifies that the table is to be partitioned by boundary values for storage in multiple RDAREAs. The maximum number of RDAREAs into which a table can be partitioned is 1,024, exclusive of duplications.

column-name

Specifies the name of the column for which boundary values are to be specified. The column data types that can be specified are the data types on which storage condition comparison operations can be performed. When `PARTITIONED BY` *column-name* is specified, the resulting partitioning will be single-column partitioning.

The following rules apply to `BY` *column-name*:

1. Values in the column that is specified in column-name cannot be updated.
2. The column specified in column-name should be a `NOT NULL` column (`NOT NULL` constrained, `FIX` specification, cluster key, or primary key).
3. If a cluster key is specified for any column, boundary values cannot be specified for any other columns.
4. If multiple columns constitute a cluster key, boundary values cannot be specified for any of those columns except the first column.
5. A repetition column cannot be specified.

boundary-value

Specifies a boundary value for determining where the table's rows are to be partitioned. Specify a literal as a *boundary-value*.

The following rules apply to boundary values:

1. Any of the following items cannot be specified in *literal*:
   - Character string literals, national character string literals, or mixed character string literals with a length of 0
   - Character string literals with a length of 256 bytes or greater, national character string literals with a length of 128 characters or greater, or mixed or character string literals with a length of 256 bytes or greater
   - Hexadecimal character string literals
2. Boundary values must be specified in ascending order; they must all be distinct values.
3. A maximum value should not be specified in the boundary value that is specified last.
4. Specify table storage RDAREAs and boundary values alternately so that the specification begins and ends with a table storage RDAREA.
5. The maximum number of table storage RDAREAs is 3,000.
6. The same table storage RDAREA can be specified multiple times, provided that it is not specified twice in succession.
7. The first RDAREA for which a boundary value is specified will store the

rows whose value is less than or equal to the specified boundary value. In each subsequent RDAREA (except for the last one), rows are stored that have a value greater than the previously specified boundary value and less than or equal to the next boundary value that is specified. The last RDAREA that is specified stores rows with a value that is greater than any of the previously specified boundary values.

PARTITIONED BY MULTIDIM

Specify this operand when partitioning table data into a column (first dimension partitioning column) and partitioning boundary value data into another column (second dimension partitioning column). Partitioning by this type of specification is called *matrix partitioning*, and tables that are partitioned in this manner are referred to as *matrix-partitioned tables*.

Defining a matrix-partitioned table requires HiRDB Advanced High Availability.

first-dimension-column-name::=*column-name*

Specifies a first-dimension partitioned column name.

The following rules apply to first-dimension partitioned column names:

1. The specified column should be made NOT NULL. The following methods can be used to make a column NOT NULL:
   - Defining the column as a FIX table
   - Specifying NOT NULL in the column definition
   - Defining a cluster key or the primary key
2. The values in the specified column cannot be updated.
3. A repetition column cannot be specified in *column-name*.
4. For the data type of the column specified in *column-name*, see the data types of columns that can be compared under storage conditions.

first-dimension-boundary-value-list::=*boundary-value-list*

boundary-value-list::=((*boundary-value*)[,(*boundary-value*)]...)

Specifies the column boundary value list that was specified in *first-dimension-column-name*.

In *boundary-value*, specify the boundary value to be used for partitioning the rows in a table. The following rules apply to boundary values:

1. Specify a literal in *boundary-value*.
2. Specify boundary values in ascending order.
3. Any of the following items cannot be specified in *boundary-value*:

- Character string literals, national character string literals, or mixed character string literals with a length of 0
- Character string literals and mixed or character string literals with a length of 256 bytes or greater
- National character string literals with a length of 128 characters or greater
- Hexadecimal character string literals

The maximum value cannot be specified in the boundary value that is specified last.

second-dimension-column-name::=*column-name*

Specifies a second-dimension partitioned column name.

For details about rules, see the explanation of the first-dimension column name.

second-dimension-boundary-value-list::=*boundary-value-list*

boundary-value-list::=(*boundary-value*)[,(*boundary-value*)]...

Specify a list of column boundary values that were specified in *second-dimension-column-name*.

For rules on boundary values, see the explanation of boundary values for the first-dimension boundary-value list.

[FIX] HASH [*hash-function-name*]

BY *second-dimension-column-name*[, *second-dimension-column-name*]...

second-dimension-column-name::= *column-name*

Specifies the names of the hash function and second-dimension column to be used.

For details about specification methods and rules, see the [FIX] HASH item.

matrix-partitioned-table-storage-RDAREA-specification::=*two-dimensional-storage-RDAREA-specification*

two-dimensional-storage-RDAREA-specification::=

(*matrix-partitioning-RDAREA-list*

[, *matrix-partitioning-RDAREA-list*]...)

matrix-partitioning-RDAREA-list::=

(*RDAREA-name*[, *RDAREA-name*]...)

When defining a matrix-partitioned table, a cluster key, the primary key, a

BLOB column, or an abstract data type of the BLOB attribute for a matrix-partitioned table, specifies the RDAREA that stores them.

The following rules apply to the RDAREA for the storage of matrix-partitioned tables:

1. For rules on RDAREA names, see the explanation of RDAREAs under each operand.
2. The number of RDAREAs specified per matrix-partitioned RDAREA list is *the number of boundary values specified in second-dimension boundary value list + 1*. If a hash function is used in *second-dimension-partitioning-column*, the number of RDAREAs is a user-specified number.
3. The specifiable number of matrix-partitioned RDAREA lists is *the number of boundary values specified in first-dimension boundary value list + 1*.
4. When defining a cluster key, the primary key, a BLOB column, and an abstract data type having the BLOB attribute for a matrix-partitioned table, specify RDAREAs in correspondence with the RDAREAs for storing matrix-partitioned tables.
5. The maximum number of RDAREAs that can be specified, exclusive of duplicates, is 1,024.
6. The total number of specifiable RDAREAs is 3,000.
7. Cluster keys cannot be specified in a single column.
8. If a cluster key extends over two or more columns, the columns specified for partitioning must be included in the same order from the beginning.

[FIX] HASH

Specifies that the table is to be partitioned by means of a hash function for storage in multiple RDAREAs. The same table storage RDAREA name can be specified more than once. This operand is also specified when using a hash function in *second-dimension-partitioning-column*.

When a table is to be partitioned by flexible hash partitioning, specify HASH only; for FIX hash partitioning, specify FIX HASH.

[*hash-function-name*]

Specifies the hash function to be used for hash partitioning of the table.

If a hash function name is omitted, the following hash function is assumed, depending on the partitioning method involved:

- If the table is partitioned by using a hash function, HASH1 is assumed.
- If a hash function is specified in a second-dimension partitioning column in the matrix-partitioning table, HASH6 is assumed.

BY *column-name*[, *column-name*] ...

Specifies the names of the columns to be operated on by the hash function. The column data types that can be specified are data types that are eligible for storage condition comparison operations.

Specifying one column name only results in single-column partitioning; specifying multiple columns results in multicolumn partitioning.

For a single-column partitioned table:

- If there is a column for which a cluster key is specified, no other columns can be specified.
- If the cluster key includes more than one column, no columns other than the first cluster key column can be specified.

For a multicolumn partitioned table:

- A cluster key cannot be specified for a single column.
- The cluster key columns must include all columns that are specified for partitioning, beginning with the first column and in the same sequence.

Partitioning a second-dimension partitioning column in a matrix-partitioning table into single columns:

- A cluster key cannot be specified for a single column.
- If a cluster key is multiple columns, a column other than the second column from the beginning cannot be specified as a column name.

Partitioning a second-dimension partitioning column in a matrix-partitioning table into multiple columns:

- A cluster key cannot be specified for a single column.
- If a cluster key is multiple columns, the second and subsequent columns from the beginning must be specified in the same sequence (the second and subsequent columns from the beginning of the cluster key constituent columns need not all be included).

If there is a column for which the primary key is specified, whether or not the primary key can be defined depends on how the table is partitioned. For details about primary key definability (UNIQUE specifiability) see *Table 3-22 Specifiability of UNIQUE in conjunction with row-partitioning of a table*.

The following rules apply to BY *column-name*:

1. A maximum of 16 columns can be specified for multicolumn partitioning. If a hash function is specified in second-dimension partitioning columns of a matrix-partitioned table, the maximum number of columns that can be specified is 15.
2. The same column name cannot be specified more than once for multicolumn partitioning.
3. For multicolumn partitioning, specify a combination of columns that have values that are not mutually dependent.
4. A column specified in *column-name* should be a NOT NULL column (NOT NULL constrained, FIX specification, cluster key, or primary key).
5. If flexible hash partitioning is specified for a flexible hash partitioned table or for second-dimension partitioning columns of a matrix-partitioned table, a cluster key with the UNIQUE specification, the primary key, or an index with the UNIQUE specification cannot be specified.
6. For FIX hash-partitioning, the values in a column specified in *column-name* cannot be updated during data updating.
7. A repetition column cannot be specified.
8. If a column name is specified in a second-dimension partitioning column of a matrix-partitioned table, the column name specified in the first-dimension column name cannot be specified.

5) *table-option* ::=

{PCTFREE = {*percentage-of-free-area*

|([*percentage-of-free-area*],*percentage-of-free-pages-in-segment*)}

|{LOCK ROW|LOCK PAGE}

|SUPPRESS [DEC [IMAL]]

|WITHOUT ROLLBACK

|INDEX LOCK {NONE|PAGE}

|SEGMENT REUSE {[*number-of-segments*[{K|M}]]|NO}

|INSERT ONLY [WHILE {*date-interval-data*|*labeled-duration*} BY *column-name*]}

{PCTFREE = {*percentage-of-free-area*

|([*percentage-of-free-area*], *percentage-of-free-pages-in-segment*)}

The same option cannot be specified more than once for the same table.

The specification of parentheses is critical in the meaning of the PCTFREE specification:

`PCTFREE = 30` specifies that 30% of the area is to be left unused (free area).

`PCTFREE = (,30)` specifies that 30% of the pages in each segment are to be left unused (free pages).

percentage-of-free-area

Specifies, in the range 0 to 99, a percentage of free area to be allocated in the database when the table is initialized. The default is 30 (%).

The percentage of unused space is applied when the database load utility or the database reorganization utility is executed. During other addition or update operations, such as the execution of the `INSERT` or `UPDATE` statements, the value `PCTFREE = (0,0)` is assumed.

The following rules apply to specifying *percentage-of-free-area*:

1. When a cluster key is defined, `PCTFREE` should be specified for the following purposes:

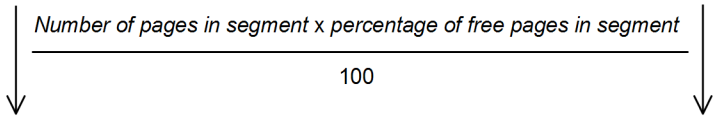
   - To create free space in the table so that data to be inserted after data initialization or reorganization can be clustered.
   - If the table contains variable-length data, to create free space in the table so that data that is updated after data initialization or reorganization and that results in increasing a row length can be stored as close together as possible.

2. `PCTFREE = (0,0)` should be specified for a fixed-length table that does not have a cluster key.
3. When `PCTFREE` is specified for a fixed-length table that does not have a cluster key and updating increases the record length within a page, the specified free area in the page will be used.
4. A high free area percentage should be specified for a table containing variable-length data and for which a cluster key is defined if rows will be added frequently after the table has been created or if frequent updating will result in increased row lengths.

percentage-of-free-pages-in-segment

Specifies, in the range 0 to 50, a percentage of free pages to be allocated in each segment when the table is created. The default is 10 (%).

The following formula can be used to calculate the percentage of free pages in a segment that should be specified:



The specified percentage of free pages in a segment is applied when the database load utility or the database reorganization utility is executed.

If the addition of rows occurs frequently in a table for which a cluster key is defined, or updates that increase the row length occur frequently, and if the data from the increase cannot fit in the unused area on the page, this value should be specified.

{LOCK ROW |LOCK PAGE}

Specifies the minimum unit of locked resources for retrieval and updating operations. Specification of LOCK ROW means that the row is the minimum locked resources unit; specification of LOCK PAGE means that the page is the minimum locked resources unit. The default is LOCK ROW.

LOCK ROW

Specifies that the row is the minimum unit for resources locking.

LOCK PAGE

Specifies that the page is the minimum unit for resources locking.

SUPPRESS [DEC[IMAL]]

For a non-FIX table, specifies that when data is stored in the table, part of the data can be omitted. The SUPPRESS option is useful for reducing the database's storage space requirements when the number of effective digits in the data to be stored in a column of the table is less than the column's defined length. For decimal-type data, the effective digits excludes leading zeros.

DEC[IMAL]

Specifies that when decimal-type data is stored in the table, leading zeros are to be omitted.

The following rules apply to the DECIMAL option:

1. If SUPPRESS is specified and DECIMAL is omitted, DECIMAL is assumed.
2. When the number of effective digits in decimal-type data is equal to the defined length, the data is stored with a length equal to the defined length + 1. Note that this increases the data length compared to the case where the SUPPRESS option is specified.
3. Note also that if the defined precision (total number of digits) of

decimal-type data is 1, the length of the stored data is greater than when the SUPPRESS option is omitted.

4. The DECIMAL attribute for the abstract data type is not subject to this function.

WITHOUT ROLLBACK

Specifies that the table is to be defined such that whenever updating of the table (including additions and deletions) is completed, the locking of the rows in the table is to be released without waiting for a commit of the updating transaction.

The following rules apply to the WITHOUT ROLLBACK option:

1. The following table indicates the applicability of row-locking and rollback during the updating (including additions and deletions) of rows in a given table.

Table 3-29: Applicability of row-locking and rollback during the updating (including additions and deletions) of rows with a WITHOUT ROLLBACK specification

<table>
<tr><th colspan="2" rowspan="2">Object of operation</th><th colspan="4">Operation on the table</th></tr>
<tr><th>Row insertion</th><th colspan="2">Updating a column in a row</th><th>Row deletion</th></tr>
<tr><th></th><th></th><th>Updated value same as pre-update value</th><th>Updated value different from pre-update value</th><th></th></tr>
<tr><td rowspan="2">WITHOUT ROLLBACK-specified table with an index definition</td><td>The column to be updated is an index constituent column.</td><td rowspan="2">Row-locking is released when the transaction terminates. Even after processing, the transaction can be rolled back if it is not finished.</td><td rowspan="2">Row-locking is released upon completion of the update. After processing, the update cannot be rolled back.</td><td>Cannot be executed.</td><td rowspan="2">Row-locking is released when the transaction terminates. Even after processing, the transaction can be rolled back if it is not finished.</td></tr>
<tr><td>The column to be updated is not an index constituent column.</td><td>Row-locking is released when the transaction terminates. After processing, the update cannot be rolled back.</td></tr>
<tr><td colspan="2">WITHOUT ROLLBACK-specified table without an index definition</td><td colspan="4">Row-locking is released upon completion of the update. After processing, the update cannot be rolled back.</td></tr>
</table>

2. This option is ignored during execution of the database load utility and the database reorganization utility.
3. When this option is specified, the FIX option should also be specified for the table. This option is not specifiable if SHARE is specified.
4. This option is not compatible with a BLOB column definition or with a LOCK PAGE specification.

INDEXLOCK {NONE| PAGE}

This option is provided for compatibility with XDM/RD; it is ignored if specified.

Index key value no-lock is specified in the pd_indexlock_mode operand of the system definition; for details of the pd_indexlock_mode operand, see the manual *HiRDB Version 9 System Definition*.

SEGMENT REUSE { [*number-of-segments* [{K|M}]] |NO}

This operand is specified when using the free space reusage facility on the table being defined.

Because of the overhead caused by the processing required to reuse free space, use the free space reusage facility to give priority to storage efficiency over performance.

For details about the free space reusage facility, see the *HiRDB Version 9 Installation and Design Guide*.

number-of-segments [{K|M}]

When using the free space reusage facility and setting an upper limit on the number of segments for the table, specify the limit segment count. The *number-of-segments* operand is specified in the 1 to 268,435,440 range as an unsigned integer. Units K (kilo), or M (mega) can be specified.

The use of this operand can improve row insertion efficiency for tables that are subject to frequent row insertions or deletions, and the storage efficiency in the specified segments.

number-of-segments not specified

The operand *number-of-segments* can be omitted when using the free space reusage facility on a table and an upper limit is not set on the number of segments in the table.

Use this operand when a table is subject to frequent row insertions or deletions and only the specified table is to be stored in the RDAREA. This operand also improves row insertion performance and the storage efficiency of free space in the RDAREA in which the specified table is to be stored.

NO

This operand is specified when not using the free space reusage facility.

Specify NO for tables that are not subject to frequent row insertions or deletions.

The following rules apply to SEGMENT REUSE:

1. The free space reusage facility has no effect on LOB columns, abstract data type columns of the LOB attribute, or indexes.
2. The free space reusage facility cannot be specified for rebalancing tables.

INSERT ONLY [WHILE {*date-interval-data*|*labeled-duration*} BY *column-name*]

This option is specified when making a given table into a falsification-prevented table. For details about falsification-prevented tables, see the *HiRDB Version 9 Installation and Design Guide*.

If a table is made into a falsification-prevented table, its values cannot be updated. However, its updatable columns can be updated.

Not all columns in a falsification-prevented table can be made into updatable columns.

For a falsification-prevented table, you can specify a period in which any deletion of rows is prohibited (a *deletion-prevented duration*). When specifying a deletion-prevented duration, specify a period in WHILE, and in *column-name*, specify an insert history maintenance column (a DATE-type column that is a SYSTEM GENERATED column). If a deletion-prevented duration is not specified for a given falsification-prevented table rows can no longer be deleted from the table, permanently.

date-interval-data

Specifies a deletion-prevented duration in the decimal representation of date interval data. For details about decimal representation of date interval data, see *1.4.4 Decimal representation of date interval data*.

Note that date interval data can be specified in positive values only.

labeled-duration

In *labeled-duration*, specifies a deletion-prevented duration. For details about labeled durations, see *2.11 Date operations*.

Only positive integer literals, not enclosed in parentheses, can be specified in the value expression of *labeled-duration*.

column-name

Specifies a DATE-type column that is SYSTEM GENERATED.

The deletion-prevented duration should include the date when a row was inserted. The deletion-prevented duration should be calculated according to the *Rules for addition and subtraction of date data and date interval data* in *2.11 Date operations*. The last day of deletion prevention and the deletion-allowed date can be calculated as follows:

- Last day of deletion prevention = *row insertion date* + *deletion-prevented duration* - 1
- Deletion-allowed data = row insertion date + deletion-prevented duration

The following table indicates the relationship between last day of deletion prevention and the deletion-allowed date for specified values of the row insertion date and deletion prevented duration.

Table 3-30: Relationship between last day of deletion prevention and the deletion-allowed date

Date of row insertion	Specified value for deletion-prevented duration	Last day of deletion prevention	Deletion-allowed date
2002-03-01	1 year#1	2003-02-28	2003-03-01
1995-03-01	1 year#1	1996-02-29	1996-03-01
2002-02-28	1 month#2	2002-03-27	2002-03-28
2002-05-01	1 day#3	2002-05-01	2002-05-02

#1: Actual specification format: `00010000.` for date interval data, `1 YEAR` for labeled duration

#2: Actual specification format: `00000100.` for date interval data, `1 MONTH` for labeled duration

#3: Actual specification format: `00000001.` for date interval data, `1 DAY` for labeled duration

6) *table-restriction-definition* ::= {*multicolumn-uniqueness-constraint-definition*

[*index-option* [*index-option*]]

| {*multicolumn-referential-constraint-definition* [*constraint-name-definition*]

| [*constraint-name-definition*] *multicolumn-referential-constraint-definition* }

| {*multicolumn-referential-constraint-definition* [*constraint-name-definition*]

| [*constraint-name-definition*] *multicolumn-referential-constraint-definition*}}

This operand specifies a uniqueness constraint, check constraint, and referential constraint for multiple columns.

The position in which *constraint-name-definition* is specified is determined by the specification value of the system common definition `pd_constraint_name` operand or the specification value of the client environment variable `PDCNSTRNTNAME`. The following table indicates allowable specification positions for the constraint name definition.

Table 3-31: Specification positions for the constraint name definition

Client environment variable		System common definition		
		pd_constraint_name		
		Not specified	LEADING	TRAILING
PDCNSTRNTNAME	Not specified	Before	Before	After
	LEADING	Before	Before	Before
	TRAILING	After	After	After

Legend:

Before: A constraint name definition is specified before a constraint definition (standard SQL specifications).

After: A constraint name definition is specified after a constraint definition (XDM/RD-compatible specifications).

A multicolumn referential constraint cannot be specified if `SHARE` and `FIX` are specified.

7) *column-definition* ::=

column-name data-type [`ARRAY` [*maximum-number-of-elements*]]

[`NO SPLIT`]

[{*column-data-suppression-specification*|[*column-recovery-restriction*]

{IN {*LOB-column-storage-RDAREA-name*

|(*LOB-column-storage-RDAREA-name*)

|((*LOB-column-storage-RDAREA-name*)

[, (*LOB-column-storage-RDAREA-name*)]...)

|*matrix-partitioned-LOB-column-storage-RDAREA-specification*}

|*abstract-data-type-LOB-column-storage-*

RDAREA-specification}}]

[*plug-in-specification*]

[DEFAULT *clause*]

[*column-restriction*]...

[*updatable-column-attribute*]

column-name

Specifies a name for a column that is to compose the table. Each column name must be unique.

data-type

Specifies the data type of the column; see *1.2 Data types* for an explanation of data types.

Neither a super-type abstract data type with BLOB defined nor the BOOLEAN data type can be specified.

If the specified data type is an abstract data type, no authorization identifier is specified, and the default authorization identifier does not have an abstract data type of the same name, and if there is an abstract data type of the same name in the MASTER authorization identifier, that abstract data type is assumed to have been specified.

ARRAY [*max-number-of-elements*]

When a repetition column is being defined, specifies the maximum number of elements, as an unsigned integer in the range 2 to 30,000.

A repetition column cannot be specified for any of the following data types:

- CHAR or VARCHAR for which a character set is specified
- BLOB
- BINARY
- Abstract data type

NO SPLIT

Specifies that when the actual data length of a variable-length character string exceeds 255 bytes, one row of data is to be stored on a single page. In some cases, the NO SPLIT option will reduce the database's storage space requirements. This is called the *no-split option*; for details of the no split option, see the *HiRDB Version 9 Installation and Design Guide*.

The no-split option is applicable only to variable-length character string data types (VARCHAR, NVARCHAR, and MVARCHAR).

LOB-column-storage-RDAREA-name

Specifies the name of the user LOB RDAREA for storing BLOB column data.

The following rules apply to LOB column storage RDAREAs:

1. If the BLOB data type is specified for a column, a LOB column storage RDAREA must be specified for it. A LOB column storage RDAREA cannot be specified for a column of a non-BLOB data type.
2. When a table is partitioned, you must specify the same number of user LOB RDAREAs as the number of partitions into which the table is to be split. Thus, the partitioning must be specified so that the user RDAREAs and the user LOB RDAREAs at the same server will be in the same sequence. An example is shown below:

```
CREATE TABLE MOVIE (ID INT NOT NULL,
       IMAGE BLOB IN ((LU01),(LU02))
       IN ((RU01) ID<120,(RU02))
```

LU01, LU02, RU01, and RU02 denote RDAREA names.

RU01 and LU01, and RU02 and LU02, are RDAREAs for the respective servers.

1. Before CREATE TABLE can be execute LOB column storage RDAREA must be created using the database initialization utility or must be added using the database structure modification utility.
2. The same RDAREA name cannot be used when specifying multiple LOB column storage RDAREA names. For a row-partitioned table, hash-partitioned table, or matrix-partitioned table that specifies a boundary value, specify an LOB column storage RDAREA name for each table storage RDAREA and LOB column storage RDAREA. If duplicate table storage RDAREA names are specified, specify a duplicate LOB column storage RDAREA at the same position.

   In all other cases, duplicate LOB column storage RDAREAs cannot be specified.

matrix-partitioned-LOB-column-storage-RDAREA-specification::=*two-dimensional-storage-RDAREA-specification*

This operand is specified when defining a matrix-partitioned table.

For specification methods, see the explanation of two-dimensional storage RDAREA names in *PARTITIONED BY MULTIDIM*. For details about RDAREAs that can be specified, see the explanation of LOB column storage RDAREA names.

8) *column-data-suppression-specification*::=SUPPRESS

Specifies that column data is to be suppressed. This option reduces the amount of disk

space that is required when much of the data is in fixed-size character format with trailing spaces.

The following rules apply to column data suppression specifications:

1. If the last character in column data is the space, a search is made for contiguous spaces to the left and the data is stored in the database by suppressing the string of consecutive spaces that are found as a result of this search. If there is a break in the string of consecutive spaces, no spaces to the left of the break are suppressed.
2. When the data type is CHAR or MCHAR, four or more one-byte spaces are suppressed.
3. When the data type is NCHAR, three or more double-byte spaces are suppressed.
4. This option cannot be specified for a FIX table.
5. Column data suppression cannot be specified if the data type is an abstract data type.
6. This option can be specified only when the data type is CHAR, MCHAR, or NCHAR.
7. Column data suppression cannot be specified for a repetition column.
8. One byte of additional information is added to a column for which this option is specified even if data suppression does not occur.
9. Specification of this option is invalid in the following cases:
   - Data type is CHAR and the column data size is shorter than CHAR(5)
   - Data type is MCHAR and the column data size is shorter than MCHAR(5)
   - Data type is NCHAR and the column data size is shorter than NCHAR(3)

   If spaces are exhausted, any space preceding that location is not suppressed.

9) *column-recovery-restriction* ::=RECOVERY [{ALL|PARTIAL|NO}]

Specifies the database update logging method for a LOB column storage RDAREA or a LOB storage RDAREA within an abstract data type definition.

LOB column storage RDAREA:

If the BLOB data type is specified, specifies the database update logging method for user LOB RDAREAs. This option cannot be specified for columns with a data type other than BLOB.

LOB storage RDAREA within an abstract data type definition:

If the BLOB attribute is in an abstract data type definition, specifies the database update logging method for user LOB RDAREAs.

ALL

Specifies that the user LOB RDAREA is to be operated in the log acquisition mode. Operation in this mode means that a database update log needed for rollback and rollforward is acquired.

`PARTIAL`

Specifies that the user LOB RDAREA is to be operated in the pre-update log acquisition mode. Operation in this mode means that a database update log needed for rollback is acquired.

`NO`

Specifies that the user LOB RDAREA is to be operated in the no-log mode. Operation in this mode means that a database update log is not to be acquired.

The UAP execution method and the method of recovering a user LOB RDAREA when an error occurs depend on the specified update log acquisition mode for the database. For details of operation in the no-log mode, see the *HiRDB Version 9 System Operation Guide*.

10) *abstract-data-type-LOB-column-storage-RDAREA-specification* `:::=`

`ALLOCATE` (*attribute-name*[`.`*attribute-name*]...

`IN` {*LOB-attribute-storage-RDAREA-name*

|(*LOB-attribute-storage-RDAREA-name*)

|((*LOB-attribute-storage-RDAREA-name*)

[`,` (*LOB-attribute-storage-RDAREA-name*)])

|*matrix-partitioned-LOB-attribute-storage-RDAREA-specification*}

[`,` *attribute-name*[*..attribute-name*]...

`IN` {*LOB-attribute-storage-RDAREA-name*

|(*LOB-attribute-storage-RDAREA-name*)

|((*LOB-attribute-storage-RDAREA-name*)

[`,` (*LOB-attribute-storage-RDAREA-name*)])

|*matrix-partitioned-LOB-attribute-storage-RDAREA-specification*}]...)

attribute-name[*..attribute-name*]

Specifies the names of attributes that comprise an abstract data type. If the attribute of the abstract data type is the abstract data type and if the attribute of nested abstract data types includes a `LOB`-type attribute, specify the attribute name of the `LOB` type.

Specify an attribute name in the following cases:

- If the data type of the attribute of an abstract data type is the LOB type
- If the attribute of an abstract data type is the abstract data type and if the attribute of nested abstract data types is the LOB-type attribute (specify the attribute name of that LOB type)

LOB-attribute-storage-RDAREA-name

Specifies the name of a user LOB RDAREA for storage of BLOB-attribute data, located at any level of the abstract data type.

The following rules apply to LOB attribute storage RDAREAs:

1. If an abstract data type with the BLOB type is specified as a data type, a LOB RDAREA must be specified for each BLOB attribute. A LOB attribute storage RDAREA cannot be specified for an attribute whose data type is not of the BLOB type.
2. When a table is being partitioned, specify the same number of user LOB RDAREAs as the number of partitions into which the table is to be partitioned. You must ensure that the user RDAREAs and user LOB RDAREAs on the same server are specified in the same order.
3. Required LOB attribute storage RDAREAs must be created or added in advance with the database initialization utility or the database structure modification utility.
4. The same RDAREA name cannot be used when specifying multiple LOB attribute storage RDAREA names. For a row-partitioned table, hash-partitioned table, or matrix-partitioned table that specifies a boundary value, specify an LOB attribute storage RDAREA name for each table storage RDAREA and LOB attribute storage RDAREA. If duplicate table storage RDAREA names are specified, specify a duplicate LOB attribute storage RDAREA at the same position.

   In all other cases, duplicate LOB attribute storage RDAREAs cannot be specified.

matrix-partitioned-LOB-attribute-storage-RDAREA
::=*two-dimensional-storage-RDAREA-specification*

This operand is specified when defining a matrix-partitioned table.

For specification methods, see the explanation of two-dimensional storage RDAREA names in *PARTITIONED BY MULTIDIM*. For details about RDAREAs that can be specified, see the explanation of LOB attribute storage RDAREA names.

11) *plug-in-specification* ::= PLUGIN *plug-in-option*

plug-in-option

Specifies as a character string literal (of up to 255 bytes) parameter information to be passed to the plug-in facility for a column that is defined as an abstract data type for which the plug-in facility is implemented. Hexadecimal character string literals cannot be specified as parameter information. For details of the parameter information, see the manual for the applicable plug-in.

12) DEFAULT *clause* ::=DEFAULT [*predefined-value*]

If a value is omitted during the insertion of data, the value that was specified in DEFAULT *clause* is assumed.

The following rules apply to DEFAULT *clause*:

1. The DEFAULT clause and the WITH DEFAULT clause cannot be specified for the same item.
2. The DEFAULT clause cannot be specified for items of the BLOB type, the abstract data type, or the BINARY type with a defined length of 32,001 bytes or greater.
3. The DEFAULT clause cannot be specified for a repetition column.

predefined-value ::={ *literal* | USER | CURRENT_DATE | CURRENT DATE

| CURRENT_TIME | CURRENT TIME

| CURRENT_TIMESTAMP [(*fractional-second-precision*)] [USING BES]

| CURRENT TIMESTAMP [(*fractional-second-precision*)] [USING BES]

| NULL }

The default value must be a value that can be inserted into a specified item.

The following rules apply to *predefined-value*:

1. If *predefined-value* is omitted, the predefined value for WITH DEFAULT is assumed.
2. In *predefined-value*, a data type in which the effective upper digits are nullified during insertion cannot be specified.
3. NULL as a predefined value cannot be specified in a NOT NULL constrained column.
4. When specifying a DEFAULT clause for a column that uses other than the default character set, the following default values cannot be specified:
   - CURRENT_DATE (CURRENT DATE)
   - CURRENT_TIME (CURRENT TIME)
   - CURRENT_TIMESTAMP (CURRENT TIMESTAMP)
5. If USER, CURRENT_DATE (CURRENT DATE), CURRENT_TIME (CURRENT TIME), or CURRENT_TIMESTAMP [(*fractional-second-precision*)]

(CURRENT TIMESTAMP[(*fractional-second-precision*)]) is specified, the following value is assigned:

- USER:

  The value of the authorization identifier of the execution user who inserted the row is assigned.

- CURRENT_DATE (CURRENT DATE):

  The date of row insertion is assigned. The database load utility (pdload), however, assigns the date the utility is started.

- CURRENT_TIME (CURRENT TIME):

  The time of row insertion is assigned. The database load utility (pdload), however, assigns the time the utility is started.

- CURRENT_TIMESTAMP [(*fractional-second-precision*) [USING BES]]
  (CURRENT TIMESTAMP [(*fractional-second-precision*)] [USING BES])

  Assigns the time stamp at the time a row was inserted. However, for the database load utility (pdload), the time stamp at the time the utility was started is assigned. In operations using a multi-front-end server configuration on HiRDB/Parallel Server, the time stamp is obtained from the front-end server to which the UAP established a connection.

  If USING BES is specified, the current time stamp is acquired from the back-end server that manages the RDAREA in which update rows or insertion rows are stored in the case of a HiRDB/Parallel Server; for a HiRDB/Single Server, the current time stamp is acquired from the single server.

  If USING BES is omitted, the current time stamp is acquired from the front-end server in the case of a HiRDB/Parallel Server; and the current time stamp is acquired from the single server in the case of a HiRDB/Single Server.

  In the default-value acquisition server type specification, a column for which USING BES is specified cannot be specified in *partitioning-key*.

13) *column-restriction* ::=
{*NOT-NULL-constraint*|*single-column-uniqueness-constraint-definition*|
[*index-option* [*index-option*]]}

| {*single-column-check-constraint-definition* [*constraint-name-definition*]

| [*constraint-name-definition*] *single-column-check-constraint-definition*}

| {*single-column-referential-constraint-definition* [*constraint-name-definition*]

| [*constraint-name-definition*] *single-column-referential-constraint-definition*}}

The following restrictions can be specified for a column:

- NOT NULL constraint
- Single-column uniqueness constraint definition
- Single-column check constraint definition
- Single-column referential constraint definition

The position in which *constraint-name-definition* is specified is determined by the specification value in the system common definition pd_constraint_name operand or the specification value in the client environment variable PDCNSTRNTNAME. For details, see *Table 3-31* on table constraint definitions.

A single-column referential constraint cannot be specified if both SHARE and FIX are specified.

14) *updatable-column-attribute*: : = UPDATE [ONLY FROM NULL]

Specify this operand when defining a falsification-prevented table or defining an updatable column in a table that will be changed into a falsification-prevented table.

The updatable column attribute is valid only with a falsification-prevented table.

For details about the falsification-prevented table, see the *INSERT ONLY option*. For details about how to change a given table into a falsification-prevented table, see the *INSERT ONLY option* in *CHANGE* in *ALTER TABLE*.

The following rules apply to the updatable clause attribute:

1. The attribute cannot be specified for columns for which SYSTEM GENERATED is specified.
2. The attribute cannot be specified for any of the following non-updatable columns:

- Cluster key constituent column
- Partitioning key constituent column (exclusive of partitioning key constituent columns in a flexible hash partitioning table)

UPDATE

Specify this operand when defining an updatable column in a falsification-prevented table.

UPDATE ONLY FROM NULL

Specify this operand when defining a column in a falsification-prevented table in which row values can be updated only once from the null value to a non-null value.

The following table lists the updatability of column values in a falsification-prevented table for which UPDATE ONLY FROM NULL is specified.

Column value before update	Column value after update	Updatability
Null value	Null value	Y
Null value	Non-null value	Y
Non-null value	Null value	N
Non-null value	Non-null value#	N

Legend:

Y: Updatable

N: Not updatable

Note

Repetition columns can be updated only from the null value (a value in which the number of elements is 0) to unsubscripted columns.

#: Includes the same value as a pre-update value.

The following rules apply to the UPDATE ONLY FROM NULL operand :

- This operand cannot be specified for columns for which NOT NULL is specified.
- This operand cannot be specified for FIX tables.
- This operand cannot be specified for the primary key or for cluster key constituent columns.
- This operand cannot be specified for partitioning key constituent columns.
- This operand cannot be specified for BLOB type columns and for BINARY type columns with a minimum defined length of 32,001 bytes or greater.

If the attribute is specified, the column value of the updatable column attribute can be updated under the following conditions:

Table type	UPDATE specification		UPDATE ONLY FROM NULL specification		No specification	
	Specifiable	Column value updatable	Specifiable	Column value updatable	Specifiable	Column value updatable
Non-falsification-prevented table	Y	Y	Y	Y	--	Y
Falsification-prevented table	Y	Y	Y	Y#	--	N

Legend:

Y: Updatable

N: Not updatable

--: Not applicable

#: Can be updated only once from the null value to a non-null value.

15) *NOT-NULL-constraint-specification*::=

{[NULL

|NOT NULL [WITH DEFAULT[SYSTEM GENERATED]]]#2

|[NOT NULL] WITH DEFAULT[SYSTEM GENERATED]]}#1

#1: For a column of a FIX table, a cluster key column, or a column that belongs to the primary key.

#2: For a column other than the above.

NULL

Specifies that the null value is to be permitted in the specified column.

The NULL option cannot be specified for a column of a FIX table, a cluster key column, or a column that belongs to the primary key.

NOT NULL

Specifies the NOT NULL constraint, which means that the column cannot contain the null value.

NOT NULL cannot be specified for an abstract data type or a repetition column.

WITH DEFAULT

If the column names to be inserted and the insertion values are omitted during data

loading using either the INSERT statement or the database creation utility, the WITH DEFAULT option must be specified when the default values are inserted into NOT NULL constrained columns.

The following rules apply to the WITH DEFAULT option:

1. NOT NULL can be omitted when specifying WITH DEFAULT for a FIX table.
2. WITH DEFAULT cannot be omitted if the data type is the abstract data type.
3. The following table lists column default values that are assigned when WITH DEFAULT is specified.

Table 3-32: Default values for a column with the WITH DEFAULT clause

Data type		Column default value
INTEGER		0
SMALLINT		
FLOAT		
SMALLFLT		
DECIMAL		
CHAR		Space
NCHAR		
MCHAR		
VARCHAR	Default character set Character set other than UTF-16	Single-byte space
	Character set UTF-16	One-character space
NVARCHAR		One-character space
MVARCHAR		One-byte space
DATE		Current date when a row is added
TIME		Current time when a row is added
TIMESTAMP		Current time stamp when a row is added
INTERVAL YEAR TO DAY		0 years, 0 months, 0 days
INTERVAL HOUR TO SECOND		0 hours, 0 minutes, 0 seconds

Data type	Column default value
BLOB	Data with a length of 0 bytes
BINARY	

Note

When the WITH DEFAULT *clause* is not specified, the null value becomes the default value for the column.

SYSTEM GENERATED

This option can be specified when the data type of the column is either the DATE type or TIME type. The column for which SYSTEM GENERATED is specified is called an *insert history maintenance column*. Columns for which SYSTEM GENERATED is specified are used to specify a deletion-prevented duration for a falsification-prevented table.

Columns for which SYSTEM GENERATED is specified receive the insertion of the current date (CURRENT_DATE) for the DATE type, or the current time (CURRENT_TIME) for the TIME type during the insertion of data by means of an INSERT statement, irrespective of whether a value is specified.

However, it is an error to specify the NULL keyword as the insertion value for a column for which SYSTEM GENERATED was specified.

16) *single-column-uniqueness-constraint-definition* : : =

{[{UNIQUE |PRIMARY}] CLUSTER KEY [{ASC |DESC}]
[IN {*index-storage-RDAREA-name*
|(*index-storage-RDAREA-name*)
|((*index-storage-RDAREA-name*)
[, (*index-storage-RDAREA-name*)]...)}]
|PRIMARY KEY [{ASC |DESC}]
[IN {*index-storage-RDAREA-name*
|(*index-storage-RDAREA-name*)
|((*index-storage-RDAREA-name*)
[, (*index-storage-RDAREA-name*)]...)}]}
{[{UNIQUE |PRIMARY}] CLUSTER KEY [{ASC |DESC}]
[IN {*index-storage-RDAREA-name*
|(*index-storage-RDAREA-name*)

|((*index-storage-RDAREA-name*)

[, (*index-storage-RDAREA-name*)]...)}]

Specifies that the column is to be defined as a cluster key.

The following rules apply to the cluster key:

1. None of the following data types can be specified for a column that composes a cluster key:
   - `BLOB`
   - `BINARY`
   - Abstract data type
2. A repetition column cannot be specified as a column that composes a cluster key.
3. When a cluster key is specified, an index is defined for the specified column. A defined index cannot be deleted.
4. The `NOT NULL` option is assumed for a cluster key column.
5. Duplicated data cannot be inserted into a cluster key for which `UNIQUE` or `PRIMARY` is specified.
6. Only one cluster key can be defined per table.
7. When a cluster key is specified, the `HASHA` to `HASHF` hash functions cannot be specified.
8. The total length of columns comprising a cluster key must satisfy the following formula:

   Total length of columns

   ≤ MIN((*page size of index storage RDAREAs* ÷ 2) - 1242, 4036)
9. The following rules apply to specifying a cluster key on a partitioned table:

key range-partitioning (*storage-condition*, *boundary-value*)

Specify partitioning keys for columns that compose a cluster key.

hash-partitioning

single-column-partitioning

Specify partitioning keys for columns that compose a cluster key.

multicolumn-partitioning

A cluster key cannot be specified for a single column.

matrix-partitioning

A cluster key cannot be specified for a single column.

`UNIQUE`

Specifies the restriction that the value in each row in the cluster key column must be unique (i.e., no value can be duplicated in the cluster key column).

`PRIMARY`

Specifies that a column belonging to the cluster key is being defined as the primary key.

`ASC`

Specifies that the cluster key index is to be generated in ascending order of the key values.

`DESC`

Specifies that the cluster key index is to be generated in descending order of the key values.

index-storage-RDAREA-name

Specifies the name of an RDAREA in which the cluster key index is to be stored.

If the index is to be partitioned by rows for storage, an index storage RDAREA name must be specified for each component into which the table is to be partitioned.

On HiRDB/Parallel Server, the operand is subject to the following restrictions:

- If `SHARE` is specified, the index storage RDAREA name should be a shared RDAREA.
- If `SHARE` is not specified, the index storage RDAREA name cannot be specified as a shared RDAREA.

The following rules apply to the index storage RDAREAs:

1. Each specified RDAREA either must have already been created by the database initialization utility or must be added by the database structure modification utility.
2. If no index storage RDAREA names are specified, the index is stored in the same RDAREAs that store the defined table.
3. The same RDAREA name cannot be used when specifying multiple index storage RDAREA names. For a row-partitioned table, hash-partitioned table, or matrix-partitioned table that specifies a boundary value, specify an index storage RDAREA name for each table

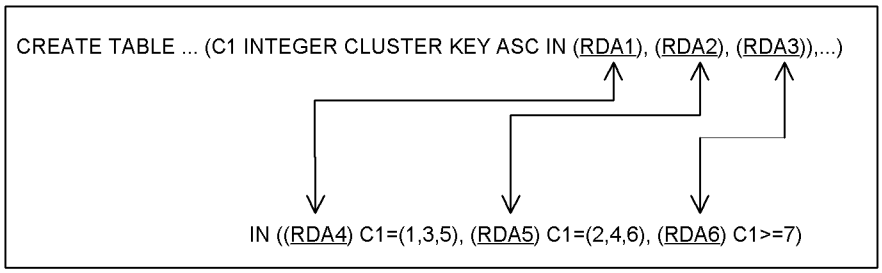
storage RDAREA and index storage RDAREA. If duplicate table storage RDAREA names are specified, specify a duplicate index storage RDAREA in the same position.

4. A user RDAREA that stores a rebalancing table cannot be specified. Similarly, if a cluster key is defined for a rebalancing table, the specification of an RDAREA name cannot be omitted.

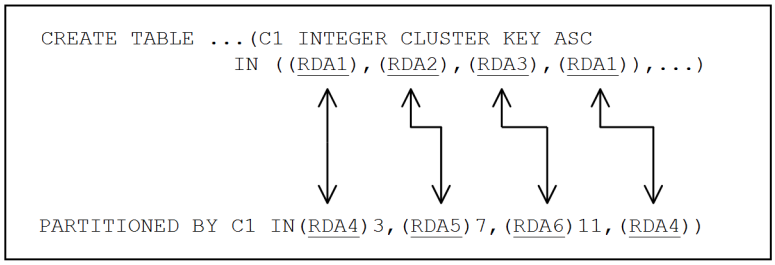
5. The number of index storage RDAREA names must equal the number of table partitions that are stored (tables are partitioned by rows). In this case, the specified index storage RDAREAs are used in the same order in which the table storage RDAREAs are specified. All specified RDAREAs must be in the same server.

   Examples are provided below:

Example 1: Partitioning by rows based on storage conditions



Example 2: Partitioning by rows based on a boundary value



Legend:

RDA1 to RDA6: Indicates the RDAREA names.

⇕ : The arrows indicate that RDAREAs RDA4, RDA5, and RDA6 of the table are associated with RDAREAs RDA1, RDA2, and RDA3 of the index, respectively.

RDA1 to RDA6 denote RDAREA names.

The arrows indicate that RDAREAs RDA4, RDA5, and RDA6 of the RDAREAs in

the table are associated with RDAREAs RDA1, RDA2, and RDA3 of the index, respectively.

```
PRIMARY KEY [{ASC |DESC}]
      [IN {index-storage-RDAREA-name
          |(index-storage-RDAREA-name)
          |((index-storage-RDAREA-name)
            [, (index-storage-RDAREA-name)]...)}]
```

Specifies that the column specified in *column-name* is being defined as the primary key.

The following rules apply to the primary key:

1. The following data types cannot be specified for the columns that make up the primary key:
   - BLOB
   - BINARY
   - Abstract data type
2. A repetition column cannot be specified as a column comprising the primary key.
3. Specifying a primary key causes an index to be defined for the specified columns. An index defined in this manner cannot be deleted.
4. NOT NULL is assumed for a column that comprises the primary key.
5. Duplicated data cannot be inserted into a primary key column.
6. Only one primary key can be defined per table.
7. For definability (UNIQUE specifiability) of row-partitioned tables, see *Table 3-22*.
8. The total length of columns comprising the primary key must satisfy the following formula:

   Total length of columns

   ≤ MIN((*page size of index storage RDAREAs* ÷ 2) - 1242, 4036)
9. For a matrix-partitioned table, the primary key cannot be specified for a single column.

ASC

Specifies that the primary key index is to be created in ascending order.

DESC

Specifies that the primary key index is to be created in descending order.

index-storage-RDAREA-name

Specifies the name of an RDAREA for storing the primary key index.

The specified index storage RDAREA should be a user RDAREA.

On HiRDB/Parallel Server, the operand is subject to the following restrictions:

- If SHARE is specified, the index storage RDAREA name should be a shared RDAREA.
- If SHARE is not specified, the index storage RDAREA name cannot be specified as a shared RDAREA.

The following rules apply to the index storage RDAREAs:

1. The RDAREA must be created in advance using the database initialization utility or added using the database structure modification utility.
2. If no RDAREA name is specified, the index will be stored in the RDAREAs that store the table that is being defined. However, in the case of a row-partitioned table in a HiRDB/Single Server or a row-partitioned table at one back-end server in a HiRDB/Parallel Server, the index will be stored as follows:

   For a single-column partitioned table

   If a column other than the partitioning key is specified as the primary key, the index is stored in the first table storage RDAREA for which a partitioning condition is specified.

   For a multicolumn partitioned table

   The index is stored in the first table storage RDAREA for which a partitioning condition is specified.

   Matrix-partitioned tables

   The primary key cannot be specified for a single column.
3. The same RDAREA name cannot be used when specifying multiple index storage RDAREA names. For a row-partitioned table, hash-partitioned table, or matrix-partitioned table that specifies a boundary value, specify an index storage RDAREA name for each table storage RDAREA and index storage RDAREA. If duplicate table storage RDAREA names are specified, specify a duplicate index storage RDAREA in the same position.
4. A user RDAREA that stores a rebalancing table cannot be specified.

Similarly, if a cluster key is defined for a rebalancing table, specification of an RDAREA name cannot be omitted.

5. If a table is stored on a split basis into multiple RDAREAs, the index storage RDAREAs must be specified by one of the following methods:

   (a) Specifying a column specified as a partitioning key in a single-column-partitioned table, at the beginning of the primary key

   (b) Specifying all columns specified as a partitioning key in a multicolumn-partitioned table, in the same order at the beginning of the primary key

   (c) Matrix-partitioned tables

   (d) Other than items (a) to (c), specifying all columns specified in a partitioned key as constituent columns of the primary key (in any order)

   For items (a) to (c), specify RDAREA names in a number equal to the number of table storage RDAREA names. In this case, the destination of index storage corresponds with the order in which table storage RDAREAs are specified.

   For (d), specify RDAREA names in a number equal to the number of servers that are stored by partitioning tables. For a HiRDB/Single Server, specify only one RDAREA name; for a HiRDB/Parallel Server, specify an RDAREA per back-end server that contains a table.

   In any other cases, the primary key cannot be defined for a table that is stored on a partitioned basis.

17) *index-option* ::={PCTFREE=*percentage-of-free-area*|UNBALANCED SPLIT}

PCTFREE =*percentage-of-free-area*

Specifies the percentage of unused space to be left in index pages when an index is created. The specifiable range of values is 0 to 99, and the default is 30.

When created in batch by the database load utility and the database reorganization utility, indexes are created in a percentage equal to the percentage of unused space. In other addition or update operations by INSERT or UPDATE statements, the default PCTFREE=0 is assumed.

If rows are to be added frequently after an index has been created, a high percentage of unused space should be specified.

UNBALANCED SPLIT

Specifies that the key values are to be allocated unevenly among the pages when a page is split.

If the location where a key value is to be inserted is in the first half of the page to

be split, more empty space is allocated to the left-side page after the split. If the key value insertion location is in the second half of the page, more empty space is allocated to the right-side page after the split. This is called an unbalanced index split.

For details of unbalanced index splits, see the *HiRDB Version 9 System Operation Guide*.

18) *single-column-check-constraint-definition* : := CHECK (*search-condition*)

Specify this operand when defining a check constraint for the column specified in *column-name*.

search-condition

Specifies the condition that constrains the value of a column. If this condition is FALSE, no insertion or updating can be performed on the table.

The column specified in *search-condition* should be one that was specified in *column-name*.

The following items cannot be specified in *search-condition*:

- Subquery
- Set function or SQL/XML set function
- Window function
- Repetition column
- Function call
- USER
- CURRENT DATE, CURRENT_DATE
- CURRENT TIME, CURRENT_TIME
- CURRENT TIMESTAMP, CURRENT_TIMESTAMP
- ? parameter, embedded variable
- CAST specification specifying conversion from TIME to TIMESTAMP
- Scalar function VARCHAR_FORMAT specifying TIME type in a value expression
- Value expression in the abstract data type
- System-defined scalar function
- Scalar function IS_USER_CONTAINED_IN_HDS_GROUP
- Structured repetition predicate

- XML constructor function
- SQL/XML scalar function

19) *single-column-referential-constraint-definition*::= *reference-specification*

Specify this operand when defining the column specified in *column-name* as a foreign key.

The following rules apply to foreign keys:

1. If a foreign key is specified for a column, the following data types cannot be specified on the column:
   - BLOB
   - BINARY
   - Abstract data type
2. A repetition column cannot be specified for a foreign key column.
3. The foreign key and the primary key that references it must agree in all of the following items:
   - Corresponding data type
   - Corresponding data length

20) *multicolumn-uniqueness-constraint-definition*::=

{[{UNIQUE|PRIMARY}] CLUSTER KEY (*column-name* [{ASC|DESC}]
[, *column-name* [{ASC|DESC}]] ...)
[IN {*index-storage-RDAREA-name*
|(*index-storage-RDAREA-name*)
|(*index-storage-RDAREA-name*)
[,(*index-storage-RDAREA-name*)]...)
|*matrix-partitioned-index-storage-RDAREA-specification*}]
|PRIMARY KEY [{ASC|DESC}]
[IN {*index-storage-RDAREA-name*
|(*index-storage-RDAREA-name*)
|((*index-storage-RDAREA-name*)
[, (*index-storage-RDAREA-name*)]...)
|*matrix-partitioned-index-storage-RDAREA-specification*}]}
{[{UNIQUE|PRIMARY}] CLUSTER KEY (*column-name* [{ASC|DESC}]

[, *column-name* [{ASC|DESC}]] ...)

[IN {*index-storage-RDAREA-name*

|(*index-storage-RDAREA-name*)

|(*index-storage-RDAREA-name*)

[,(*index-storage-RDAREA-name*)]...)

|*matrix-partitioned-index-storage-RDAREA-specification*}]

Specifies that multiple columns are to be defined as a cluster key.

The following rules apply to cluster keys:

1. None of the following data types can be specified for the columns that comprise a cluster key:
   - BLOB
   - BINARY
   - Abstract data type
2. If a cluster key is comprised of multiple columns, the following data types, in addition to item 1, cannot be specified:
   - FLOAT
   - SMALLFLT
3. A repetition column cannot be specified as a column that composes a cluster key.
4. When a cluster key is specified, an index is defined for the specified columns. When a defined index is deleted, the associated table should also be deleted.
5. The NOT NULL option is assumed for the columns that comprise a cluster key.
6. Duplicated data cannot be inserted into a cluster key for which UNIQUE or PRIMARY is specified.
7. Only one cluster key can be defined per table.
8. The maximum number of columns comprising a cluster key for which PRIMARY is specified is 16.
9. When a cluster key is specified, the HASHA to HASHF hash functions cannot be specified.
10. The total length of columns comprising an index must satisfy the following formula:

    Total length of columns

≤ MIN((*page size of index storage RDAREAs* ÷ 2) - 1242, 4036)

11. The following rules apply to specifying a cluster key for a partitioned table:

key range-partitioning (*storage-condition*, *boundary-value*)

Specify partitioning keys for columns that compose a cluster key.

hash-partitioning

single-column-partitioning

Specify partitioning keys at the beginning of the columns that compose a cluster key.

multicolumn-partitioning

Specify this item so that it includes all partitioning keys in the same sequence, from the beginning of the columns that compose a cluster key.

Matrix-partitioning

Specify this item so that it includes all partitioning keys in a first-dimension partitioned column and second-dimension partitioned column sequence from the beginning of the columns that compose a cluster key.

When multicolumn-partitioning a second-dimension partitioned clause, specify this item so that it includes all partitioning keys in a first-dimension partitioned column - second-dimension partitioned column sequence.

`UNIQUE`

Specifies the restriction that the value in each row in the cluster key column must be unique (i.e., no value can be duplicated in the cluster key column).

`PRIMARY`

Specifies that a multicolumn cluster key is being defined as the primary key.

column-name

Specifies the name of a column that defines the cluster key.

The names of all columns that comprise the cluster key must be unique. When a cluster key is defined to consist of multiple columns, the key values are generated in the order in which the columns are specified.

`ASC`

Specifies that the index of the cluster key values is to be generated in ascending order of the values.

`DESC`

Specifies that the index of the cluster key values is to be generated in

descending order of the values.

index-storage-RDAREA-name

Specifies the name of an RDAREA in which the cluster key index is to be stored.

If the index is to be partitioned by values for storage, an index storage RDAREA name must be specified for each component into which the table is partitioned.

The specified index storage RDAREAs must be user RDAREAs.

The following rules apply to the index storage RDAREAs:

1. Each specified RDAREA either must have been created by the database initialization utility or must be added by the database structure modification utility.
2. If no index storage RDAREA names are specified, the index is stored in the same RDAREAs that store the defined table.
3. The same RDAREA name cannot be used when specifying multiple index storage RDAREA names. For a row-partitioned table, hash-partitioned table, or matrix-partitioned table that specifies a boundary value, specify an index storage RDAREA name for each table storage RDAREA and index storage RDAREA. If duplicate table storage RDAREA names are specified, specify a duplicate index storage RDAREA in the same position.
4. The number of index storage RDAREAs must equal the number of table partitions that are stored (tables are partitioned by rows). In this case, the specified index storage RDAREAs are used in the same order in which the table storage RDAREAs are specified. All specified RDAREAs must be in the same server.

   For an example, see the section on index storage RDAREA names for the single-column uniqueness restriction definition.
5. A user RDAREA that stores a rebalancing table cannot be specified. Similarly, if a cluster key is defined for a rebalancing table, specification of an RDAREA name cannot be omitted.

matrix-partitioned-index-storage-RDAREA-specification **::=***two-dimensional-storage-RDAREA-specification*

This operand is specified when defining a matrix-partitioned table.

For specification methods, see the explanation of two-dimensional storage RDAREA names in *PARTITIONED BY MULTIDIM*. For details about RDAREAs that can be specified, see the explanation of index storage

RDAREA names.

PRIMARY KEY (*column-name* [{ASC |DESC}]
[, *column-name* [{ASC |DESC}]]...)
[IN {*index-storage-RDAREA-name*
|(*index-storage-RDAREA-name*)
|((*index-storage-RDAREA-name*)
[, (*index-storage-RDAREA-name*)]...)
|*matrix-partitioned-index-storage-RDAREA-specification*}]

Specifies multiple columns that are being defined as the primary key.

The following rules apply to the primary key:

1. The following data types cannot be specified for the columns that make up the primary key:
   - BLOB
   - BINARY
   - Abstract data type
2. When the primary key is composed of multiple columns, the following data types, in addition to item 1, cannot be specified:
   - FLOAT
   - SMALLFLT
3. A repetition column cannot be specified as a column comprising the primary key.
4. Specifying a primary key causes an index to be defined for the specified columns. To delete the defined index, the entire table must be deleted.
5. NOT NULL is assumed for a column that comprises the primary key.
6. Duplicated data cannot be inserted into a primary key column.
7. Only one primary key can be defined per table.
8. The maximum number of columns comprising a cluster key for which PRIMARY is specified is 16.
9. For definability of the primary key (UNIQUE specifiability) for row-partitioned tables, see *Table 3-22 Specifiability of UNIQUE in conjunction with row-partitioning of a table*.
10. The total length of columns comprising an index must satisfy the following

formula:

Total length of columns

≤ MIN((*page size of index storage RDAREAs* ÷ 2) - 1242, 4036)

column-name

Specifies the name of a column for which the primary key is to be defined.

The column names that comprise the primary key must all be unique. When multiple columns are defined for the primary key, the key values are created in the order in which the columns are specified.

`ASC`

Specifies that the primary key index is to be created in ascending order.

`DESC`

Specifies that the primary key index is to be created in descending order.

index-storage-RDAREA-name

Specifies the name of the RDAREA that stores the index for the primary key.

1. If *index-storage-RDAREA-name* is omitted, the index also is stored in the RDAREA that stores the defined table. However, for a row-partitioned table on HiRDB/Single Server, or for a row-partitioned table on the same back-end server on HiRDB/Parallel Server, indexes are stored as follows:

   For a single-column-partitioned table:

   The RDAREA that stores the defined table also stores the index.

   For a multicolumn-partitioned table:

   The multicolumn-partitioned table is stored in the first table storage RDAREA for which partitioning conditions are specified, unless all columns that are specified as partitioning keys are specified in the same order from the beginning of the primary key.

   For a matrix-partitioned table:

   The table cannot be defined unless all columns that are specified as partitioning keys are specified in the same order from the beginning of the primary key. If all columns that are specified as partitioning keys are not specified from the beginning of the primary key in the same order, specify the name of an index storage RDAREA.

For an explanation of index storage RDAREA names and rules regarding those names, see the explanation of index storage RDAREA names in `PRIMARY KEY` of the single-column uniqueness constraint definition.

matrix-partitioned-index-storage-RDAREA-specification::=*two-dimensional-storage-RDAREA-specification*

This operand is specified when defining a matrix-partitioned table.

For details about how to specify this, see the explanation of two-dimensional storage RDAREA names in *PARTITIONED BY MULTIDIM*. For details about the RDAREA name to be specified, see the explanation of index storage RDAREA names for `[{UNIQUE|PRIMARY}] CLUSTER KEY` for the multicolumn uniqueness constraint definition.

21) *multicolumn-check-constraint-definition*::= `CHECK(`*search-condition*`)`

Specify this operand when defining a check constraint for multiple columns.

search-condition

Specifies the condition under which multiple columns are restricted. If this condition is `FALSE`, no insertions or updates can be performed on the table.

In *search-condition*, specify the column that was specified in the table definition.

For restrictions in a search condition, see *search-condition* in *18) Single-column-check-constraint-definition*.

22) *multicolumn-referential-constraint-definition*::= `FOREIGN KEY(`*column-name*[`,` *column-name*]...`)` *reference-specification*

Specify this operand when defining multiple columns as a foreign key.

The following rules apply to foreign keys:

1. When a foreign key is specified, the following data types cannot be specified for columns that compose the foreign key:
   - `BLOB`
   - `BINARY`
   - Abstract data type
2. A repetition column cannot be specified for a column that composes a foreign key.
3. The foreign key and the primary key that is referenced must be identical in all of the following items:
   - Corresponding data type
   - Corresponding data length
   - Number of columns

   If more than one column is specified as a foreign key, HiRDB checks the

correspondence between columns in the order in which they are specified.

`FOREIGN KEY (`*column-name*`[, `*column-name*`]...) `*reference-specification*

Specifies the names of columns that compose a foreign key.

A maximum of 16 column names can be specified.

All specified column names must be distinct.

23) *storage-condition* `::=`

column-name {= |< > | ^= |!= |< |<= | > | > =} {*literal* |(*literal*[`,` *literal*])}

Specifies conditions for storing the table in multiple RDAREAs on a split basis (row-partitioning of the table).

The following rules apply to the storage conditions:

1. Multiple literals can be specified only when the = comparison operator is used.
2. When multiple literals are specified, the same value cannot be specified more than once.
3. Columns with the following data types can be used in comparison operations:
   - `INTEGER`
   - `SMALLINT`
   - `DECIMAL`
   - `FLOAT`
   - `SMALLFLT`
   - `CHARACTER`#1
   - `VARCHAR`#1
   - `NCHAR`#2
   - `NVARCHAR`#2
   - `MCHAR`#1
   - `MVARCHAR`#1
   - `DATE`
   - `TIME`
   - `TIMESTAMP`#4
   - `INTERVAL YEAR TO DAY`#3

- INTERVAL HOUR TO SECOND[#3]

#1: Comparison operation can be specified only on columns with a defined length not exceeding 255 bytes.

#2: Comparison operation can be specified only on columns with a defined length not exceeding 127 characters.

#3: The DCVALUES column in the SQL_DIV_TABLE data dictionary table (storage assignment condition value) contains corrected values:

Examples

19921225. → 19930025.

99981315. → 99990115.

#4: This item cannot be specified if the fractional second precision is greater than 0. Also, it cannot be specified if a default value including USING BES is specified in the DEFAULT clause.

4. The following items cannot be specified in *literal*:
   - Character string literals, national character string literals, or mixed character string literals with a length of 0.
   - Character string literals with a length of 256 bytes or greater, national character string literals with a length of 128 characters or greater, or mixed character string literals with a length of 256 bytes or greater
   - Hexadecimal character string literals
5. The value specified in column-name cannot be updated.
6. The number of literals specified in each storage condition must be such that the total number of literals specified in all storage conditions does not exceed 5,000. If a storage condition is omitted, the number of literals used is counted as 1.
7. If a cluster key is specified for a column, storage conditions cannot be specified for any other columns.
8. The column specified in column-name must be NOT NULL (NOT NULL constraint, FIX specification, or a cluster key).
9. When a multicolumn cluster key is specified, a storage condition cannot be specified for columns other than the leading column.
10. A storage condition cannot be specified for a repetition column.

24) *hash-function -name* : : =

{HASH1 |HASH2 |HASH3 |HASH4 |HASH5 |HASH6 |HASH0 |HASHA |HASHB |HASHC |HASHD |HASHE |HASHF}

If the table is not a rebalancing table, or when a hash function is specified in the second dimension of a matrix-partitioned table:

Specify one of the hash functions HASH1 through HASH6 or HASH0.

HASH6 is usually specified, because it provides the most uniform hashing. However, some partitioning key data is not amendable to uniform hashing, in which case another hash function should be specified.

Specify HASH0 to use the year and month values to rotate the data storage destination RDAREA every month.

Rebalancing table:

Specify one of the hash functions HASHA to HASHF.

HASHF is usually specified, because it provides the most uniform hashing. However, some partitioning key data is not amendable to uniform hashing, in which case another hash function should be specified.

HASH1, HASHA

This hash function can be used for column hash partitioning for all data types. It hashes by using all bytes# of the data in all the columns specified for partitioning. HASH1 can be specified for columns whose data length is at least 0 bytes.

HASH2, HASHB

This hash function can be used for column hash partitioning for all data types. It hashes by using all bytes# of the data in all the columns specified for partitioning. HASH2 can be specified for columns whose data length is at least 0 bytes.

Specify this if data is not being stored uniformly in an RDAREA when HASH1 or HASHA is specified.

HASH3, HASHC

This hash function can be used only when the data type of the columns specified for partitioning is INTEGER or SMALLINT. Hashing is performed using the last 2 bytes# of each partitioning column. HASH3 can be specified for columns whose data size is at least 2 bytes.

HASH4, HASHD

This hash function can be used only when the data type of the columns specified for partitioning is DATE. Hashing is performed using the first 4 bytes# of each partitioning column. HASH4 can be specified for columns whose data size is at least 4 bytes.

HASH5, HASHE

This hash function can be used only when the data type of the columns specified

for partitioning is TIME. Hashing is performed using the first 3 bytes# of each partitioned column. HASH4 can be specified for columns whose data size is at least 3 bytes.

HASH6, HASHF

This hash function can be used for column hash partitioning for all data types. It is well suited for DECIMAL applications. It hashes by using all data bytes# in all the columns specified for partitioning. HASH6 can be specified for columns whose data length is at least 0 bytes.

HASH0

Specify this hash function to use the year and month values in the partitioning column to rotate and allocate the data storage destination RDAREA every month.

In order to specify this hash function, the partitioning key must be one column and the data type must be DATE, TIMESTAMP, CHAR(8) (CHAR(16) when using the UTF-16 character set), or CHAR(6) (CHAR(12) when using the UTF-16 character set)). For a character format with a length of eight characters, specify YYYYMMDD, and for a character format with a length of six characters, specify YYYYMM.

Use the following range of values for YYYY, MM, and DD:

YYYY: 0001 to 9999 (year)

MM: 01 to 12 (month)

DD: 01 to the last day of the specified month of the specified year (day)

#: For the VARCHAR, MVARCHAR, or C type, hashing is performed by ignoring trailing spaces. If the sign portion is F in the DECIMAL, INTERVAL YEAR TO DAY, or INTERVAL HOUR TO SECOND type, hashing is performed by converting F to C.

25) *reference-specification* : := REFERENCES *referenced-table* [*referential-constraint-operation-specification*]

Specifies the referenced table to be referenced. When specifying a constraint operation, specify *referential-constraint-operation-specification*.

If *reference-specification* is specified, the operation rules in the table below apply to tables with a specified *reference-specification* (referencing table). For a table referencing a table with a specified *reference-specification* (referenced table), the operation rules in *Table 3-34* and *Table 3-35* apply.

Table 3-33: Operation on referencing tables with a reference-specification specification

Operation on foreign key constituent columns	Relationship between rows of a foreign key constituent column and rows of the referenced table referenced by the foreign key		Results
Add (INSERT)	A row having a primary key constituent column value equal to the value of the foreign key constituent column for the row to be inserted exists in the referenced table.		Y
	A row having a primary key constituent column value equal to the value of the foreign key constituent column for the row to be inserted does not exist in the referenced table.	The null value is found in the foreign key constituent column for the row to be inserted.	Y
		The null value is not found in the foreign key constituent column for the row to be inserted.	N
Update (UPDATE)	A row having a primary key constituent column value equal to the value of the updated foreign key constituent column exists in the referenced table.		Y
	A row having a primary key constituent column value equal to the value of the updated foreign key constituent column does not exist in the referenced table.	The null value is found in the updated foreign key constituent column.	Y
		The null value is not found in the updated foreign key constituent column.	N

Legend:

Y: Can be operated on the referencing table.

N: A constraint violation error occurs.

referenced-table::= *table-name*

Specifies the name of the table to be referenced.

The following rules apply to table names:

- Specify the name of the table that has the primary key.
- The table name should be a base table.
- The table owned by the user should be specified.
- The table identifier being defined cannot be specified.

referential-constraint-operation-specification::= {*delete-operation* [*update-operation*] | *update-operation* [*delete-operation*]}

Specifies an operation that is performed in synchronization with the updating or deletion of the primary key.

delete-operation::= ON DELETE *reference-operation*

Specifies the operation to be performed when a row in the referenced table is deleted.

update-operation::= ON UPDATE *reference-operation*

Specifies the operation to be performed when a row in the referenced table is updated.

reference-operation

CASCADE

Specify this operand when an operation on the primary key is to be communicated to a foreign key to maintain data integrity.

The following table describes operations that are allowed on referenced tables and the influence of those operations on the referencing table.

Table 3-34: Allowable operations on referenced tables with CASCADE specification, and the influence of those operations on the referencing table

Operations on the primary key constituent column of the referenced table referenced by the foreign key constituent column	Relationship between rows of a foreign key constituent column and rows of the referenced table referenced by the foreign key	Impact on referencing table
Delete (DELETE)	A row having a foreign key constituent column value equal to the value of the primary key constituent column for the row to be deleted exists in the referencing table.	Deletes row
	A row having a foreign key constituent column value equal to the value of the primary key constituent column for the row to be deleted does not exist in the referencing table.	No impact
Update (UPDATE)	A row having a foreign key constituent column value equal to the value of the primary key constituent column for the pre-update row exists in the referencing table.	Updates using a value equal to the primary key
	A row having a foreign key constituent column value equal to the value of the primary key constituent column for the pre-update row does not exist in the referencing table.	No impact

RESTRICT

Specify this operand to check whether or not operations on the primary key affect the foreign key and to restrict the operations so that data integrity can be maintained.

The following table lists allowable operations on the referenced table.

Table 3-35: Operations on the referenced table with RESTRICT specification

Operations on the primary key constituent column of the referenced table referenced by the foreign key constituent column	Relationship between rows of a foreign key constituent column and rows of the referenced table referenced by the foreign key	Results
Delete (DELETE)	A row having a foreign key constituent column value equal to the value of the primary key constituent column for the row to be deleted exists in the referencing table.	N
	A row having a foreign key constituent column value equal to the value of the primary key constituent column for the row to be deleted does not exist in the referencing table.	Y
Update (UPDATE)	A row having a foreign key constituent column value equal to the value of the primary key constituent column for the pre-update row exists in the referencing table.	N
	A row having a foreign key constituent column value equal to the value of the primary key constituent column for the pre-update row does not exist in the referencing table.	Y

Legend:

Y: Operation can be performed on the referenced table.

N: A restriction violation error occurs.

The default for the reference restriction operation itself is ON DELETE RESTRICT ON UPDATE RESTRICT.

The default for *delete-operation* is ON DELETE RESTRICT; the default for *update-operation* is ON UPDATE RESTRICT.

If CASCADE is specified for the referential constraint operation, HiRDB generates a trigger to perform restriction operations.

The following table lists the names of triggers that are created. All trigger names are 21 bytes in length.

Table 3-36: Names of triggers created by HiRDB

Referential constraint operation	Trigger name
delete-operation	(DRA*yyyymmddhhmmssth*)
update-operation	(URA*yyyymmddhhmmssth*)

Note: The *yyyymmddhhmmssth* part of the trigger name is the time stamp when the trigger is generated (it contains information down to 1/100th of a second). When defining consecutive tables for which a reference operation specifies the CASCADE reference restriction, the name of the trigger being created has the same name, and a KFPA11803-E error may occur. In this case, re-execute the table definition that caused the error.

The SQL compile option for a trigger is the same as the default SQL compile option that is in effect in the trigger definition. For details about the default value, see *CREATE TRIGGER (Define a trigger)* in this chapter.

If more than one foreign key is specified, restrictions are performed in the following sequence:

1. CASCADE
2. RESTRICT

If CASCADE is specified more than once, CASCADE restriction is performed in the order in which CASCADE is specified in tables.

If RESTRICT is specified more than once, HiRDB determines the order in which restriction is performed so that an optimal restrict check can be performed, and RESTRICT is performed in that order.

26) *constraint-name-definition* ::= CONSTRAINT *constraint-name*

Specify this operand when defining a constraint name for a specified constraint.

constraint-name

Duplicate constraint names cannot be specified in a given schema.

If *constraint-name* is omitted, HiRDB assigns a default constraint name.

The following table lists constraint names that HiRDB assigns by default:

Table 3-37: Default constraint names assigned by HiRDB

Type		Constraint name	Notes
Referential constraint	Single column check constraint definition	The name of the column for which a constraint is specified	None
	Multicolumn referential constraint definition	The first column name specified in the foreign key	None
Check constraint		CK_*table-number_yyyymmddhhmmssth*	30 characters, fixed (table number: 10 characters, time: 16 characters)

Note

The *yyyymmddhhmmssth* part of the constraint name is the time stamp when the constraint is defined (containing information up to 1/100 second).

The table number is 10 characters long, right justified, and zero filled on the right.

Constraint names specified by users carry the potential for duplication. Therefore, constraint names in the above format should not be specified.

27) `WITH PROGRAM`

When defining a foreign key, specify this operand to disable the applicable function, procedure, or an SQL object for which a trigger is enabled. If a foreign key is not defined, any `WITH PROGRAM` specification is ignored. The following table lists objects that are disabled by this operand.

Table 3-38: Disabled objects

Version used to create object	Object description	
	Object type	Disabling condition
07-00 or later	Function, procedure, and trigger objects	When the object contains an `UPDATE` or `DELETE` statement that uses a table specified in `REFERENCES`
Before 07-00		When the object contains an SQL table that uses a table specified in `REFERENCES`

Common rules

1. Up to 500 sequence generators can be defined in an RDAREA for a table.
2. Columns for which a cluster key is defined cannot be updated.

   However, if a column for which a cluster key is defined contains a variable-length

character type column with a defined length of 256 bytes or greater, the column can be updated. If the cluster key is updated, the updated row can lose its clustering effect.

3. The null value cannot be inserted into a table for which a cluster key is defined.
4. Columns that belong to a cluster key cannot be updated.
5. An RDAREA that is already assigned to a BLOB column or BLOB attribute cannot be specified as a LOB column storage RDAREA or LOB attribute storage RDAREA.
6. The same columns cannot be specified as constituent columns in the CLUSTER KEY clause and the PRIMARY KEY clause. To define the same columns as cluster key and primary key constituents, specify the PRIMARY CLUSTER KEY clause. In this context, *same columns* means columns that satisfy all the following conditions:
   - The lists of column names specified in CLUSTER KEY and PRIMARY KEY clauses and number of specified columns are identical.
   - Either the ascending order/descending order specifications are all in agreement or they are all in reverse.
7. RDAREAs using the inner replica facility and those not using the facility cannot be specified on a mixed basis in the table storage RDAREAs, LOB column storage RDAREAs, LOB attribute storage RDAREAs, or index storage RDAREAs. When specifying an RDAREA to which the inner replica facility is applied, specify the name of the original RDAREA.
8. For execution conditions for CREATE TABLE using the inner replica facility, see the manual *HiRDB Version 9 Staticizer Option*.
9. HiRDB Dataextractor and HiRDB Datareplicator should not be used to affect data on the following tables:
   - Falsification-prevented tables
   - Tables containing columns for which NOT NULL-constrained and SYSTEM GENERATED are specified
10. If a cluster key or the primary key is defined, the index identifier for the index being defined is determined according to the rules listed in the following table:

Table 3-39: Index identifier that is defined

Specification item	Index identifier
CLUSTER	(CLUSTER table number)
PRIMARY	(PRIMARY table number)

Specification item	Index identifier
PRIMARY CLUSTER	(PRI-CLS table number)

Note

The table number part is a value consisting of 10 characters, right-justified and zero-filled on the left.

A fixed value is specified as a total of 19 characters (of which 9 characters are the parentheses and the characters shown previously).

11. A maximum of 30,000 columns can be specified per table.

The sum of the column lengths (data lengths) must satisfy the formulas shown below.

The table below lists the lengths of the columns (data lengths).

- Table without FIX specification (table manipulation)

$$\sum_{i=1}^{n}(2+ai) \leq \text{Page length in storage RDAREA} - 56$$

(*ai*: Column length; *n*: Number of columns as indicated above)

- Table with FIX specification (table definition)

$$\sum_{i=1}^{n} a_i < \left\downarrow \frac{\text{Page length in storage RDAREA}}{1000} \right\downarrow \times 1000$$

(a_i: Column length; *n*: Number of columns indicated above)

Table 3-40: Predefined-type data lengths

Classification	Data type and condition	Data length (in bytes)
Numeric data	INTEGER	4
	SMALLINT	2
	LARGE DECIMAL(*m*,*n*)[#1]	$\downarrow m \div 2 \downarrow + 1$[#2]
	FLOAT or DOUBLE PRECISION	8
	SMALLFLT or REAL	4

Classification	Data type and condition			Data length (in bytes)
Character data	CHARACTER(*n*)			*n*[#3]
	VARCHAR(*n*)	*d* ≦ 255	Elements of a repetition column	*d* + 2
			Other than the above	*d* + 1
		d ≧ 256		6
	VARCHAR(*n*) No-split option specified	*n* ≦ 255	Attributes of abstract data type	*d* + 3
			Elements of a repetition column	*d* + 2
			Other than the above	*d* + 1
		n ≧ 256		6
National character data	NCHAR(*n*) or NATIONAL CHARACTER(*n*)			2 x *n*[#4]
	NVARCHAR(*n*)	*d* ≦ 127	Elements of a repetition column	2 x *d* + 2
			Other than the above	2 x *d* + 1
		d ≧ 128		6
	NVARCHAR(*n*) No-split option specified	*n* ≦ 127	Attributes of abstract data type	2 x *d* + 3
			Elements of a repetition column	2 x *d* + 2
			Other than the above	2 x *d* + 1
		n ≧ 128		6
Mixed character data	MCHAR(*n*)			*n*[#3]
	MVARCHAR(*n*)	*d* ≦ 255	Elements of a repetition column	*d* + 2
			Other than the above	*d* + 1
		d ≧ 256		6
	MVARCHAR(*n*) No-split option specified	*n* ≦ 255	Attributes of abstract data type	*d* + 3
			Elements of a repetition column	*d* + 2
			Other than the above	*d* + 1
		n ≧ 256		6
Date data	DATE			4

Classification	Data type and condition		Data length (in bytes)
Time data	TIME		3
Time stamp data	TIMESTAMP(*n*)		7 + (*n* ÷ 2)
Date interval data	INTERVAL YEAR TO DAY		5
Time interval data	INTERVAL HOUR TO SECOND		4
Large-object data	BLOB		9
Binary data	BINARY(*n*)	*n* ≤ 255	*d* + 3
		n ≥ 256	15

Legend:

d: Actual data length (number of characters)

m, *n*: Positive integers

#1: This is a fixed-point number with a total of *m* digits with *n* decimal places. The default for *m* is 15.

#2: If SUPPRESS DECIMAL is specified as a table option during the definition of the table, the data length is (↓*k* ÷ 2 ↓ + 2), where *k* denotes the number of effective digits at the time of data storage (the number of digits exclusive of the leading zeros). The SUPPRESS DECIMAL option should not be used in the following case, where *a* denotes the total value of data lengths of the columns in the table when either SUPPRESS DECIMAL or the column data suppress specification is not used:

32717 < (*a* + *number-of-columns-in-table* x 2 + 8)

#3: If column data suppression is specified and data is suppressed, *n* is (*n* - *b* + 4). Data suppression is executed only if the last character of column data is a space and if there are 4 or more single-byte spaces that are contiguous with the last character, at the time of the column data suppress specification. *b* denotes the number of spaces that are contiguous with the last character in the column data.

If, however, column data suppression is specified and no data is suppressed, one byte of additional information is added per column.

The column data suppress specification should not be used in the following case, where *a* denotes the total value of the data lengths of the columns in the table when either SUPPRESS DECIMAL or the column data suppress specification is not

used:

32717 < (*a* + *number-of-columns-in-table* x 2 + 8)

#4: If column data suppression is specified and data is suppressed, *2* x *n* is (2 x *n* - 2 x *b* + 5). Data suppression is executed only if the last character of column data is a space and if there are 3 or more double-byte spaces that are contiguous with the last character, at the time of the column data suppress specification. *b* denotes the number of spaces that are contiguous with the last character in the column data.

If, however, column data suppression is specified and no data is suppressed, one byte of additional information is added per column.

The column data suppress specification should not be used in the following case, where *a* denotes the total value of the data lengths of the columns in the table when either `SUPPRESS DECIMAL` or the column data suppress specification is not used:

32717 < (*a* + *number-of-columns-in-table* x 2 + 8)

Rules on referential constraints

1. A given referenced table cannot be referenced from a foreign key for the same foreign key constituent column (even though it may not have the same sorting order).
2. A foreign key cannot be specified for a table that was defined by specifying `WITHOUT ROLLBACK`.
3. If the table defined by specifying `WITHOUT ROLLBACK` contains the primary key, foreign keys that reference the primary key cannot be defined.
4. Foreign keys cannot be specified for a shared table.
5. Foreign keys cannot be specified for a falsification-prevented table.
6. A maximum of 255 foreign keys can be defined per table.
7. A maximum of 255 foreign keys can be defined per primary key.
8. The character set of the foreign key must be the same as the character set of the primary key referenced from the foreign key.

Rules on check constraints

1. A maximum of 254 check constraints can be defined in a table. The maximum allowable sum of Boolean operators (except for `AND` and `OR` in `WHEN` search conditions of `CASE` expressions) specified in check constraints in a table and the number of check constraints is also 254.
2. When specifying more than one condition, Hitachi recommends that check constraints be defined on the basis of separate conditions instead of grouping the

conditions into a single check constraint. In this manner, if a constraint violation arises, the violating condition can easily be determined from a constraint name.

3. A check constraint definition cannot be specified for a falsification-prevented table.
4. If the BLOB type is specified in a search condition in a check constraint, or a BINARY type column with a defined length of 32,001 bytes or greater is specified, the following SQL statement cannot be executed:
   - Updating by concatenation operations in an UPDATE statement of the BLOB type or the BINARY type with a defined length of 32,001 bytes or greater, specified in the search condition in the check constraint.

Notes

1. The CREATE TABLE statement cannot be specified from an X/Open-compliant UAP running under OLTP.
2. PCTFREE=(0,0) must be specified to set 0% for both the table's unused space percentage and the percentage of free pages per segment.
3. To pass or receive date data using CHAR(10) by means of the row-unit interface, the columns must be defined using CHAR(10) instead of the date data type.
4. To pass or receive time data using CHAR(8) by means of the row-unit interface, the columns must be defined using CHAR(8) instead of the time data type.
5. When passing or receiving time stamp data in CHAR in 19, 22, 24, or 26 bytes by using the by-row interface, specify columns in CHAR in 19, 22, 24, or 26 bytes without using the time stamp data type.
6. When you define a falsification-prevented table, Hitachi recommends that only the falsification-prevented table is stored in the table storage RDAREA. If pdrorg terminates abnormally relative to the falsification-prevented table, the affected RDAREA cannot be released from the hold status until the reorganization process is complete. Consequently, if other tables and indexes are stored in the RDAREA storing the falsification-prevented table, those tables and indexes also become unavailable if pdrorg fails.
7. If table definition changes are made for a referencing table specifying CASCADE as a reference operation, the trigger, generated by HiRDB, for performing constraint operations can be disabled in some cases. The trigger is disabled under the following conditions:
   - Condition under which the generated trigger is disabled when the referential constraint operation specification is ON UPDATE CASCADE
     - When table definitions are changed for a referencing table (changing SPLIT for the table or modifying the default for the table)

- When an index is defined for a referencing table
- When the index for a referencing table is deleted
- When a trigger with a trigger timing `UPDATE` is defined for a referencing table
- When a trigger with a trigger timing `UPDATE` defined for a referencing table is deleted
- When a table definition is changed for the primary key constituent column of the referenced table referenced by a referencing table

- Condition that disables the trigger that is generated when the referential constraint operation specification is `ON DELETE CASCADE`
  - When table definitions are changed for a referencing table (changing `SPLIT` for a column or modifying the default for a column)
  - When an index is defined for a referencing table
  - When the index for a referencing table is deleted
  - When a trigger with a trigger timing `DELETE` is defined for a referencing table
  - When a trigger with a trigger timing `DELETE` defined for a referencing table is deleted

  Any disabled trigger should be re-created using `ALTER ROUTINE`.

8. If more than one referential constraint specifying `ON UPDATE CASCADE` as a reference operation is specified, the same table name should not be specified for the referenced table.

   However, the above rule does not apply if all of the following conditions are met:

   - The applicable multiple foreign key constituent columns with a reference specification are not duplicated.
   - Check constraints and referential constraints related to the applicable multiple foreign key constituent elements with a reference specification are not defined.

Examples

1. Define a stock table (`STOCK`):

```
CREATE TABLE STOCK
    (PCODE CHAR(4),PNAME NCHAR(8),
     COLOR NCHAR(1),PRICE INTEGER,SQTY INTEGER)
```

2. Define a stock table (`STOCK`) with the following conditions:

- Table is a FIX table
- Table data is to be stored in a user RDAREA (RDA1)
- Because the inventory table is a fixed-length table without a cluster key, the percentage of free area and the percentage of free pages per segment are both to be 0

```
CREATE FIX TABLE STOCK
    (PCODE CHAR(4),PNAME NCHAR(8),
     COLOR NCHAR(1),PRICE INTEGER,SQTY INTEGER)
  IN RDA1
  PCTFREE=(0,0)
```

3. Define a stock table (STOCK) with the following conditions:
   - The product code column (PCODE) is to be defined as a uniqueness-constrained cluster key
   - The table data and index are to be partitioned into three RDAREAs, each with the following storage conditions:

Storage conditions	Storage RDAREAs	
	Table data	Index
101M≤PCODE≤202M	RDA1	RDA4
302S≤PCODE≤412M	RDA2	RDA5
591L≤PCODE≤591S	RDA3	RDA6

```
CREATE TABLE STOCK
   (PCODE CHAR(4)
       UNIQUE CLUSTER KEY ASC
       IN ((RDA4),(RDA5),(RDA6)),
   PNAME NCHAR(10),
   COLOR NCHAR(5),
   PRICE INTEGER,
   SQTY INTEGER)
   IN ((RDA1)PCODE<='202M',
     (RDA2)PCODE<='412M',
     (RDA3))
```

4. Define a stock table (STOCK) with the following conditions:
   - The product name column (PNAME) and the color column (COLOR) are to be defined as a cluster key, and the index for the cluster key is to be sorted in ascending order of the product names and descending order of the colors
   - The table data is to be stored in a user RDAREA (RDA1)

- The index is to be stored in a user RDAREA (RDA2)

```
CREATE TABLE STOCK
   (PCODE CHAR(4),PNAME NCHAR(10),
    COLOR NCHAR(5),PRICE INTEGER,SQTY INTEGER)
 IN RDA1
 CLUSTER KEY (PNAME ASC,COLOR DESC) IN RDA2
```

5. Define an employee table containing the abstract data type t_EMPLOYEE.

```
CREATE TABLE STAFF_TABLE
  (EMPLOYEENO INTEGER NOT NULL,
   DOCUMENT_DATA_BLOB (6000) IN ((LRDA1),(LRDA2)),
   EMPLOYEE T_EMPLOYEE ALLOCATE(PHOTOGRAPH_OF_THE_FACE
      IN ((LRDA03),(LRDA04))))
      IN ((RDA1) EMPLOYEE_NO<=700000,(RDA2))
```

6. Define an order table (ORDER) under the following conditions:

   - The table is designated as a falsification-prevented table.
   - The deletion-prevented duration is 10 years.
   - An OINSDATE column is defined as an insert history maintenance column.

```
CREATE TABLE ORDER
   (FNO CHAR(6),CCODE CHAR(5),PCODE CHAR(4),
    OQTY INTEGER,ODATE DATE,OTIME TIME,
    OINSDATE DATE NOT NULL
    WITH DEFAULT SYSTEM GENERATED)
INSERT ONLY WHILE 10 YEARS BY OINSDATE
```

7. Define an inventory table (STOCK) under the following conditions:

   - Store the table as a matrix-partitioned table on a partitioned basis in 6 user RDAREAs.
   - The following storage conditions apply:

Storage condition	Storing RDAREA
PCODE ≤ 202M AND PRICE ≤ 5000	RDA1
PCODE ≤ 202M AND PRICE > 5000	RDA2
202M < PCODE ≤ 412M AND PRICE ≤ 5000	RDA3
202M < PCODE ≤ 412M AND PRICE > 5000	RDA4
PCODE > 412M AND PRICE ≤ 5000	RDA5
PCODE > 412M AND PRICE > 5000	RDA6

```
CREATE TABLE STOCK
```

```
   (PCODE CHAR(4) NOT NULL,
    PRICE INTEGER NOT NULL)
PARTITIONED BY MULTIDIM (PCODE(('202M'),('412M')),
                         PRICE((5000)))
IN ((RDA1,RDA2),(RDA3,RDA4),(RDA5,RDA6))
```

8. Define an inventory table (`STOCK`) with the following conditions:

   - In the product code (`PCODE`) column, define a check constraint so that data cannot be inserted into the column or the column cannot be updated with a product with a size other than `S`, `M`, or `L`:

```
CREATE TABLE STOCK
    (PCODE CHAR(4) CONSTRAINT CHECK_SIZE
    CHECK(PCODE LIKE '%S' OR PCODE LIKE '%M
    OR PCODE LIKE '%L'),
    PNAME NCHAR(8),
    COL NCHAR(1),PRICE INTEGER,SQTY INTEGER)
```

9. Define a referential constraint for a single column:

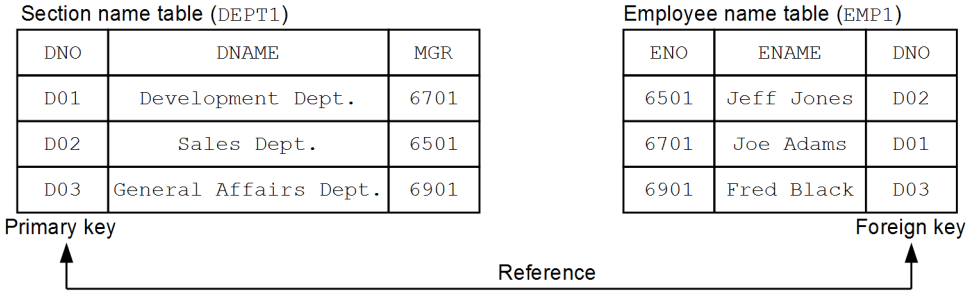
   - If a row in a parts name table is deleted, also delete the corresponding row in the name table.

     Define a referenced table (parts name table (`DEPT1`)), and designate the parts code (`DNO`) column as the primary key:

```
CREATE TABLE DEPT1
    (DNO CHAR(3) PRIMARY KEY,DNAME NVARCHAR(20),MGR
CHAR(8))
```

   Define a referencing table (employee name table (`EMP1`)), and designate the parts code (`DNO`) column as a foreign key.

```
CREATE TABLE EMP1
    (ENO CHAR(8),ENAME NVARCHAR(25),
     DNO CHAR(3)
         CONSTRAINT EMP1_K
         REFERENCES DEPT1 ON DELETE CASCADE)
```



10. Define a referential constraint for multiple columns.

    - If a row in a section name table is deleted and a row in the name table contains the same value as the section name table row being deleted, the following code suppresses the deletion of the section name table. Similarly, if a row in the section name table is to be updated and a row in the name table contains the same value as the section name table row being updated, the following code suppresses the updating of the section name table.

      Define a referenced table (section name table (DEPT2)), and designate the parts code (DNO) column and the section code (SNO) column as the primary key.

```
CREATE TABLE DEPT2
     (DNO CHAR(3),SNO CHAR(3),
     PNAME CHAR(20),SHEAD CHAR(8),
     PRIMARY KEY(DNO,SNO))
```

    Define a referencing table (employee name table (EMP2)), and designate the parts code (DNO) column and the section code (SNO) column as foreign keys:

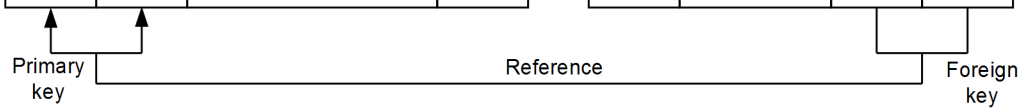
```
CREATE TABLE EMP2
     (ENO CHAR(8),ENAME CHAR(25),
     DNO CHAR(3),SNO CHAR(3),
     CONSTRAINT EMP2_K
     FOREIGN KEY(DNO,SNO) REFERENCES DEPT2)
```

Section name table (DEPT2)

DNO	SNO	SNAME	SHEAD
D01	S11	Development Dept.	6701
D02	S21	Sales Dept.	6501
D03	S31	General Affairs Dept.	6901

Employee name table (EMP2)

ENO	ENAME	DNO	SNO
6501	Jim Davis	D01	S11
6702	Adam Smith	D03	S31
6903	Sam Spade	D02	S21



CREATE TRIGGER (Define a trigger)

Function

Upon operation on a specified table (INSERT, UPDATE, and DELETE statements), CREATE TRIGGER defines an action (trigger) that automatically executes SQL statements.

Privileges

A user can define a trigger on tables that he or she owns.

Format

```
CREATE TRIGGER [authorization-identifier.]trigger-identifier
  trigger-action-time
  trigger-event
  ON [authorization-identifier.]table-identifier
  [REFERENCING old-or-new-values-alias-list]
  trigger-action
  [SQL-compile-option[SQL-compile-option]...]
  [WITH PROGRAM]

trigger-action-time ::={BEFORE|AFTER}
trigger-event ::={INSERT|DELETE|UPDATE[OF column-name[, column-name]...]}
old-or-new-values-alias-list ::=old-or-new-values-alias[old-or-new-values-alias]
old-or-new-values-alias ::={OLD [ROW] [AS] old-values-correlation-name
    |NEW [ROW] [AS] new-values-correlation-name}
old-values-correlation-name, new-values-correlation-name ::=correlation-name
trigger-action ::=[{FOR EACH ROW|FOR EACH STATEMENT}]
    [WHEN (search-condition)]
    trigger-SQL-statement
  trigger-SQL-statement ::=SQL-procedure-statement
SQL-compile-option ::={ISOLATION data-guarantee-level [FOR UPDATE EXCLUSIVE]
    |OPTIMIZE LEVEL SQL-optimization-option
     [, SQL-optimization-option]...
     |ADD OPTIMIZE LEVEL SQL-extension-optimizing-option
     [, SQL-extension-optimizing-option]...
|SUBSTR LENGTH maximum-character-length }
```

Operands

- [*authorization-identifier*.]*trigger-identifier*

authorization-identifier

Specifies the authorization identifier of the owner of the trigger being defined. The default is the authorization identifier of the user who executes CREATE TRIGGER.

trigger-identifier

Specifies the name of the trigger being defined.

- *trigger-action-time*::={BEFORE|AFTER}

BEFORE

Executes the trigger action before an operation is performed on the table.

If BEFORE is specified, function calls other than data update SQL statements (INSERT, UPDATE, or DELETE statements), the CALL statement, and the default constructor function cannot be specified in *trigger-SQL-statement*.

The trigger for which BEFORE is specified in *trigger-action-time* is called a *BEFORE trigger*.

AFTER

Executes the trigger action after an operation is performed on the table.

The trigger for which AFTER is specified in *trigger-action-time* is called an *AFTER trigger*.

- *trigger-event*::={INSERT|DELETE|UPDATE[OF *column-name*[, *column-name*]...]}

Specifies the type of operation that induces the execution of the trigger.

INSERT

Executes a trigger when a row is inserted into the table. The trigger for which INSERT is specified in *trigger-event* is called an *INSERT trigger*.

If INSERT is specified in *trigger-event*, OLD[ROW] [AS] *old-values-correlation-name* cannot be specified in *old-or-new-values-alias*.

DELETE

Executes a trigger when a row is deleted from the table. The trigger for which DELETE is specified in *trigger-event* is called a *DELETE trigger*.

If DELETE is specified in *trigger-event*, NEW[ROW] [AS] *new-values-correlation-name* cannot be specified in *old-or-new-values-alias*.

UPDATE

Executes a trigger when a row is updated in the table. The trigger for which UPDATE is specified in *trigger-event* is called an *UPDATE trigger*.

If UPDATE is specified in *trigger-event*, a trigger is executed even if the value does not change before and after the update, provided that the trigger action conditions are satisfied.

OF *column-name*[, *column-name*]...

For executing a trigger upon updating of a specific column, specify `OF` *column-name*[, *column-name*] . . . . The columns specified here are called *trigger event columns*.

The following rules apply to trigger event columns:

1. In *column-name*, specify the column name of the table for which a trigger is defined.
2. Column names cannot be specified in duplicate.
3. If a repetition column is specified, a trigger is executed when an `UPDATE` statement containing only an `ADD` or `DELETE` clause is executed for the column.
4. If a trigger event column is omitted, the default is all column names for the target table (including columns that are added after the trigger is defined). The trigger is executed when an `UPDATE` statement containing only an `ADD` or `DELETE` clause is executed.

- [*authorization-identifier*.]*table-identifier*

Specifies the table name of the base table for which a trigger is being defined.

A user can define a trigger only for the tables that he or she owns. A trigger cannot be defined for a view table.

- `REFERENCING` *old-or-new-values-alias-list*::=*old-or-new-values-alias* [*old-or-new-values-alias*]

When referencing a row before and after an update as part of a trigger definition, specifies an alias.

In *old-or-new-values-alias*, either `OLD[ROW] [AS]` *old-values-correlation-name* or `NEW[ROW] [AS]` *new-values-correlation-name* can be specified only once.

- *old-or-new-values-alias*::={`OLD [ROW] [AS]` *old-values-correlation-name*|`NEW [ROW] [AS]` *new-values-correlation-name*}

  old-values-correlation-name::=*correlation-name*

  new-values-correlation-name::=*correlation-name*

This option is specified when referencing a row either before or after updating by assigning a name to it.

The following rules apply to old or new values aliases:

1. `ROW` and `AS` have the same effect, irrespective of whether or not they are specified.
2. The same correlation name cannot be specified in both *old-values-correlation-name* and *new-values-correlation-name*.

3. The scope of a correlation name specified in either *old-values-correlation-name* or *new-values-correlation-name* is the entire trigger definition.
4. If a column qualified with *new-values-correlation-name* is updated by using a BEFORE trigger, the update takes effect in the table. However, a BEFORE trigger, cannot update any of the following columns qualified with *new-values-correlation-name*; an attempt to update such a column may produce a runtime error:

   - Columns for which SYSTEM GENERATED is specified

   The BEFORE trigger for which the trigger event is INSERT cannot update any of the following columns by qualifying them with *new-values-correlation-name*; an attempt to update such a column may produce a runtime error:

   - Columns that are specified in the partitioning key for a partitioned table (exclusive of flexible hash-partitioned tables)

   For the BEFORE trigger, even when the updating is performed by qualifying a name with *new-values-correlation-name*, the insertion value (trigger event: INSERT) or the update value (trigger event: UPDATE) before updating must be a value that can be inserted into a specified column or a value that can be used for updating the column. For example, NULL cannot be specified as an insertion or update value before updating by means of a trigger action if the specified column is NOT NULL-constrained. With regard to uniqueness constraints, however, insertion and updating can be performed if the value that is updated by a trigger action satisfies the uniqueness constraint.
5. Repetition columns and abstract data-type columns cannot be referenced by qualifying them with *new-values-correlation-name* and *old-values-correlation-name* (in a trigger definition, repetition columns and abstract data-type columns cannot be referenced in the table in which a trigger is defined).
6. ROW cannot be specified in *old-values-correlation-name* or in *new-values-correlation-name*.

OLD [ROW] [AS] *old-values-correlation-name*

This operand is specified when referencing a row, before updating it, by assigning a name to it.

The value stored in the column qualified with *old-values-correlation-name* is a value that was in effect before the SQL statement that caused the trigger was executed. For the UPDATE statement, it is a pre-update value; for the DELETE statement, it is the column value of the row to be deleted.

NEW [ROW] [AS] *new-values-correlation-name*

This operand is specified when referencing a row, after updating it, by assigning

a name to it.

The value stored in the column qualified with *new-values-correlation-name* is a value that is the result of execution of the SQL statement that caused the trigger. For the UPDATE statement, it is a post-update value; for the INSERT statement, it is the inserted value. If, however, a column qualified with *new-values-correlation-name* is updated during the trigger event, the updated value is inherited.

- *trigger-action* : : =

  [{FOR EACH ROW|FOR EACH STATEMENT}]

  [WHEN (*search-condition*)]

  trigger-SQL-statement

FOR EACH ROW

This option is specified when executing a trigger by updated row. The trigger for which FOR EACH ROW is specified in *trigger-action* is called a *trigger by row*. The FOR EACH ROW option causes the execution of the trigger each time the one row in the table is updated.

FOR EACH STATEMENT

This option is specified when executing a trigger by SQL statement. The trigger for which FOR EACH STATEMENT is specified in *trigger-action* is called a *trigger by statement*.

FOR EACH STATEMENT causes the execution of the trigger by each SQL statement, in which case the trigger is executed even where there are no rows to be updated.

FOR EACH STATEMENT cannot be specified together with *old-or-new-values-alias-list*.

search-condition

When a trigger event occurs, the trigger SQL statement is executed if the condition specified here is true. Such a search condition is called a *trigger action condition*.

If the operand *trigger-action-condition* is omitted, the trigger SQL statement is always executed when a specified trigger event occurs. Any of the following items cannot be specified in the *trigger-action-condition* operand:

- Subqueries
- Set function or SQL/XML set function
- Window function
- Repetition columns

- Embedded variables, `?` parameters, SQL variables, and SQL parameters
- Columns of any of the following data types:
  - `BLOB` (however, the item can be specified in the scalar functions `LENGTH` and `POSITION`, and in function calls for a user-defined function).
  - `BINARY` with a maximum length of 32,001 bytes or greater (however, the item can be specified in the scalar functions `LENGTH` and `POSITION`, and in function calls for a user-defined function).
  - Abstract data type
- Structured repetition predicates
- Plug-in functions
- Value expressions producing a result that is an abstract data type (not specifiable in a value expression)

  When referencing a column of the table for which a trigger is defined in a trigger action condition, qualify the column with *old-or-new-values-correlation-name* (non-qualified columns cannot be specified, and columns qualified with a table name cannot be specified).
- XML constructor function
- SQL/XML scalar function

trigger-SQL-statement

Specifies an SQL procedure statement. For details about SQL procedure statements, see *7. Routine Control SQL*.

SQL procedure statements that are specified as a trigger SQL statement are subject to the following restrictions:

1. The table name of a table for which a trigger is defined cannot be specified.
2. Columns in the table for which a trigger is defined cannot be specified either without qualifying them or by qualifying them with a table name (columns qualified with an old- or new-values alias can be specified). A column qualified with an old- or new-values correlation name can be specified in the trigger SQL statement in the same locations as locations where an SQL parameter can be specified in an SQL statement. For details about locations where an SQL parameter can be specified, see *1.6 Embedded variables, indicator variables, ? parameters, SQL parameters, and SQL variables*. Such a column, however, cannot be specified in the `LIMIT` clause.
3. `ROLLBACK`, `COMMIT`, and `PURGE TABLE` statements cannot be specified. Calling a procedure specifying any of these statements using a `CALL` statement can produce a runtime error.

4. JAVA stored procedures and the GET_JAVA_STORED_ROUTINE_SOURCE specification cannot be specified. Calling a procedure specifying a JAVA stored procedure using a CALL statement can produce a runtime error.

■ *SQL-compile-option*::={ISOLATION *data-guarantee-level* [FOR UPDATE EXCLUSIVE]

|OPTIMIZE LEVEL *SQL-optimization-option*

[, *SQL-optimization-option*]...

|ADD OPTIMIZE LEVEL *SQL-extension-optimizing-option*

[, *SQL-extension-optimizing-option*]...

|SUBSTR LENGTH *maximum-character-length* }

In *SQL-compile-option*, ISOLATION, OPTIMIZE LEVEL, ADD OPTIMIZE LEVEL, and SUBSTR LENGTH can each be specified only once.

[ISOLATION *data-guarantee-level*] [FOR UPDATE EXCLUSIVE]]

Specifies an SQL data integrity guarantee level.

data-guarantee-level

A data integrity guarantee level specifies the point to which the integrity of the transaction data must be guaranteed. The following data integrity guarantee levels can be specified:

- 0

  This option is specified when the integrity of data is not to be guaranteed. Level 0 permits the referencing of data even when the data is being updated by another user, without waiting for completion of the update process. However, if the table to be referenced is a shared table and another user is executing the LOCK statement in the lock mode, the system waits until the lock condition is released.

- 1

  This option is specified when the integrity of data is to be guaranteed until the end of the retrieval process. Level 1 prevents other users from updating retrieved data until the retrieval process is completed (until HiRDB finishes viewing the page or row).

- 2

  This option is specified when the integrity of retrieved data is to be guaranteed until the end of a transaction. Level 2 prevents other users from updating retrieved data until termination of the transaction.

[FOR UPDATE EXCLUSIVE]

Specify this operand if WITH EXCLUSIVE LOCK is always to be assumed, irrespective of the data guarantee level specified in *SQL-compile-option* for a cursor or query in a procedure for which the FOR UPDATE clause is specified or assumed. If level 2 is specified in *data-guarantee-level*, WITH EXCLUSIVE LOCK is assumed for the cursor or query in a procedure for which the FOR UPDATE clause is specified or assumed, in which case it is not necessary to specify FOR UPDATE EXCLUSIVE.

Relationship to the client environment definition

PDISLLVL and PDFORUPDATEEXLOCK, if specified on CREATE TRIGGER, have no effect.

Relationship to SQL statements

If a lock option is specified in an SQL statement in a procedure, the lock option specified in the SQL statement takes precedence over any data guarantee level specified in *SQL-compile-option* or the lock option assumed due to FOR UPDATE EXCLUSIVE.

The default for this operand is Level 2.

For data guarantee levels, see the *HiRDB Version 9 UAP Development Guide*.

[OPTIMIZE LEVEL *SQL-optimization-option*[, *SQL-optimization-option*]...]

Specifies an optimization method for determining the most efficient access path by taking the condition of the database into consideration.

An SQL optimization option can be specified using either an identifier (character string) or a numeric value. For most cases, Hitachi recommends the use of an identifier.

Specifying with an identifier

OPTIMIZE LEVEL "*identifier*"[, "*identifier*"]...

Specification examples

- Applying prioritized nest-loop-join and rapid grouping processing:

  OPTIMIZE LEVEL "PRIOR_NEST_JOIN", "RAPID_GROUPING"

- Applying no optimization:

  OPTIMIZE LEVEL "NONE"

Rules

1. Specify one or more identifiers.
2. When specifying two or more identifiers, delimit them with commas

(,).

3. For details about the contents that can be specified in an identifier (optimization methods), see *Table 3-41 SQL optimization option specification values (CREATE TRIGGER)*.
4. If no optimization is to be applied, specify NONE in *identifier*. If an identifier other than NONE is specified at the same time, NONE is nullified.
5. Identifiers can be specified in both lower case and upper case characters.
6. If the same identifier is specified more than once, it is treated as if it was specified only once; however, when possible, precautions should be taken to avoid specifying a given identifier in duplicate.

Specifying with a numeric value

```
OPTIMIZE LEVEL unsigned-integer[, unsigned-integer]...
```

Specification examples

- Making multiple SQL objects, suppressing the use of AND multiple indexes, and forcing the use of multiple indexes

  For specifying unsigned integers by delimiting them with commas:

  ```
  OPTIMIZE LEVEL 4, 10, 16
  ```

  For specifying the sum of unsigned integers:

  ```
  OPTIMIZE LEVEL 30
  ```

- Adding a new value, 16, with the value 14 (4 + 10) already specified:

  ```
  OPTIMIZE LEVEL 14, 16
  ```

- Applying no optimization:

  ```
  OPTIMIZE LEVEL 0
  ```

Rules

1. When HiRDB is upgraded from a version earlier than Version 06-00 to a Version 06-00 or later, the total value specification in the earlier version also remains valid. If the optimization option does not need to be modified, the specification value for this operand need not be changed when HiRDB is upgraded to a Version 06-00 or later.
2. Specify one or more unsigned integers.

3. When specifying two or more unsigned integers, separate them with commas (,).
4. For details about the contents that can be specified in an unsigned integer (optimization methods), see *Table 3-41 SQL optimization option specification values (CREATE TRIGGER)*.
5. When not applying any optimization, specify `0` in *unsigned-integer*. However, specifying an identifier other than 0 at the same time nullifies the `0`.
6. If the same unsigned integer is specified more than once, it is treated as if it was specified only once; however, when possible, precautions should be taken to avoid specifying a given unsigned integer in duplicate.
7. When specifying multiple optimization methods, you can specify the sum of their unsigned integers. However, care should be taken not to add the value of the same optimization method multiple times (to prevent the possibility of the resulting sum from being interpreted as a separate optimization method).
8. To specify multiple optimization methods by adding their values, Hitachi recommends to separate each optimization method specification with a comma to avoid ambiguities regarding which optimization method is being specified. If a new optimization method needs to be specified after multiple optimization methods have been specified by adding their values, specify the new value by appending it, separated with a comma.

Relationship to system definitions

1. The default for the SQL optimization option is the value specified in the `pd_optimize_level` operand of the system definitions. For details about the `pd_optimize_level` operand, see the manual *HiRDB Version 9 System Definition*.
2. When the `pd_floatable_bes` operand or the `pd_non_floatable_bes` operand is specified, specification of the *Increasing the target floatable servers (back-end servers for fetching data)* option or the *Limiting the target floatable servers (back-end servers for fetching data)* option, respectively, is invalid.
3. When `KEY` is specified in the `pd_indexlock_mode` operand of the system definition (i.e., for index key value lock), specification of the *Suppressing creation of update-SQL work tables* option is invalid.

Relationship to the client environment definition

Specification of `PDSQLOPTLVL` has no effect on `CREATE TRIGGER`.

Relationship with SQL

If SQL optimization is specified in an SQL statement, the SQL optimization specification takes precedence over SQL optimization options. For SQL optimization specifications, see *2.24 SQL optimization specification*.

SQL optimization option specification values

The following table lists the SQL optimization option specification values. For details about optimization methods, see the *HiRDB Version 9 UAP Development Guide*.

Table 3-41: SQL optimization option specification values (CREATE TRIGGER)

No.	Optimization method	Specification value	
		Identifier	Unsigned integer
1	Forced nest-loop-join	"FORCE_NEST_JOIN"	4
2	Making multiple SQL objects	"SELECT_APSL"	10
3	Increasing the target floatable servers (back-end servers for fetching data)#1, #2	"FLTS_INC_DATA_BES"	16
4	Prioritized nest-loop-join	"PRIOR_NEST_JOIN"	32
5	Increasing the number of floatable server candidates#2	"FLTS_MAX_NUMBER"	64
6	Priority of OR multiple indexes	"PRIOR_OR_INDEXES"	128
7	Group processing, ORDER BY processing, and DISTINCT set function processing at the local back-end server#2	"SORT_DATA_BES"	256
8	Suppressing the use of AND multiple indexes	"DETER_AND_INDEXES"	512
9	Rapid grouping processing	"RAPID_GROUPING"	1024
10	Limiting the target floatable servers (back-end servers for fetching data)#1, #2	"FLTS_ONLY_DATA_BES"	2048
11	Separating data collecting servers#1, #2	"FLTS_SEPARATE_COLLECT_SVR"	2064
12	Suppressing index use (forced table scan)	"FORCE_TABLE_SCAN"	4096
13	Forcing use of multiple indexes	"FORCE_PLURAL_INDEXES"	32768

No.	Optimization method	Specification value	
		Identifier	Unsigned integer
14	Suppressing creation of update-SQL work tables	"DETER_WORK_TABLE_FOR_UPDATE"	131072
15	Derivation of rapid search conditions	"DERIVATIVE_COND"	262144
16	Applying key conditions including scalar operations	"APPLY_ENHANCED_KEY_COND"	524288
17	Facility for batch acquisition from functions provided by plug-ins	"PICKUP_MULTIPLE_ROWS_PLUGIN"	1048576
18	Facility for moving search conditions into derived table	"MOVE_UP_DERIVED_COND"	2097152

#1: If both *Increasing the target floatable servers (back-end servers for fetching data)* and *Limiting the target floatable servers (back-end servers for fetching data)* are specified together, the respective optimization method does not take effect; instead, the specification operates as a separating data collecting server.

#2: When specified on a HiRDB/Single Server, this option has no effect.

[ADD OPTIMIZE LEVEL *SQL-extension-optimizing-option*[, *SQL-extension-optimizing-option*]...]

Specifies an optimization method for determining the most efficient access path by taking the condition of the database into consideration.

An SQL extension optimizing option can be specified using either an identifier (character string) or a numeric value.

Specifying with an identifier

```
ADD OPTIMIZE LEVEL "identifier"[, "identifier"]...
```

Specification examples

- Applying *Optimizing mode 2 based on cost* and *Hash join, subquery hash execution*:

  ```
  ADD OPTIMIZE LEVEL "COST_BASE_2", "APPLY_HASH_JOIN"
  ```

- Applying no optimization:

  ```
  ADD OPTIMIZE LEVEL "NONE"
  ```

Rules

1. Specify one or more identifiers.
2. When specifying two or more identifiers, delimit them with commas (,).
3. For details about the contents that can be specified in an identifier (optimization methods), see *Table 3-42 SQL extension optimizing option specification values (CREATE TRIGGER)*.
4. If no optimization is to be applied, specify `NONE` in *identifier*.
5. Identifiers can be specified in both lower case and upper case characters.
6. If the same identifier is specified more than once, it is treated as if it was specified only once; however, when possible, precautions should be taken to avoid specifying a given identifier in duplicate.

Specifying with a numeric value

```
ADD OPTIMIZE LEVEL unsigned-integer[, unsigned-integer]...
```

Specification examples

- Applying *Optimizing mode 2 based on cost* and *Hash join, subquery hash execution*:

  ```
  ADD OPTIMIZE LEVEL 1, 2
  ```

- Applying no optimization:

  ```
  ADD OPTIMIZE LEVEL 0
  ```

Rules

1. Specify one or more unsigned integers.
2. When specifying two or more unsigned integers, separate them with commas (,).
3. For details about the contents that can be specified in an unsigned integer (optimization methods), see *Table 3-42 SQL extension optimizing option specification values (CREATE TRIGGER)*.
4. When not applying any optimization, specify `0` in *unsigned-integer*.
5. If the same unsigned integer is specified more than once, it is treated as if it was specified only once; however, when possible, precautions should be taken to avoid specifying a given unsigned integer in duplicate.

Relationship to system definitions

The default for the SQL extension optimizing option is the value specified in the pd_additional_optimize_level operand of system definitions. For details about the pd_additional_optimize_level operand, see the manual *HiRDB Version 9 System Definition*.

Relationship to the client environment definition

Specification of PDADDITIONALOPTLVL has no effect on CREATE TRIGGER.

Relationship with SQL

If SQL optimization is specified in an SQL statement, the SQL optimization specification takes precedence over SQL optimization options. For SQL optimization specifications, see *2.24 SQL optimization specification*.

SQL extension optimizing option specification values

The following table lists the SQL optimization option specification values. For details about optimization methods, see the *HiRDB Version 9 UAP Development Guide*.

Table 3-42: SQL extension optimizing option specification values (CREATE TRIGGER)

No.	Optimization method	Specification value	
		Identifier	Unsigned integer
1	Application of optimizing mode 2 based on cost	"COST_BASE_2"	1
2	Hash join, subquery hash execution	"APPLY_HASH_JOIN"	2
3	Facility for applying join conditions including value expression	"APPLY_JOIN_COND_FOR_VALUE_EXP"	32

Note

Items 2 and 3 take effect when *Application of optimizing mode 2 based on cost* is specified.

[SUBSTR LENGTH *maximum-character-length*]

Specifies the maximum number of bytes for representing a single character.

The value specified for the maximum character length must be in the range from 3 to6.

This operand is valid only when utf-8 is specified for the character code type in

the pdntenv command (pdsetup command for the UNIX edition); it affects the length of the result of the SUBSTR scalar function. For details about SUBSTR, see *2.16.1(20) SUBSTR*.

Relationships to system definition

When SUBSTR LENGTH is omitted, the value specified in the pd_substr_length operand in the system definition is assumed. For details about the pd_substr_length operand, see the manual *HiRDB Version 9 System Definition*.

Relationship to client environmental definition

The specification of PDSUBSTREN has no applicability to CREATE TRIGGER. For details about PDSUBSTRLEN, see the manual *HiRDB Version 9 UAP Development Guide*.

Relationship to the character code type specified in the pdntenv or pdsetup command

This operand is valid only when utf-8 is specified for the character code type.

For all other character code types, only a syntax check is performed and the specification is ignored.

- WITH PROGRAM

When defining a trigger, if there is a function, a procedure, or an SQL object for which a trigger is in effect and that uses the table for which the trigger is being defined, specify this option when nullifying the SQL object:

- If *trigger-event* is INSERT, a function, a procedure, or a trigger that inserts rows into the table for which the trigger is being defined
- If *trigger-event* is UPDATE, a function, a procedure, or a trigger that updates rows in the table for which the trigger is being defined
- If *trigger-event* is DELETE, a function, a procedure, or a trigger that deletes rows from the table for which the trigger is being defined

All functions and procedures that created an SQL object in HiRDB of Version 07-00 or earlier are nullified even if they do not meet the above conditions.

Common rules

1. Defining a trigger causes the creation of an SQL object called a *trigger action procedure*, which is stored in the data dictionary LOB RDAREA. For this reason, before defining a trigger, you need to allocate sufficient space in the LOB RDAREA. For details about how to estimate the required space for a data dictionary LOB RDAREA, see the *HiRDB Version 9 Installation and Design Guide*.

2. The routine identifier for the trigger action procedure takes the following name with a length of 22 bytes:

   '(TRIG*yyyymmddhhmmssth*)'

   yyyymmddhhmmssth: trigger-definition-time-stamp (in 10 milliseconds)

3. The specific name for the trigger action procedure is the same as [*authorization-identifier* .] *trigger-action-procedure-routine-identifier*. The trigger action procedure routine identifier is stored in the SPECIFIC_NAME column of the SQL_TRIGGERS dictionary table.

4. The user defining a trigger must have the privilege necessary for the execution of the trigger SQL statement. Whether the user defining a trigger has the requisite privilege is checked when the trigger is defined and executed.

5. If WITH PROGRAM is omitted, a trigger cannot be defined if there is a function or a procedure that uses the table for which the trigger is being defined and needs to modify the SQL object, and there is an SQL object for which a trigger is in effect.

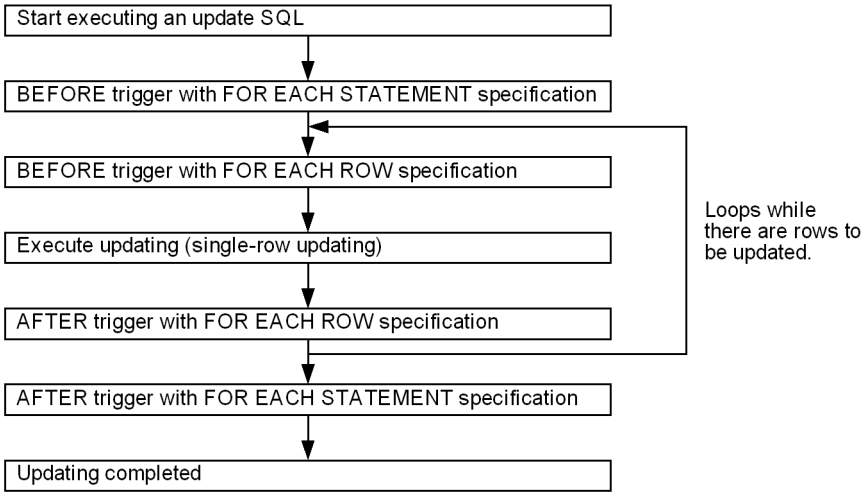
6. When specifying an SQL compile option in ALTER TRIGGER or ALTER ROUTINE, make sure that any SQL statements created by incorporating the SQL compile option into the source CREATE TRIGGER statement of the trigger that is regenerating the SQL object do not exceed the maximum allowable length for SQL statements.

7. If the SQL object being executed becomes nullified, CREATE TRIGGER cannot be executed from within a Java procedure.

Rules on executing a trigger-inducing SQL statement

1. A trigger occurs only on a specified trigger event (UPDATE or DELETE statement); it does not occur on a PURGE TABLE statement, the database load utility, the database reorganization facility, or on re-initialization of the RDAREA.

2. The sequence in which a trigger is executed is shown as follows:



3. If multiple triggers having the same trigger action time, trigger event, and trigger action unit (by statement or by row) are defined, the trigger actions are executed in the order they are defined (in the order of the values of the `CREATE_TIME` columns in the `SQL_TRIGGERS` table).
4. If an error occurs during execution of a trigger, the affected transaction is nullified (a rollback is triggered implicitly). If, during the execution of a trigger-inducing SQL statement, an error occurs after a trigger is executed once, the affected transaction is nullified irrespective of whether the trigger is being executed or whether the transaction is to be nullified. However, a rollback does not occur for a table for which `WITHOUT ROLLBACK` is specified, after completion of row updating (including additions and deletions). For details, see the rules applicable to `WITHOUT ROLLBACK` in *CREATE TABLE (Define table)* in this chapter.
5. The user executing the trigger-inducing SQL statement need not have the privilege necessary for execution of the trigger SQL statement.
6. Other triggers can also be executed upon encountering an SQL statement in the trigger SQL statement (in nesting of triggers). Only 16 levels of triggers can be nested. If a 16th level trigger causes an operation that sets off other triggers, an error may occur.
7. A row-by-row trigger for which `UPDATE` is specified in *trigger-event* is executed only once relative to the row to be updated. Even when triggers are started by nesting, and the same trigger is started as an extension of the nesting, the row-by-row trigger is executed only once, the first time.
8. Any of the following items cannot be specified on a table for which the `UPDATE` trigger is defined:

- An update using a component specification on a table for which an UPDATE trigger specifying a new-values correlation name is defined.
- An update of the BLOB type using a concatenation operation or the BINARY type with a defined length of 32,001 bytes or greater, for a table for which the UPDATE trigger is defined

9. When referencing a table that is updated by a trigger operation or its extension in an SQL subquery that causes a row-unit AFTER trigger, make sure that the subquery does not include an external reference.

Notes

1. CREATE TRIGGER cannot be specified from an X/Open compliant UAP running under OLTP.
2. When you specify an update SQL statement (UPDATE, DELETE, or INSERT statement) in a trigger SQL statement, the following SQL errors can occur during the execution of the specified update SQL statement:
   - The table specified in the update SQL statement is included in the SQL statement that executes the FOR EACH ROW trigger specifying the update SQL statement.
   - The table specified in the update SQL statement is included in the FOR EACH ROW trigger definition (including the nesting of multiple trigger definitions) that executes the FOR EACH ROW trigger definition specifying the update SQL statement.

   In this case, either specify index key-value-non-locking in system definitions or change the trigger definition.
3. Specifying an update SQL statement in a trigger SQL statement can affect the retrieval of the SQL during the execution of the specified update SQL statement. To avoid any impact on the retrieval of the SQL statement being executed, search conditions and data need to be modified so that the update SQL statement does not match the retrieval conditions for the SQL statement being executed. Special care should be exercised if trigger definitions are nested.
4. If a function, a procedure, or an SQL object for which a trigger is in effect is nullified through the specification of WITH PROGRAM, any rows related to the nullified function, procedure, or trigger are deleted from the SQL_ROUTINE_RESOURCES dictionary table.
5. Before executing the function, procedure, or trigger SQL object that was nullified through the specification of WITH PROGRAM, you need to re-create the nullified function, procedure, and trigger SQL object by executing ALTER ROUTINE, ALTER PROCEDURE, or ALTER TRIGGER.
6. The data guarantee level of the trigger SQL statement in the trigger, the SQL

optimization option, the SQL extension optimizing option, and the maximum character length are determined by what is specified when the trigger is being defined or modified, and are not affected by the system definition or client environment variable definition that is in effect when the trigger action is executed.

7. The trigger action is also executed if the operation specified in *trigger-event* is executed on the view table for which the table specified in ON [*authorization-identifier*.]*table-identifier* is the base table.
8. If the trigger SQL object is not in effect and any of the following SQL statements is executed by using the table defining that trigger, an error occurs irrespective of whether the trigger action condition is true or false. Similarly, if the index for the trigger SQL object is not in effect and any of the following SQL statements is executed by using the table that defined the trigger, an error occurs if the trigger action condition is true:
   - If *trigger-event* is INSERT, an INSERT statement that inserts rows into the table defining the trigger
   - If *trigger-event* is UPDATE, an UPDATE statement that updates rows in the table defining the trigger
   - If *trigger-event* is DELETE, a DELETE statement that deletes rows from the table defining the trigger
9. Nested triggers can degrade performance; to the maximum possible extent, nesting of triggers should be avoided.
10. When creating nested triggers, creating triggers from the leading trigger can result in an error during the creation of nested triggers. A WITH PROGRAM specification can create nested triggers, but in this case the first trigger is in a nullified state. Therefore, when creating nested triggers, create them in sequence beginning with the trigger that is at the end of the nesting levels.

Examples

The CREATE TRIGGER example uses an inventory history table (HSTOCK), a Glasgow inventory table, and a Edinburgh inventory table in addition to an inventory table (STOCK). The Glasgow and Edinburgh inventory tables have the same organization as the inventory table. The inventory history table is organized as follows:

Inventory history table (HSTOCK)

PCODE (Product code)	BQUANTITY (Pre-update quantity)	AQUANTIYT (Post-update quantity)	HDATE (Update date)	HTIME (Update time)
101M	26	10	2003-06-20	10:15:23

1. After a row is inserted into the inventory table (STOCK), define a trigger

(INSERTTRIG1) that inserts information on the inserted row into the inventory history table (HSTOCK):

```
CREATE TRIGGER INSERTTRIG1
  AFTER INSERT ON STOCK
  REFERENCING NEW ROW X1
  FOR EACH ROW
  INSERT INTO HSTOCK
    VALUES(X1.PCODE,NULL,X1.SQTY,
           CURRENT_DATE,CURRENT_TIME)
```

2. After the quantity (SQTY) is updated in the inventory table (STOCK), define a trigger (INSERTTRIG2) that inserts pre- and post-update values into the inventory history table (HSTOCK):

```
CREATE TRIGGER INSERTTRIG2
  AFTER UPDATE OF SQTY ON STOCK
  REFERENCING NEW ROW X1 OLD ROW Y1
  FOR EACH ROW
  INSERT INTO HSTOCK
    VALUES(Y1.PCODE,Y1.SQTY,X1.SQTY,
           CURRENT_DATE,CURRENT_TIME)
```

3. After a row is deleted from the inventory table (STOCK), define a trigger (INSERTTRIG3) that inserts information on the deleted row into the inventory history table (HSTOCK):

```
CREATE TRIGGER INSERTTRIG3
  AFTER DELETE ON STOCK
  REFERENCING OLD ROW Y1
  FOR EACH ROW
  INSERT INTO HSTOCK
    VALUES(Y1.PCODE,Y1.SQTY,NULL,
    CURRENT_DATE,CURRENT_TIME)
```

4. Use a routine control SQL (a compound statement) in *trigger-action*. After the inventory table (STOCK) is updated, define a trigger (UPDATELOCAL) that puts the update into effect in the Glasgow and Edinburgh inventory tables. If a compound statement is not used, two triggers that are set off upon the updating of the inventory table must be defined.

```
CREATE TRIGGER UPDATELOCAL
  AFTER UPDATE OF SQTY ON STOCK
  REFERENCING NEW ROW X1 OLD ROW Y1
  BEGIN
    UPDATE glasgow-inventory-table SET SQTY=X1.SQTY
      WHERE PCODE=Y1.PCODE;
    UPDATE edinburgh-inventory-table SET SQTY=X1.SQTY
      WHERE PCODE=Y1.PCODE;
  END
```

5. Use a routine control SQL (an assignment statement) in *trigger-action*. Define a trigger (SETPRICE) that assigns the following amount to the row to be inserted into the inventory table (STOCK): an amount equal to the unit price (PRICE) to be inserted plus 50.00 dollars if the product code (PCODE) is 101M, 201M, or an amount equal to the unit price to be inserted multiplied by 1.2 if the (PCODE) is not 101M, 201M, or 301M:

```
CREATE TRIGGER SETPRICE
  BEFORE INSERT ON STOCK
  REFERENCING NEW ROW AS X1
  FOR EACH ROW
  SET X1.PRICE=CASE X1.PCODE
      WHEN '101M' THEN X1.PRICE + 50
      WHEN '201M' THEN X1.PRICE + 50
      WHEN '301M' THEN X1.PRICE + 50
      ELSE X1.PRICE * 1.2
    END
```

6. Use an SQL diagnostic statement (SIGNAL statement) in *trigger-action*. Define a trigger (SIGNALTRIG) that suppresses the deletion of rows from the inventory table before rows in the inventory table (STOCK) are deleted:

```
CREATE TRIGGER SIGNALTRIG
  BEFORE DELETE ON STOCK
  SIGNAL SQLSTATE '99001'
```

CREATE TYPE (Define type)

Function

`CREATE TYPE` defines an abstract data type.

Privileges

Owner of a schema

A user can define abstract data types that will be owned by that user.

Format

```
CREATE TYPE [authorization-identifier.] data-type-identifier
                [subtype-clause]
                [default-constructor-option]
                [member-list]
subtype-clause::=UNDER [authorization-identifier.] data-type-identifier
default-constructor-option::=CONSTRUCTOR {PRIVATE|PROTECTED|PUBLIC}

member-list::=(member[,member]...)
    member::={attribute-definition|routine-declaration}
        attribute-definition::=[encapsulation-level]
                                    attribute-name data-type [NO SPLIT]
        encapsulation-level::= {PRIVATE|PROTECTED|PUBLIC}
        routine-declaration::=[encapsulation-level] routine-body
        routine-body::={function-body|procedure-body}
```

Operands

- [*authorization-identifier*.]*data-type-identifier*

authorization-identifier

Specifies the authorization identifier of the owner of the abstract data type that is being defined.

data-type-identifier

Specifies a name for the abstract data type being defined.

- *subtype-clause*::=`UNDER` [*authorization-identifier*.]*data-type-identifier*

A subtype clause is specified when what is being defined is a subtype that will inherit a specified abstract data type. The subtype clause is used to specify the authorization identifier and data type identifier of the abstract data type that will become the super-type for the abstract data type being defined.

Specifying a subtype clause causes all the attributes and routines defined in the

super-type to be inherited by the abstract data type that is being defined.

authorization-identifier.

Specifies the authorization identifier of the owner of the super-type abstract data type.

If the authorization identifier is omitted, the default authorization identifier does not have an abstract data type of the same name, and an abstract data type of the same name exists in the MASTER authorization identifier, that abstract data type is assumed to have been specified.

data-type-identifier

Specifies the super-type abstract data type.

- *default-constructor-option* ::=CONSTRUCTOR {PRIVATE|PROTECTED|PUBLIC}

Specifies the encapsulation level for the default constructor function. The default is PRIVATE.

For the default constructor function, a function that has the same name as the defined abstract data type is defined. The default constructor function takes no arguments. This function returns values of an abstract data type in which null values are set for the attributes of the abstract data type.

PRIVATE

Specifies that the default constructor function of the abstract data type can be used only in this abstract data type definition statement.

PROTECTED

Specifies that the abstract data type is partially encapsulated, which means that the default constructor function of the abstract data type can be used only in the abstract data type being defined and in subtype definitions for that abstract data type.

PUBLIC

Specifies that the default constructor function of the abstract data type can be used regardless of the inheritance.

- *attribute-definition* ::= [*encapsulation-level*] *attribute-name data-type* [NO SPLIT]

Specifies an attribute that makes up the abstract data type.

encapsulation-level

Specifies one of three encapsulation levels. The encapsulation level can be specified for routines that code operations on attributes and abstract data types. The default is the encapsulation level that is next higher than the encapsulation

level in the definition part. If no encapsulation level is specified at the first level, PUBLIC is assumed as the default.

PRIVATE

This option encapsulates the specified attribute, in which case the attribute can be used only in the definition of the abstract data type.

PROTECTED

This option partially encapsulates the specified attribute, in which case the attribute can be used only in the definition of the abstract data type and in the definition of all subtypes of that abstract data type.

PUBLIC

This option allows the specified attribute to be used regardless of any inheritance relationships that may be in effect.

attribute-name

Specifies the name of the attribute of the abstract data type.

data-type

Specifies the data type of the attribute of the abstract data type.

If the specified data type is an abstract data type, no authorization identifier is specified, and the default authorization identifier does not have an abstract data type of the same name, and if there is an abstract data type of the same name in the MASTER authorization identifier, that abstract data type is assumed to have been specified.

The following data types cannot be specified:

- CHAR or VARCHAR for which a character set is specified

NO SPLIT

When the actual data length of a variable-length character string is at least 256 bytes, specifies that each row is to be stored on a single page.

In some cases, the NO SPLIT option reduces the database storage space requirements. This option is called the *no-split option*; for details, see the *HiRDB Version 9 Installation and Design Guide*.

The no-split option is applicable only to the variable-length character string types (VARCHAR, NVARCHAR, and MVARCHAR).

■ *routine-declaration* ::=[*encapsulation-level*] *routine-body*

routine-body ::={*function-body* | *procedure-body*}

The routine declaration is where the routine in which data manipulations are specified

is written. The same encapsulation levels as for attribute definition are applicable to routine declarations.

Write the function or procedure in the routine body. For details about functions and procedures, see *CREATE [PUBLIC] FUNCTION (Define function, define public function)* or *CREATE [PUBLIC] PROCEDURE (Define procedure, define public procedure)* in this chapter.

Common rules

1. When an authorization identifier is specified in a routine, it must be the authorization identifier of the user who executes the routine.
2. The procedure in a `CALL` statement or the function call function specified in the function body or procedure body must be one of the following:
   - A function or procedure that was already defined outside of the `CREATE TYPE` statement
   - A function or procedure that was already defined in a routine declaration prior to declaration of this routine
3. The attribute name of each abstract data type must be unique among all abstract data types that have an interconnected inheritance relationship.
4. The maximum permissible number of generations for inherited subtypes is 30,000.
5. An abstract data type to be defined with `CREATE TYPE` must meet the expressions below. The following table indicates the length of the attributes (data lengths).

$$2 \times n + \sum_{i=1}^{n} A_i \leq 32757$$

n : Number of abstract data type attributes defined with `CREATE TYPE`

$\sum_{i=1}^{n} A_i$: Sum of the data lengths of the member list attributes

Table 3-43: Data lengths

Category	Data type	Date length (bytes)
Numeric data	INTEGER (4-byte binary integer)	4
	SMALLINT (two-byte binary integer)	2
	[LARGE] DECIMAL [*m*, *n*]# (packed decimal floating-point number)	↓*m*/2↓ + 1
	FLOAT (8-byte floating point number)	8
	SMALLFLT (4-byte floating point number)	4
Character data	CHARACTER[*n*] (*n*-byte fixed-length character string)	*n*
	VARCHAR[*n*] (*n*-byte variable-length character string)	35
National character data	NCHAR[*n*] (*n*-character fixed-length national character string)	2*n*
	NVARCHAR[*n*] (*n*-character variable-length national character string)	35
Mixed character data	MCHAR[*n*] (*n*-byte fixed-length mixed character string)	*n*
	MVARCHAR[*n*] (*n*-byte variable-length mixed character string)	35
Date data	DATE (date)	4
Date interval data	INTEVAL YEAR TO DAY (date interval)	5
Time data	TIME (time)	3
Time interval data	INTERVAL HOUR TO SECOND (time interval)	4
Time stamp data	TIMESTAMP[*p*] (fractional second with *p* digits) (time stamp)	7 + *p*/2
Large object data	BLOB (binary string in *n* bytes)	35
Abstract data	User-defined data type defined in CREATE TYPE	35

m, *n*: Positive integers.

p: Integer 0, 2, 4, or 6

#: This is a fixed-point number with a total of *m* digits and *n* decimal places. If *m* is omitted, the default that is assumed is 15.

6. When the SQL compile option is specified in an ALTER PROCEDURE or ALTER ROUTINE statement, the length of the SQL statement that is created by incorporating the SQL compile option into the source CREATE PROCEDURE statement of the routine being re-created must not exceed the maximum

permissible length for SQL statements.

7. CREATE TYPE cannot be executed from within a Java procedure if the execution results in invalidation of the SQL object being executed.

Notes

1. The CREATE TYPE statement cannot be specified from an X/Open-compliant UAP running under OLTP.
2. The following shows the invalidation conditions for the SQL object associated with the type definition:
   - When you define an abstract data type, if a valid SQL object of a function, procedure, or trigger uses an abstract data type (all subtypes of the highest data type) with the same inheritance relationship as the abstract data type being defined, the SQL object is invalidated.
   - When you define an abstract data type, if the same data type as the one you are defining uses MASTER as its authorization identifier, any valid SQL object that belongs to the authorization identifier of the abstract data type being defined becomes invalid if it is in a function, procedure, or trigger that uses the MASTER authentication identifer. In addition, if a valid SQL object is in a public function or public procedure that has the same authorization identifier as that of the abstract data type being defined, the SQL object in that public function or public procedure becomes invalid.
   - For details about invalidation of an SQL object when specifying a function body and defining a function, see the notes under *CREATE [PUBLIC] FUNCTION (Define function, define public function)* in this chapter.
   - For details about invalidation of an SQL object when specifying a procedure body and defining a procedure, see the notes under *CREATE [PUBLIC] PROCEDURE (Define procedure, define public procedure)* in this chapter.
3. If an SQL object for which functions, procedures, and triggers are in effect is nullified, any rows of nullified functions, procedures, and triggers in the SQL_ROUTINE_RESOURCES dictionary table are deleted.
4. For a function that was invalidated for any of the reasons in Note 2, if a function satisfying either of the following conditions is used in the view definition, the type definition results in an error:
   - Abstract data type is used in the data type of an argument.
   - Abstract data type is used in the data type of a return value.
5. Before executing the SQL object for nullified functions, procedures, and triggers, you need to execute ALTER ROUTINE, ALTER PROCEDURE, or ALTER TRIGGER to re-create the SQL object for the functions, procedures, and triggers.

6. To access a view table that references an invalidated function associated with the type definition, you must execute `ALTER ROUTINE` to re-create the SQL object of the function.
7. The subtype of an abstract data type that is provided by the developer of a plug-in cannot be defined by inheriting an abstract data type that is registered in the system through use of the `pdplgrgst` command.

CREATE [PUBLIC] VIEW (Define view, define public view)

Function

CREATE VIEW defines a view table.

CREATE VIEW can also be used to define *public* view tables, which are available to all users without them having to qualify the table identifier with an authorization identifier.

(1) CREATE VIEW (Define view)

Privileges

Users who can use derived query expressions

These users can specify available derived query expressions and define their own view tables.

The following users are permitted to use derived query expressions (for details of derived query expressions, see *2.2 Query expressions*; for details of the privilege for specifying a query in a derived query expression, see the privileges topic in *2.3 Query specification*).

- Owner of a base table or view table
- User with the SELECT privilege on base tables, or view tables

If a user defines a view table by receiving the SELECT privilege for tables owned by another user, only the receiving user, not the user who granted the SELECT privilege, can perform view definitions from that view table.

Format

```
CREATE [READ ONLY] VIEW [authorization-identifier.]table-identifier
       [(column-name [,column-name]...)]
  AS derived-query-expression

 derived-query-expression::= query-expression-body
 query-expression-body :: = {query-specification
             |(query-expression-body)
             |query-expression-body {UNION | EXCEPT}[ALL]
              {query-specification|(query-expression-body)}}
```

Operands

- [READ ONLY]

Specifies that the view table being defined is to be a read-only table.

- [*authorization-identifier*.]*table-identifier* [(*column-name*[, *column-name*]...)]

authorization-identifier

Specifies the authorization identifier of the user who is to own the view table being defined.

table-identifier

Specifies a name for the view table being defined.

This name must not already be specified for a table (base table or view table) owned by the user specified as the owner of the view table being defined.

column-name

Specifies a name for a column that is to comprise the view table.

If no column name is specified, the default column names are as explained below:

- If no set operation is specified in the derived query expression, the column names of the columns for the derived table specified in the query specification (when AS *column-name* is specified, the column names specified in the AS clause) will be the column names of the columns that constitute the view table.
- When a set operation is specified in the derived query expression, the column names of the columns for the derived table specified in the first query specification (when AS *column-name* is specified, the column names specified in the AS clause) of the derived query expression will be the column names of the columns that constitute the view table.

A column name must be specified if the derived table specified in the derived query expression contains either multiple columns with the same name or unnamed columns.

The following rules apply to the column names:

1. When the column in the derived table is a column derived from one of the following items, and when the specification of AS *column-name* is omitted, the column becomes a nameless column:
   - Scalar operation
   - Function call
   - Set functions
   - Literals
   - USER
   - CURRENT_DATA

- CURRENT_TIME
- CURRENT_TIMESTAMP[(p)]
- Component specification

2. Each column name must be unique. The number of column names specified should be the same as the number of columns in the derived table obtained as a result of the derived query expression.
3. The maximum number of column names that can be specified is 30,000.

■ *derived-query-expression*

Specifies a derived query expression that expresses the contents of the definition of a view table (for details of search conditions, see the following sections in this manual: *2.2 Query expressions*).

The following rules apply to the derived query expression:

1. [*table-specification* .]ROW cannot be specified in a selection expression in a derived query expression for a view definition.
2. A subscripted repetition column in a directly contained SELECT clause cannot be specified in a derived query expression for a view definition.
3. Even though * or table.* is specified in a SELECT clause that is directly contained in a derived query expression, columns that are added to the base table for a view table after definition of the view table will not be added to the view table.
4. Base tables, and view tables can be specified in a derived query expression for a view definition. Similarly, a new table based on a view table can be defined.
5. Embedded variables and the ? parameter cannot be specified in a search condition in a derived query expression for a view definition.
6. The following value expressions cannot be specified in a derived query expression in the view definition:
   - XML constructor function
   - SQL/XML scalar function
   - SQL/XML predicate
   - SQL/XML set function

Common rules

1. A view table can be a read-only view table or a writable view table. Operations such as inserting, updating, and deleting rows, or specifying the FOR UPDATE clause in a cursor declaration, cannot be performed on a read-only view table.

The following view tables are read-only:

- View tables that are defined by specifying READ ONLY in a view definition statement
- View tables that include a table join, SELECT DISTINCT, a GROUP BY clause, a HAVING clause, or a set function with respect to the outermost query specification in a view definition statement.
- View tables that are defined by specifying the same column in the base table more than once in a SELECT clause in a view definition statement.
- View tables that include a value expression other than a column specification in the SELECT clause in the outermost query specification in a view definition statement
- Of the view tables defined before Version 07-02, a view table that contains a subquery that specifies in the FROM clause the same table as the FROM clause in the outermost query specification in a view definition statement (if the table specified in the FROM clause is a view table, includes tables that are a base for the view table).

  Note: To make the view table an updatable table, delete the view table, and then redefine it.
- A view table that is defined by specifying a derived table in the FROM clause in the outermost query specification in a view definition statement.
- A view table defined by specifying a set operation in a view definition statement.

All other view tables (those that are not read-only) are writable view tables.

2. The table that is specified in the derived query expression for a view definition serves as the base table that comprises the view table. The table that is specified in a FROM clause in the derived query expression for a view definition is the base table from which the view table is derived.
3. The table that is specified in a FROM clause contained in the outermost query in a derived query expression for a view definition is the base table that is subject to operations on that view table.

   The owner of an updatable view table directly inherits the following access privileges with respect to the base table subject to operations on the view table:

   - SELECT privilege
   - INSERT privilege
   - DELETE privilege
   - UPDATE privilege

The owner of a read-only view table inherits only the INSERT privilege with respect to the base table that is subject to operations on the view table.

The owner of a view table that is defined from tables owned by that user can grant and revoke the same access privileges for other users.

4. Rows that are added or updated using a view table need not satisfy the search conditions specified in the derived query expression during definition of the view table. However, rows that do not satisfy the search conditions specified in the derived query expression during definition of the view table cannot be searched, updated, or deleted by using the view table.
5. The attributes (data type, data length, any NOT NULL constraints that may be in effect, and maximum number of elements) of columns comprising a view table will be the same as the attributes of the corresponding columns in the derived table specified in the derived query expression for the view definition.
6. The table that is the base table for a view table must be defined before the view definition can be executed.
7. View definition statements cannot be specified from an X/Open compliant UAP running under OLTP.
8. A column in a view table, defined in terms of a date, time, or time stamp literal represented in a character string, or a date interval or time interval literal represented in a decimal number, are treated in terms of the data type for that column, even when the column is specified in a location where date, time, time stamp, date interval, or time interval data is required; it is not converted into the respective required data type, except when such a column is specified as an argument in the scalar function DATE, TIME, or TIMESTAMP.

   Example:

   ```
   UPDATE T1 SET C1=(SELECT VC2 FROM V1 WHERE VC1='E')
   ... Cannot be specified.
   ```

   (C1 is a column of date data type, VC2 is a column of VARCHAR(10) defined in terms of a literal that is represented in a date character string)
9. A subscripted repetition column in a directly contained SELECT clause cannot be specified in the derived query expression for a view definition.
10. If a CASE expression is specified in a selection expression in the outermost query specification, a repetition column cannot be specified in the CASE expression in a derived query expression for a view definition.
11. Any of the following items cannot be specified in a selection expression in a derived query expression in a view definition:

- WRITE specification
- GET_JAVA_STORED_ROUTINE_SOURCE specification
- Window function

12. If a view table defined by specifying function call is operated, user-defined functions that are candidates for function call are solely user-defined functions that were defined before the view table was defined.
13. If the platform is moved from a 32-bit mode HiRDB to a 64-bit mode HiRDB using the SQL object migration utility, view tables meeting all of the following conditions may produce different search results before and after migration:

    (1) Defining the view table by specifying a user-defined function in a derived query expression in the view definition

    (2) After defining a view table in (1), defining a user-defined function that can be a calling candidate for the user-defined function specified in the derived query expression in (1)

    (3) The user-defined function defined in (2) has higher calling priority than the user-defined function specified in the derived query expression in (1)

    After performing steps (1) to (3) above, use the SQL object migration utility to move from 32-bit mode HiRDB to 64-bit mode HiRDB.

    For details about the rules for determining the user-defined function to be called, see *2.20(6) Rules for determining the function to be called and the data type of the result*.

Examples

1. From a stock table (STOCK), define a view table (VSTOCK1) composed of the product code (PCODE), quantity in stock (SQTY), and unit price (PRICE) columns for the rows containing socks in the product name (PNAME) column; assume that the columns are sorted on product code, quantity in stock, and unit price:

```
CREATE VIEW VSTOCK1
   AS SELECT PCODE,SQTY,PRICE
         FROM STOCK
         WHERE PNAME=N'socks'
```

2. Define a read-only view table (VSTOCK2) with the same organization as the STOCK stock table:

```
CREATE READ ONLY VIEW VSTOCK2
   AS SELECT * FROM STOCK
```

(2) CREATE PUBLIC VIEW (Define public view)

Privileges

Owner of a base table

The owner of a base table can define a public view owned by him or her from his or her own base table.

Owner of a view table

The owner of a view table can define a public view owned by him or her from a view table based on his or her own base table.

Format

```
CREATE PUBLIC[READ ONLY] VIEW table-identifier
    [(column-name[, column-name]...)]
 AS derived-query-expression

derived-query-expression::= query-expression-body
query-expression-body::= {query-expression
           | (query-expression-body)
           | query-expression-body {UNION|EXCEPT} [ALL]
            {query-expression | (query-expression-body)}}
```

Operands

For a description of operands other than PUBLIC and *table-identifier*, see *(1) CREATE VIEW (Define view)* under *CREATE [PUBLIC] VIEW (Define view, define public view)* in this chapter.

- PUBLIC

Specify this operand when defining a view table as a public view.

When a view table is defined as a public view, a single view table can be used by more than one user by specifying a table identifier, without the need to define view tables of identical contents for different users.

- *table-identifier*

Specifies the name of the public view to be defined.

In *table-identifier*, a name identical to a previously defined public view cannot be specified.

Common rules

1. As the name of a public view being defined, the same table identifier as that for a previously defined table (base table or view table) can be specified. However, if

a table identifier is used by omitting an authorization identifier, the tables (base tables, or view tables) owned by the user who is executing the UAP take precedence over the public view. When qualifying a public view, in *authorization-identifier*, specify the word PUBLIC in upper case characters enclosed in double quotation marks (").

2. For details about other rules, see *(1) CREATE VIEW (Define view)* under *CREATE [PUBLIC] VIEW (Define view, define public view)* in this chapter.

Notes

1. In columns (such as the TABLE_SCHEMA column in the SQL_TABLES table) that store the owner of a dictionary table, the word PUBLIC is assigned. The authorization identifier used to define the public view is stored in the TABLE_CREATOR column in the SQL_TABLES table.
2. If a definition SQL statement specifying a table (base table or view table) with the same name as the public view being used is issued while preprocessing on an SQL statement using the public view is in effect, the definition SQL statement goes into a lock release wait state.
3. After defining a procedure and a trigger using a public view, if a table (base table or view table) having the same name as the public view is defined, the procedure and the trigger are not disabled, and they operate as the procedure and trigger that use the public view. However, if the procedure and trigger are re-created (including the case in which a procedure with a nullified index is internally created by HiRDB), they operate as the procedure and trigger that use the table (base table or view table) having the same name as the public view.
4. Public views can be deleted using DROP PUBLIC VIEW.

Examples

1. From an inventory table (STOCK) define a public view comprised of a row for which the product name (PNAME) is socks, and columns of product code (PCODE), quantity in stock (SQTY), and unit price (PRICE), such that the columns are sorted by product code, quantity in stock, and unit price:

```
CREATE PUBLIC VIEW PVSTOCK1
    AS SELECT PCODE,SQTY,PRICE
    FROM STOCK
    WHERE PNAME = N'socks'
```

DROP AUDIT (Delete an audit target event)

Function

DROP AUDIT deletes definitions that match the target audit event defined in CREATE AUDIT from the targets of auditing.

Privileges

Users with the audit privilege

These users can execute DROP AUDIT definition statements.

Format

No.	Format
1	DROP AUDIT [AUDITTYPE {PRIVILEGE | EVENT | ANY}]
2	FOR *operation-type*
9	[*selection-option*]
3	[WHENEVER {SUCCESSFUL | UNSUCCESSFUL | ANY}]

Details about items

No.	Format
2	○*operation-type*::= ANY | SESSION [{*session-type* |ANY}] | PRIVILEGE [{*privilege-operation-type* | ANY}] | DEFINITION [{*object-definition-event-type* | ANY}] | ACCESS [{*object-operation-event-type* | ANY}] | UTILITY [{*utility-event-type* | ANY}]

No.	Format
9	*selection-option*::=ON *object-name* *object-name*::= {FUNCTION *authorization-identifier*.*routine-identifier* |INDEX *authorization-identifier*.*index-identifier* |LIST *authorization-identifier*.*table-identifier* |PROCEDURE *authorization-identifier*.*routine-identifier* |RDAREA *RDAREA-name* |SCHEMA *authorization-identifier* |TABLE [*authorization-identifier*.]*table-identifier* |TRIGGER *authorization-identifier*.*trigger-identifier* |TYPE *authorization-identifier*.*data-type-identifier* |VIEW *authorization-identifier*.*table-identifier* ||SEQUENCE *authorization-identifier*.*sequence-generator-identifier*}
4	*session-type*::= {CONNECT | DISCONNECT | AUTHORIZATION}
5	*privilege-operation-type*::= {GRANT | REVOKE}
6	*object-definition-event-type*::= {CREATE | DROP | ALTER}
7	*object-operation-event-type*::= {SELECT | INSERT | UPDATE | DELETE | PURGE | ASSIGN | CALL |LOCK | NEXT VALUE}
8	*utility-event-type*::= {PDLOAD | PDRORG | PDEXP | PDCONSTCK}

Operands

For details about each item, see *CREATE AUDIT (Define the target audit event)* in this chapter.

1) AUDITTYPE {PRIVILEGE | EVENT | ANY}

Specifies the audit trail to be deleted.

2) FOR *operation-type*

Specifies the operation type to be deleted from the audit object.

3) WHENEVER {SUCCESSFUL | UNSUCCESSFUL | ANY}

Deletes the WHENEVER clause specification specified in CREATE AUDIT from *audit-object*.

4) *session-type*

Specifies the connection to HiRDB, user modifications made while connected, or disconnection operation to be deleted from the object of the audit.

5) *privilege-operation-type*

Specifies the privilege operations on HiRDB to be deleted from the object of the audit.

6) *object-definition-event-type*

Specifies the object definition operations on HiRDB to be deleted from the object of the audit.

7) *object-operation-event-type*

Specifies the object operations on HiRDB to be deleted from the object of the audit.

8) *utility-event-type*

Specifies the utility event on HiRDB to be deleted from the object of the audit.

9) *selection-option*

Specifies a selection option to be deleted from the audit object.

Rules

1. For the security audit facility, see the *HiRDB Version 9 System Operation Guide*.
2. Recording actual audit trails requires either the specification of the system definition `pd_audit` operand or the execution of the `pdaudbegin` command.
3. An audit trail from the execution of `CREATE AUDIT` or `DROP AUDIT` is always recorded, provided that the security audit facility is enabled.
4. `DROP AUDIT` can be executed by the same specification as the combination of `AUDITTYPE`, `FOR` <*operation-type*>, `WHENEVER` statements that were specified in `CREATE AUDIT`.

   Example:

   For deleting `CREATE AUDIT AUDITTYPE EVENT FOR SESSION`, simply specify `DROP AUDIT AUDITTYPE EVENT FOR SESSION`.
5. `DROP AUDIT` cannot be used in such a way as to only delete a part of a defined audit target range from the audit object. If such a specification is attempted, the system generates the `KFPA11909-E` message.

   Example 1:

   If all audit events are defined as audit objects in `CREATE AUDIT FOR ANY`, `DROP AUDIT FOR ACCESS` cannot be executed by specifying `SELECT` because `SELECT` as it applies to a table is deleted from the audit object.

To delete SELECT as it applies to a table from the audit object, define a required audit object other than SELECT for a table, and then execute DROP AUDIT FOR ANY.

Example 2:

If all audit events during a privilege check are defined as audit objects in CREATE AUDIT AUDITTYPE PRIVILEGE FOR ANY, the task cannot be executed by specifying DROP AUDIT AUDITTYPE PRIVILEGE because the PRIVILEGE in AUDITTYPE is deleted from the audit object.

Notes

1. DROP AUDIT cannot be specified from an X/Open-compliant UAP running under OLTP.

Examples

1. Delete the target of the audit defined in CREATE AUDIT FOR ANY.

```
DROP AUDIT FOR ANY
```

2. Delete the target of the audit defined in CREATE AUDIT FOR SESSION.

```
DROP AUDIT FOR SESSION CONNECT
```

3. Delete the target of the audit defined in CREATE AUDIT FOR PRIVILEGE.

```
DROP AUDIT FOR PRIVILEGE GRANT
```

4. Delete the target of the audit defined in CREATE AUDIT FOR DEFINITION.

```
DROP AUDIT FOR DEFINITION CREATE WHENEVER ANY
```

5. Delete the target of the audit defined in CREATE AUDIT FOR ACCESS.

```
DROP AUDIT FOR ACCESS INSERT
```

6. Delete the target of the audit defined in CREATE AUDIT AUDITTYPE ANY FOR ANY.

```
DROP AUDIT AUDITTYPE ANY FOR ANY
```

7. Delete the target of the audit defined in `CREATE AUDIT AUDITTYPE PRIVILEGE FOR ANY`.

```
DROP AUDIT AUDITTYPE PRIVILEGE FOR ANY
```

8. Delete the target of the audit defined in `CREATE AUDIT AUDITTYPE EVENT FOR ANY ON TABLE "USER1"."T1"`.

```
DROP AUDIT AUDITTYPE EVENT FOR ANY ON TABLE "USER1"."T1"
```

DROP CONNECTION SECURITY (Delete the connection security facility)

Function

`DROP CONNECTION SECURITY` deletes security items related to the connection security facility.

Privileges

User with DBA privileges

Users with DBA privileges can execute `DROP CONNECTION SECURITY` definition statements.

Format

```
DROP CONNECTION SECURITY FOR security-object[, security-object]

security-object::={CONNECT|PASSWORD}
```

Operands

- *security-object*::= {`CONNECT` | `PASSWORD`}

In *security-object*, `CONNECT` and `PASSWORD` can each be specified only once.

`CONNECT`

Specify this operand when deleting settings related to a consecutive certification failure limit.

`PASSWORD`

Specify this operand when deleting settings related to password character limit enhancement.

Notes

1. If settings related to the consecutive certification failure limit are deleted, all users are released from the consecutive certification failure account lock state (the null value is assigned to the consecutive certification failure account lock date and time, and to the permitted number of consecutive certification failures in `SQL_USERS`).
2. If settings related to the password character limit are deleted, all users are released from the password-invalid account lock state (the null value is assigned to the password-invalid account lock date and time of `SQL_USERS`).

3. An undefined security object cannot be specified.
4. An error may occur if an attempt is made to delete both CONNECT and PASSWORD when either CONNECT or PASSWORD, but not both, is defined. Any defined items that are specified as objects of deletion are not deleted.

Examples

1. Delete settings related to password character limit enhancement.

```
DROP CONNECTION SECURITY FOR PASSWORD
```

DROP DATA TYPE (Delete user-defined data type)

Function

DROP DATA TYPE deletes an abstract data type.

Privileges

Owners of abstract data types

A user can delete an abstract data type owned by that user.

Users with the DBA privilege

These users can delete abstract data types owned by other users.

Format

```
DROP DATA TYPE [authorization-identifier.]data-type-identifier
        [WITH PROGRAM]
```

Operands

- [*authorization-identifier*.]*data-type-identifier*

authorization-identifier

Specifies the authorization identifier of the owner of the data type that is to be deleted.

data-type-identifier

Specifies the identifier of the data type that is to be deleted.

- WITH PROGRAM

When deleting an abstract data type, this option is specified to nullify an SQL object for which any of the following functions, procedures, and triggers are in effect:

- An SQL object for which functions, procedures, and triggers are in effect that use an abstract data type (all subtypes of the highest-level data type) that has the same inheritance relationship as the abstract data type to be deleted
- An SQL object for which functions, procedures, and triggers are in effect that use the same functions and procedures as those defined in the abstract data type to be deleted
- An SQL object for which functions, procedures, and triggers are in effect that use a function having the same owner, routine identifier, or number of parameters as the function that was defined in the abstract data type to be deleted

If WITH PROGRAM is omitted and if there is an SQL object for which any of the

following functions, procedures, and triggers are in affect, the abstract data type cannot be deleted:

- An SQL object for which functions, procedures, and triggers are in effect that use an abstract data type that has the same inheritance relationship as the abstract data type to be deleted
- An SQL object for which functions, procedures, and triggers are in effect that use the functions and procedures that were defined in the abstract data type to be deleted
- An SQL object for which functions, procedures, and triggers are in effect that use a function having the same owner, routine identifier, and number of parameters as the function defined in the abstract data type to be deleted

Common rules

1. The specified abstract data type is not deleted if there are base tables, indexes, abstract data types, routines in an abstract data type, routines, and triggers that use the specified abstract data type.
2. If a function defined in the abstract data type definition is used in a view definition, that abstract data type cannot be deleted.
3. Of the functions invalidated by the `WITH PROGRAM` specification, if a function that meets either of the following conditions is used in the view definition, the abstract data type cannot be deleted:
   - Abstract data type is specified in the data type of an argument.
   - Abstract data type is specified in the data type of a return value.
4. If an SQL object being executed is invalidated by deleting the abstract data type, `DROP DATA TYPE` cannot be executed from within a Java procedure.

Notes

1. The `DROP DATA TYPE` statement cannot be specified from an X/Open-compliant UAP running under OLTP.
2. If the specified abstract data type contains an abstract data type function, its plug-in information will also be deleted.
3. If an SQL object for which functions, procedures, or triggers are in effect is nullified by specifying `WITH PROGRAM`, any rows associated with the nullified functions, procedures, or triggers in the `SQL_ROUTINE_RESOURCES` dictionary table are deleted.
4. Before executing the SQL object associated with the function, procedure, or trigger that was nullified by specifying `WITH PROGRAM`, you need to re-create the function, procedure, or trigger by executing `ALTER ROUTINE`, `ALTER`

PROCEDURE, or ALTER TRIGGER.

5. Before executing the SQL object associated with the trigger that was nullified by specifying WITH PROGRAM, you need to execute either ALTER TRIGGER or ALTER ROUTINE to re-create the trigger SQL object. However, to execute an SQL object associated with a trigger that was using the function and procedure that were defined in the deleted abstract data type definition, you need to perform either of the following operations:

   - Redefine the abstract data type and execute either ALTER TRIGGER or ALTER ROUTINE to re-create the trigger SQL object.
   - Delete the nullified trigger by using DROP TRIGGER, and then redefine the trigger by using CREATE TRIGGER so that the deleted functions and procedures are not reused in the deleted abstract data type definition. If triggers satisfying all of the following conditions exist, use DROP TRIGGER to delete them all, and redefine the triggers using CREATE TRIGGER in the order in which they were defined, so that there is no change in the sequence of execution of trigger actions.

     Conditions:

     - The defined trigger is later than the nullified trigger.
     - The nullified trigger is the same as the defined table.
     - The nullified trigger is the same as the trigger event (INSERT, UPDATE, or DELETE) (for UPDATE, the nullified trigger is considered to be the same as the trigger event regardless of whether a trigger event column is specified or the contents of the specification).
     - The nullified trigger has the same trigger action timing (BEFORE or AFTER) as the trigger event.
     - The nullified trigger has the same trigger action units (units of rows or statements) as the trigger event.

6. To access a view table that uses a function invalidated by specifying WITH PROGRAM, you must execute ALTER ROUTINE to re-create the SQL object of the function.

Example

Delete the SGML type defined by the user with authorization identifier USER1:

```
DROP DATA TYPE USER1.SGML
```

DROP [PUBLIC] FUNCTION (Delete function, delete public function)

Function

`DROP FUNCTION` deletes a function.

All users can delete these functions (public functions) without having to modify the routine identifier with the authorization identifier.

(1) DROP FUNCTION (Delete function)

Privileges

Owner of a function

The owner of a function can delete that function.

Users with the DBA privilege

These users can delete functions that are owned by other users.

Format

```
DROP FUNCTION [authorization-identifier.]routine-identifier
                ([data-type [,data-type]...])
        [WITH PROGRAM]
```

Operands

- [*authorization-identifier*.]*routine-identifier*

authorization-identifier

Specifies the authorization identifier of the owner of the function that is to be deleted.

routine-identifier

Specifies the name of the function that is to be deleted.

- *data-type*

Specifies the data type that was specified in the parameter for the function that is to be deleted. If the data type specified in a parameter of the function to be deleted is fixed-length or variable-length character data type and the character set specification is specified, specify the character set specification here.

- `WITH PROGRAM`

When deleting a function, this option is specified to nullify an SQL object for which any of the following functions, procedures, and triggers are in affect:

- SQL objects for which functions, procedures, and triggers are in effect that use

the function being deleted

- SQL objects for which functions, procedures, and triggers are in effect that use a function that has the same owner, routine identifier, and number of parameters as the function being deleted

If WITH PROGRAM is omitted, and if there is an SQL object for which any of the following functions, procedures, and triggers are in effect, that function cannot be deleted:

- SQL objects for which functions, procedures, and triggers are in effect that use the function being deleted
- SQL objects for which functions, procedures, and triggers are in effect that use a function that has the same owner, routine identifier, and number of parameters as the function being deleted

Common rules

1. DROP FUNCTION cannot be executed from within a Java procedure if the deletion results in invalidation of the SQL object being executed.
2. A function that is defined in an abstract data type cannot be deleted.
3. If there are multiple functions with the same name, specify the data type parameters so that the functions to be deleted can be identified uniquely.
4. If there are view tables (including public views) that use a specified function, the function cannot be deleted.
5. Of the functions invalidated by the WITH PROGRAM specification, if a function that meets either of the following conditions is used in the view definition, the function cannot be deleted:
   - Abstract data type is used in the data type of an argument.
   - Abstract data type is used in the data type of a return value.

Notes

1. DROP FUNCTION cannot be specified from an X/Open-compliant UAP running under OLTP.
2. If an SQL object for which functions, procedures, or triggers are in effect is nullified by specifying WITH PROGRAM, any rows associated with the nullified functions, procedures, or triggers in the SQL_ROUTINE_RESOURCES dictionary table are deleted.
3. Before executing the SQL object associated with the function, procedure, or trigger that was nullified by specifying WITH PROGRAM, you need to re-create the function, procedure, or trigger by executing ALTER ROUTINE, ALTER PROCEDURE, or ALTER TRIGGER.

4. To execute the SQL object associated with the trigger that was nullified by specifying WITH PROGRAM, you need to perform either of the following operations; however, if the trigger was using a function having the same owner, routine identifier, and number of parameters as the function that was deleted, you can re-create the trigger SQL object by executing either ALTER TRIGGER or ALTER ROUTINE:

   - Redefine the function, and re-create the trigger SQL object by executing either ALTER TRIGGER or ALTER ROUTINE.
   - Delete the nullified trigger by using DROP TRIGGER, and then redefine the trigger by using CREATE TRIGGER so that the deleted functions are not reused. If triggers satisfying all of the following conditions exist, use DROP TRIGGER to delete them all, and redefine the triggers using CREATE TRIGGER in the order in which they were defined so that there is no change in the sequence of execution of trigger actions:

     Conditions:

     - The defined trigger is later than the nullified trigger.
     - The nullified trigger is the same as the defined table.
     - The nullified trigger is the same as the trigger event (INSERT, UPDATE, or DELETE) (for UPDATE, the nullified trigger is considered to be the same as the trigger event regardless of whether a trigger event column is specified or the contents of the specification).
     - The nullified trigger has the same trigger action timing (BEFORE or AFTER) as the trigger event.
     - The nullified trigger has the same trigger action units (units of rows or statements) as the trigger event.

5. If the function that was being used in the trigger action conditions was deleted by specifying WITH PROGRAM, an error occurs, not only during the execution of the SQL object associated with the nullified trigger, but also during the preprocessing of the SQL statement that induces the trigger.
6. To access a view table that uses a function invalidated by specifying WITH PROGRAM, you must execute ALTER ROUTINE to re-create the SQL object of the function.

(2) DROP PUBLIC FUNCTION (Delete public function)

Privileges

Owner of a public function

The owner of a (defined) public function can delete that function.

Users with the DBA privilege

These users can delete public functions that are owned by other users.

Format

```
DROP PUBLIC FUNCTION routine-identifier([data-type [,data-type]...])
              [WITH PROGRAM]
```

Operands

For descriptions of operands other than PUBLIC, *routine-identifier*, and WITH PROGRAM, see *(1) DROP FUNCTION (Delete function)* under *DROP [PUBLIC] FUNCTION (Delete function, delete public function)* in this chapter.

- PUBLIC

Specifies that a public function is to be deleted.

- *routine-identifier*

Specifies the name of the public function that is to be deleted.

- WITH PROGRAM

Specifies the following valid SQL objects of a function, procedure, or trigger when deleting a public function:

- A valid SQL object of a function, procedure, or trigger that uses the public function being deleted
- A valid SQL object of a function, procedure, or trigger that uses a public function with the same number of routine identifiers and parameters as the public function being deleted

If WITH PROGRAM is omitted and there exists one of the following SQL objects of a function, procedure, or trigger, that function cannot be deleted:

- A valid SQL object of a function, procedure, or trigger that uses the public function being deleted
- A valid SQL object of a function, procedure, or trigger that uses a function with the same number of routine identifiers and parameters as the public function being deleted

Common rules

1. If the SQL object being executed is invalidated, DROP PUBLIC FUNCTION cannot be executed from within a Java procedure.
2. For details about other common rules, see *(1) DROP FUNCTION (Delete*

function) under *DROP [PUBLIC] FUNCTION (Delete function, delete public function)* in this chapter.

Notes

1. `DROP PUBLIC FUNCTION` cannot be specified from an X/Open-compliant UAP running under OLTP.
2. For other notes, see *(1) DROP FUNCTION (Delete function)* under *DROP [PUBLIC] FUNCTION (Delete function, delete public function)* in this chapter.

DROP INDEX (Delete index)

Function

DROP INDEX deletes an index.

Privileges

Owner of the index

The owner of an index can delete that index.

Users with the DBA privilege

A user with the DBA privilege can delete indexes owned by other users.

Format

```
DROP INDEX [authorization-identifier.]index-identifier [WITH PROGRAM]
```

Operands

- [*authorization-identifier*.]*index-name* [WITH PROGRAM]

authorization-identifier

Specifies the authorization identifier of the user who owns the index.

When this operand is omitted, the authorization identifier of the user who is executing the command is assumed.

index-identifier

Specifies the name of the index to be deleted.

WITH PROGRAM

Specifies that when the index is deleted, the SQL objects for active procedures and triggers that use the index are to be made inactivate.

When WITH PROGRAM is omitted, the index cannot be deleted if a procedure and triggers that uses the index is still active in an SQL object.

Common rules

1. DROP INDEX cannot be executed from within a Java procedure if the deletion results in invalidation of the SQL object being executed.
2. If the index to be deleted, the table from which the index is to be deleted, the index defined for the table, and columns of the abstract data type containing LOB columns and LOB attributes that are defined for the table are stored in an RDAREA to which the inner replica facility is applied, the index can be deleted, provided that certain conditions are met. For details about DROP INDEX execution

conditions under a condition in which the inner replica facility is used, see the manual *HiRDB Version 9 Staticizer Option*.

Notes

1. The DROP INDEX statement cannot be specified from an X/Open-compliant UAP running under OLTP.
2. If an SQL object for which procedures or triggers are in effect is nullified by specifying WITH PROGRAM, any information associated with the nullified procedures or triggers in the SQL_ROUTINE_RESOURCES dictionary table is deleted.
3. Before executing the SQL object associated with the procedures and triggers that were nullified by specifying WITH PROGRAM, you need to re-create the SQL object associated with the procedures and triggers by executing ALTER ROUTINE, ALTER PROCEDURE, or ALTER TRIGGER.
4. Regardless or whether or not WITH PROGRAM is specified, if there are procedures and triggers (exclusive of triggers that are defined for the table) associated with the table for which the index to be deleted is defined, any index information in the SQL object is nullified, in which case the triggers can no longer be executed. Because the procedure cannot be executed from another procedure or trigger, you need to re-create the SQL object.

Example

Delete the index (IDX1) defined for the product code (PCODE) column of a stock table (STOCK):

```
DROP INDEX IDX1
```

DROP [PUBLIC] PROCEDURE (Delete procedure, delete public procedure)

Function

`DROP PROCEDURE` deletes a procedure.

All users can delete these procedures (public procedures) without having to modify the routine identifier with the authorization identifier.

(1) DROP PROCEDURE (Delete procedure)

Privileges

Owner of the procedure

This user can delete his or her own procedures.

Users with the DBA privilege

These users can delete procedures belonging to other users.

Format

```
DROP PROCEDURE [authorization-identifier.]routine-identifier
          [WITH PROGRAM]
```

Operands

- [*authorization-identifier*.]*routine-identifier*

authorization-identifier

Specifies the authorization identifier of the user who owns the procedure being deleted.

When this operand is omitted, the authorization identifier of the user who is executing the command is assumed.

routine-identifier

Specifies the routine name of the procedure being deleted.

- `WITH PROGRAM`

When deleting a procedure, this option is specified if there is an SQL object for which functions, procedures, and triggers are in effect that use the procedure.

If `WITH PROGRAM` is omitted, and if there is an SQL object associated with functions, procedures, and triggers that use the function, the function cannot be deleted.

Common rules

1. If there is an SQL routine, or a routine or a trigger in the abstract data type that calls a specified procedure, the procedure is not deleted.
2. Procedures defined in an abstract data type cannot be deleted.
3. In the following cases, DROP PROCEDURE cannot be executed from within a Java procedure:
   - The SQL object being executed will be invalidated or deleted.
   - The Java procedure being executed will be deleted.

Notes

1. The DROP PROCEDURE statement cannot be specified from an X/Open-compliant UAP running under OLTP.
2. If an SQL object for which functions, procedures, or triggers are in effect is nullified by specifying WITH PROGRAM, any rows associated with the nullified functions, procedures, or triggers in the SQL_ROUTINE_RESOURCES dictionary table are deleted.
3. Before executing the SQL object associated with the function, procedure, or trigger that was nullified by specifying WITH PROGRAM, you need to re-create the function, procedure, or trigger by executing ALTER ROUTINE, ALTER PROCEDURE, or ALTER TRIGGER.
4. To execute the SQL object associated with the trigger that was nullified by specifying WITH PROGRAM, you need to perform either of the following operations:
   - Redefine the procedure, and execute either ALTER TRIGGER or ALTER ROUTINE to re-create the SQL object.
   - Delete the nullified trigger by using DROP TRIGGER, and then redefine the trigger by using CREATE TRIGGER so that the deleted procedures are not reused. If triggers satisfying all of the following conditions exist, use DROP TRIGGER to delete them all, and redefine the triggers using CREATE TRIGGER in the order in which they were defined so that there is no change in the sequence of execution of trigger actions:

     Conditions:

     - The defined trigger is later than the nullified trigger.
     - The nullified trigger is the same as the defined table.
     - The nullified trigger is the same as the trigger event (INSERT, UPDATE, or DELETE) (for UPDATE, the nullified trigger is considered to be the same as the trigger event regardless of whether a trigger event column is specified or

the contents of the specification).

• The nullified trigger has the same trigger action timing (BEFORE or AFTER) as the trigger event.

• The nullified trigger has the same trigger action units (units of rows or statements) as the trigger event.

5. The identifier specified in the trigger action procedure cannot be specified in *routine-identifier*. To delete the trigger, you need to execute DROP TRIGGER.

(2) DROP PUBLIC PROCEDURE (Delete public procedure)

Privileges

Owner of the procedure

The owner of a (defined) public procedure can delete that procedure.

Users with the DBA privilege

These users can delete public procedures that are owned by other users.

Format

```
DROP PUBLIC PROCEDURE routine-identifier
                [WITH PROGRAM]
```

Operands

For descriptions of operands other than PUBLIC and *routine-identifier*, see *(1) DROP PROCEDURE (Delete procedure)* under *DROP [PUBLIC] PROCEDURE (Delete procedure, delete public procedure)* in this chapter.

■ PUBLIC

Specifies that a public procedure is to be deleted.

■ *routine-identifier*

Specifies the routine name of the public procedure to be deleted.

Common rules

1. In the following cases, DROP PUBLIC PROCEDURE cannot be executed from within a Java procedure:
   - The SQL object being executed will be invalidated or deleted.
   - The Java procedure being executed will be deleted.
2. For details about other common rules, see *(1) DROP PROCEDURE (Delete procedure)* under *DROP [PUBLIC] PROCEDURE (Delete procedure, delete*

public procedure) in this chapter.

Notes

1. `DROP PUBLIC PROCEDURE` cannot be specified from an X/Open-compliant UAP running under OLTP.
2. For other notes, see *(1) DROP PROCEDURE (Delete procedure)* under *DROP [PUBLIC] PROCEDURE (Delete procedure, delete public procedure)* in this chapter.

DROP SCHEMA (Delete schema)

Function

DROP SCHEMA deletes a schema defined in a schema definition.

Privileges

Owner of the schema

The owner of a schema can delete that schema.

Users with the DBA privilege

A user with the DBA privilege can delete schemas owned by other users.

Format

```
DROP SCHEMA authorization-identifier [WITH PROGRAM]
```

Operands

- *authorization-identifier* [WITH PROGRAM]

authorization-identifier

Specifies the authorization identifier of the user who owns the schema.

WITH PROGRAM

When deleting a schema, specify this operand to disable another user's valid SQL object for a function, procedure, or trigger that uses a base table, view table, index, abstract data type, routine, trigger, or sequence generator in a specified schema.

If WITH PROGRAM is omitted and another user has a valid SQL object for a function, procedure, or trigger that uses a base table, view table, index, abstract data type, procedure, trigger, or sequence generator in a schema, that schema cannot be deleted.

Common rules

1. All base tables, view tables (including public views), indexes, comments, access privileges, routines (including public procedures and public functions for which an authorization identifier is defined with DROP SCHEMA), triggers, abstract data types, index types, and sequence generators in a schema of a specified authorization identifier are deleted.
2. Even when WITH PROGRAM is specified, if another user has a valid SQL object for a function, procedure, or trigger that uses an abstract data type, function, procedure, trigger, or sequence generator in a specified schema, that schema

cannot be deleted.

3. The schema cannot be deleted if another user has a view table or public view of that uses the function of a specified schema.
4. The schema cannot be deleted if there is a table or index of another user that uses the abstract data type in the schema.
5. In the following cases, DROP SCHEMA cannot be executed from within a Java procedure:
   - The SQL object being executed will be invalidated or deleted.
   - The Java procedure being executed will be deleted.
6. If tables in the specified schema, indexes, and columns of the abstract data type containing LOB columns and LOB attributes that are defined for the table are stored in an RDAREA to which the inner replica facility is applied, the schema can be deleted, provided that certain conditions are met. For details about execution conditions for DROP SCHEMA under a condition in which the inner replica facility is used, see the manual *HiRDB Version 9 Staticizer Option*.
7. If a specified schema contains a falsification-prevented table and the falsification-prevented table contains rows, that schema cannot be deleted.
8. Users with the DBA privilege can delete schemas belonging to an auditor. However, if there is an audit trail table in the schema belonging to an auditor, the schema owned by the auditor cannot be deleted.

Notes

1. The DROP SCHEMA statement cannot be specified from an X/Open-compliant UAP running under OLTP.
2. If a valid SQL object of a function, procedure, trigger, or sequence generator is disabled by specifying WITH PROGRAM, the information about the disabled function, procedure, trigger, or sequence generator in the SQL_ROUTINE_RESOURCES dictionary table is deleted.
3. To execute a valid SQL object of a function, procedure, trigger, or sequence generator that has been disabled by specifying WITH PROGRAM, you must execute ALTER ROUTINE, ALTER PROCEDURE, or ALTER TRIGGER, to recreate the valid SQL object of the function, procedure, trigger, or sequence generator.

Example

Delete the schema owned by the user whose authorization identifier is USER1:

```
DROP SCHEMA
    AUTHORIZATION USER1
```

DROP SEQUENCE (Delete sequence generator)

Function

`DROP SEQUENCE` deletes a sequence generator.

Privileges

The owner of a sequence generator to be specified

The owner of a sequence generator can delete that sequence generator.

Users with the DBA privilege

These users can delete their own sequence generators and sequence generators owned by other users.

Format

```
DROP SEQUENCE [authorization-identifier.]sequence-generator-identifier[WITH PROGRAM]
```

Operands

- [*authorization-identifier*.]*sequence-generator-identifier*

authorization-identifier

Specifies the authorization identifier of the user that owns the sequence generator.

If this is omitted, the authorization identifier of the executing user is assumed.

sequence-generator-identifier

Specifies the name of the sequence generator to be deleted.

`WITH PROGRAM`

Specify this operand to disable another user's valid SQL object for a procedure or trigger that uses a sequence generator when deleting a sequence generator.

The sequence generator cannot be deleted if there is a valid SQL object of a trigger or procedure that uses the sequence generator and `WITH PROGRAM` is omitted.

Common rules

1. If the SQL object being executed is invalidated, `DROP PUBLIC FUNCTION` cannot be executed from within a Java procedure.

Notes

1. DROP SEQUENCE cannot be specified from an X/Open-compliant UAP running under OLTP.
2. If a valid SQL object of a procedure or trigger is disabled by specifying WITH PROGRAM, the information about the disabled procedure or trigger in the SQL_ROUTINE_RESOURCES dictionary table is deleted.
3. To execute an SQL object of a procedure or trigger that has been disabled by specifying WITH PROGRAM, you must execute ALTER ROUTINE or ALTER PROCEDURE to recreate the SQL object of the procedure or trigger.

Examples

Deletes a sequence generator (SEQ1).

```
DROP SEQUENCE SEQ1
```

DROP TABLE (Delete table)

Function

DROP TABLE deletes a table.

Privileges

Owner of the table

The owner of a table can delete that table.

Users with the DBA privilege

A user with the DBA privilege can delete tables owned by other users.

Format

```
DROP TABLE [authorization-identifier.]table-identifier [WITH PROGRAM]
```

Operands

- [*authorization-identifier*.]*table-identifier* [WITH PROGRAM]

authorization-identifier

Specifies the authorization identifier of the user who owns the table.

When this operand is omitted, the authorization identifier of the user who is executing the command is assumed.

table-identifier

Specifies the name of the table to be deleted.

WITH PROGRAM

Specify this operand when deleting a table and when disabling any function that uses the table in an SQL procedure statement, a procedure, or an SQL object with a trigger that is in effect.

If WITH PROGRAM is omitted, the table cannot be deleted if there is a function, a procedure, or an SQL object with an effective trigger that uses the table (exclusive of triggers for performing a referential constraint operation that are internally defined for the referenced table referenced by the table being deleted).

If a referencing table is deleted, any of the functions, procedures, and SQL objects with a trigger that is in effect shown in the following table are disabled:

Table 3-44: Disabled objects

Version in which the object was created	Description of object	
	Object type	Nullifying condition
07-00 or later	Function, procedure, or trigger object	The object contains either an UPDATE or DELETE statement that uses a referenced table referenced by the referencing table.
Before 07-00		The object contains an SQL statement that uses a referenced table referenced by the referencing table.

Common rules

1. Deleting a table also causes any indexes, view tables (including public views), comments, access privileges, and triggers defined for the table to be deleted.

   If a referencing table is deleted, any triggers for performing referential constraint operations internally defined on the referenced table, are also deleted.

2. DROP TABLE cannot be executed from within a Java procedure if the deletion results in invalidation of the SQL object being executed.

3. If tables, indexes defined for a table, and columns of the abstract data type containing LOB columns and LOB attributes that are defined for a table are stored in an RDAREA to which the inner replica facility is applied, such items can be deleted, provided that certain conditions are met. For details about execution conditions for DROP TABLE under a condition in which the inner replica facility is used, see the manual *HiRDB Version 9 Staticizer Option*.

4. If the frozen update specification is specified for a LOB column defined for a table or for an RDAREA that stores columns of the LOB attribute, that table cannot be deleted.

5. If a specified table is a falsification-prevented table and the falsification-prevented table contains rows, that table cannot be deleted.

6. Users with the DBA privilege can delete tables owned by other users, but not audit trail tables owned by an auditor.

7. Referenced tables that are referenced by a foreign key cannot be deleted.

Notes

1. The DROP TABLE statement cannot be specified from an X/Open-compliant UAP running under OLTP.

2. If an SQL object for which functions, procedures, or triggers are in effect is nullified by specifying WITH PROGRAM, any information associated with the nullified functions, procedures, or triggers in the SQL_ROUTINE_RESOURCES

dictionary table is deleted.

3. Before executing the SQL object associated with the function, procedure, or trigger that was nullified by specifying `WITH PROGRAM`, you need to re-create the function, procedure, or trigger by executing `ALTER ROUTINE`, `ALTER PROCEDURE`, or `ALTER TRIGGER`.
4. To execute the SQL object associated with the trigger that was nullified by specifying `WITH PROGRAM`, you need to perform either of the following operations:
   - Redefine the table, and re-create the trigger SQL object by executing either `ALTER TRIGGER` or `ALTER ROUTINE`.
   - Delete the nullified trigger by using `DROP TRIGGER`, and then redefine the trigger by using `CREATE TRIGGER` so that the deleted tables are not reused. If triggers satisfying all of the following conditions exist, use `DROP TRIGGER` to delete them all, and redefine the triggers using `CREATE TRIGGER` in the order in which they were defined so that there is no change in the sequence of execution of trigger actions:

     Conditions:

     - The defined trigger is later than the nullified trigger.
     - The nullified trigger is the same as the defined table.
     - The nullified trigger is the same as the trigger event (`INSERT`, `UPDATE`, or `DELETE`) (for `UPDATE`, the nullified trigger is considered to be the same as the trigger event regardless of whether a trigger event column is specified or the contents of the specification).
     - The nullified trigger has the same trigger action timing (`BEFORE` or `AFTER`) as the trigger event.
     - The nullified trigger has the same trigger action units (units of rows or statements) as the trigger event.

Example

Delete a stock table (`STOCK`):

```
DROP TABLE STOCK
```

DROP TRIGGER (Delete a trigger)

Function

`DROP TRIGGER` deletes a trigger.

Privileges

Owner of a trigger

This user can delete his or her own triggers.

Users with the DBA privilege

These users can delete their own triggers and triggers owned by other users.

Format

```
DROP TRIGGER [authorization-identifier.]trigger-identifier[WITH PROGRAM]
```

Operands

- [*authorization-identifier*.]*trigger-identifier*[WITH PROGRAM]

authorization-identifier

Specifies the authorization identifier of the owner of the trigger to be deleted.

The default is the authorization identifier of the user issuing `DROP TRIGGER`.

trigger-identifier

Specifies the name of the trigger to be deleted.

`WITH PROGRAM`

When deleting a trigger, this option is specified to nullify an SQL object for which functions, procedures, and triggers that use the trigger are in effect.

If `WITH PROGRAM` is omitted, and if there is an SQL object for which functions, procedures, and triggers that use a trigger are in effect, the trigger cannot be deleted.

Common rule

1. `DROP TRIGGER` cannot be executed from within a Java procedure if the following conditions are met:
   - The SQL object being executed is nullified or deleted.
   - The Java procedure being executed is deleted.

Notes

1. DROP TABLE cannot be specified from an X/Open compliant UAP running under OLTP.
2. If an SQL object for which functions, procedures, or triggers are in effect is nullified by specifying WITH PROGRAM, any information associated with the nullified functions, procedures, or triggers in the SQL_ROUTINE_RESOURCES dictionary table is deleted.
3. Before executing the SQL object associated with the function, procedure, or trigger that was nullified by specifying WITH PROGRAM, you need to re-create the function, procedure, or trigger by executing ALTER ROUTINE, ALTER PROCEDURE, or ALTER TRIGGER.

Example

Delete the trigger (TRIG1).

```
DROP TRIGGER TRIG1
```

DROP [PUBLIC] VIEW (Delete view table, delete public view table)

Function

DROP VIEW deletes a view table.

All users can delete these view tables (public view) without having to modify the table identifier with the authorization identifier.

(1) DROP VIEW (Delete view table)

Privileges

Owner of the view table

The owner of a view table can delete that view table.

Users with the DBA privilege

Users with the DBA privilege can delete view tables owned by other users.

Format

```
DROP VIEW [authorization-identifier.]table-identifier [WITH PROGRAM]
```

Operands

- [*authorization-identifier*.]*table-identifier* [WITH PROGRAM]

authorization-identifier

Specifies the authorization identifier of the user who owns the view table.

When this operand is omitted, the authorization identifier of the user who is executing the command is assumed.

table-identifier

Specifies the name of the view table to be deleted.

WITH PROGRAM

When deleting a view table, this operand is specified to nullify the SQL object for which functions, procedures, and triggers that use the view table are in effect.

If WITH PROGRAM is omitted, and if there is an SQL object for which functions, procedures, and triggers that use a view table are in effect, that view table cannot be deleted.

Common rules

1. DROP VIEW deletes the access privileges for the view table as well as any view tables that are defined from the view table that is to be deleted.

2. DROP VIEW cannot be executed from within a Java procedure if the deletion results in invalidation of the SQL object being executed.

Notes

1. The DROP VIEW statement cannot be specified from an X/Open-compliant UAP running under OLTP.
2. If an SQL object for which functions, procedures, or triggers are in effect is nullified by specifying WITH PROGRAM, any information associated with the nullified functions, procedures, or triggers in the SQL_ROUTINE_RESOURCES dictionary table is deleted.
3. Before executing the SQL object associated with the function, procedure, or trigger that was nullified by specifying WITH PROGRAM, you need to re-create the function, procedure, or trigger by executing ALTER ROUTINE, ALTER PROCEDURE, or ALTER TRIGGER.
4. To execute the SQL object for the trigger that was nullified by specifying WITH PROGRAM, you need to perform one of the following operations:
   - Redefine the view table, and re-create the trigger SQL object by executing either ALTER TRIGGER or ALTER ROUTINE.
   - Delete the nullified trigger by using DROP TRIGGER and then redefine the trigger by using CREATE TRIGGER so that the deleted view table is not used. However, if there are triggers that satisfy all of the following conditions, they should all be deleted by using DROP TRIGGER and redefined by using CREATE TRIGGER in the order in which they were defined so that the order in which trigger actions are executed does not change:

     Conditions:

     - The defined trigger is later than the nullified trigger.
     - The nullified trigger is the same as the defined table.
     - The nullified trigger is the same as the trigger event (INSERT, UPDATE, or DELETE) (for UPDATE, the nullified trigger is considered to be the same as the trigger event regardless of whether a trigger event column is specified or the contents of the specification).
     - The nullified trigger has the same trigger action timing (BEFORE or AFTER) as the trigger event.
     - The nullified trigger has the same trigger action units (units of rows or statements) as the trigger event.

Example

Delete view table VSTOCK1 defined from stock table STOCK:

```
DROP VIEW VSTOCK1
```

(2) DROP PUBLIC VIEW (Delete public view)

Privileges

Owner (definer) of a specified public view

A user can delete any public view that he or she owns (defines).

User with DBA privileges

These users can delete public views owned by other users.

Format

```
DROP PUBLIC VIEW table-identifier [WITH PROGRAM]
```

Operands

For descriptions of operands other than PUBLIC and *table-identifier*, see *(1) DROP VIEW (Delete view table)* under *DROP [PUBLIC] VIEW (Delete public view, delete public view table)* in this chapter.

- PUBLIC

Specify this operand when deleting a public view.

- *table-identifier*

Specifies the name of the public view to be deleted.

Common rules

For details about the common rules, see *(1) DROP VIEW (Delete view table)* under *DROP [PUBLIC] VIEW (Delete public view, delete public view table)* in this chapter.

Notes

For the notes, see *(1) DROP VIEW (Delete view table)* under *DROP [PUBLIC] VIEW (Delete public view, delete public view table)* in this chapter.

Example

Delete the public view (PVSTOCK1) defined for an inventory table (STOCK).

```
DROP PUBLIC VIEW PVSTOCK1
```

GRANT Format 1 (Grant privileges)

Function

GRANT format 1 grants to users the DBA, schema definition, CONNECT, and private user RDAREA usage privileges (see subsection *(1)* below); it also grants access privileges to users (see subsection *(2)* below).

(1) GRANT DBA (grant DBA privilege), GRANT SCHEMA (grant schema definition privilege), GRANT CONNECT (grant CONNECT privilege), and GRANT RDAREA (grant RDAREA usage privilege)

Privileges

Users with the DBA privilege

A user with the DBA privilege can grant the DBA, schema definition, CONNECT, and private user RDAREA usage privileges.

Users with the CONNECT privilege

A user with the CONNECT privilege can change passwords.

Format

```
GRANT {DBA TO authorization-identifier[, authorization-identifier]...
          [IDENTIFIED BY password[, password]...]
       |SCHEMA TO authorization-identifier[, authorization-identifier]...
       |CONNECT TO authorization-identifier[, authorization- identifier]...
          [IDENTIFIED BY password[, password]...]
       |RDAREA RDAREA-name [RDAREA-name]...
        TO {authorization-identifier[, authorization-identifier]...|PUBLIC}}
```

Operands

- DBA TO *authorization-identifier*[, *authorization-identifier*]...

  [IDENTIFIED BY *password*[, *password*]...]

DBA TO

Specifies that the DBA privilege is to be granted to one or more users.

authorization-identifier[, *authorization-identifier*]...

Specifies the authorization identifiers of the users to whom the DBA privilege is to be granted.

IDENTIFIED BY *password*[, *password*]...

Specifies passwords for the users to whom the DBA privilege is to be granted.

The following table indicates the relationship between the IDENTIFIED BY clause of the GRANT DBA statement and the user privilege.

Table 3-45: Relationship between the IDENTIFIED BY clause of the GRANT DBA statement and the user privilege

User privilege		IDENTIFIED BY clause of GRANT DBA statement	
		Specified	Not specified
Already has CONNECT privilege	Has password	Grants DBA privilege and changes to the specified password.	Grants DBA privilege.
	No password	Grants DBA privilege and changes to the specified password.	KFPA11571 error
No CONNECT privilege		Grants CONNECT privilege, DBA privilege, and the specified password.	KFPA11571 error

- SCHEMA TO *authorization-identifier*[, *authorization-identifier*]...

SCHEMA TO

Specifies that the schema definition privilege is to be granted to one or more users.

authorization-identifier[, *authorization-identifier*]...

Specifies the authorization identifiers of the users to whom the schema definition privilege is to be granted.

- CONNECT TO *authorization-identifier*[, *authorization-identifier*]...

  [IDENTIFIED BY *password*[, *password*]...]

CONNECT TO

Specifies that the CONNECT privilege is to be granted to one or more users.

authorization-identifier[, *authorization-identifier*]...

Specifies the authorization identifiers of the users to whom the CONNECT privilege is to be granted.

IDENTIFIED BY *password*[, *password*]...

Specifies passwords for the users to whom the CONNECT privilege is to be granted.

The following table indicates the relationship between the IDENTIFIED BY clause and of the GRANT CONNECT statement the user privilege.

Table 3-46: Relationship between the IDENTIFIED BY clause of the GRANT CONNECT statement and the user privilege

User privilege			Password character restriction definition	IDENTIFIED BY clause of GRANT CONNECT statement	
				Specified	Not Specified
Already has CONNECT privilege	Has password	Has DBA privilege	--	Changes to the specified password.	KFPA11571-E error
		No DBA privilege	No	Changes to the specified password.	Changes to No password.
			Yes	Changes to the specified password.	KFPA19634-E error
	No password	No DBA privilege	No	Changes to the specified password.	Changes to No password.
			Yes	Changes to the specified password.	KFPA19634-E error
No CONNECT privilege			No	Grants CONNECT privilege and password.	Grants CONNECT privilege only with No password.
			Yes	Grants CONNECT privilege and password.	KFPA19634-E error

Legend:

--: Not applicable.

- RDAREA *RDAREA-name* [*RDAREA-name*]...

  TO {*authorization-identifier*[, *authorization-identifier*]...|PUBLIC}

RDAREA *RDAREA-name*[, *RDAREA-name*]

Specifies the names of the RDAREAs for which the usage privilege is to be granted.

authorization-identifier[, *authorization-identifier*]...

Specifies the authorization identifiers of the users to whom the RDAREA usage privilege is to be granted for the specified RDAREAs.

PUBLIC

Specifies that the specified RDAREAs are to be public user RDAREAs.

Common rules

1. The usage privilege cannot be granted to a replica RDAREA through the use of

GRANT RDAREA.

2. For details about a GRANT execution condition when the inner replica facility is in use, see the manual *HiRDB Version 9 Staticizer Option*.
3. If the password character limit enhancement facility is used, this command checks the password character limit during the execution of the GRANT DBA or GRANT CONNECT statement.

Notes

1. A maximum of 16 private user RDAREAs and 1,600 authorization identifiers can be specified.
2. The schema definition privilege cannot be granted to users who do not have the DBA or CONNECT privilege.
3. The RDAREA usage privilege for public user RDAREAs cannot be granted to individual users, nor can RDAREAs granted to individual users be defined as public user RDAREAs (PUBLIC cannot be specified for such an RDAREA).
4. The DBA privilege includes the CONNECT privilege.
5. When the DBA privilege is granted to a user, a password must also be specified for that user. Even when the DBA privilege is granted, a user who does not have a password cannot use the DBA privilege.
6. A user can change a password assigned to that user by specifying the GRANT CONNECT option. In this case, the individual who issues the GRANT statement can change his or her password solely on the basis of his or her CONNECT privilege.
7. The GRANT statement cannot be specified from an X/Open-compliant UAP running under OLTP.
8. When specifying more than one authorization identifier in the GRANT DBA and GRANT CONNECT statements and granting privileges to multiple users simultaneously, you cannot partially omit the password specified in the IDENTIFIED BY clause; either specify passwords in a number equal to the number of specified authorization identifiers or omit the entire IDENTIFIED BY clause. However, if no IDENTIFIED BY clauses are specified in the GRANT statements, a specification without a password that attempts to register a DBA will result in an error (see *Table 3-45* and *Table 3-46*), and all GRANT statements will be disabled. In this case, execute the command by specifying the error-generating user with the IDENTIFIED BY clause in another GRANT statement.
9. If multiple authorization identifiers are specified in the GRANT DBA and GRANT CONNECT statements while using the password character limit facility, resulting in privileges being granted to multiple users at that same time, if even one person enters a specification that violates the password character limit, an error will

result (see *Table 3-45* and *Table 3-46*), and all GRANT statements will be disabled.

10. The GRANT DBA statement cannot be used to grant DBA privileges to or change passwords for a user who is in a password-invalid account lock state. In this case, the GRANT CONNECT statement should be used to clear the password-invalid account lock state, and then the GRANT DBA statement should be executed.
11. The DBA privilege cannot be granted to auditors.
12. If the RDAREA usage privilege for an RDAREA is granted solely to an auditor, other users cannot be granted the RDAREA usage privilege for that RDAREA.

Examples

1. Grant the DBA privilege to the user whose authorization identifier is USER1 and assign the password PSWD:
```
GRANT DBA TO USER1 IDENTIFIED BY PSWD
```

2. Grant the schema definition privilege to the user whose authorization identifier is USER2:
```
GRANT SCHEMA TO USER2
```

3. Grant the CONNECT privilege to the user whose authorization identifier is USER3 and assign the password PSWD:
```
GRANT CONNECT TO USER 3
     IDENTIFIED BY PSWD
```

4. Change the password for the user (*authorization-identifier*: USER3) to ABCD.
```
  GRANT CONNECT TO USER3
      IDENTIFIED BY ABCD
```

5. Grant the RDAREA usage privilege (for RDAREAs RDA1 and RDA2) to users whose authorization identifiers are USER4, USER5, and USER6:
```
GRANT RDAREA RDA1,RDA2
  TO USER4,USER5,USER6
```

6. Define RDAREA RDA3 as a public user RDAREA:
```
GRANT RDAREA RDA3 TO PUBLIC
```

(2) GRANT access privilege (grant access privileges)

Privileges

Owner of the table

This user can grant to other users his or her access privilege with respect to base tables that the user owns, and view tables owned by user and defined from his or her base table, or view table. However, if the user receives the SELECT privilege with respect to a table owned by another user and defines a view table, subsequently he or she cannot grant this access privilege to other users.

Format

```
GRANT {access-privilege[, access-privilege] |ALL [PRIVILEGES]}
  ON [authorization-identifier.]table-identifier
  TO {authorization-identifier[, authorization-identifier]...|PUBLIC}

access-privilege::={SELECT|INSERT|DELETE|UPDATE}
```

Operands

- {*access-privilege*[, *access-privilege*] |ALL [PRIVILEGES]}

access-privilege

Specifies an access privilege that is to be granted for a specified table. Identical access privileges cannot be specified.

ALL [PRIVILEGES]

Specifies that all applicable access privileges are to be granted for the specified table.

If the specified table is a view table, the ALL option grants all access privileges possessed by the owner of the view table.

- *access-privilege*::={SELECT|INSERT|DELETE|UPDATE}

SELECT

Specifies that the SELECT privilege is to be granted.

INSERT

Specifies that the INSERT privilege is to be granted.

DELETE

Specifies that the DELETE privilege is to be granted.

UPDATE

Specifies that the UPDATE privilege is to be granted.

- ON [*authorization-identifier*.]*table-identifier*

authorization-identifier

Specifies the authorization identifier of the owner of the table for which the specified access privileges are to be granted.

To grant access privileges to public views, specify the word PUBLIC in *authorization-identifier*, in uppercase characters enclosed in double quotation marks (").

table-identifier

Specifies the name of the table for which the specified access privileges are to be granted.

- TO {*authorization-identifier*[, *authorization-identifier*]...|PUBLIC}

authorization-identifier

Specifies the authorization identifier of a user to whom the specified access privileges for the specified table are to be granted. A maximum of 1,600 user authorization identifiers can be specified. Duplicate authorization identifiers are not allowed.

PUBLIC

Specifies that the specified access privileges to the specified table are to be granted to all users.

Common rule

1. For details about the GRANT execution conditions under the inner replica facility, see the manual *HiRDB Version 9 Staticizer Option*.

Notes

1. The GRANT statement cannot be specified from an X/Open-compliant UAP running under OLTP.
2. INSERT, UPDATE, or DELETE privileges on audit trails or view tables based on an audit trail table cannot be granted to users other than an auditor.

Examples

1. Grant the SELECT access privilege for the stock (STOCK) table to the user whose authorization identifier is USER1:
```
GRANT SELECT
 ON STOCK TO USER1
```
2. Grant all access privileges for the stock table (STOCK) to all users:
```
GRANT ALL
 ON STOCK TO PUBLIC
```

GRANT Format 2 (Change auditor's password)

Function

`GRANT` format 2 changes the auditor's password.

Privileges

Users with the audit privilege

These users can change the password owned by an auditor.

Format

```
GRANT AUDIT IDENTIFIED BY password
```

Operands

- *password*

Specifies a new password for the auditor.

If the password is to be case-sensitive, specify it by enclosing it in double quotation marks (`"`).

Examples

Change the auditor's password to `a0h7Fc3`:

```
GRANT AUDIT IDENTIFIED BY "a0h7Fc3"
```

REVOKE (Revoke privileges)

Function

REVOKE revokes the DBA, schema definition, CONNECT, and private user RDAREA usage privileges (see subsection *(1)* below); it also revokes access privileges granted to users (see subsection *(2)* below).

(1) REVOKE DBA (revoke DBA privilege), REVOKE SCHEMA (revoke schema definition privilege), REVOKE CONNECT (revoke CONNECT privilege), and REVOKE RDAREA (revoke RDAREA usage privilege)

Privileges

Users with the DBA privilege

A user with the DBA privilege can revoke the DBA, schema definition, CONNECT, and private user RDAREA usage privileges.

Format

```
REVOKE {DBA FROM authorization-identifier[, authorization-identifier]...
       |SCHEMA FROM authorization-identifier
          [, authorization-identifier]...
       |CONNECT FROM authorization-identifier
          [, authorization-identifier]...
       | [RDAREA RDAREA-name [,RDAREA-name]...
          FROM {authorization-identifier
                  [, authorization-identifier]...|PUBLIC}
```

Operands

- DBA FROM *authorization-identifier*[, *authorization-identifier*]...

DBA FROM

Specifies that one or more users' DBA privilege is to be revoked.

authorization-identifier[, *authorization-identifier*]...

Specifies the authorization identifiers of the users whose DBA privilege is to be revoked.

- SCHEMA FROM *authorization-identifier*[, *authorization-identifier*]...

SCHEMA FROM

Specifies that one or more users' schema definition privilege is to be revoked.

authorization-identifier[, *authorization-identifier*]...

Specifies the authorization identifiers of the users whose schema definition privilege is to be revoked.

- CONNECT FROM *authorization-identifier*[, *authorization-identifier*]...

CONNECT FROM

Specifies that one or more users' CONNECT privilege is to be revoked.

authorization-identifier[, *authorization-identifier*]...

Specifies the authorization identifiers of the users whose CONNECT privilege is to be revoked.

- RDAREA *RDAREA-name*[, *RDAREA-name*]...

  FROM {*authorization-identifier*[, *authorization-identifier*]...|PUBLIC}

RDAREA-name[, *RDAREA-name*]

Specifies the names of RDAREAs for which the RDAREA usage privilege is to be revoked.

authorization-identifier[, *authorization-identifier*]...

Specifies the authorization identifiers of the users whose usage privilege for the specified RDAREAs is to be revoked.

PUBLIC

Specifies that the usage privilege for the specified RDAREAs as public user RDAREAs is to be revoked.

Notes

1. REVOKE can revoke a privilege that has not been granted or a privilege that has already been revoked.
2. When revoking a user's privilege for an RDAREA, if that user has a table, index, or sequence generator in that RDAREA, their privilege cannot be revoked.
3. The schema privilege for a specific schema cannot be revoked if the schema contains tables.
4. A maximum of 16 private user RDAREAs can be specified.
5. A maximum of 1,600 authorization identifiers can be specified.
6. Privileges granted with the PUBLIC option of GRANT must be revoked with the PUBLIC option of REVOKE.
7. A user cannot revoke his or her own DBA privilege.
8. The CONNECT privilege of a user who has either the DBA privilege or a schema cannot be revoked.

9. Revoking the CONNECT privilege also revokes the schema definition privilege.
10. The REVOKE statement cannot be specified from an X/Open-compliant UAP running under OLTP.
11. The auditor's schema definition privilege or CONNECT privilege cannot be revoked.

Examples

1. Revoke the DBA privilege for the user whose authorization identifier is USER1:
```
REVOKE DBA FROM USER1
```

2. Revoke the schema definition privilege for the user whose authorization identifier is USER2:
```
REVOKE SCHEMA FROM USER2
```

3. Revoke the CONNECT privilege for the user whose authorization identifier is USER3:
```
REVOKE CONNECT FROM USER3
```

4. Revoke RDAREA usage privileges for RDAREAs RDA1 and RDA2 for the users whose authorization identifiers are USER4, USER5, and USER6:
```
REVOKE RDAREA RDA1,RDA2
  FROM USER4,USER5,USER6
```

5. Redefine public user RDAREA RDA3 as a private user RDAREA and grant RDAREA usage privilege to USER1:
```
REVOKE RDAREA RDA3 FROM PUBLIC
GRANT RDAREA RDA3 TO USER1
```

(2) REVOKE access privilege (revoke access privileges)

Privileges

Owner of the table

The owner of a table can revoke an access privilege granted with GRANT access privilege.

Format

```
REVOKE {access-privilege[, access-privilege]...|All [privilege]}
   ON  [authorization-identifier.]table-identifier
   FROM [authorization-identifier[, authorization-identifier]...| PUBLIC}
     [WITH PROGRAM]
```

access-privilege::={SELECT|INSERT|DELETE|UPDATE}

Operands

- {*access-privilege*[, *access-privilege*]...|All [privilege]}

access-privilege

Specifies an access privilege for a specified table that is to be revoked. Identical access privileges cannot be specified.

ALL [PRIVILEGES]

Specifies that all applicable access privileges for a specified table are to be revoked.

- *access-privilege* : := {SELECT|INSERT|DELETE|UPDATE}

SELECT

Specifies that the SELECT privilege is to be revoked.

INSERT

Specifies that the INSERT privilege is to be revoked.

DELETE

Specifies that the DELETE privilege is to be revoked.

UPDATE

Specifies that the UPDATE privilege is to be revoked.

- ON [*authorization-identifier* .]*table-identifier*

authorization-identifier

Specifies the authorization identifier of the owner of the table whose access privileges are to be revoked.

To revoke access privileges to public views, specify the word PUBLIC in *authorization-identifier*, in uppercase characters enclosed in double quotation marks (").

table-identifier

Specifies the name of the table to which access privileges are to be revoked.

- FROM [*authorization-identifier*[, *authorization-identifier*] ... |PUBLIC} [WITH PROGRAM]

authorization-identifier

Specifies the authorization identifier of a user whose access privileges for the specified table are to be revoked. A maximum of 1,600 user authorization identifiers can be specified. Duplicate authorization identifiers are not allowed.

PUBLIC

Specifies that the privilege granted by means of the PUBLIC option of GRANT is to be revoked.

WITH PROGRAM

When revoking the SELECT privilege for a table that is the base for a view table, this option is specified to nullify the SQL object for which functions, procedures, and triggers are in effect that use the view table that is deleted by the revocation of the SELECT privilege.

Common rules

1. A user cannot revoke his or her own access privileges.
2. REVOKE can revoke a privilege that has not been granted or a privilege that has already been revoked.
3. Any privilege granted using the PUBLIC option of GRANT must be revoked using the PUBLIC option of REVOKE.

   If a privilege was granted by specifying the PUBLIC option and it is desired to block access by a specific user, the privilege must be revoked using the PUBLIC option and regranted to the desired users by specifying their authorization identifiers.
4. When the table owner revokes the SELECT privilege that was granted to another person, any view table defined by that person using that table is deleted.
5. The following shows the rules for revoking access privileges to tables used in a view definition or the access privileges to the base table for a view table.

   Defining view table V1:

```
CREATE VIEW V1[1] (VPCODE,VPNAME,VPRICE)
   AS SELECT PCODE, PNAME, PRICE
         FROM STOCK[2] WHERE PCODE =
         (SELECT PCODE FROM ORDERS[3])
```

   Defining view table V2:

```
CREATE VIEW V2[4] (VSPCODE, VSDATE_IN_STOCK)
   AS SELECT PCODE, DATE_IN_STOCK
         FROM INTO_STOCK[5]
```

   Defining view table VV1 from view table V1:

```
CREATE VIEW VV1[6] (VVPCODE, VVPRICE)
   AS SELECT VPCODE, VPRICE
         FROM V1[1] WHERE PCODE =
         (SELECT VSPCODE FROM V2[4])
```

```
WHERE VSDATE_IN_STOCK >
      DATE('1995-09-21'))
```

- When the SELECT privilege of the owner of a view table for a table ([2], [3], [5]) used in a view definition is revoked, both the view table itself ([1], [4]) and any other view tables ([6]) that are defined in terms of the view table are also deleted.

Example 1:

When the SELECT privilege of the owner of the view table for ORDERS table ([5]) is revoked, both V1 ([4]) and VV1 ([6]) are deleted.

Example 2:

When the SELECT privilege of the owner of the view table for INTO_STOCK table ([5]) is revoked, both V2 ([4]) and VV1 ([6]) are deleted.

- When the access privilege of the owner of the view table for a base table ([2], [5]) for a view table is revoked, the access privilege to the view table itself is also revoked ([1], [4]) as well as the access privileges to any other view tables ([6]) that are defined in terms of the view table. In this context, the term *access privilege* refers to privileges other than the SELECT privilege.

Example 1:

When the access privilege of the owner of the view table for STOCK table ([2]) is revoked, access privileges to V1 ([1]) and VV1 ([6]) are also revoked.

Example 2:

When the access privilege of the owner of the view table for INTO_STOCK table ([5]) is revoked, the access privilege to V2 ([4]) is also revoked.

To regrant an access privilege that has been revoked, reassign the access privilege to the base table on which the privilege-revoke view table is defined, delete the view table, and then redefine the view table.

For Example 1, grant access privileges to the owner of the STOCK view table ([2]), delete V1 ([1]) (this also deletes VV1), and then redefine V1 ([1]) and VV1 ([6]) in the indicated order.

For Example 2, grant access privileges to the owner of the INTO_STOCK view table ([5]), delete V2 ([4]), and then redefine V2 ([4]).

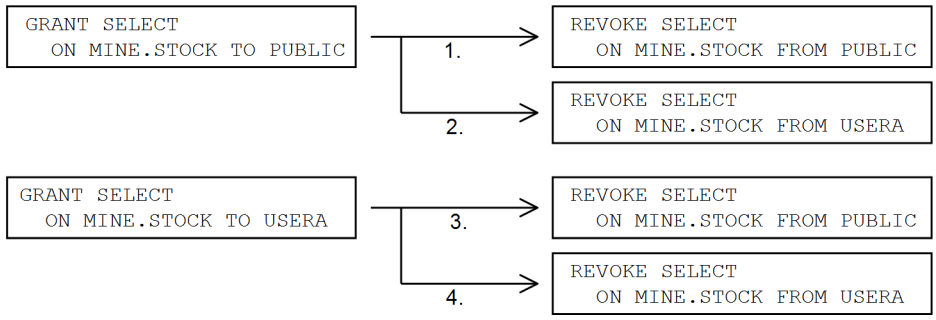
6. If WITH PROGRAM is omitted, and if there is an SQL object for which a procedure and trigger are in effect that use the view table that is deleted by the revocation of the SELECT privilege, that privilege cannot be deleted.

7. REVOKE cannot be executed from within a Java procedure if execution results in invalidation of the SQL object being executed.

Note

1. The results of revocation of privileges granted to PUBLIC and to a specific user vary depending on the combination. The following figure shows the possible combinations.

   Figure 3-1: Revocation of privileges granted to PUBLIC and to a specific user



Explanation:
1. Privileges granted to all users are revoked.
2. No privileges are granted to USERA, so nothing is revoked.
3. No privileges are granted to any user, so nothing is revoked.
4. Privileges granted to USERA are revoked.

2. The REVOKE statement cannot be specified from an X/Open-compliant UAP running under OLTP.
3. If WITH PROGRAM is specified to nullify an SQL object for which a function, a procedure, and a trigger are in effect, any information in the SQL_ROUTINE_RESOURCES dictionary table about the nullified function, procedure, and trigger is deleted.
4. Before executing the SQL object of a function, procedure, and trigger that is nullified by specifying WITH PROGRAM, you need to execute ALTER ROUTINE, ALTER PROCEDURE, or ALTER TRIGGER to re-create the SQL object for which the function, procedure, and trigger are in effect.

Examples

1. Revoke the privilege (SELECT privilege) to retrieve the stock table (STOCK) for the user whose authorization identifier is USER1:
```
REVOKE SELECT ON STOCK FROM USER1
```

2. For the stock table (STOCK), revoke only the DELETE privilege from the access privileges that were granted to all users:
```
REVOKE DELETE ON STOCK FROM PUBLIC
```

Chapter 4. Data Manipulation SQL

This chapter explains the syntax and structure of the data manipulation SQL.

PURGE TABLE statement (Delete all rows)
Single-row SELECT statement (Retrieve one row)
Dynamic SELECT statement Format 1 (Retrieve dynamically)
Dynamic SELECT statement Format 2 (Retrieve dynamically)
UPDATE statement Format 1 (Update data)
UPDATE statement Format 2 (Update data)
UPDATE statement Format 3, Format 4 (Update row using an array)
Preparable dynamic UPDATE statement: locating Format 1 (Update data using a preprocessable cursor)
Preparable dynamic UPDATE statement: locating Format 2 (Update data using a preprocessable cursor)
Assignment statement Format 1 (Assign a value to an SQL variable or SQL parameter)
Assignment statement Format 2 (Assign a value to an embedded variable or a ? parameter)

General rules

Types and functions of the data manipulation SQL

The data manipulation SQL performs operations on table data (retrieving, adding, deleting, and updating).

The following figure shows the types and functions of the data manipulation SQL.

Table 4-1: Types and functions of the data manipulation SQL

Type	Function
ALLOCATE CURSOR statement (Allocate a cursor)	Allocates the cursor to the SELECT statement preprocessed by the PREPARE statement or a group of result sets returned by a procedure.
ASSIGN LIST statement (Create list)	Creates a list from a base table.
CALL statement (Call procedure)	Calls a procedure.
CLOSE statement (Close cursor)	Closes a cursor.
DEALLOCATE PREPARE statement (Nullify the preprocessing of SQL statements)	Nullifies the SQL statement preprocessed by the PREPARE statement, and releases any allocated SQL statement identifier or extended statement name.
DECLARE CURSOR (Declare cursor)	Declares a cursor so that the results of a retrieval by the SELECT statement can be fetched row by row with the FETCH statement.
DELETE statement (Delete rows)	Deletes either the rows that satisfy specified search conditions or the row indicated by a cursor.
Preparable dynamic DELETE statement: locating (Delete row using a preprocessable cursor)	Deletes the row pointed to by a specified cursor. This command is used for dynamic execution.
DESCRIBE statement (Receive retrieval information and I/O information)	Returns to the SQL descriptor area retrieval information, output information, or input information on SQL preprocessed by the PREPARE statement.
DESCRIBE CURSOR statement (Receive cursor retrieval information)	Returns retrieval information for the cursor that references a result set returned by a procedure to the SQL descriptor area.
DESCRIBE TYPE statement (Receive definition information on a user-defined data type)	Returns to the SQL descriptor area definition information (attribute data codes and data lengths) on a user-defined data type that is directly or indirectly contained in SQL retrieval item information that is preprocessed by a PREPARE statement.
DROP LIST statement (Delete list)	Deletes a list.
EXECUTE statement (Execute SQL)	Executes an SQL preprocessed by the PREPARE statement.

Type	Function
EXECUTE IMMEDIATE statement (Prepare and execute SQL)	Prepares and executes an SQL provided in a character string.
FETCH statement (Fetch data)	Advances to the next row the cursor that indicates the row to be fetched, and reads column values in that row into the embedded variables specified in the INTO clause.
FREE LOCATOR statement (Invalidate locator)	Nullifies the locator.
INSERT statement (Insert rows)	Inserts rows into a table (a row can be inserted by direct specification of values or the SELECT statement can insert one or more rows).
OPEN statement (Open cursor)	Opens a cursor. Locates the cursor declared in a DECLARE CURSOR statement or a cursor allocated by an ALLOCATE CURSOR statement before the first row of retrieval results so that the retrieval results can be fetched.
PREPARE statement (Prepare SQL)	Prepares an SQL provided in a character string and assigns a name (SQL statement identifier or extended statement name) to the SQL.
PURGE TABLE statement (Delete all rows)	Deletes all rows in a base table.
Single-row SELECT statement (Retrieve one row)	Retrieves table data (fetches only one row of data from a table without using a cursor).
Dynamic SELECT statement (Retrieve dynamically)	Retrieves table data. The dynamic SELECT statement is preprocessed by the PREPARE statement. After a cursor is declared in a DECLARE CURSOR statement or it is allocated in an ALLOCATE CURSOR statement, the cursor is used to fetch retrieval results row by row.
UPDATE statement (Update data)	Updates the values of columns in the rows that satisfy specified search conditions or in the row indicated by a cursor.
Preparable dynamic UPDATE statement: locating (Update data using a preprocessable cursor)	Updates the value of a specified column of the row pointed to by a specified cursor. This command is used for dynamic execution.
Assignment statement (Assign a value)	Assigns a value to an assign-to item.

ALLOCATE CURSOR statement Format 1 (Allocate a statement cursor)

Function

Defines and allocates a cursor to the SELECT statement (dynamic SELECT statement) preprocessed by a PREPARE statement.

Privileges

None.

Format 1: Allocating a cursor to the SELECT statement (dynamic SELECT statement) preprocessed by a PREPARE statement

```
ALLOCATE extended-cursor-name CURSOR [WITH HOLD] FOR extended-statement-name
```

Operands

- *extended-cursor-name*

  Specifies the extended cursor name for the cursor to be allocated.

  For extended cursor names, see *2.27 Extended cursor name*.

- [WITH HOLD]

  Specify this operand when allocating a cursor as a holdable cursor. A holdable cursor, however, cannot be used in the following cases:

  - When specifying a table containing an abstract data type for which a plug-in is used in a FROM clause
  - A query for a named derived table that was derived by specifying a function call using a plug-in
  - A retrieval through a list

  For holdable cursors, see the *HiRDB Version 9 UAP Development Guide*.

- *extended-statement-name*

  Within the scope of this command, specifies an extended statement name that identifies the SELECT statement preprocessed by a PREPARE statement.

  For extended statement names, see *2.26 Extended statement name*.

Common rules

1. Any allocated cursor is closed.

2. If more than one cursor is allocated to an extended statement name that identifies a given SELECT statement, you cannot specify both a cursor allocation with a WITH HOLD specification and a cursor allocation without that specification.

Notes

1. An error may occur if a specified extended cursor name is already allocated in the prevailing scope.

Examples

1. Allocate a cursor (*extended-cursor-name*: cr (*scope*: GLOBAL)) to fetch rows, row by row, from the inventory table (STOCK):

```
PREPARE GLOBAL :sel FROM 'SELECT * FROM STOCK'
ALLOCATE GLOBAL :cr CURSOR FOR GLOBAL :sel
```

2. While retrieving all rows from the inventory table (STOCK) using a cursor (*extended-cursor-name*: cr (*scope*: GLOBAL, *value*: CR1)), dynamically update the unit price (PRICE) in the row at the cursor position to a 10% discount.

```
Set any name as the embedded variable :sel
PREPARE GLOBAL :sel FROM 'SELECT * FROM STOCK FOR UPDATE'
Assign CR1 to the embedded variable cr
ALLOCATE GLOBAL :cr CURSOR FOR GLOBAL :sel
PREPARE PRE1 FROM
'UPDATE SET PRICE = 10% discount value on the unit price WHERE CURRENT
OF GLOBAL CR1'
OPEN GLOBAL :cr
FETCH GLOBAL :cr INTO Name of the variable into which columns are fetched
EXECUTE PRE1
CLOSE GLOBAL :cr
DEALLOCATE PREPARE GLOBAL :sel
```

ALLOCATE CURSOR statement Format 2 (Allocate a result set cursor)

Function

Allocates a cursor to a group of ordered result sets that were returned from a procedure.

Privileges

None.

Format 2: Allocating a cursor to the group of result sets returned from a procedure

```
ALLOCATE extended-cursor-name FOR
   PROCEDURE [authorization-identifier.]routine-identifier
```

Operands

- *extended-cursor-name*

Specifies the extended cursor name for the cursor to be allocated.

For extended cursor names, see *2.27 Extended cursor name*.

- [*authorization-identifier*.] *routine-identifier*

authorization-identifier

Specifies the authorization identifier of the owner of the procedure that returned the result set to which a cursor is to be allocated.

If the cursor is allocated for a public procedure, specify upper-case `PUBLIC` enclosed in double quotation marks (`"`) as the authorization identifier.

routine-identifier

Specifies the name of a routine in the procedure that returned the result set to which a cursor is to be allocated.

Common rules

1. Specify a procedure that has already been called in the current SQL session (from the time HiRDB is connected to the time it is disconnected).
2. If a procedure specified in the current SQL session is called two times or more, the cursor is allocated to the group of result sets that was returned by the last called procedure.

3. An error may occur if a cursor is already allocated to the group of result sets returned by the last called procedure among the specified procedures.
4. The following return codes are assigned if a specified procedure does not return any result sets:
   - A return code `100` to the `SQLCODE` area of the SQL communications area
   - A return code `100` to the `SQLCODE` variable
   - A return code `02001` to the `SQLSTATE` variable
5. When the `ALLOCATE CURSOR` statement is executed, the cursor references the first result set among the result sets returned by the procedure, and it can fetch the data in the result set using a `FETCH` statement. The `CLOSE` statement is executed to reference the second result set and beyond. Upon execution of the `CLOSE` statement, if any of the following return codes is assigned, it indicates that another result set exists, and the cursor references the next result set:
   - A return code `121` to the `SQLCODE` area of the SQL communications area
   - A return code `121` to the `SQLCODE` variable
   - A return code `0100D` to the `SQLSTATE` variable

   On the other hand, upon execution of the `CLOSE` statement, if any of the following return codes is assigned, it indicates that another result set does not exist, in which case the extended cursor name does not identify any cursor:
   - A return code `100` to the `SQLCODE` area of the SQL communications area
   - A return code `100` to the `SQLCODE` variable
   - A return code `02001` to the `SQLSTATE` variable
6. The definition of the allocated cursor is the same as the cursor declaration in the procedure that generated the result set being referenced.
7. The allocated cursor remains open.
8. The allocated cursor is located at the cursor position that would be in effect when the procedure terminated.
9. If the cursor is allocated for a public procedure, specify upper-case `PUBLIC` enclosed in double quotation marks (`"`) as the authorization identifier.

Notes

1. An error may occur if a specified extended cursor name is already allocated in the prevailing scope.

Examples

See *1.9.3 Results-set return facility* for examples.

ASSIGN LIST statement Format 1 (Create list)

Function

The Format 1 ASSIGN LIST statement creates a list from a base table.

Privileges

A user who has the SELECT privilege for a base table can create a list from that base table.

Format 1: Creating a list from a base table

```
ASSIGN LIST list-name FROM ([authorization-identifier.] table-identifier)
       [WHERE search-condition]
       [WITHOUT LOCK [{WAIT|NOWAIT} ]]
       [ {WITH ROLLBACK|NO WAIT} ]
```

Operands

- *list-name*

Specifies a name for the list that is to be created.

If an existing list name is specified, the existing list is deleted and a new list is created.

- ([*authorization-identifier*.] *table-identifier*)

Specifies the authorization identifier of the table's owner and the name of the base table from which the list to be created.

The following tables cannot be specified in *table-identifier*:

- Shared tables
- Tables with WITHOUT ROLLBACK specification
- View tables
- Tables with columns specified in character sets other than the default character set

- [WHERE *search-condition*]

Specifies conditions for determining the rows that are to be retrieved.

If no search conditions are specified, the list will be created from all the rows in the specified table.

The ANDNOT logical operator can be specified in an ASSIGN LIST statement search condition. The ANDNOT logical operator is evaluated in the same priority order as AND.

The following table lists the results of ANDNOT logical operations:

Left side	Right side		
	T	F	?
T	F	T	T#
F	F	F	F
?	F	?	?

T: TRUE

F: FALSE

?: Indeterminate

#: C1 = V1 ANDNOT V2 = C2 is not equivalent to C1 = V1 AND NOT C2 = V2.

If *predicate-1* = (*predicate-2* ANDNOT *predicate-3*), the set satisfying *predicate-1* is the difference between the set satisfying *predicate-2* and the set satisfying *predicate-3*.

If *predicate-1* = (*predicate-2* AND NOT *predicate-3*), the set satisfying *predicate-1* is the result of the AND logical operation between *predicate-2* and NOT *predicate-3*.

Therefore, in the case of AND NOT, the result indicated by the asterisk in the above truth table will be ? (indeterminate).

The following roles apply to search condition statements:

1. None of the following items can be specified in a search condition:
   - A subquery (table subqueries without an external reference in an IN predicate without a NOT specification can be specified).
   - An arithmetic operation, date operation, time operation, concatenation operation, scalar function, CASE expression, or CAST specification
   - Comparison of one column against another when written with a comparison operator as follows:

     column-specification comparison-operator column-specification
   - A comparison predicate whose both sides are a literal, USER, CURRENT_DATE, CURRENT_TIME, or CURRENT_TIMESTAMP[(p)]
   - A BETWEEN predicate where *value-expression-1* is a column specification and either *value-expression-2* or *value-expression-3* is a column specification
   - A component specification

- A function call (functions provided by a plug-in in which index-using logic is implemented can be specified)
- Logical predicates `IS FALSE` and `IS UNKNOWN`
- `XMLQUERY`, `XMLSERIALIZE`, `XMLPARSE`, and the `XMLEXISTS` predicate in which no XML query context items are specified in the column specification.

2. A table name or a correlation name cannot be specified in column-name in a search condition.
3. If a repetition column is specified in a search condition, the subscript ANY must also be specified.
4. An index (except for an index having an exception key for a column that is specified to the `IS NULL` predicate) must be defined for all columns that are specified in a search condition. For a structured repetition predicate, a multicolumn index that contains all repetition columns in its constituent columns, specified in a search condition in the structured repetition predicate, must be defined.
5. When evaluating the predicate for the non-repetition column by using an index that contains both a repetition column and a non-repetition column as constituent columns, the index can be used if a condition is specified for one of the repetition columns.
6. Either of the following indexes must be defined on a column in an `IN` predicate with a table subquery specification. Notice that case (b) is limited to the situation in which the SQL optimization mode is optimizing mode 2 based on cost:

   (a) Single-column index

   (b) Column in the `IN` predicate with a table subquery specification that is not the first column in a multicolumn index

   Even if the column in the `IN` predicate with a table subquery specification is not the first column in a multicolumn index, a column need not be defined if one of the following is specified in an index constituent column preceding this column: a comparison predicate (=), the `NULL` predicate (`IS NULL`), or an `IN` predicate (`IN`) for which the right-hand side is a value specification. This rule, however, excludes indexes that contain a repetition column as an index constituent column. For the `IN` predicate for which a value is specified on the right side, the number of value specifications must satisfy one of the following conditions:

   - If `IN` is in only one column, the number of value specifications is 5 or less
   - If `IN` is in two or more columns, the product of numbers of value specifications specified in the columns is 5 or less.

7. If no search conditions are specified, an index (other than a plug-in index or an index that has an exception key) must be defined for one of the columns (other

than a repetition column) for each of the tables from which the list is derived.

8. A predicate in which a repetition column is specified cannot be negated by the NOT logical operator.
9. A predicate that includes the ANDNOT logical operator cannot be negated by the NOT logical operator.
10. Logical predicates cannot be negated by NOT.
11. The IN predicate for which a table subquery is specified cannot be negated by the NOT Boolean operator.
12. If the XMLEXISTS predicate is specified in the search condition, the XQuery query must satisfy the usage conditions of the index. For details about the usage conditions for indexes, see the *HiRDB Version 9 Installation and Design Guide*.

- [WITHOUT LOCK [{WAIT | NOWAIT}]]

Omission of this operand specifies that once data has been retrieved, it must be guaranteed until completion of the transaction.

WITHOUT LOCK [WAIT]

Specifies that once data has been retrieved, it does not have to be guaranteed until completion of the transaction. The WITHOUT LOCK [WAIT] option causes HiRDB to release the lock without waiting for completion of the transaction, thus improving the system's capacity for concurrent execution.

WITHOUT LOCK NOWAIT

Specifies immediate lock release when data being updated by another user must be referenced immediately, or the integrity of data after it has been retrieved once need not be maintained before the transaction has been completed.

When the WITHOUT LOCK NOWAIT option is specified, HiRDB does not perform locking. Because a table that is being updated can be retrieved without waiting for locking, the system's capacity for concurrent execution can be improved. However, retrieving data while it is being updated may produce incorrect results.

- [WITH ROLLBACK | NOWAIT]

Omission of this operand specifies that if the table to be retrieved is being used by another user, this operation is to be placed on hold until the other transaction is completed, and then this operation is to be executed (except when the WITHOUT LOCK NOWAIT option is specified).

WITH ROLLBACK

Specifies that this transaction is to be canceled and invalidated if the table to be retrieved is being used by another user.

NO WAIT

Specifies that if the table to be retrieved is being used by another user, this transaction is to be flagged as an error but is not to be canceled. However, locking applied during execution of this SQL statement is not released.

Common rules

1. A list can be used only by its owner; lists are not subject to privilege definition or revocation. If the executing user is not the owner of the list, the `SET SESSION AUTHORIZATION` statement can be used to change the executing user.
2. Lists are not deleted when exported from HiRDB.
3. All lists are deleted automatically when HiRDB stops.
4. The disposition of lists when an error occurs is indicated below (an attempt to retrieve such a list results in an error); lists that cannot be retrieved must be either deleted or re-created:
   - In the event of abnormal termination of all units:

     All lists are deleted automatically.
   - In the event of abnormal termination of some but not all units:

     Lists created at the abnormally terminated units cannot be retrieved.
5. While a list is in existence, none of the following operations can be performed on the base table from which the list was derived:
   - Schema deletion
   - Table deletion
   - Table definition modification
6. When a list is created, the number of rows created is returned to `SQLERRD[2]` of `SQLCA`.
7. Lists cannot be specified in an SQL routine.
8. Although the `ASSIGN LIST` statement can be executed dynamically, it cannot be executed by embedding it directly in a host program.
9. The same user cannot manipulate a list by connecting to HiRDB concurrently in multiple sessions.
10. The maximum number of lists that can be stored in a list RDAREA is determined by the maximum number of lists registered (500 to 50,000), which is specified by the database initialization utility (`pdinit`) or the database structure modification utility (`pdmod`).

Notes

1. Lists are created in list RDAREAs at the same server where the base table is

located. Before a list can be created, a list RDAREA must have been created.

2. A list that is exported from HiRDB is not deleted. Lists that are no longer needed should be deleted with the `DROP LIST` statement.
3. If any of the following operations is performed on the base table from which a list was derived while the list is still in existence, the list must be re-created:
   - `PURGE TABLE` statement
   - Reorganization of the table
   - Data loading in the load mode
   - RDAREA reinitialization

Example

See the examples provided in the section on the *ASSIGN LIST statement Format 2 (Create list)*.

ASSIGN LIST statement Format 2 (Create list)

Function

The Format 2 ASSIGN LIST statement copies a list or performs a set operation on lists in order to create a list from the results. The Format 2 ASSIGN LIST statement can also rename a list.

Privileges

Owner of the list

The owner of a list can create lists from that list.

Format 2: Creating a list from a list; renaming a list

```
ASSIGN LIST list-name FROM list-name-1
     [{{AND|OR|AND NOT|ANDNOT} list-name-2|FOR ALTER LIST}]
```

Operands

- *list-name*

Specifies a name for the list that is to be created.

If an existing list name is specified, the existing list is deleted and a new list is created.

- *list-name-1* {{AND | OR | AND NOT | ANDNOT} *list-name-2* | FOR ALTER LIST}]

The lists that are specified in *list-name-1* and *list-name-2* must be derived from the same base table.

The list names specified in *list-name*, *list-name-1*, and *list-name-2* must be unique.

list-name-1

Specifies a list that is to be copied.

list-name-1 AND *list-name-2*

Specifies lists from which a set product is to be derived.

list-name-1 OR *list-name-2*

Specifies lists from which a set union is to be derived.

list-name-1 {AND NOT | ANDNOT} *list-name-2*

Specifies lists from which the set difference is to be derived.

list-name-1 FOR ALTER LIST

Specifies a new name for the list.

Common rules

See the rules for Format 1.

Notes

See the notes for Format 1.

Examples

Table T1

C1	C2	CHECK1	CHECK2
A	1	Y	Y
B	2	Y	Y
C	3	Y	Y
D	4	Y	N
E	5	Y	N
F	6	N	Y
G	7	N	N
H	8	N	N

1. Create from table `T1` a list (`LIST1`) of those rows with the value `Y` in the `CHECK1` column:
```
ASSIGN LIST LIST1 FROM (T1) WHERE CHECK1='Y'
```

List1

C1	C2	CHECK1	CHECK2
A	1	Y	Y
B	2	Y	Y
C	3	Y	Y
D	4	Y	N
E	5	Y	N

2. Create from table `T1` a list (`LIST2`) of those rows with the value `Y` in the `CHECK2` column:
```
ASSIGN LIST LIST2 FROM (T1) WHERE CHECK2='Y'
```

List2

C1	C2	CHECK1	CHECK2
A	1	Y	Y
B	2	Y	Y
C	3	Y	Y
F	6	N	Y

3. Create a list (`LIST3`) from the set product of `LIST1` and `LIST2`:
```
ASSIGN LIST LIST3 FROM LIST1 AND LIST2
```

List3

C1	C2	CHECK1	CHECK2
A	1	Y	Y
B	2	Y	Y
C	3	Y	Y

CALL statement (Call procedure)

Function

The CALL statement calls a procedure.

Privileges

Users with the DBA or CONNECT privilege

These users can call procedures. If SQL statements are to be executed in a procedure, the user needs to have the privilege to execute all the SQL.

Format

```
CALL [authorization-identifier.]routine-identifier ([argument
          [, argument]...])
argument ::= {{IN|OUT|INOUT} :embedded-variable[:indicator-variable]
               |[{IN|OUT|INOUT}] {SQL-variable|SQL-parameter
                                          |?-parameter}
               |[IN] value-expression}
```

Operands

- [*authorization-identifier*.] *routine-identifier*

authorization-identifier

Specifies the authorization identifier of the owner of the procedure being called.

If a public procedure is called, specify upper-case PUBLIC enclosed in double quotation marks (") as the authorization identifier.

routine-identifier

Specifies the routine name of the procedure being called.

- *argument* ::= {{IN|OUT|INOUT} :*embedded-variable*[:*indicator-variable*]

  |[{{IN|OUT|INOUT}} *SQL*-variable|SQL-*parameter*

  |{{IN|OUT|INOUT}] *?-parameter*

  |[IN] *value-expression*}

Specifies arguments for a parameter of the procedure to be called. IN, OUT, or INOUT specifies the I/O mode (parameter mode) for the parameter for a procedure specified in a CREATE PROCEDURE statement.

The following table lists the specification rules for IN, OUT, and INOUT.

Table 4-2: Specification rules for IN, OUT, and INOUT

Argument specification	Type of CALL statement		
	Dynamic execution	UAP embedding	In routine definition
:*embedded-variable*[: *indicator-variable*]	N	Y#2	N
SQL-variable, *SQL-parameter*	N	N	Y#1
Component specification based on *SQL-variable*, *SQL-parameter*	N	N	Y#1
?-parameter	Y#1	N	N
Value expression other than above	Y#3	Y#3	Y#3

Y: Can be specified.

N: Cannot be specified.

#1: `IN`, `OUT`, or `INOUT` may be specified; or the specification may be omitted.

#2: `IN`, `OUT`, or `INOUT` must be specified.

#3: `IN` can be specified or may be omitted.

Common rules

1. Arguments are associated with parameters by the order in which they are specified.
2. The data type of an argument must be compatible with the data type of the parameter with which the argument is associated.
3. If the parameter mode of the associated parameter is `OUT` or `INOUT` and the `NULL` value is output as a parameter value, the receiving argument must have an indicator variable.
4. `?` parameters that are specified as arguments for parameters with an `IN`, `OUT`, or `INOUT` parameter mode become, respectively, the input `?` parameter, the output `?` parameter, or the input `?` parameter and output `?` parameter.

   The `?` parameter in the value expression will be the input `?` parameter.
5. The `BOOLEAN` data type cannot be specified for an input or output parameter.
6. An embedded variable (indicator variable) and the `?` parameter must be a simple structure.
7. The following cannot be specified as an argument when the parameter mode is `IN`:

- A value expression containing a set function
- A value expression containing the window function
- A value expression containing a column specification
- A value expression containing a component specification that exhibits the attribute of an abstract data type column

8. A component specification that exhibits the attribute of an abstract data type column cannot be specified as an argument when the parameter mode is `OUT` or `INOUT`.
9. In *argument*, a `SUBSTR` scalar function with a result data type of `BLOB` or `BINARY` with a maximum length of 32,001 bytes or greater cannot be specified as a single value expression.
10. A subquery cannot be specified in a value expression specified as an argument.
11. If a procedure in which the number 1 or greater is specified in the `DYNAMIC RESULT SETS` clause of a procedure definition, the procedure returns one of the groups of result sets listed in the following table. If, however, the number of the result sets listed in the following table is greater than the number specified in the `DYNAMIC RESULT SETS` clause, only result sets up to the number specified in the `DYNAMIC RESULT SETS` clause are returned.

Table 4-3: Result sets returned by a procedure and their order

Procedure type	Returned result sets	Order in which result sets are returned
Java procedure	The result set specified in a `java.sql.ResultSet[]` type parameter for the Java method specified in a foreign routine specification in the procedure definition.	Order of parameter specifications
SQL procedure	Of the result set cursors declared in the procedure, the result set of cursors that are open when the procedure terminates.	Order in which cursors are opened

12. If the called procedure returns a result set, one of the following return codes is assigned:
    - A return code `120` to the `SQLCODE` area of the SQL communications area
    - A return code `120` to the `SQLCODE` variable
    - A return code `0100C` to the `SQLSTATE` variable

However, if the number of result set cursors that are open at the time of procedure termination is greater than the number specified in the `DYNAMIC RESULT SETS`

clause, a code 0100E is assigned to the SQLSTATE variable.

Notes

1. If index information in an SQL object in the procedure is invalidated by addition or deletion of an index, the procedure cannot be executed. In such a case, the SQL object in the procedure must be recreated.
2. Procedures using PURGE TABLE, COMMIT, and ROLLBACK statements cannot be used in the following environment:
   - Calling a procedure from a UAP running under OLTP
3. For details about default values when the authorization identifier is omitted, see *1.1.9 Schema path*.

CLOSE statement (Close cursor)

Function

The CLOSE statement closes a cursor and terminates fetching of retrieval results by the FETCH statement.

Privileges

None.

Format

CLOSE {*cursor-name* | *extended-cursor-name*}

Operands

- {*cursor-name* | *extended-cursor-name*}

cursor-name

Specifies the cursor that is to be closed, which must be a cursor opened by the OPEN statement.

extended-cursor-name

Specifies the name of an extended cursor that identifies the cursor to be closed. The cursor to be closed is one that is opened by the OPEN statement or one that is allocated to a result set returned by a procedure and one that references the result set.

For extended cursor names, see *2.27 Extended cursor name*.

Common rules

1. Executing any of the SQL statements listed below closes all cursors that are open at that time. All cursors are also closed when an error with implicit rollback occurs.

   • Definition SQL statements (when YES is specified for PDCMMTBFDDL in the client environment definition)

   • PURGE TABLE statement

   • COMMIT statement

   • DISCONNECT statement

   • ROLLBACK statement

   • PREPARE statement (when YES is specified for PDPRPCRCLS in the client environment definition)

- Internal DISCONNECT (termination of a UAP without executing a DISCONNECT statement)

Note that holdable cursors are not closed when a COMMIT statement is executed. When a PURGE TABLE statement is executed and a table opened using a holdable cursor is set to check pending status, the holdable cursor is closed.

2. If the CLOSE statement is executed on the cursor allocated to the group of result sets returned by a procedure by the ALLOCATE CURSOR statement Format 2, and if there is a result set next to the result set currently referenced, the currently referenced result set is closed. In that case, the cursor references another result set, and the following return codes are assigned:

   - A return code 121 to the SQLCODE area of the SQL communications area
   - A return code 121 to the SQLCODE variable
   - A return code 0100D to the SQLSTATE variable

   In this case, the cursor remains open.

   If another result set does not exist, the currently referenced result set is closed, and the following return codes are assigned:

   - A return code 100 to the SQLCODE area of the SQL communications area
   - A return code 100 to the SQLCODE variable
   - A return code 02001 to the SQLSTATE variable

   In this case, the extended cursor name ceases to identify any cursor.

   For details about the series of operations to be performed when a cursor is allocated to the group of result sets returned from a procedure, see *1.9.3 Results-set return facility*.

Note

A cursor name, similar to an embedded variable name, is effective within a compile-unit module. Therefore, multiple SQLs related to the same cursor cannot be used in multiple modules.

Example

Close cursor CR1:

```
CLOSE CR1
```

DEALLOCATE PREPARE statement (Nullify the preprocessing of SQL)

Function

Nullifies the SQL statement preprocessed by the PREPARE statement, and releases the SQL statement identifier or the extended statement name from its allocated state.

Privileges

None.

Format

```
DEALLOCATE PREPARE {SQL-statement-identifier|extended-statement-name}
```

Operands

- {*SQL-statement-identifier* | *extended-statement-name*}

SQL-statement-identifier

Specifies the name that was assigned to identify the SQL statement preprocessed by the *SQL-statement-identifier* PREPARE statement.

For SQL statement identifiers, see *1.1.7 Specification of names*.

HiRDB reserved words can also be used. If a HiRDB reserved word is used, the SQL statement identifier, even if it is identical to the reserved word, should not be enclosed in double quotation marks ("). However, the words SELECT and WITH cannot be used.

extended-statement-name

Specifies the extended statement name that was assigned to identify the SQL statement preprocessed by the PREPARE statement.

For extended statement names, see *2.26 Extended statement name*.

Common rules

1. If a cursor exists that was either declared by DECLARE CURSOR for the SQL statement identified by a specified SQL statement identifier or an extended statement name, or that was allocated by an ALLOCATE CURSOR statement, and if that cursor is open, the cursor cannot be nullified.
2. All cursors that are declared for or allocated to a specified SQL statement or extended statement name and that are closed are also nullified. In addition, all

preprocessed SQL statements that reference those cursors are also nullified.

Notes

1. If the SQL statement for the nullified SQL statement identifier is a holdable cursor, the SQL statement identifier is not enabled, even if it is rolled back.
2. The SQL statement identifier, similar to embedded variable names, is a name that is in effect within compile-by-compile modules; more than one SQL statement with respect to a given SQL statement identifier cannot be used across multiple modules.

Examples

Deallocate the results of preprocessing of the SQL statement identified by the SQL statement identifier (`PRESQL`) specified in a `PREPARE` statement.

```
DEALLOCATE PREPARE PRESQL
```

DECLARE CURSOR Format 1 (Declare cursor)

Function

DECLARE CURSOR declares a cursor to be used by the FETCH statement to fetch on a row-by-row basis the results of a retrieval by a query specification.

In Format 1, this SQL statement declares a cursor for a direct cursor specification.

Privileges

None.

Format 1: Declaring a cursor relative to a direct cursor specification

```
DECLARE cursor-name CURSOR [WITH HOLD][ {WITH RETURN | WITHOUT
RETURN} ] FOR
(Cursor-Specification-Format-1)
   (Query-Expression)
      (Query-Specification)
      {SELECT [{ALL|DISTINCT}]{selection-expression
                                          [,selection-expression]...|*}
      (Table-Expression)
      FROM table-reference [, table-reference]...
             [WHERE search-condition]
             [GROUP BY value-expression [, value-expression]...]
             [HAVING search-condition]
             | query-expression }
      [ORDER BY {column-specification|sort-item-specification-number}
             [{ASC|DESC}]
           [, {column-specification|sort-item-specification-number}
                [{ASC|DESC}]...]]
      [LIMIT { [offset, ] {row_count | ALL}
           | {row_count | ALL} [OFFSET offset] } ]
(Lock-Option)
  [[(WITH {SHARE|EXCLUSIVE} LOCK
     |WITHOUT LOCK [{WAIT|NOWAIT}]}]
  [{WITH ROLLBACK|NO WAIT}]]
    [FOR {UPDATE [OF column-name [, column-name]...] [NOWAIT]|READ
ONLY}]
    [UNTIL DISCONNECT]
```

Operands

- *cursor-name*

Specifies the name of the desired cursor.

When a cursor name is specified in a UAP, it must not be enclosed in double quotation marks, even when the cursor name is the same as an SQL reserved word. However, when a cursor name that is the same as an SQL reserved name is specified in a procedure, it must be enclosed in double quotation marks.

For details about cursor names, see *1.1.7 Specification of names*.

- [WITH HOLD]

Specifies that a holdable cursor is to be used.

Because WITH HOLD provides the same function as specifying UNTIL DISCONNECT, see the section on *UNTIL DISCONNECT* for an explanation. The result is the same whether WITH HOLD or UNTIL DISCONNECT is specified.

- [{WITH RETURN | WITHOUT RETURN}]
- In a cursor declaration in an SQL statement, these operands specify the returnability of the result set for the cursor.
- A cursor declared by specifying WITH RETURN is referred to as a result set cursor.
- If the procedure terminates when the result set cursor declared in an SQL procedure is still open, the result set for the cursor cannot be returned to the calling source for the procedure.
- A value greater than or equal to 1 should be specified in the DYNAMIC RESULT SETS clause of the definition of the SQL procedure that declares a result set cursor.
- For details about how to use a returned result set, see *1.9.3 Results-set return facility*.
- *Cursor-Specification-Format-1*

Specifies the cursor that expresses the contents of a query.

See *2.1.1 Cursor specification: Format 1* for cursor specifications, *2.2 Query expressions* for query expressions, *2.3 Query specification* for query specifications, *2.5 Table expressions* for table expressions, and *2.7 Search conditions* for search conditions.

- *Lock-Option*

Specifies the lock mode for specifying queries, and the action to be taken by the system when the necessary resources for performing a query are being used exclusively by another user.

See *2.19 Lock option* for details about the lock option.

- [FOR {UPDATE [OF *column-name* [, *column-name*]...] [NOWAIT]|READ ONLY}]

The FOR UPDATE [OF *column-name* [, *column-name*]] clause is called the FOR

UPDATE clause.

FOR UPDATE

In the case of a table that is being searched using the cursor, specifies that a row on which the cursor is used can only be updated or deleted, and a row on which the cursor is not used can be updated, deleted, or added.

If a module contains the UPDATE statement for updating rows using the specified cursor or contains the DELETE statement for deleting rows and the FOR UPDATE OF clause is omitted, then FOR UPDATE is assumed as the default, thus enabling updating, adding, or deleting of any column.

This operand should be omitted for a table being retrieved using the cursor that has no rows to be updated or deleted using that cursor or any other cursors, and no rows to be updated, deleted, or added without using a cursor. If a lock option in the SQL statement is omitted, the lock option is determined by a specified value in PDISLLVL or a specified value for the data guarantee level specified in *SQL-compile-option*. However, if YES is indicated for PDFORUPDATEEXLOCK or if FOR UPDATE EXCLUSIVE is specified after the data guarantee level in *SQL-compile-option*, the lock option for the cursor with that specification is assumed to be WITH EXCLUSIVE LOCK. For details, see the *HiRDB Version 9 UAP Development Guide*.

OF *column-name* [, *column-name*]

Specifies the columns to be updated in a table being searched using the cursor when only the rows retrieved with that cursor are to be updated.

This operand can also specify columns that are not specified in a selection expression of the SELECT statement. A column can be specified only once in a SELECT statement.

If a table being searched using the cursor has no rows to be updated or deleted using that cursor or any other cursors and no rows to be updated, deleted, or added without using a cursor, this operand should be omitted.

When a column name is specified, instead of using the column name that was specified in AS *column-name*, specify the column of the table that was specified in the FROM clause for the outermost query specification.

[NOWAIT]

Produces the same behavior as if the FOR UPDATE clause was specified and WITH EXCLUSIVE LOCK NO WAIT was specified as the lock option. However, if the lock option is specified, NOWAIT cannot be specified. For details about operations when WITH EXCLUSIVE LOCK NO WAIT is specified as the lock option, see *2.19 Lock option*.

FOR READ ONLY

In the case of a table being searched using the cursor, specifies that the rows are to be updated either using another cursor or by specifying a direct search condition. The purpose of this specification is to ensure that the update operation performed during searching is not affected by the retrieval results.

- [UNTIL DISCONNECT]

Specifies that a holdable cursor is to be used.

The function provided by this specification is exactly the same as specifying WITH HOLD. The result is the same whether WITH HOLD or UNTIL DISCONNECT is specified.

For details about holdable cursors, see the *HiRDB Version 9 UAP Development Guide*.

The following rules apply to holdable cursors:

1. A holdable cursor cannot be used in the following cases:
   - When a column of the abstract data type using a plug-in is specified
   - When a function call using a plug-in is specified
   - A query with respect to a named derived table that was derived by specifying a function call using a plug-in
2. Definition SQL statements cannot be executed while a holdable cursor is open.
3. If, after an OPEN statement is executed for a SELECT statement using a holdable cursor, a PURGE TABLE statement is executed for a table used in the SELECT statement, the cursor is placed into closed status.
4. If, after an OPEN statement is executed for a SELECT statement using a holdable cursor and before a DISCONNECT is performed, another user issues a definition SQL statement for a table used in the SELECT statement, the definition SQL statement is placed into lock-wait status. Similarly, if, during the period when preprocessing relative to a SELECT statement using a holdable cursor is still in effect, another user issues a definition SQL statement for a table that is being used in the SELECT statement, the definition SQL statement is placed into lock-wait status.

Common rules

1. The declared cursor remains closed.
2. The value (the value of an embedded variable, an SQL variable, or an SQL parameter value that is specified in the SELECT statement of DECLARE CURSOR) that is in effect when the OPEN statement for the cursor is executed remains in effect from the time the cursor is opened until it is closed. To modify such values, the cursor must be closed and then reopened.
3. A maximum of 1,023 cursors can be declared per UAP.
4. If either the cursor specification or the cursor lock option contains any of the

following specifications, updating and deletion using that cursor cannot be performed and the FOR UPDATE clause cannot be specified:

1. UNION [ALL] or EXCEPT [ALL]

2. A table specified in the FROM clause of the outermost query specification in the FROM clause of a subquery.

3. Joined tables in the outermost query specification.

4. A derived table in a FROM clause in the outermost query specification

5. SELECT DISTINCT in the outermost query specification

6. A GROUP BY clause in an outermost query specification

7. A HAVING clause in an outermost query specification

8. A set function on the outermost query specification

9. The window function on the outermost query specification

10. Any of the following view tables in the FROM clause of the outermost query specification:

- A view table defined by specifying (1) to (9) above in CREATE VIEW
- A view table defined by specifying a value expression other than a column specification in the SELECT clause of the outermost query specification in a view definition statement
- A view table for which READ ONLY is specified in CREATE VIEW

11. WITHOUT LOCK NOWAIT

12. A query specification name specified in the FROM clause of the outermost query specification in the query expression body in which a WITH clause is specified

5. The FOR READ ONLY clause cannot be specified for a cursor if rows are to be updated or deleted using that cursor.

6. When a FOR READ ONLY clause is specified, the following restrictions apply:

   1. Scalar operations, function calls, and component specifications that produce results in any of the following data types cannot be specified in a selection expression:

      - BLOB
      - BINARY with a maximum length of 32,001 bytes or greater
      - BOOLEAN
      - Abstract data type

2. Only a column specification can be specified for an output BLOB value with a WRITE specification in a selection expression.

3. A GET_JAVA_STORED_ROUTINE_SOURCE specification cannot be specified.

Rule related to referential constraints

1. A holdable cursor that is used to retrieve a table in which a foreign key is defined is closed when the table being retrieved goes into check pending status.

Notes

1. Similar to an embedded variable, a cursor name is effective within a compile-unit module. Multiple SQLs relative to the same cursor cannot be used in different modules.
2. A cursor declaration must be coded before any SQL statement that references the cursor name used in the declaration.
3. Because DECLARE CURSOR is not an executable statement, no return code is returned to SQLCODE (and return code testing should not be performed).
4. By applying the work table creation suppression feature of the update SQL statement in the SQL optimization option and using the index key-value no-lock facility, you can update, add, or delete rows while using a cursor for which neither FOR UPDATE nor FOR UPDATE OF is specified.
5. Specifying FOR READ ONLY may cause HiRDB to create a work table. In this case, the FOR READ ONLY processing may be subject to restrictions depending on the row length of the work table. For details about work table row lengths, see the *HiRDB Version 9 Installation and Design Guide.*

Examples

1. Declare cursor CR1 to fetch rows, one row at a time, from stock table STOCK:
```
DECLARE CR1 CURSOR FOR
  SELECT PCODE,PNAME,COLOR,PRICE,SQTY
    FROM STOCK
```

2. Declare cursor CR1 for stock table STOCK to fetch rows, one row at a time, in which the unit price (PRICE) is $50 or greater:
```
DECLARE CR1 CURSOR FOR
  SELECT * FROM STOCK
    WHERE PRICE  = 50
```

3. Use cursor CR1 to retrieve all rows from stock table STOCK, apply a 10% discount to the unit price (PRICE), and then insert rows:
```
DECLARE CR1 CURSOR FOR
  SELECT * FROM STOCK
    FOR UPDATE
OPEN CR1
```

```
FETCH CR1 INTO <Name of variable into which column is fetched>
UPDATE STOCK
  SET PRICE = <Value of unit price after 10% discount>
    WHERE CURRENT OF CR1
INSERT INTO STOCK VALUES (<Insertion values for columns>)
CLOSE CR1
```

4. Use cursor `CR1` to retrieve all rows from stock table `STOCK`, and apply a 10% discount to the unit price (`PRICE`):

```
DECLARE CR1 CURSOR FOR
  SELECT * FROM STOCK
    FOR UPDATE OF PRICE
OPEN CR1
FETCH CR1 INTO <Name of variable into which column is fetched>
UPDATE STOCK
  SET PRICE = <Value of unit price after 10% discount>
    WHERE CURRENT OF CR1
CLOSE CR1
```

5. Use cursor `CR1` to retrieve all rows from stock table `STOCK`, and delete, without using the cursor, rows whose product name (`PNAME`) is sweater:

```
DECLARE CR1 CURSOR FOR
  SELECT * FROM STOCK
    FOR READ ONLY
OPEN CR1
FETCH CR1 INTO <Name of variable into which column is fetched>
DELETE FROM STOCK
    WHERE PNAME=N'sweater'
CLOSE CR1
```

DECLARE CURSOR Format 2 (Declare cursor)

Function

DECLARE CURSOR declares a cursor to be used by the FETCH statement to fetch on a row-by-row basis the results of a retrieval by a query specification.

In Format 2, this SQL statement declares a cursor for a SELECT statement preprocessed by a PREPARE statement (dynamic SELECT statement).

Privileges

None.

Format 2: Declaring a cursor relative to a SELECT statement (dynamic SELECT statement) preprocessed by the PREPARE statement

```
DECLARE cursor-name CURSOR
   [WITH HOLD][ {WITH RETURN | WITHOUT RETURN} ] FOR
SQL-statement-identifier
```

Operands

- *cursor-name*

Specifies the name of the desired cursor.

When a cursor name is specified in a UAP, it must not be enclosed in double quotation marks, even when the cursor name is the same as an SQL reserved word. However, when a cursor name that is the same as an SQL reserved name is specified in a procedure, it must be enclosed in double quotation marks.

See *1.1.7 Specification of names* for details about cursor names.

- [WITH HOLD]

Specifies that a holdable cursor is to be used. A holdable cursor cannot be used in the following cases:

- When a column of the abstract data type using a plug-in is specified
- When a function call using a plug-in is specified
- A query with respect to a named derived table that was derived by specifying a function call using a plug-in
- A search using a list

For details about holdable cursors, see the *HiRDB Version 9 UAP Development Guide*.

- [{WITH RETURN | WITHOUT RETURN}]

In a cursor declaration in an SQL statement, these operands specify the returnability of the result set for the cursor. A cursor declared by specifying `WITH RETURN` is referred to as a result set cursor. If the procedure terminates when the result set cursor declared in an SQL procedure is still open, the result set for the cursor cannot be returned to the calling source for the procedure. For details about how to use a returned result set, see *1.9.3 Results-set return facility*.

- *SQL-statement-identifier*

Specifies the SQL statement identifier assigned to the `SELECT` statement that was preprocessed by the `PREPARE` statement.

Common rules

1. The declared cursor remains closed.
2. A declared cursor cannot be used in the `UPDATE` or `DELETE` statement.
3. When multiple cursor declarations are made for the same SQL statement identifier, cursor declarations with and without the `WITH HOLD` specification cannot be mixed.

Rule related to referential constraints

1. A holdable cursor that is used to retrieve a table in which a foreign key is defined is closed when the table being retrieved goes into check pending status.

Notes

1. Similar to an embedded variable, a cursor name is effective within a compile-unit module. Multiple SQLs relative to the same cursor cannot be used in different modules.
2. The corresponding `PREPARE` statement must be coded before the cursor declaration.
3. A cursor declaration must be coded before any SQL statement that references the cursor name used in the declaration.
4. Because `DECLARE CURSOR` is not an executable statement, no return code is returned to `SQLCODE` (and return code testing should not be performed).
5. By applying the work table creation suppression feature of the update SQL statement in the SQL optimization option and using the index key-value no-lock facility, you can update, add, or delete rows while using a cursor for which neither `FOR UPDATE` nor `FOR UPDATE OF` is specified.

Example

Declare cursor `CR1` to fetch rows, one row at a time, that have been specified by a prepared `SELECT` statement (SQL statement identifier is `SEL`):

```
DECLARE CR1 CURSOR FOR SEL
```

DELETE statement Format 1 (Delete rows)

Function

The DELETE instruction deletes from a table the rows that satisfy a specified search condition or the row indicated by a cursor.

Privileges

A user who has the DELETE privilege for a table can delete rows from that table.

However, if a subquery is specified in the search condition, the user needs the SELECT privilege for the table for which the subquery is specified.

Format

```
DELETE FROM [authorization-identifier.]table-identifier
            [IN (RDAREA-name-specification)][ [AS] correlation-name]
            [SQL-optimization-specification-for-used-index]
            [WHERE {search-condition|CURRENT OF {cursor-name |
extended-cursor-name }}]
            [WITH ROLLBACK]
```

Operands

- [*authorization-identifier*.]*table-identifier*

authorization-identifier

Specifies the authorization identifier of the user who owns the table.

MASTER cannot be specified as an authorization identifier. For the case in which the authorization identifier is omitted, see *1.1.8 Qualifying a name*.

table-identifier

Specifies a table containing the row to be deleted.

The following rules apply to table identifiers:

1. Row insertion, updating, or deletion cannot be performed on a read-only view table.
2. For details about read-only view tables, see the common rules under *CREATE [PUBLIC] VIEW (Define view, define public view)* in Chapter 3.
3. If the deletion of a row from a view table is specified, HiRDB deletes the row from the base table which is subject to the view table operation.
4. The scope of *table-name* is the entire DELETE statement.

- [IN (*RDAREA-name-specification*)]

IN

Specifies the RDAREA to access.

RDAREA-name-specification ::= *value-specification*

Of the RDAREAs that contain the table specified in the table identifier, specify the name of the RDAREA to access as a *value-specification* of type VARCHAR, CHAR, MVARCHAR, or MCHAR. If multiple RDAREA names are specified, separate them with a comma (,). RDAREA names must be unique; an error occurs if duplicate RDAREA names are specified. For details about what characters are allowed in RDAREA names in *value-specification*, see *1.1.7 Specification of names*. Note also that leading and trailing whitespace is ignored in RDAREA names specified in *value-specification*. If the RDAREA name is enclosed in double quotation marks ("), only whitespace outside the double quotation marks is ignored.

If *cursor-name* or *extended-cursor-name* is specified, specify the same set of RDAREAs (in any order) as the RDAREAs specified in the cursor declared in the cursor declaration. An error occurs if they are not specified.

If specifying an RDAREA that uses the inner replica facility, specify the original RDAREA name. To target the replica RDAREA, use the change current database command (pddbchg command), or the PDDBACCS operand in the client environment definition, to switch the RDAREA to access to the replica RDAREA.

- [AS]*correlation-name*

Specify this operand when using a correlation name for the table to be deleted.

The scope of the correlation name is the entire DELETE statement. The table identifier to be deleted does not have a scope.

- *SQL-optimization-specification-for-used-index*

For details about the SQL optimization specification for a used index, see *2.24 SQL optimization specification*.

- WHERE {*search-condition*|CURRENT OF {*cursor-name* | *extended-cursor-name*} }

WHERE

When the WHERE clause is omitted, all rows in the specified table are deleted and the W warning flag is set in SQLWARN4 in the SQL Communications Area.

search-condition

Specifies criteria for selecting the rows to be deleted.

Embedded variables, ? parameters, SQL variables, and SQL parameters can be specified in a search condition. Only ? parameters can be specified in a search

condition in a DELETE statement that is preprocessed by the PREPARE statement.

SQL variables or SQL parameters are used in an SQL procedure. For details about specification values in a Java procedure, see *JDBC drivers or SQLJ* in the *HiRDB Version 9 UAP Development Guide*. For details about search conditions, see *2.7 Search conditions*.

- *cursor-name*

Specifies the name of the cursor that indicates the row to be deleted.

The cursor name specified in *cursor-name* is a cursor declared in a cursor declaration.

The cursor specified in cursor-name must be positioned at a row opened by the OPEN statement and closed by the FETCH statement. Any OPEN, FETCH, CLOSE, or DELETE statement that is directed at the cursor should be executed in the same transaction (except when a holdable cursor is used).

After the DELETE statement has executed, the cursor specified in cursor-name becomes a cursor that does not point to any row. To update or delete the row following the deleted row, the FETCH statement must be executed on the cursor in order to advance the cursor.

extended-cursor-name

This operand specifies the extended cursor name that identifies the cursor that points to the row to be deleted.

The extended cursor name that identifies the cursor allocated by the ALLOCATE CURSOR statement should be specified. However, a result set cursor cannot be specified.

The cursor identified in *extended-cursor-name* should be open and positioned on the row to be deleted.

The cursor identified in *extended-cursor-name* does not have any row that it can point to, after the execution of the DELETE statement. If any row after the deleted row is to be updated or deleted, the FETCH statement should be executed on the cursor to move it.

If *extended-cursor-name* is specified, an extended cursor must be specified for queries that specify the FOR UPDATE clause. For details about the FOR UPDATE clause, see the FOR UPDATE clause under *Operands* in *Dynamic SELECT statement Format 1 (Retrieve dynamically)* in this chapter.

For extended cursor names, see *2.27 Extended cursor name*.

- WITH ROLLBACK

  Specifies that if the table to be deleted is being used by another user, the transaction issued by that user is to be cancelled and invalidated.

If the WITH ROLLBACK option is omitted and the table to be deleted is being used by another user, the current user must wait until the transaction issued by the other user is completed.

Common rules

1. Normal execution of the DELETE statement sets the number of deleted rows in SQLERRD[2] in the SQL Communications Area.
2. If there are no rows to be deleted, the system returns the following return codes:
   - Return code 100 to SQLCODE in the SQL Communications Area
   - Return code 100 to the SQLCODE variable
   - Return code '02000' to the SQLSTATE variable
3. If the user LOB RDAREA that stores a LOB column or LOB attribute is in the frozen update status, the LOB column or LOB attribute cannot be deleted (an attempt to delete it causes an already frozen error).
4. If the table is a falsification prevented table and the rows that satisfy specified search conditions include a row that is subject to a deletion prohibition duration, HiRDB deletes none of the rows satisfying the search conditions, and generates an error.
5. If the table is a falsification prevented table and the row pointed to by a specified cursor is subject to a deletion prohibition duration, HiRDB generates an error without deleting the row.
6. If the DELETE statement is executed on a table with a WITHOUT ROLLBACK specification, the timing at which the row-locking is released can vary according to whether an index is defined. For details, see the rules on WITHOUT ROLLBACK in *CREATE TABLE (Define table)*.
7. Before deleting a row in a shared table, the LOCK statement for the table should be executed in the lock mode. An attempt to delete a row in a shared table without executing the LOCK statement can cause an error that prevents the row from being deleted. For details about how to update shared tables, see the *HiRDB Version 9 Installation and Design Guide*. For objects of locking in the execution of the LOCK statement, see the notes in *LOCK statement (Lock control on tables)*.
8. If *RDAREA-name-specification* is specified, you cannot use an index in which the number of partitions is different from the number of partitions of the table. When defining an index for queries that specify *RDAREA-name-specification*, define an index that has the same number of partitions as the number of partitions of the table.

Rules on referential constraints

1. For rules on deleting rows in a referenced table or referencing table, see the

reference operation in *CREATE TABLE (Define table)*.

2. For the deletion of rows in a referenced table for which constraint operations are defined in RESTRICT, the referencing table is referenced to determine whether the value of the primary key constituent column in the rows to be deleted is included in the value of a foreign key constituent column in the referencing table. The data guarantee level during the search through the referencing table assumes the share mode. For this reason, if during the deletion of rows in the referenced table for which constraint operations are defined in RESTRICT, operations are performed on the referencing table by another transaction, the row deletion action goes into a wait state until the transaction is settled.
3. If any combination of the following conditions occurs, data incompatibility can occur between the referenced table and the referencing table subject to referential constraints. Such incompatibility can also occur regardless of whether the constraint operation is RESTRICT or CASCADE. For rules on referential constraints, see the *HiRDB Version 9 Installation and Design Guide*.
   - The transaction involving the deletion of rows in the referencing table is different from the transaction that updates or deletes rows in the referenced table.
   - The above two transactions are executed simultaneously.
   - The value of the primary key constituent column for the row in the referencing table to be deleted is equal to the value of the foreign key constituent column of the row in the referenced table to be updated or deleted.
   - The transaction that deletes referencing table rows is committed, and the transaction that updates or deletes referenced table rows is rolled back.

Notes

1. Similar to an embedded variable, a cursor name is effective within a compile-unit module. Multiple SQLs relative to the same cursor cannot be used in different modules.
2. If the table is a falsification prevented table and a deletion prohibition duration is specified, an overflow occurs if the sum of the date of insertion of the row to be deleted and the deletion prohibition duration exceeds December 31, 9999.

Examples

1. Delete from stock table STOCK those rows whose product code (PCODE) column is '302S':

```
DELETE FROM STOCK
       WHERE PCODE = '302S'
```

2. Delete from stock table STOCK those rows whose product code (PCODE) column

is read into embedded variable :XPCODE:

```
DELETE FROM STOCK
       WHERE PCODE = :XPCODE
```

3. Delete the row specified by cursor CR1 from stock table STOCK:

```
DELETE FROM STOCK
       WHERE CURRENT OF CR1
```

DELETE statement Format 2 (Delete row using an array)

Function

Deletes rows that satisfy specified search conditions from a table. Multiple deletion actions can be executed in batch by specifying an embedded variable of an array format.

Privileges

A user who has the DELETE privilege for a table can delete rows from that table.

However, if a subquery is specified in the search condition, the user needs the SELECT privilege for the table for which the subquery is specified.

Format: Deleting several times using an embedded variable array

```
FOR : embedded-variable
DELETE FROM [authorization-identifier.]table-identifier
        [IN (RDAREA-name-specification)][[AS] correlation-name]
        [used-index-SQL-optimization-specification]
        WHERE search-condition
        [WITH ROLLBACK]
```

Operands

See Format 1 for details about the operands and operand rules other than search conditions and FOR, IN (*RDAREA-name-specification*).

- FOR : *embedded-variable*

Specifies the embedded variable in which the number of times deletion operations are performed using an embedded variable array is assigned. An embedded variable of the SMALLINT type should be specified. The allowable range is from 1 to 4,096, no greater than the number of elements in the embedded variable array or in the indicator variable array. Zero and negative values are not allowed. An out-of-range value can produce a run-time error.

embedded-variable-array

This is the embedded variable declared in the array format. Specify an array variable to be used as a search condition using a value other than the NULL value. Values to be used as search conditions should be assigned to the elements of the variable array. If a value to be used as a search condition contains the NULL value, both *embedded-variable-array* and *indicator-variable-array* should be specified.

indicator-variable-array

This is the indicator variable declared in the array format. Values indicating

whether the values of the elements in *embedded-variable-array* are the NULL value should be assigned to the corresponding elements in *indicator-variable-array*. For values that can be assigned, see *1.6.5 Setting a value for an indicator variable*.

- [IN (*RDAREA-name-specification*)]

If *RDAREA-name-specification* is specified as an embedded variable, it must be specified as an embedded variable array. Specifying an embedded variable that is not an array results in an error. See Format 1 for details about other operand rules.

- WHERE *search-condition*

search-condition

Specifies the criteria by which the rows to be deleted are selected. An embedded variable not in the array format cannot be specified in *search-condition*.

Common rules

1. One or more variable arrays should be specified in a clause other than the FOR clause. An error may occur if such an array is not specified.
2. Specifying an embedded variable not in the array format in a clause other than the FOR clause can cause an error.
3. The data type of *embedded-variable-array* should be the data type of the corresponding column or a convertible data type.
4. The number of elements in the embedded variable array or the indicator variable array should be in the range of 1 to 4,096. Specifying an out-of-range value can cause an error. The number of elements should be greater than the maximum value specified in FOR:*embedded-variable*.
5. The elements that are evaluated in one deletion operation in a given embedded variable array and indicator variable array are the elements having the same element number.
6. Because it contains embedded variable arrays and indicator variable arrays, the DELETE statement Format 2 cannot be preprocessed by a PREPARE statement. For details about how to execute DELETE statement Format 2 dynamically, see *EXECUTE statement Format 2 (Execute an SQL statement using an array)*.
7. DELETE using an array cannot be used in a procedure.
8. DELETE using an array cannot accept the BLOB type, the BINARY type with a maximum length of 32,001 bytes or greater, or the abstract data type.
9. Upon normal completion of the DELETE statement, the number of rows deleted is assigned to the SQLERRD[2] area of the SQL communications area.
10. If the row to be deleted is not found, the following return codes are assigned:

- A return code of 100 to the SQLCODE area of the SQL communications area
- A return code of 100 to the SQLCODE variable
- A return code of 02000 to the SQLSTATE variable

11. If the given table is a falsification-prevented table, and if the rows satisfying specified search conditions include a row in the deletion prohibited duration, an error occurs, and the command terminates without deleting any of the rows satisfying the search conditions.
12. If an error occurs in any of the rows to be deleted, the DELETE statement is rolled back.
13. If the DELETE statement is executed on a table with a WITHOUT ROLLBACK specification, the timing at which the row-locking is released varies according to whether or not an index is defined. See the rules on WITHOUT ROLLBACK in *CREATE TABLE (Define table)*.
14. Before deleting rows in a shared table, the LOCK statement in the lock mode should be executed on the table. An attempt to delete rows in a shared table without executing the LOCK statement can cause an error and a failure to delete the rows. For details about how to perform updates on a shared table, see the *HiRDB Version 9 Installation and Design Guide*. For objects of locking in the execution of the LOCK statement on a shared table, see the notes in *LOCK statement (Lock control on tables)*.
15. If *RDAREA-name-specification* is specified, you cannot use an index in which the number of partitions is different from the number of partitions of the table. When defining an index for queries that specify *RDAREA-name-specification*, define an index that has the same number of partitions as the number of partitions of the table.

Rules on referential constraints

1. For rules on deleting rows in a referenced table or referencing table, see the reference operation in *CREATE TABLE*.
2. For the deletion of rows in a referenced table for which constraint operations are defined in RESTRICT, the referencing table is referenced to determine whether the value of the primary key constituent column in the rows to be deleted is included in the value of a foreign key constituent column in the referencing table. The data guarantee level during the search through the referencing table assumes the share mode. For this reason, if during the deletion of rows in the referenced table for which constraint operations are defined in RESTRICT operations are performed on the referencing table by another transaction, the row deletion action goes into a wait state until the transaction is settled.
3. If any combination of the following conditions occurs, data incompatibility can occur between the referenced table and the referencing table subject to referential

constraints. Such incompatibility can also occur regardless of whether the constraint operation is RESTRICT or CASCADE. For rules on referential constraints, see the *HiRDB Version 9 Installation and Design Guide*.

- The transaction involving the deletion of rows in the referencing table is different from the transaction that updates or deletes rows in the referenced table.
- The above two transactions are executed simultaneously.
- The value of the primary key constituent column for the row in the referencing table to be deleted is equal to the value of the foreign key constituent column of the row in the referenced table to be updated or deleted.
- The transaction that deletes referencing table rows is committed, and the transaction that updates or deletes referenced table rows is rolled back.

Examples

1. Execute, in batch, several deletion operations on rows of the inventory table (STOCK) by value of the product code (PCODE) that is assigned to an array variable in the C language:

```
XDELETE_NUM = 5;
EXEC SQL FOR :XDELETE_NUM
      DELETE FROM STOCK
WHERE PCODE = :XPCODE:IPCODE;
```

Preparable dynamic DELETE statement: locating (Delete row using a preprocessable cursor)

Function

Deletes the row pointed to by a cursor. This command is used to delete a row by means of an EXECUTE statement after performing preprocessing by a PREPARE statement, or to perform preprocessing and execution at once by means of an EXECUTE IMMEDIATE statement.

Privileges

A user who has the DELETE privilege for a table can delete rows from that table.

Format

```
DELETE [FROM [authorization-identifier.]table-identifier
      [IN (RDAREA-name-specification)][[AS] correlation-name]
      [used-index-SQL-optimization-specification]]
            WHERE CURRENT OF GLOBAL cursor-name
         [WITH ROLLBACK]
```

Operands

For details about operands other than GLOBAL or *cursor-name*, and for rules on operands, see *DELETE statement Format 1*.

- WHERE CURRENT OF GLOBAL *cursor-name*

GLOBAL

Specifies GLOBAL as the scope for *cursor-name*.

cursor-name

Specifies the name of the cursor that points to the row to be deleted.

The cursor specified in *cursor-name* is the cursor identified by the extended cursor name specified in the ALLOCATE CURSOR statement. The value of the extended cursor name specified in the ALLOCATE CURSOR statement should be specified in *cursor-name*. However, a result set cursor cannot be specified in *cursor-name*.

At execution time, the cursor specified in *cursor-name* must be open and must be positioned on the row to be deleted.

The cursor identified in *cursor-name* does not have any row that it can point to, after the execution of the DELETE statement. If any row after the deleted row is to

be updated or deleted, the FETCH statement should be executed on the cursor to move it.

If *extended-cursor-name* is specified, an extended cursor must be specified for queries that specify the FOR UPDATE clause. For details about the FOR UPDATE clause, see the FOR UPDATE clause under *Operands* in *Dynamic SELECT statement Format 1 (Retrieve dynamically)* in this chapter.

Common rules

1. After performing preprocessing using a PREPARE statement, use the EXECUTE statement to execute, or use the EXECUTE IMMEDIATE statement to preprocess and execute at once.
2. When omitting a table identifier, make sure that before preprocessing is performed, the ALLOCATE CURSOR statement is used to allocate the cursor to the dynamic SELECT statement. In this operation, the table that is the object of retrieval specified in the dynamic SELECT statement to which the cursor is allocated is assumed. When specifying a table identifier, it is not necessary that the cursor be allocated to the dynamic SELECT statement before the preprocessing.
3. The common rules on the DELETE statement Format 1 are applicable to the other common rules.

Rules on referential constraints

1. The rules on the DELETE statement Format 1 are applicable.

Notes

1. The notes on the DELETE statement Format 1 are applicable.

Examples

1. Delete the row specified with the cursor (*cr* (value: CR1)) from the inventory table (STOCK):

```
PREPARE :sel FOR 'SELECT * FROM STOCK FOR UPDATE'
<Assign CR1 to the embedded variable cr>
ALLOCATE CURSOR GLOBAL :cr FOR GLOBAL :sel
OPEN GLOBAL :cr
FETCH GLOBAL :cr INTO <Name of the variable into which columns are
fetched>
PREPARE PRE1 FOR
    'DELETE FROM STOCK WHERE CURRENT OF GLOBAL CR1'
EXECUTE PRE1
DEALLOCATE PREPARE GLOBAL :sel
```

DESCRIBE statement Format 1 (Receive retrieval information and I/O information)

Function

Receives into an SQL descriptor area the retrieval item information or output ? parameter information (such as data code and data length) of an SQL statement pre-processed by a PREPARE statement.

For details about the information to be received, see the *HiRDB Version 9 UAP Development Guide*. If the SQL statement pre-processed by the PREPARE statement does not have SQL retrieval item information or output ? parameter information, 0 is set in the SQLD area of the SQL descriptor area.

Privileges

None.

Format 1: Receiving retrieval information or output ? parameter information

```
DESCRIBE [OUTPUT]{ SQL-statement-identifier | extended-statement-name}
INTO
            [:]SQL-descriptor-area-name
                        [[:]Column-Name-Descriptor-Area-name]
            [TYPE[:]Type-Name-Descriptor-Area-name]
            [CHARACTER_SET[:]character-set-descriptor-area-name]
```

Operands

- {*SQL-statement-identifier* | *extended-statement-name*}
- *SQL-statement-identifier*

Specifies the SQL statement identifier that was specified in the PREPARE statement.

- *extended-statement-name*

Specifies the extended statement name that identifies the SQL statement preprocessed by the PREPARE statement.

For extended statement names, see *2.26 Extended statement name*.

- [:]*SQL-descriptor-area-name*[[:]*Column-Name-Descriptor-Area-name*]

SQL-descriptor-area-name

Specifies the name of the SQL descriptor area that is to receive SQL retrieval item information (if the preprocessed SQL is a SELECT statement) or output ? parameter information (if the preprocessed SQL is a CALL statement).

For SQL descriptor areas, see the *HiRDB Version 9 UAP Development Guide*.

Column-Name-Descriptor-Area-name

Specifies the Column Name Descriptor Area that is to receive the names retrieval items or the parameter names of routines.

For Column Name Descriptor Areas, see the *HiRDB Version 9 UAP Development Guide*.

- [TYPE:]*Type-Name-Descriptor-Area-name*]

Type-Name-Descriptor-Area-name

Specifies the name of the Type Name Descriptor Area for receiving a user-defined data type name for a retrieval item.

For Type Name Descriptor Areas, see the *HiRDB Version 9 UAP Development Guide*.

- [CHARACTER_SET[:]*character-set-descriptor-area-name*]

character-set-descriptor-area-name

Specify a character set descriptor area in which the character set name for retrieval item information (if the preprocessed SQL is a SELECT statement), or output ? parameter information (if the preprocessed SQL is a CALL statement) is to be stored.

For details about character set descriptor areas, see the *HiRDB Version 9 UAP Development Guide*.

Common rules

1. Before executing the DESCRIBE statement, the UAP should set the number of SQLVARs (SQLN areas) in the SQL descriptor area.
2. Both SQLDATA and SQLIND are cleared when the DESCRIBE statement executes. Therefore, executing the DESCRIBE statement is executed, appropriate values must be set in SQLDATA and SQLIND.
3. When a Column Name Descriptor Area is specified, the WITH SQLNAME OPTION must be specified in the associated PREPARE statement.
4. Specify a Column Name Descriptor Area name only when the names of retrieval items or the parameter names of routines are to be received. However, the parameter name of a routine can be received only when a ? parameter is specified singly in an argument of the CALL statement. If a value expression containing a ? parameter is specified, the length of the name of the Column Name Descriptor Area is 0.
5. When a Type Name Descriptor Area name is specified, WITH [ALL] TYPE OPTION must be specified in the corresponding PREPARE statement.

6. Specify a Type Name Descriptor Area name only when a user-defined data type name is to be received for a retrieval item.
7. Specify a character set descriptor area only if a character set has to be specified for retrieval item information (if the preprocessed SQL is a SELECT statement), or output ? parameter information (if the preprocessed SQL is a CALL statement) for which the data type is character data.

Notes

1. As an embedded variable name, an SQL statement identifier is effective in a compile-unit file. Multiple SQL statements referencing the same SQL statement identifier cannot be used in multiple modules.
2. If the output ? parameter is a procedure of a user-defined data type, the data type name information will not be set in the output ? parameter of the user-defined data type.
3. Even if a DESCRIBE [OUTPUT] statement is not used, the same information as that obtained by using DESCRIBE [OUTPUT] can be obtained by specifying OUTPUT in a PREPARE statement.

Examples

1. Specify retrieval item information on the SELECT statement (SQL statement identifier: PRESQL) preprocessed by the PREPARE statement, the names of retrieval items, or the parameter names of routines in the SQL descriptor area and in the Column Name Descriptor Area:
```
DESCRIBE PRESQL INTO :SQLDA :SQLCNDA
```

2. Specify retrieval item information and the names of retrieval items for the SELECT statement (extended statement name: pre) preprocessed by the PREPARE statement in the SQL descriptor area and the Column Name Descriptor Area.

```
PREPARE GLOBAL :pre FOR :sel WITH SQLNAME OPTION
DESCRIBE GLOBAL :pre INTO :SQLDA :SQLCNDA
```

DESCRIBE statement Format 2 (Receive retrieval information and I/O information)

Function

Receives into the SQL descriptor area the input ? parameter information of an SQL statement pre-processed by a PREPARE statement (data code, data length, etc.).

For details about the information to be received, see the *HiRDB Version 9 UAP Development Guide*. If the SQL statement pre-processed by the PREPARE statement does not have input ? parameter information, 0 is set in the SQLD area of the SQL descriptor area.

Privileges

None.

Format 2: Receiving input ? parameter information

```
DESCRIBE INPUT { SQL-statement-identifier | extended-statement-name} INTO
  [:]SQL-descriptor-area-name [[:]Column-Name-Descriptor-Area-name]
  [CHARACTER_SET[:]character-set-descriptor-area-name]
```

Operands

- {*SQL-statement-identifier* | *extended-statement-name*}

SQL-statement-identifier

Specifies the SQL statement identifier specified in the PREPARE statement.

extended-statement-name

Specifies the extended statement name that identifies the SQL statement preprocessed by the PREPARE statement.

For extended statement names, see *2.26 Extended statement name*.

- [:]*SQL-descriptor-area-name*[[:]*Column-Name-Descriptor-Area-name*]

SQL-descriptor-area-name

Specifies the name of the SQL descriptor area into which input ? parameter information is to be set.

For SQL descriptor areas, see the *HiRDB Version 9 UAP Development Guide*.

Column-Name-Descriptor-Area-name

Specifies the Column Name Descriptor Area that is to receive the names of retrieval items or the parameter names of routines.

For Column Name Descriptor Areas, see the *HiRDB Version 9 UAP Development Guide*.

- [CHARACTER_SET[:]*character-set-descriptor-area-name*]

character-set-descriptor-area-name

Specifies a character set descriptor area into which a character set name for input ? parameter information is to be stored.

For details about character set descriptor areas, see the *HiRDB Version 9 UAP Development Guide*.

Common rules

1. Before executing the DESCRIBE statement, the UAP must set the number of SQLVAR areas in SQLN in the SQL descriptor area.
2. Because SQLDATA and SQLIND are both cleared when the DESCRIBE statement is executed, values must be set after the DESCRIBE statement has executed.
3. When a Column Name Descriptor Area name is specified, the WITH SQLNAME OPTION must be specified in the corresponding PREPARE statement.
4. Specify a Column Name Descriptor Area name only for receipt of the name of a retrieval item or a routine parameter name. A routine parameter name can be received only if a ? parameter is specified by itself in an argument of the CALL statement. If a value expression including a ? parameter is specified, the length of the Column Name Descriptor Area name will be 0.
5. Specify a character set descriptor area only when a character set has to be specified for input ? parameter information for which the data type is character data.

Notes

1. The SQL statement identifier, similar to an embedded variable name, is effective only within a compile-unit file. Multiple SQLs relative to the same SQL statement identifier cannot be used in multiple modules.
2. Even if a DESCRIBE INPUT statement is not used, the same information as that obtained by using DESCRIBE INPUT can be obtained by specifying INPUT in a PREPARE statement.

DESCRIBE CURSOR statement (Receive cursor retrieval information)

Function

Receives cursor query retrieval item information for referencing the result set returned from a procedure into the SQL descriptor area (data code, data length, etc.).

For details about the information to be received, see the *HiRDB Version 9 UAP Development Guide*.

Privileges

None.

Format 1: Receiving cursor retrieval item information

```
DESCRIBE [OUTPUT] CURSOR extended-cursor-name STRUCTURE  INTO
      [:] SQL-descriptor-area[CHARACTER_SET[:]character-set-descriptor-area-name]
```

Operands

For operands other than the `CURSOR` *extended-cursor-name* `STRUCTURE` operand, see *DESCRIBE statement Format 1*.

- `CURSOR` *extended-cursor-name* `STRUCTURE`

extended-cursor-name

Specifies the name of the extended cursor that identifies the cursor allocated to the group of result sets that was returned by a procedure in `ALLOCATE CURSOR` statement Format 2.

For extended cursor names, see *2.27 Extended cursor name*.

- `[:]`*SQL-descriptor-area-name*`[CHARACTER_SET[:]`*character-set-descriptor-area-name*`]`

SQL-descriptor-area-name

Specifies the name of the SQL descriptor area into which cursor retrieval item information is to be stored.

For SQL descriptor areas, see the *HiRDB Version 9 UAP Development Guide*.

character-set-descriptor-area-name

Specifies a character set descriptor area into which a character set name is to be stored for cursor retrieval item information.

For details about character set descriptor areas, see the *HiRDB Version 9 UAP Development Guide*.

Common rules

1. Before executing the DESCRIBE statement, the UAP should assign the number of SQLVAR (SQLN areas) in the SQL descriptor area.
2. Because the SQLDATA and SQLIND are cleared when the DESCRIBE statement is executed, if the DESCRIBE statement is used, a value should be assigned after the statement is executed.

Example

1. Assign retrieval item information on the cursor (extended cursor name: *cr*, scope: GLOBAL) allocated to the group of result sets returned by the procedure PROC1 to *SQL-descriptor-area*.

```
CALL PROC1()
ALLOCATE GLOBAL :cr FOR PROCEDURE PROC1
DESCRIBE CURSOR GLOBAL :cr STRUCTURE INTO :SQLDA
```

DESCRIBE TYPE statement (Receive definition information on user-defined data type)

Function

When a user-defined data type is contained directly or indirectly# in retrieval item information for an SQL statement preprocessed by a PREPARE statement, the DESCRIBE TYPE statement is used to receive the definition information (attribute data codes and data lengths) on the user-defined data type into the SQL descriptor area.

When preprocessing is performed without specifying the WILL ALL TYPE OPTION in the PREPARE statement, definition information on the user-defined data type cannot be received. In such a case, if all encapsulation levels of the attributes of the user-defined data type are other than PUBLIC, 0 is set in the SQLD area of the SQL descriptor area.

#: A user-defined data type is contained directly when the column containing the retrieval item information has a user-defined data type. A user-defined data type is contained indirectly when the column containing the retrieval item information has a user-defined data type and also contains attributes of the user-defined data type (i.e., nested user-defined data type).

Privileges

None.

Format

```
DESCRIBE TYPE :embedded-variable-1 :embedded-variable-2
                FOR { SQL-statement-identifier | extended-statement-name}
INTO
              [:]SQL-descriptor-area-name
                            [[:]Column-Name-Descriptor-Area-name]
              [TYPE [:]Type-Name-Descriptor-Area-name]
```

Operands

- TYPE : *embedded-variable-1* : *embedded-variable-2* FOR {*SQL-statement-identifier* | *extended-statement-name*}

embedded-variable-1

Specifies a VARCHAR(8) embedded variable that stores the name of the owner of the user-defined data type.

embedded-variable-2

Specifies a VARCHAR(30) embedded variable that stores the data type identifier of the user-defined data type.

SQL-statement-identifier

Specifies the SQL statement identifier that was specified in the PREPARE statement.

- [:] *SQL-descriptor-area-name* [[:] *Column-Name-Descriptor-Area-name*] [TYPE [:] *Type-Name-Descriptor-Area-name*]

SQL-descriptor-area-name

Specifies the name of the SQL descriptor area that receives attribute information on the user-defined data type. For details about descriptor areas, see the *HiRDB Version 9 UAP Development Guide*.

extended-statement-name

Specifies the extended statement name that identifies the SQL statement preprocessed by the PREPARE statement.

For extended statement names, see *2.26 Extended statement name*.

Column-Name-Descriptor-Area-name

Specifies the name of the Column Name Descriptor Area that receives the attribute name for the user-defined data type. For details about Column Name Descriptor Areas, see the *HiRDB Version 9 UAP Development Guide*.

Type-Name-Descriptor-Area-name

When the attribute of the user-defined data type is a user-defined data type, specifies the name of the Type Name Descriptor Area that receives the name of the user-defined data type of user-defined type. For details about Type Name Descriptor Areas, see the *HiRDB Version 9 UAP Development Guide*.

Common rules

1. The number of SQLVARs (SQLN area) in the SQL descriptor area must be assigned by the UAP before the DESCRIBE TYPE statement is executed.
2. The following items are assigned: data type of the attribute of the user-defined data type (assigned in the SQL descriptor area), name of the attribute (assigned in the Column Name Descriptor Area), and the name of the user-defined data type when the data type of the attribute is the user-defined type (assigned in the Type Name Descriptor Area).
3. If a specified abstract data type inherited the attributes of a higher-level data type, information on the inherited attributes will also be assigned. In such a case, attributes are set in the following order: inherited attributes first, then attributes specific to the abstract data type.
4. User-defined data types cannot be specified, other than those that are directly or indirectly included in the SQL retrieval items specified in the PREPARE statement.

5. Definition information on an attribute cannot be obtained if the encapsulation level of the attribute is not PUBLIC.
6. When a Column Name Descriptor Area is specified, WITH SQLNAME OPTION must be specified in the corresponding PREPARE statement.

Note

An SQL statement identifier, as embedded variables, is effective in a compile-unit file. Multiple SQL statements with the same SQL statement identifier cannot be used in multiple modules.

Example

Specify attribute information, the attribute name, and the user-defined data type for the attribute of a user-defined data type (embedded variable 1 is WUSERID; embedded variable 2 is WTYPENAME) obtained by a SELECT statement (SQL statement identifier is PRESQL) preprocessed by a PREPARE statement in the following areas: an SQL descriptor area, a Column Name Descriptor Area, and a Type Name Descriptor Area:

```
DESCRIBE TYPE :WUSERID :WTYPENAME
    FOR PRESQL INTO :SQLDA :SQLCNDS :SQLTNDA
```

DROP LIST statement (Delete list)

Function

The DROP LIST statement deletes a list or all lists owned by the user.

Privileges

Creator of the list

The creator of a list can delete the created list.

Format

```
DROP {LIST list-name|ALL LIST}
```

Operands

- LIST *list-name*

Specifies the name of a list that is to be deleted.

- ALL LIST

Specifies that all the lists owned by the user are to be deleted.

Common rules

1. The DROP LIST statement is not subject to the ROLLBACK statement.
2. Although the DROP LIST statement can be executed dynamically, it cannot be executed by embedding it directly in a host program.
3. The same user cannot manipulate a list by connecting to HiRDB concurrently in multiple sessions.

Note

Even if there are no lists to be deleted, executing DROP ALL LIST does not result in an error.

Examples

1. Delete a dependent list (FAMILYT):
```
DROP LIST FAMILYT
```

2. Delete all lists owned by the user:
```
DROP ALL LIST
```

EXECUTE statement Format 1 (Execute SQL)

Function

The EXECUTE statement executes an SQL preprocessed by the PREPARE statement.

Privileges

None.

Format 1: Executing preprocessed SQL statements

```
EXECUTE { SQL-statement-identifier | extended-statement-name}
 [{INTO:embedded-variable [:indicator-variable]
          [, :embedded-variable [:indicator-variable]]...
   |INTO DESCRIPTOR [:]SQL-descriptor-area-name
           [CHARACTER_SET[:]character-set-descriptor-area-name]}]
 [{USING:embedded-variable [:indicator-variable]
          [, :embedded-variable [:indicator-variable]]...
   |USING DESCRIPTOR [:]SQL-descriptor-area-name
    [CHARACTER_SET[:]character-set-descriptor-area-name]}]
```

Operands

- {*SQL-statement-identifier* | *extended-statement-name*}

SQL-statement-identifier

Specifies the SQL statement identifier assigned to the SQL preprocessed by the PREPARE statement.

extended-statement-name

Specifies the extended statement name that identifies the SQL statement preprocessed by the PREPARE statement.

For extended statement names, see *2.26 Extended statement name*.

- INTO:*embedded-variable*[:*indicator-variable*] [, :*embedded-variable*

  [:*indicator-variable*]]...

embedded-variable

If retrieval results from the single-row SELECT statement preprocessed by the PREPARE statement are to be received, or the CALL statement preprocessed by the PREPARE statement includes an output ? parameter, and if the resulting value is to be received into an embedded variable, specifies an embedded variable that receives the value of the column of the retrieval results or the output ? parameter.

indicator-variable

Specifies the indicator variable to which a value is returned, indicating whether the value of the column of the retrieval results returned to the embedded variable or the value of the output parameter is the null value.

The indicator variable should be declared in the embedded SQL declaration section as an embedded variable having the SMALLINT data type.

If you omit the indicator variable, a null value cannot be returned to the embedded variable.

- INTO DESCRIPTOR[:]*SQL-descriptor-area-name*

  [CHARACTER_SET[:]*character-set-descriptor-area-name*]

SQL-descriptor-area-name

When retrieving the results from a single-line SELECT statement preprocessed by a PREPARE statement, or the value of the output ? parameter in a CALL statement that uses the SQL descriptor area, specify the name of the SQL descriptor area that will contain the name of the variable that is to receive the name of the column containing the retrieval results, or the value of the output ? parameter.

character-set-descriptor-area-name

When retrieving the results from a single-line SELECT statement preprocessed by a PREPARE statement, or the value of the output ? parameter in a CALL statement that uses the SQL descriptor area, specify the name of the character set descriptor area into which the character set name of the variable that is to receive the name of the column of the retrieval results, or the value of the output ? parameter, is to be stored.

For details about character set descriptor areas, see the *HiRDB Version 9 UAP Development Guide*.

- USING:*embedded-variable*[:*indicator-variable*][,:*embedded-variable*

  [:*indicator-variable*]]...

embedded-variable

If an SQL statement preprocessed by the PREPARE statement contains an input ? parameter and a value is to be assigned to the parameter from an embedded variable, specifies the embedded variable containing the value.

The number of embedded variables specified in the USING clause must be the same as the number of input ? parameters contained in the SQL that is executed by the EXECUTE statement. The embedded variables and the input ? parameters are associated with each other in the order in which they are listed.

The data type of an embedded variables specified in the USING clause must be compatible with the data types that are permitted for input ? parameters.

indicator-variable

Specifies an indicator variable that indicates whether or not the value of the paired embedded variable is the null value.

The indicator variable should be declared in the embedded SQL declaration section as an embedded variable with the SMALLINT data type.

If no indicator variable is specified, the value of that embedded variable is assumed to be a non-null value.

- USING DESCRIPTOR [:] *SQL-descriptor-area-name*

  [CHARACTER_SET[:] *character-set-descriptor-area-name*]

SQL-descriptor-area-name

If the SQL preprocessed by the PREPARE statement contains an input ? parameter, specifies the name of the SQL descriptor area containing the information that you will assign for that input ? parameter.

character-set-descriptor-area-name

When storing the value in the SQL descriptor area for an input ? parameter specified in the SQL preprocessed by the PREPARE statement, specifies the name of the character set descriptor area into which the character set name of the variable used to assign the value of the input ? parameter is to be stored.

For details about character set descriptor areas, see the *HiRDB Version 9 UAP Development Guide*.

Common rules

1. An SQL executed by the EXECUTE statement must be preprocessed by the PREPARE statement.
2. The PREPARE statement and the EXECUTE statement that executes the SQL preprocessed by the PREPARE statement must be executed in the same transaction.
3. As an embedded variable name, an SQL statement identifier is effective in a compile-unit file. Multiple SQL statements referencing the same SQL statement identifier cannot be used in multiple modules.
4. The number of embedded variables specified in the INTO clause, the number of columns of retrieval results from the single-row SELECT statement executed by the EXECUTE statement, and the number of output ? parameters included in the CALL statement must all be equal. In the SQL statement to be executed in a single-row SELECT statement, if the number of embedded variables is not equal to the number of columns, a warning flag W is assigned to the SQLWARN3 area of the SQL communications area. Notice that the embedded variables, the columns of retrieval results, and the output ? parameters are put in correspondence with

one another from the beginning in the order in which they are listed.

5. The data type of the embedded variables specified in the INTO clause should be the data type allowed in the corresponding retrieval result columns or output ? parameters.
6. If the data fetched into the embedded variable of the fixed-length character string (including national character strings and mixed character strings) specified in the INTO clause is shorter than the defined length of the embedded variable, the data is inserted left-justified, and the remaining characters are blank-filled.
7. If the value of retrieval result columns or the value of the output ? parameter in the CALL statement is the null value, the value of the corresponding embedded variable may be unpredictable.
8. If the embedded variable specified in the INTO clause is character data type and uses the default character set, and the character set of the retrieval results column or output ? parameter of the CALL statement uses a different character set for character data type, it is automatically converted to the character set of the embedded variable.
9. If the embedded variable specified in the USING clause is character data type and uses the default character set, and it is different from the character set of the input ? parameter, it is automatically converted to the character set of the input ? parameter.

Notes

1. The EXECUTE statement can be executed any number of times once the SQL is preprocessed by the PREPARE statement.
2. For details about SQL descriptor area settings, see the *HiRDB Version 9 UAP Development Guide*.

Examples

1. Execute an SQL (SQL statement identifier is PRESQL) preprocessed by the PREPARE statement; specify an embedded variable (QUESTION) that assigns a value to a ? parameter in PRESQL:
```
EXECUTE PRESQL USING :QUESTION
```

2. Execute the SQL (*SQL-statement-identifier*: PRESQL) preprocessed by the PREPARE statement. The user must specify the SQL descriptor area (SQLDA) that is to store the information by which a value is assigned to the ? parameter of PRESQL.
```
EXECUTE PRESQL USING DESCRIPTOR SQLDA
```

3. Execute the SQL (*SQL-statement-identifier*: PRESQL) preprocessed by the PREPARE statement.
```
CALL PROC1(?,?,?)
```

The parameter mode for the first SQL parameter of the procedure PROC1 must be set to IN, the parameter mode for the second SQL parameter must be set to INOUT, and the parameter mode for the third SQL parameter must be set to OUT. The embedded variable XPARAM1 must be specified for the first SQL parameter, the embedded variable XPARAM2 for the second SQL parameter, and the embedded variable XPARAM3 for the third SQL parameter.

```
EXECUTE PRESQL
   INTO  :XPARAM2, :XPARAM3
   USING :XPARAM1, :XPARAM2
```

EXECUTE statement Format 2 (Execute an SQL statement using an array)

Function

The EXECUTE statement executes the SQL statements, preprocessed by the PREPARE statement, in multiple rows in batch using an array.

Privileges

None.

Format 2: Batch execution of multiple rows or multiple times of preprocessed (INSERT, UPDATE, or DELETE) statement

```
EXECUTE {SQL-statement-identifier|extended-statement-name}
    {USING :array-of-embedded-variables[:array-of-indicator-variables]
          [, :array-of-embedded-variables[:array-of-indicator-variables]]...
     |USING DESCRIPTOR [:]SQL-descriptor-area-name
                        [CHARACTER_SET[:]character-set-descriptor-area-name]}
    BY :embedded-variable[ROWS]
```

Operands

- {*SQL-statement-identifier* | *extended-statement-name*}

SQL-statement-identifier

Specifies the SQL identifier assigned to the SQL statement that was preprocessed by the PREPARE statement.

extended-statement-name

Specifies the extended statement name that identifies the SQL statement preprocessed by the PREPARE statement.

For extended statement names, see *2.26 Extended statement name*.

- USING :*array-of-embedded-variables*[:*array-of-indicator-variables*][, :*array-of-embedded-variables*[:*array-of-indicator-variables*]]...

array-of-embedded-variables

When assigning the value of an input ? parameter included in the SQL statement that was preprocessed by the PREPARE statement, specifies the embedded variable having the value to be assigned to the input ? parameter in an array format.

Make the number of embedded variable arrays specified in the USING clause equal to the number of input ? parameters that are included in the SQL statement executed by the EXECUTE statement. The array of embedded variables and the input ? parameters are put in correspondence from the beginning in their respective collating sequences.

The data type of the array of embedded variables specified in the USING clause should be the data type allowed for the corresponding input ? parameter.

array-of-indicator-variables

Specifies an indicator variable in an array format, indicating whether the value of *embedded-variable* is the null value.

The array of indicator variables should be declared in the embedded SQL declare section as an embedded variable in an array format having the SMALLINT data type.

If the *array-of-indicator-variables* operand is omitted, HiRDB assumes that the value of *array-of-indicator-variables* is a non-null value.

- USING DESCRIPTOR [:]*SQL-descriptor-area-name*

  [CHARACTER_SET [:]*character-set-descriptor-area-name*]

SQL-descriptor-area-name

When using the SQL descriptor area to assign a value to the input ? parameter in the SQL preprocessed by the PREPARE statement, specifies the name of the SQL descriptor area that is to be used to store the variable being passed to the input ? parameter.

character-set-descriptor-area-name

When using the SQL descriptor area to assign a value to the input ? parameter in the SQL preprocessed by the PREPARE statement, specifies the name of the character set descriptor area that specifies the character set name used for the variable being passed to the input ? parameter.

For details about character set descriptor areas, see the *HiRDB Version 9 UAP Development Guide*.

- BY :*embedded-variable* [ROWS]

embedded-variable

If the SQL statement preprocessed by the PREPARE statement is an INSERT statement, assigns the number of rows to be processed (inserted). If the SQL statement preprocessed by the PREPARE statement is either an UPDATE or DELETE statement, specifies an embedded variable in which the number of operations (updating or deleting) to be performed is assigned. Specify a SMALLINT type embedded variable.

The following range of values can be specified:

- Specifying `USING` *:array-of-embedded-variables* `[`*:array-of-indicator-variables*`] [,` *:array-of-embedded-variables* `[`*:array-of-indicator-variables*`]]` in the operand:

  The range of values is 1 to 4,096.

- Specifying `USING DESCRIPTOR [:]` *SQL-descriptor-area-name* in the operand:

  The range of values is 1 to 30,000.

Zero and negative values cannot be assigned. Assigning an out-of-range value may cause a runtime error.

The word `ROW` can be omitted without changing the meaning.

Common rules

1. The SQL statement executed by the `EXECUTE` statement must be preprocessed in advance using the `PREPARE` statement.
2. The `EXECUTE` statement Format 2 can be used only if the SQL statement preprocessed by the `PREPARE` statement is the `INSERT`, `UPDATE`, or `DELETE` statement shown below:

   - `INSERT INTO` [*authorization-identifier*.] *table-identifier* [(*column-value* [, *column-value*]...)]

     {`VALUES` (*insertion-value* [, *insertion-value*]...)

     | *query-expression-body* }

     [`WITH ROLLBACK`]

   - `INSERT INTO` [*authorization-identifier*.] *table-identifier* (`ROW`)

     {`VALUES (:`*embedded-variable-array* [`:`*indicator-variable-array*]`)`

     | *query-expression-body* }

     [`WITH ROLLBACK`]

   - `UPDATE` [*authorization-identifier*.] *table-identifier* [[`AS`] *correlation-name*]

     [*used-index-SQL-optimization-specification*]

     `SET` {*update-object* = *update-value*

     | (*update-object* , *update-object* [, *update-object*]) = *row-subquery*}

     [, {*update-object* = *update-value*

     | (*update-object* , *update-object* [, *update-object*]...) = *row-subquery*}]...

[WHERE *search-condition*]

[WITH ROLLBACK]

- UPDATE [*authorization-identifier*.] *table-identifier* [[AS] *correlation-name*]

  SET ROW = *row-update-value*

  [*used-index-SQL-optimization-specification*]

  [WHERE *search-condition*]

  [WITH ROLLBACK]

- DELETE FROM [*authorization-identifier*.] *table-identifier*

  [[AS] *correlation-name*]

  [*used-index-SQL-optimization-specification*]

  [WHERE *search-condition*]

  [WITH ROLLBACK]

If the SQL statement preprocessed by the PREPARE statement is not in the above format, a runtime error may occur.

3. The PREPARE statement and the EXECUTE statement that executes the SQL statement preprocessed by the PREPARE statement should be executed in the same transaction.
4. The SQL statement identifier, similar to the embedded variable, is a name that is effective in the file in the compile unit. More than one SQL statement associated with the same SQL statement identifier cannot be used across multiple modules.
5. Specifying USING :*array-of-embedded-variables*[:*array-of-indicator-variables*][, :*array-of-embedded-variables*[:*array-of-indicator-variables*]]... in the operands is subject to the following rules:
   - The number of elements in *array-of-embedded-variables* and *array-of-indicator-variables* should be in the 1 to 4,096 range. An out-of-range value may cause an error.
   - The number of elements in *array-of-embedded-variables* and *array-of-indicator-variables* should be greater than or equal to the maximum number of rows specified in BY :*embedded-variable*[ROWS].
6. Specifying USING DESCRIPTOR [:]*SQL-descriptor-area-name* in the operands is subject to the following rules:
   - For assigning the value of an input ? parameter to the area indicated by SQLDATA in the SQL descriptor area specified in *SQL-descriptor-area-name*, see the *HiRDB Version 9 UAP Development*

Guide. The number of elements in the array should be greater than or equal to the maximum number of elements specified in `BY :embedded-variable[ROWS]`.

- Assign a value consistent with the data type to the `SQLSYS` area of the SQL descriptor area specified in *SQL-descriptor-area-name*.

  - For a variable-length character string type (`VARCHAR`, `NVARCHAR`, or `MVARCHAR`), assign a length equal to one element, including the area for storing the length of the character string and any gap between elements that arises from a boundary alignment.

    Example:

    In the C language, the following array variable of the `VARCHAR` type requires the value `sizeof(vchr[0])` to be assigned to `SQLSYS`:

    ```
    struct {
        short   len;
        char    str[257];
    } vchr[128];
    ```

  - For other data types, assign the value `0`.

7. The `BLOB` type, `BINARY` with a maximum length of 32,001 bytes or greater, and the abstract data type cannot be specified.
8. If the array of embedded variables specified in the `USING` clause are character data type that use the default character set, and if a different character set is used for the input `?` parameter, it is automatically converted to the character set used for the input `?` parameter.

Notes

1. The `EXECUTE` statement can be executed any number of times per SQL statement that is preprocessed by the `PREPARE` statement.
2. The `EXECUTE` statement Format 2 processes the number of rows specified in `BY :embedded-variable`. Therefore, a value equal to the specified number of rows should be assigned to the area pointed to by `SQLDATA` to which the value of *array-of-embedded-variables*, *array-of-indicator-variables*, or *input-?-parameter* is assigned.

Examples

1. Insert 50 rows of data into the inventory table in batch, assigned to the array variable in the C language:

```
EXEC SQL BEGIN DECLARE SECTION;
  short   XINSERT_NUM;
```

```
  long    XPCODE[50];
  short   IPCODE[50];
  char    XPNAME[50][17];
  short   IPNAME[50];
EXEC SQL END DECLARE SECTION;

EXEC SQL
  PREPARE PRESQL FROM
      'INSERT INTO STOCK(PCODE, PNAME) VALUES (?, ?)';

                :
  Assign a value to each element of the array variable.
                :

XINSERT_NUM = 50;
EXEC SQL
  EXECUTE PRESQL USING :XPCODE:IPCODE, :XPNAME:IPNAME
      BY :XINSERT_NUM ROWS;
```

2. Update the values of the product code (PCODE) and quantity in stock (SQTY) specified in the array variable written in C to the values listed in the following tables:

Table 4-4: Product code and quantity in stock stored in table (before updating)

Product code	New quantity in stock
'101M'	40
'101L'	70
'201M'	15
'202M'	28
'302S'	7

Table 4-5: Product code and quantity in stock subject to updating (assigned to an embedded variable array)

Product code	New quantity in stock
'101M'	35
'101L'	62
'201M'	13
'202M'	10
'302S'	6

Table 4-6: Product code and quantity in stock stored in table (after updating)

Product code	New quantity in stock
'101M'	35
'101L'	62
'201M'	13
'202M'	10
'302S'	6

```
    EXEC SQL BEGIN DECLARE SECTION;
          short   XUPDATE_NUM;
          char    XPCODE[5][5];
          short   IPCODE[5];
          long    XSQTY[5];
          short   ISQTY[5];
    EXEC SQL END DECLARE SECTION;

EXEC SQL
PREPARE PRESQL FROM
      'UPDATE STOCK SET SQTY = ? WHERE PCODE = ?';
    . . . Assign value to elements of variable array . . .
        Assign {'101M','101L','201M','202M','302S'} to XPCODE
        Assign {35,62,13,10,6} to XSQTY
XUPDATE_NUM = 5;
EXEC SQL
EXECUTE PRESQL USING :XSQTY:ISQTY, :XPCODE:IPCODE
BY :XUPDATE_NUM
```

EXECUTE IMMEDIATE statement (Preprocess and execute SQL)

Function

The EXECUTE IMMEDIATE statement preprocesses and executes an SQL provided in a character string.

Privileges

None.

Format

```
EXECUTE IMMEDIATE {'character-string'|:embedded-variable}
 [ {INTO :embedded-variable [:indicator-variable]
    [, : embedded-variable[: indicator-variable]]...
  |INTO DESCRIPTOR [:]SQL-descriptor-area-name
          [CHARACTER_SET[:]character-set-descriptor-area-name]}]
 [{USING : embedded-variable[: indicator-variable]
      [, : embedded-variable[: indicator-variable]]...
  |USING DESCRIPTOR [:]SQL-descriptor-area-name
    [CHARACTER_SET[:]character-set-descriptor-area-name]}]
```

Operands

- {'*character-string*'|:*embedded-variable*}

character-string

Specifies directly as a character literal the character string representing the SQL to be executed.

An apostrophe within the character literal specification of an SQL to be executed must be specified as two apostrophes in succession.

embedded-variable

Specifies as an embedded variable the character string of the SQL to be executed.

An embedded variable must be preceded by a colon (:).

- INTO: *embedded-variable* [: *indicator-variable*] [, : *embedded-variable* [: *indicator-variable*]]...

embedded-variable

When retrieval results from a single-row SELECT statement are to be received or the CALL statement contains an output ? parameter and its value is to be received into *embedded-variable*, specifies the embedded variable that receives the value of the column of retrieval results or the value of the output ? parameter.

indicator-variable

Specifies an indicator variable to which a value indicating whether the value of a column of retrieval results returned to the embedded variable is the null value.

The indicator variable should be declared in the embedded SQL declaration section as an embedded variable having a SMALLINT data type.

If *indicator-variable* is omitted, the null value cannot be received.

- INTO DESCRIPTOR [:] *SQL-descriptor-area-name*

  [CHARACTER_SET[:] *character-set-descriptor-area-name*]

SQL-descriptor-area-name

When receiving the results from a single-line SELECT statement, or the value of the output ? parameter from a CALL statement that uses an SQL descriptor area, specifies the name of the SQL descriptor area that specifies the variable to receive the column value of the retrieval results, or the value of the output ? parameter.

character-set-descriptor-area-name

When receiving the results from a single-line SELECT statement, or the value of the output ? parameter from a CALL statement that uses an SQL descriptor area, specifies the name of the character set descriptor area that specifies the character set name of the variable used to receive the column value of the retrieval results, or the value of the output ? parameter.

For details about character set descriptor areas, see the *HiRDB Version 9 UAP Development Guide*.

- USING: *embedded-variable* [:*indicator-variable*] [, :*embedded-variable* [:*indicator-variable*]]...

embedded-variable

If the SQL statement includes an input ? parameter and if its value is given by an embedded variable, specifies the embedded variable that contains the value to be assigned to the input ? parameter.

The number of embedded variables specified in the USING clause must be equal to the number of input ? parameters included in the SQL statement that is executed by the EXECUTE IMMEDIATE statement. Notice that the embedded variables and the input ? parameters are placed in correspondence with each other from the beginning in the order in which they are listed.

The embedded variable specified in the USING clause must have a data type that is allowed for the corresponding input ? parameter.

indicator-variable

Specifies an indicator variable that indicates whether the value of the embedded

variable is the null value.

The indicator variable should be declared in the embedded SQL declaration section as an embedded variable having a SMALLINT data type.

If *indicator-variable* is omitted, the value of the indicator variable is assumed to be a non-null value.

- USING DESCRIPTOR [:] *SQL-descriptor-area-name*

  [CHARACTER_SET[:] *character-set-descriptor-area-name*]

SQL-descriptor-area-name

When assigning a value to the input ? parameter using an SQL descriptor area, specifies the name of the SQL descriptor area that specifies the variable used to assign the value of the input ? parameter.

character-set-descriptor-area-name

When assigning a value to the input ? parameter using an SQL descriptor area, specifies the name of the character set descriptor area that specifies the character set name of the variable used to assign the value of the input ? parameter.

For details about character set descriptor areas, see the *HiRDB Version 9 UAP Development Guide*.

Common rules

1. Neither the SQL prefix nor the SQL terminator can be specified in the SQL character string that is to be executed.
2. The maximum length of an SQL to be executed is 2,000,000 bytes. If the SQL character string is specified as a character literal, the maximum length is the maximum length of character literals in the UAP description language.
3. The type of an embedded variable is the following structure:

```
struct {
   long  xxxxxxx;   /* Effective length of SQL statement */
   char  yyyyyyy[n]; /* SQL statement storage area */
} zzzzzzz;
```

   The characters xxxxxxx indicate the effective length of the character string stored in character array yyyyyyy.

   $1 \leq$ (value of xxxxxxx) ≤ 2000000

   The effective length of a character string does not include the characters \0 that indicate the end of the character string.

   n is any value.
4. The number of embedded variables specified in the INTO clause should be equal

to the number of retrieval result columns. If the SQL statement being executed is a single-row SELECT statement, and if the number of embedded variables and the number of columns are not equal, a warning flag W is assigned to the SQLWARN3 area of the SQL communications area. Notice that the embedded variables, the columns of retrieval results, or the embedded variables and the output ? parameters, are placed in correspondence from the beginning in the order in which they are listed.

5. The embedded variable specified in the INTO clause should have a data type that is identical to the data type of the corresponding column or the output parameter, or a convertible data type.
6. If the data fetched into an embedded variable of a fixed-length character string (including national character strings and mixed character strings) specified in the INTO clause is shorter than the defined length of the embedded variable, the data is inserted left-justified, and the remaining data is blank-filled.
7. If the value of the retrieval result column or the value of the output ? parameter of the CALL statement is the null value, the value of the corresponding embedded variable may be unpredictable.
8. If the embedded variable specified in the INTO clause is character data type that uses the default character set, and a different character set is used for the retrieval result column, or output ? parameter of the CALL statement, for character data type, it is automatically converted to the character set of the embedded variable.
9. If the embedded variable specified in the USING clause is character data type that uses the default character set, and it is different from the character set used for the input ? parameter, it is automatically converted to the character set used for the input ? parameter.

Notes

The EXECUTE IMMEDIATE statement produces the same result as executing the SQL shown below:

When an SQL specified in a character string is to be executed multiple times, the PREPARE statement should be used to preprocess the SQL and then the EXECUTE statement should be used to execute it repeatedly.

- PREPARE *SQL-statement-identifier* FROM {'*character-string*' |:*embedded-variable*}
- EXECUTE *SQL-statement-identifier*

The following SQLs can be preprocessed and executed using the EXECUTE IMMEDIATE statement:

- Data manipulation SQLs:

  ASSIGN LIST statement, CALL statement, DELETE statement,

preparable-dynamic DELETE *statement*:*locating*, DROP LIST statement, INSERT statement, PURGE TABLE statement, *single-row-SELECT-statement*, UPDATE statement, and *preparable-dynamic* UPDATE *statement*:*locating*

- Control SQLs:

  LOCK TABLE statement

- Definition SQLs:

  ALTER INDEX, ALTER PROCEDURE, ALTER ROUTINE, ALTER TABLE, ALTER TRIGGER, COMMENT, CREATE AUDIT, CREATE CONNECTION SECURITY, CREATE FUNCTION, CREATE INDEX, CREATE PROCEDURE, CREATE SCHEMA, CREATE SEQUENCE, CREATE TABLE, CREATE TRIGGER, CREATE TYPE, CREATE VIEW, DROP AUDIT, DROP CONNECTION SECURITY, DROP DATA TYPE, DROP FUNCTION, DROP INDEX, DROP PROCEDURE, DROP SCHEMA, DROP SEQUENCE, DROP TABLE, DROP TRIGGER, DROP VIEW, GRANT, and REVOKE statements

Examples

1. Preprocess and execute SQL 'PURGE TABLE STOCK' that is provided in a character string:
   ```
   EXECUTE IMMEDIATE 'PURGE TABLE STOCK'
   ```

2. Preprocess and execute an SQL statement that is defined as an embedded variable (:STOCKX):
   ```
   EXECUTE IMMEDIATE :STOCKX
   ```

3. Preprocess SQL 'SELECT PNAME FROM STOCK WHERE PCODE = ?' given as a character string, and read the retrieval results into the embedded variable (:XPNAME) and indicator variable (:IPNAME). In this operation, specify the embedded variable (:XPCODE) and the indicator variable (:IPCODE) that store information on the value to be assigned to the ? parameter.

   ```
   EXECUTE IMMEDIATE 'SELECT PNAME FROM STOCK WHERE PCODE = ?'
         INTO :XPNAME:IPNAME
         USING :XPCODE:IPCODE
   ```

FETCH statement Format 1 (Fetch data)

Function

The FETCH statement advances to the next row the cursor that indicates the row to be fetched and reads the column values in that row into the embedded variables specified in the INTO clause.

Privileges

None.

Format 1: Reading one line of retrieval results into variables

```
FETCH {cursor-name | extended-cursor-name} INTO
    {:embedded-variable [:indicator-variable]
      | [statement-label.] SQL-variable-name
      | [[authorization-identifier.] routine-identifier.] SQL-parameter-name
      | [statement-label.] SQL-variable-name..attribute-name
                [..attribute-name]...
      | [[authorization-identifier.] routine-identifier.] SQL-parameter-name
                ..attribute-name[..attribute-name]...}
        [, {:embedded-variable [:indicator-variable]
          | [statement-label.] SQL-variable-name
          | [[authorization-identifier.] routine-identifier.]
            SQL-parameter-name
          | [statement-label.] SQL-variable-name..attribute-name
                [..attribute-name]...
          | [[authorization-identifier.] routine-identifier.]
            SQL-parameter-name
                ..attribute-name[..attribute-name]...} ]...
```

Operands

- {*cursor-name* | *extended-cursor-name*}

cursor-name

Specifies the name of the cursor being used to fetch retrieval results.

extended-cursor-name

Specifies the extended cursor name of the cursor into which retrieval results are fetched.

For extended cursor names, see *2.27 Extended cursor name*.

- {:*embedded-variable*[:*indicator-variable*]

|[*statement-label* .]*SQL-variable-name*

|[[*authorization-identifier* .]*routine-identifier* .]*SQL-parameter-name*

|[*statement-label* .]*SQL-variable-name*..*attribute-name*

[..*attribute-name*]...

|[[*authorization-identifier* .]*routine-identifier* .] *SQL-parameter-name*

..*attribute-name*[..*attribute-name*]...}

[, {:*embedded-variable*[: *indicator-variable*]

|[*statement-label* .]*SQL-variable-name*

|[[*authorization-identifier* .]*routine-identifier* .]

SQL-parameter-name

|[*statement-label* .]*SQL-variable-name*..*attribute-name*

[..*attribute-name*]...

|[[*authorization-identifier* .]*routine-identifier* .]

SQL-parameter-name

..*attribute-name*[..*attribute-name*]...}]...

embedded-variable

Specifies an embedded variable into which a non-null column value is to be read.

To receive a column value that is the null value, an embedded variable and an indicator variable must both be specified.

indicator-variable

Specifies the indicator variable that is to receive the value returned by the system indicating whether or not the column value read into the paired embedded variable is the null value.

[*statement-label* .]*SQL-variable-name*

[[*authorization-identifier* .]*routine-identifier* .] *SQL-parameter-name*

Specifies an SQL variable or an SQL parameter that is to receive the value of a column in the SQL procedure.

If an authorization identifier is specified in the SQL procedure statement of a public procedure definition, specify upper-case `PUBLIC` enclosed in double quotation marks (`"`) as the authorization identifier.

[*statement-label* .] *SQL-variable-name*..*attribute-name* [..*attribute-name*]

[[*authorization-identifier* .]

routine-identifier .]*SQL-parameter-name* . . *attribute-name*[. . *attribute-name*]

Specify these operands in order to receive the value of an attribute in a column within the SQL procedure.

If an authorization identifier is specified in the SQL procedure statement of a public procedure definition, specify upper-case `PUBLIC` enclosed in double quotation marks (`"`) as the authorization identifier.

Common rules

1. Unless allocated by the `ALLOCATE CURSOR` statement Format 2 (allocate a result set cursor), the cursor specified in the `FETCH` statement should be opened using the `OPEN` statement.
2. The number of retrieval result columns (the number of selection expressions specified in the `SELECT` statement specified in either a cursor declaration or an `ALLOCATE CURSOR` statement) must be equal to the number of embedded variables, SQL variables, or SQL parameters specified in the `INTO` clause of the `FETCH` statement. If they are not, there will be fewer column values to be read into the embedded variables. In such a case, a warning flag (`W`) is set in `SQLWARN3` in the SQL Communications Area.
3. The data type of each embedded variable specified in the `INTO` clause must be either the same as the data type of the corresponding retrieval item or a data type that can be converted into that data type.
4. If the embedded variable is character data type that uses the default character set, and the retrieval result is character data type that uses a different character set, it is automatically converted to the character set used by the embedded variable.
5. If data fetched into the embedded variable of a fixed-length character string (including a national character string or a mixed character string) is shorter than the length of the retrieval item, the data is left-justified in the embedded variable and trailing blanks are added.
6. If there are no rows to be fetched, the system returns the following return codes:
   - Return code 100 to `SQLCODE` in the SQL Communications Area
   - Return code 100 to the `SQLCODE` variable
   - Return code `'02000'` to the `SQLSTATE` variable

   However if a row that existed when the list was created during a search using the list or the value of an attribute is deleted or updated, codes `110`, `110`, and `'R2000'` will be set, respectively.

Notes

1. If a retrieval result column contains the null value, the value of the corresponding

embedded variable is unpredictable.

2. The cursor name, similar to an embedded variable name, is effective only within a compile-unit module. Multiple SQLs relative to the same cursor cannot be used in multiple files.

Examples

Use a cursor (CR1) to read retrieval results from a stock table (STOCK) into the embedded variables and indicator variables associated with the product code (PCODE), product name (PNAME), color (COLOR), unit price (PRICE), and quantity-in-stock (SQTY) columns:

1. With embedded variables specified:
```
FETCH CR1
  INTO :XPCODE,:XPNAME,:XCOLOR,
       :XPRICE,:XSQTY
```

2. With embedded variables and indicator variables specified:
```
FETCH CR1
  INTO :XPCODE:IPCODE,
       :XPNAME:IPNAME,
       :XCOLOR:ICOLOR,
       :XPRICE:IPRICE,
       :XSQTY:ISQTY
```

FETCH statement Format 2 (Fetch data)

Function

Fetches one or more rows of the search results into the area specified in the SQL descriptor area.

Privileges

None.

Format 2: Reading one or more lines of retrieval results into specified receive areas in the SQL descriptor area

```
FETCH {cursor-name | extended-cursor-name}
    USING DESCRIPTOR [:]SQL-descriptor-area-name
        [CHARACTER_SET[:]character-set-descriptor-area-name]

    [BY : embedded-variable [ROWS]]
```

Operands

- {*cursor-name* | *extended-cursor-name*}

cursor-name

Specifies the name of the cursor being used to fetch the retrieval results

extended-cursor-name

Specifies the extended cursor name of the cursor into which retrieval results are fetched.

For extended cursor names, see *2.27 Extended cursor name*.

- [:]*SQL-descriptor-area-name*

Specifies the name of the SQL descriptor area to be used in order for the UAP to receive the retrieval results.

- [CHARACTER_SET[:]*character-set-descriptor-area-name*]

Specifies the name of the character set descriptor area to be used to specify the character set name of the retrieval results.

- [BY : *embedded-variable* [ROWS]]

By using an array that is set in the SQL descriptor area, specifies the embedded variable in which the size of the FETCH area is set in terms of the number of elements. A SMALLINT data type embedded variable with a value in the range 1 to 30000 should be specified. An error may result if 0 or a negative value is set. The behavior of the

FETCH statement cannot be guaranteed if a client library older than Version 05-03 is used.

Common rules

1. Unless allocated by the ALLOCATE CURSOR statement Format 2 (allocate a result set cursor), the cursor specified in the FETCH statement should be opened using the OPEN statement.
2. The information needed by the UAP for the execution of the FETCH statement should be assigned to the SQL descriptor area specified in *SQL-descriptor-area-name*. For details about the SQL descriptor area, see the *HiRDB Version 9 UAP Development Guide*.
3. If there are no rows to be fetched, the system returns the following return codes:
   - Return code 100 to SQLCODE in the SQL Communications Area
   - Return code 100 to the SQLCODE variable
   - Return code '02000' to the SQLSTATE variable

   However if a row that existed when the list was created during a search using the list or the value of an attribute is deleted or updated, codes 110, 110, and 'R2000' will be set, respectively.
4. The data type of a receive area in the specified SQL descriptor area must be either the data type of the corresponding retrieval item or a data type that can be converted into that data type.
5. If the retrieval result is character data and the character set of the retrieval result is different from the character set specified in the character set descriptor area, the retrieval results are automatically converted into the character set specified in the character set descriptor area.
6. When BY : *embedded-variable* [ROWS] is specified, a value that corresponds with the data type in the SQLSYS area of the SQL descriptor area should be set.
   - Variable-length character string type (VARCHAR, NVARCHAR, or MVARCHAR):

     Set an area that stores the length of the character string and a value equivalent to one element, including any gap between elements that is necessitated by boundary alignment.

     For example, in the case of the following array variable of VARCHAR data type, the value set in SQLSYS will be sizeof(vchr[0]):

```
struct {
    short  len;
    char   str[257];
} vchr[128];
```

   - Other data types:

Set the value 0.

Note

The cursor name, similar to an embedded variable name, is effective only within a compile-unit module. Multiple SQLs relative to the same cursor cannot be used in multiple files.

Example

Use a cursor (`CR2`) to read retrieval results from a stock table into receive areas in a specified SQL descriptor area:

```
FETCH CR2
  USING DESCRIPTOR SQLDA
```

FETCH statement Format 3 (Fetch data)

Function

Fetches multiple rows of the search results into the embedded variable specified in the `INTO` clause.

Privileges

None.

Format 3: Reading more line of retrieval results all at once into embedded variables specified in the INTO clause

```
FETCH {cursor-name | extended-cursor-name} INTO :
embedded-variable-array [:indicator-variable-array]
                    [,:embedded-variable-array [:indicator-variable-array]] ...
```

Operands

- {*cursor-name* | *extended-cursor-name*}

cursor-name

Specifies the name of the cursor being used to fetch the retrieval results.

extended-cursor-name

Specifies the extended cursor name of the cursor into which retrieval results are fetched.

For extended cursor names, see *2.27 Extended cursor name*.

- : *embedded-variable-array*[: *indicator-variable-array*]

  [, : *embedded-variable-array*[: *indicator-variable-array*]]...

embedded-variable-array

Specifies the array variables (embedded variables declared in array format) into which the values of columns that do not contain the null value are to be fetched. To receive values from columns that contain the null value, both embedded variables and indicator variables must be specified.

indicator-variable-array

Specifies the indicator variables (indicator variables declared in array format) to which the values indicating whether or not the values of the columns to be read into the embedded variables contain the null value are returned.

Common rules

1. Unless allocated by the ALLOCATE CURSOR statement Format 2 (allocate a result set cursor), the cursor specified in the FETCH statement should be opened using the OPEN statement.
2. An embedded variables array and its paired indicator variables array must have the same number of elements. If they do not, the number of rows fetched will equal the number of elements in the smaller of the specified arrays.
3. The cumulative number of rows actually fetched is set in the SQL Communications Area's SQLERRD2 in the case of the C language and in SQLERRD(3) in the case of COBOL.
4. A FETCH statement that uses arrays cannot be used in a procedure.
5. A FETCH statement that uses arrays cannot handle LOB data.
6. Block transfer cannot be specified in a FETCH statement that uses arrays.
7. If there are no more rows to be fetched, the system returns the following return codes; in such a case, the data in the preceding rows is returned:
   - Return code 100 to SQLCODE in the SQL Communications Area
   - Return code 100 to the SQLCODE variable
   - Return code '02000' to the SQLSTATE variable

   If a row that was present at the time the list was created during a retrieval through a list is deleted, or if an attribute value is deleted or updated, a return code indicating that fact is not set. In this case, HiRDB continues to perform the retrieval, ignoring the event.
8. If an event requiring a warning occurs in any of the fetched rows, the warning information is set in SQLWARN of the SQL Communications Area.
9. If an error occurs in any of the fetched rows, the data in the rows up to that row is returned.
10. A FETCH statement that uses an array cannot be used for searching unsubscripted repetition columns.

Notes

1. If the value in a column of the retrieval results is the null value, the value in the element of the corresponding embedded variable array cannot be guaranteed.
2. Array-type and non-array-type variables cannot be mixed.
3. SQL descriptor area cannot be specified.
4. The cursor is positioned at the last fetched row. If an updating operation using the

cursor is then executed, the last fetched row is updated.

Example

Use a cursor (CR3) to read retrieval results from an inventory table into an array variable in C language:

```
EXEC SQL BEGIN DECLARE SECTION;
  long  XPCODE[50];
  short IPCODE[50];
  char  XPNAME[50][17];
  short IPNAME[50];
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH CR3
    INTO :XPCODE :IPCODE,
           :XPNAME :INAME;
```

FREE LOCATOR statement (Invalidate locator)

Function

Deletes the link between a locator and the data allocated to it, and nullifies the locator.

Privileges

None.

Format

```
FREE LOCATOR locator-reference [, locator-reference]...
locator-reference:: = : embedded-variable
```

Operands

- *locator-reference*`:: = :` *embedded-variable*

Specifies the embedded variable for the locator to be nullified.

Common rule

1. Specifying an embedded variable having the value for an invalid locator causes an error. If multiple embedded variables are specified, and those embedded variables include an embedded variable having the value of an invalid locator, an error occurs, and all valid locators are nullified.

INSERT statement Format 1 (Insert row)

Function

Inserts rows into a table in units of columns. The statement can be used to insert one row by directly specifying values. In addition, this command can also insert one or more rows by using a query expression body.

Privileges (Format 1)

A user who has the INSERT privilege for a table can insert rows into that table.

However, if a query is specified in the INSERT statement, the user needs the SELECT privilege for the table for which the query is specified.

Format 1: Inserting rows into a table on a column-by-column basis

```
INSERT INTO [authorization-identifier.]
                table-identifier
      {[(column-name [, column-name]...)]
        {VALUES (insertion-value [, insertion-value]...)
       |query-expression-body}
       |DEFAULT VALUES}
      [WITH ROLLBACK]

 query-expression-body::= {query-specification
             |(query-expression-body)
             |query-expression-body {UNION|EXCEPT}[ALL]
              query-specification|(query-expression-body)}}
 query-specification ::= SELECT[{ALL|DISTINCT}]{selection-expression[,
selection-expression]...|*} table-expression
 table-expression   ::= FROM table-reference[, table-reference]...
             [WHERE search-condition]
             [GROUP BY value-expression[, value-expression]...]
             [HAVING search-condition]
```

Operands

- [*authorization-identifier*.]*table-identifier*

authorization-identifier

Specifies the authorization identifier of the owner of the table.

MASTER cannot be specified as an authorization identifier.

table-identifier

Specifies the name of the table into which rows are to be inserted.

- [(*column-name* [, *column-name*])]

Specifies the names of the columns into which data is to be inserted.

The following rules apply to column names:

1. In read-only view tables, rows cannot be inserted, updated, or deleted. For details about read-only view tables, see *Common rules* under *CREATE [PUBLIC] VIEW (Define view, define public view)* in Chapter 3.
2. When row insertion into a view table is specified, the system inserts the rows into the base table of the view table. Any columns in the view table's base table that are not associated with a column in the view table receive the null value. Therefore, in the case of a view table defined from a base table with the `FIX` attribute, rows are not inserted if there are columns of the view table's base table that are not associated with columns in the view table.
3. Any column that is not specified receives the null value.
4. The number of elements in unspecified repetition columns will be 0.
5. Not specifying any column names is equivalent to specifying all columns in the specification order of the columns when the columns were defined in the table definition.
6. A subscript cannot be specified as part of a column name.

- `VALUES` (*insertion-value* [, *insertion-value*])

insertion-value

Specifies an insertion value for a specified column. The following items can be specified:

- Value expressions
- `NULL` (represents the null value)
- `DEFAULT`
- `ARRAY` [*element-value*[, *element-value*]...][#]

#: The following can be specified in element-value:

- Value expression
- `NULL` (represents the null value)
- `DEFAULT`

The following rules apply to insertion values:

1. Insertion values must be specified in the order in which the column names are specified.

2. To assign the null value as an insertion value, NULL must be specified.
3. If DEFAULT is specified in *insertion-value* or as a value of an element in *insertion-value*, the predefined value of the column that is the target of insertion is inserted. If the DEFAULT clause is present, the insertion value is the specified predefined value. If the DEFAULT clause is not present but WITH DEFAULT is specified, the insertion value is the predefined value for WITH DEFAULT. If neither the DEFAULT clause nor WITH DEFAULT is present, NULL is the predefined value.
4. If SYSTEM GENERATED is specified for the column associated with *insertion-value*, the specified insertion value is ignored; instead, the current date (CURRENT_DATE) is inserted for the DATE type, or the current time (CURRENT_TIME) for the TIME type.
5. In the case of preprocessing using the PREPARE statement, an embedded variable or an indicator variable cannot be specified.
6. Embedded variables must be of the same structures as the column structures of the associated columns.
7. In the case of an embedded variable used to assign a value to a ? parameter, the same structure as the structure of the associated column must be specified.
8. ARRAY [*element-value* [, *element-value*] ...] can be specified only if the corresponding column is a repetition column. The maximum number of elements that can be specified is 30,000, provided that they do not exceed the maximum number of elements allowed in the column into which they are inserted. The embedded variable (indicator variable) and the ? parameter for element values should be of a simple structure.
9. A value expression containing a column name or a set function cannot be specified in an insertion value.
10. Neither a WRITE specification nor a GET_JAVA_STORED_ROUTINE_SOURCE specification can be specified in an insertion value.

- *query-expression-body* : : = {*query-specification*
  | (*query-expression-body*)
  | *query-expression-body* {UNION | EXCEPT} [ALL]
  {*query-specification* | (*query-expression-body*) } }

Specifies the query specification body that fetches the data to be inserted.

For details about query expression bodies, see *2.2 Query expressions*.

- *query-specification* : : = SELECT [{ ALL | DISTINCT}]

{*selection-expression* [, *selection-expression*]... | *} *table-expression*

For details about query specifications, see *2.3 Query specification*.

If the column into which the fetched data is to be inserted is a repetition column, specify an unsubscripted repetition column in the selection expression in the query specification associated with that column.

- *table-expression*:: = FROM *table-reference* [, *table-reference*]...

  [WHERE *search-condition*]

  [GROUP BY *value-expression* [, *value-expression*]...]

  [HAVING *search-condition*]

For table expressions, see *2.5 Table expressions*. For table referencing, see *2.6 Table reference*. For search conditions, see *2.7 Search conditions*.

- DEFAULT VALUES

Inserts a predefined value for all columns in the row to be inserted.

DEFAULT VALUES has the same meaning as the specification of the following format:
VALUES(DEFAULT,DEFAULT,...)

The number of DEFAULT specifications above is equal to the number of columns in the table that is subject to the insertion operation.

- [WITH ROLLBACK]

Specifies that if the table that is subject to the insertion operation is being used by another user, the transaction issued by that user is to be cancelled and invalidated.

When the WITH ROLLBACK option is omitted and the table subject to the insertion operation is being used by another user, the current user must wait until the transaction issued by that user is completed.

Common rules

1. Care must be taken that the embedded variable, SQL variable, or SQL parameter has either the same data type as the data type of the corresponding column or a data type that can be converted into the data type of the corresponding column.
2. The ? parameter can be specified in the INSERT statement only if the INSERT statement is preprocessed by the PREPARE statement. The value to be assigned to a ? parameter is specified in an embedded variable in the USING clause of the EXECUTE statement that is associated with the PREPARE statement that prepares the INSERT statement.
3. Embedded variables and indicator variables cannot be used in the INSERT statement or in an SQL procedure that is preprocessed by the PREPARE statement. For details about specification values in a Java procedure, see *JDBC drivers or*

SQLJ in the *HiRDB Version 9 UAP Development Guide*. For details about search conditions, see *2.7 Search conditions*.

4. The number of columns in a row to be inserted must equal the number of columns for which column names are specified.

   The values of these columns must be of either the data types of the columns or data types that can be converted into those data types. (Note that if the column into which the data is to be inserted is of the national character data type and a character string literal is specified as the insertion value, the character string literal will be treated as a national character string literal. When a character string literal is treated as a national character string literal, only the character data length is checked; the character codes are not checked.) When rows are to be inserted, the number of columns retrieved by a query specification body must be equal to the number of columns specified in *column-name*. The data types of the corresponding columns must be convertible data types. (If the column into which data is to be inserted is the national character data type and if a character string literal is specified in the selection expression for the query expression body, the character string literal is treated as a national character string literal. When a character string literal is treated as a national character string literal, only the character data length is checked; the character codes are not checked.)

5. When fixed-point or floating-point data is inserted into any of the following data type columns, any digits following the decimal point are rounded off:

   - `INTEGER`
   - `SMALLINT`

   During the insertion of fixed-point data into a column of the `DECIMAL` type, any digits below the scaling for the column are rounded off.

6. Character data greater than the length specified when the table was defined cannot be inserted.

7. Numeric data outside the range of the data types defined for a column cannot be inserted.

8. If the data to be inserted into a fixed-length character string column (including a national character string or a mixed character string column) is shorter than the length of the column, the data is left-justified in the column and trailing blanks are added.

9. The following items can be specified when data is inserted into a column of `BLOB` data type: embedded variables, `?` parameters, SQL variables, SQL parameters, `SUBSTR` scalar function, function calls, and `NULL`.

10. When data is inserted into a column of an abstract data type, embedded variables and `?` parameters cannot be specified in the values to be inserted.

11. A component specification cannot be specified in a selection expression in the query expression body.
12. When data is inserted into a column of an abstract data type, the following item cannot be specified in the values to be inserted: values of an abstract data type, including the BLOB attribute for which no LOB attribute storage RDAREA name was specified in the table definition.
13. Neither a WRITE specification nor a GET_JAVA_STORED_ROUTINE_SOURCE specification can be specified in a selection expression in the query expression body.
14. If the INSERT statement is executed on a table with a WITHOUT ROLLBACK specification, the timing for the release of row locking can vary depending on whether an index is defined. For details, see the rules on WITHOUT ROLLBACK in *CREATE TABLE (Define table)* in Chapter *3*.
15. Before inserting data into a shared table, the LOCK statement with respect to the table should be executed in the lock mode. An attempt to insert data into a shared table without executing the LOCK statement can cause an error. For details about updating a shared table, see the *HiRDB Version 9 Installation and Design Guide*. For the objects of locking during execution of the LOCK statement on a shared table, see the notes in *LOCK statement (Lock control on tables)* in Chapter *5*.
16. To insert data into a character data type column, make sure the character set used for the data being inserted is the same as the character set used for the column into which it is being inserted. However, if the data being inserted is an embedded variable (that uses the default character set), ? parameter, or a string constant, it is automatically converted into the character set used for the column into which it is being inserted.

Rules on referential constraints

1. For the rules on inserting rows into a referenced table or referencing table, see the explanation of referencing actions in *CREATE TABLE (Define table)* in Chapter *3*.

Examples

1. Insert rows with the values read into embedded variables into all columns of a stock table named STOCK:

```
INSERT INTO STOCK
    VALUES(:XPCODE,:XPNAME,:XCOLOR,
           :XPRICE,:XSQTY)
```

2. Insert rows with the values read into embedded variables into the product code (PCODE), product name (PNAME), and quantity-in-stock (SQTY) columns of a stock table named STOCK:

```
INSERT INTO STOCK
```

```
    (PCODE,PNAME,SQTY)
    VALUES(:XPCODE,:XPNAME,:XSQTY)
```

3. Insert into stock table STOCK all table data from stock table 2 (STOCK2) with the same column definition information as the STOCK stock table:
```
INSERT INTO STOCK
  SELECT * FROM STOCK2
```

4. Insert data representing 612S PANTS, and WHITE into the following columns in a stock table (STOCK): product code (PCODE), product name (PNAME), and color (COLOR):
```
INSERT INTO STOCK (PCODE,PNAME,COLOR)
  VALUES('612S',N'PANTS',N'WHITE')
```

5. Insert the following items into columns in an orders table (ORDERS): 02561, TT001, 302S, 50, current date (CURRENT_DATE), and current time (CURRENT_TIME):
```
INSERT INTO ORDERS
  VALUES('02561','TT001','302S',50,
         CURRENT_DATE,CURRENT TIME)
```

INSERT statement Format 2 (Insert row)

Function

Treats an entire row as a data item and inserts rows in units of rows into a FIX-attribute table. Can insert a single row through the direct specification of a value. In addition, this command can also insert one or more rows by using a query expression body.

Privileges (Format 2)

A user who has the INSERT privilege for a table can insert rows into that table.

However, if a query is specified in the INSERT statement, the user needs the SELECT privilege for the table for which the query is specified.

Format 2: Inserting one row into a table with the FIX attribute on a row-by-row basis, by treating the entire row as a set of data

```
INSERT INTO [authorization-identifier.]table-identifier|(ROW)
      {VALUES (row-insertion-value)

     |query-expression-body }
   [WITH ROLLBACK]
 query-expression-body:: = {query-specification
             |(query-expression-body)
             |query-expression-body {UNION|EXCEPT}[ALL]
              {query-specification|(query-expression-body)}}
               query-specification :: = SELECT
                  [{ALL|DISTINCT}]{selection-expression
                          [, selection-expression]...|*}
                             table-expression
 table-expression:: = FROM table-reference[, table-reference]...
              [WHERE search-condition]
              [GROUP BY value-expression[, value-expression]...]
              [HAVING search-condition]
```

Operands

For details about operands other than ROW and VALUES, see *INSERT statement Format 1 (Insert row)*.

- (ROW)

Specifies that data is to be inserted on a row-by-row basis. The following rules apply to the specifying of ROW:

1. The ROW operand can be specified only for a base table with the FIX attribute. The

operand ROW refers to an entire row. When ROW is specified, the system treats the entire row as a set of data and inserts data from one area. The data type of the data that is inserted should be the ROW type, regardless of the data types of the individual columns. (Variables corresponding to CHAR(*n*) (where *n* is the row length) or structures of the same length can be specified as ROW-type data; if a structure is specified, the structure should not contain any boundary alignment blanks.) The data length should be equal to the row length (the sum of the data lengths of the columns).

2. The platform on which the UAP runs and the platform on which the HiRDB server runs should have the same endian. The ROW option cannot be used between different endians. For example, if ROW is used in a Windows edition UAP, the HiRDB server should also use the Windows edition of the same endian.

■ {VALUES (*row-insertion-values*)|*query-specification*}

row-insertion-values

Specifies the values to be inserted into the row corresponding to ROW. Any of the following items can be specified:

- Embedded variables (and indicator variables)
- ? parameters
- SQL variables or SQL parameters

■ *query-expression-body* : : = {*query-specification*

| (*query-expression-body*)

| *query-expression-body* {UNION | EXCEPT} [ALL]

{*query-specification* | (*query-expression-body*) } }

Specifies the query expression body that fetches the data to be inserted. For details about query expression bodies, see *2.2 Query expressions*.

■ *query-specification* : : = SELECT [{ ALL | DISTINCT}]

{*selection-expression* [, *selection-expression*]... | * } *table-expression*

For details about query specifications, see *2.3 Query specification*.

■ *table-expression* : : = FROM *table-reference* [, *table-reference*]...

[WHERE *search-condition*]

[GROUP BY *value-expression* [, *value-expression*]...]

[HAVING *search-condition*]

For table expressions, see *2.5 Table expressions*. For table referencing, see *2.6 Table reference*. For search conditions, see *2.7 Search conditions*.

Notes

1. If the data type of a table column to be inserted is DECIMAL, or a national character string, HiRDB checks the contents of the applicable row insertion values.
2. For retrieving or updating rows (with a ROW specification), the portions that are of the date data type with respect to ROW in an embedded variable, SQL variable, or SQL parameter have a length of 4 bytes, and must be in *X'YYYYMMDD'* format.

   When date data is received in a predefined character string representation by using a by-row (ROW specification) interface, during the definition of a column, define the column as CHAR(10) rather than a date data type. Any date arithmetic operations should be specified using the DATE scalar function after the data involved has been converted into the date data type.
3. For retrieving or updating rows (with a ROW specification), the portions that are of the time data type with respect to ROW in an embedded variable, SQL variable, or SQL parameter have a length of 3 bytes, and must be in *X'hhmmss'* format.

   When time data is received in a predefined character string representation by using a by-row (ROW specification) interface, during the definition of a column, define the column as CHAR(8) rather than a time data type. Any time arithmetic operations should be specified using the TIME scalar function after the time involved has been converted into the time data type.
4. For performing a retrieval or updating by row (ROW specification), the time stamp data type portion of the embedded variable, SQL variable, or SQL parameter with respect to ROW is (7 + *p*/2) bytes long, and it should be in the X'*YYYYMMDDhhmmss*[*nn*...*n*]' format. When time stamp data is received in a predefined character string representation using a by-row (ROW specification), when defining a column, define it as CHAR with a length of 19, 22, 24, or 26 bytes rather than as a time stamp data type column.
5. If SYSTEM GENERATED is specified for the column in the table into which data is to be inserted, HiRDB ignores any data in the corresponding portion, and inserts the current date (CURRENT_DATE) for the DATE type, and the current time (CURRENT_TIME) for the TIME type.

Common rules

1. For common rules on specifying the SELECT statement, see the common rules in *INSERT statement Format 1 (Insert row)*.
2. If the INSERT statement is executed on a table with a WITHOUT ROLLBACK specification, the timing for the release of row locking can vary depending on whether an index is defined. For details, see the rules on WITHOUT ROLLBACK in *CREATE TABLE (Define table)* in Chpater *3*.

3. Before inserting data into a shared table, the LOCK statement with respect to the table should be executed in the lock mode. An attempt to insert data into a shared table without executing the LOCK statement can cause an error. For details about updating a shared table, see the *HiRDB Version 9 Installation and Design Guide*. For the objects of locking during execution of the LOCK statement on a shared table, see the notes in *LOCK statement (Lock control on tables)* in Chapter *5*.
4. If the table into which a row is to be insert has a column of character data that does not use the default character set, data cannot be inserted by row.

Rules on referential constraints

1. For rules on referential constraints in specifying the SELECT statement, see the rules on referential constraints in *INSERT statement Format 1 (Insert row)*.
2. For the insertion of rows into a referenced table, the referencing table is referenced to determine whether the value of the foreign key constituent column is included in the value of the primary key constituent column in the referenced table. The data guarantee level during the search through the referenced table, assumes the share mode. For this reason, if during the insertion of rows into the referencing table, operations are performed on the referenced table by another transaction, the row insertion operation goes into a wait state until the transaction is settled.

Example

Insert into a stock table named STOCK one row of values that are read into the embedded variable XROW:

```
INSERT INTO STOCK(ROW)
    VALUES(:XROW)
```

INSERT statement Format 3, Format 4 (Insert row using an array)

Function

Can insert rows when an embedded variable of the array format is specified.

Format 3

Inserts rows into a table by column.

Format 4

Treats an entire row as a single data item and inserts rows into a table of the FIX attribute row by row.

Privileges

A user who has the INSERT privilege for a table can insert rows into that table.

However, if a query is specified in the INSERT statement, the user needs the SELECT privilege for the table for which the query is specified.

Format 3: Inserting multiple rows by column into a table by specifying an embedded variable array

```
FOR : embedded-variable
INSERT INTO [authorization-identifier.] table-identifier
    [(column-name [, column-name]...)]
      {VALUES (insertion-value[, insertion-value]...)
      | query-expression-body}
    [WITH ROLLBACK]

query-expression-body:: = {query-specification
            | (query-expression-body)
            | query-expression-body {UNION|EXCEPT} [ALL]
              {query-specification | (query-expression-body)}}
query-specification :: = SELECT [{ALL|DISTINCT}] {selection-expression[, selection-expression]...
| * } table-expression
table-expression      :: = FROM table-reference[, table-reference]...
             [WHERE search-condition]
             [GROUP BY value-expression[, value-expression]...]
             [HAVING search-condition]
```

Format 4: Inserting multiple rows by row into a table with the FIX attribute by specifying an embedded variable array

```
FOR : embedded-variable
INSERT INTO [authorization-identifier.]table-identifier (ROW)
  {VALUES (:embedded-variable-array [:indicator-variable-array])
  | query-expression-body }
  [WITH ROLLBACK]

query-expression-body::= {query-specification
           | (query-expression-body)
           | query-expression-body {UNION|EXCEPT} [ALL]
            {query-specification | (query-expression-body)}}
query-specification ::= SELECT [{ALL|DISTINCT}] {selection-expression[, selection-expression]...
| * } table-expression
selection-expression     ::= FROM table-reference[, table-reference]...
            [WHERE search-condition]
            [GROUP BY value-expression [, value-expression]...]
            [HAVING search-condition]
```

Operands

For details about operands other than FOR, VALUES, or ROW, see *INSERT statement Format 1 (Insert row)*.

Format 3 operands

- VALUES (*insertion-value* [, *insertion-value*]...)

insertion-value

Inserts corresponding insertion values into the columns specified in *column-name*. The following items can be specified in this operand:

- : *embedded-variable-array* [: *indicator-variable-array*]
- Value expression
- NULL (represents the null value)
- DEFAULT

However, embedded variables not in the array format cannot be specified in *insertion-value*.

Format 4 operands

- (ROW)

Specify this operand when inserting data by row. Specification of the ROW operand is subject to the following rules:

1. The ROW operand can be specified only for base tables with the FIX attribute. ROW

refers to an entire row. Specifying ROW causes HiRDB to treat the entire row as a single data item and insert it from one area. The data type of the data to be inserted should be the ROW type, irrespective of the data types of columns (for the ROW type, a variable corresponding to CHAR (*n*) [where *n* is the row length] or a structure of the same length can be specified, provided that the structure does not contain a boundary alignment gap). The data length should be equal to the row length (the sum of the data lengths of the columns).

2. The platform on which the UAP runs and the platform on which the HiRDB server runs should have the same endian. The ROW option cannot be used between different endians. For example, if ROW is used in a Windows edition UAP, the HiRDB server should also use the Windows edition of the same endian.

Operands common to Formats 3 and 4

- FOR:*embedded-variable*

Specifies the embedded variable storing the number of rows to be inserted using an embedded variable array. An embedded variable of the SMALLINT type should be specified. The allowable range of values is from 1 to 4,096, less than or equal to the number of elements in the embedded variable array or in the indicator variable array. Only non-zero, positive values are allowed. An out of range value can cause a run-time error.

embedded-variable-array

Specifies the embedded variable that was declared in the array format. Specify an array variable for inserting non-null values.

The values to be inserted into the rows should be assigned to the elements of the embedded variable array. If the values to be inserted include the NULL value, specify both an embedded variable array and an indicator variable array.

indicator-variable-array

Specifies the indicator variable that was declared in the array format. Values indicating that the values to be inserted into the elements of the embedded variable array are not the NULL value should be assigned to the corresponding elements in the indicator variable array. For allowable values, see *1.6.5 Setting a value for an indicator variable*. Notice that the NULL value cannot be inserted into a base table with the FIX attribute.

Common rules

Common rules on Format 3

1. The data type of the embedded variable array should be either the data type of the corresponding column or a convertible data type.
2. The number of columns per row to be inserted should be equal to the number of columns specified in *column-name*. Their values should have the data type of the

columns or a convertible data type.

3. The following data types are not allowed in an INSERT statement using an array: BLOB type, BINARY type with a maximum length of 32,001 bytes or greater, and the abstract data type.
4. In an INSERT statement using an array, data cannot be inserted into a repetition column.
5. When fixed-point or floating-point data is inserted into a column of any of the following data types, the fractional part (below the decimal point) is truncated:
   - INTEGER
   - SMALLINT

   Also, when fixed-point data is inserted into a DECIMAL type column, any digits below the scaling for the column are truncated.
6. Character data longer than the length that was specified when the table was defined cannot be inserted into the table.
7. Numerical data outside the range of the data type defined for the column cannot be inserted into the column.
8. If the data inserted into a column of a fixed-length character string (including national character strings and mixed character strings) is shorter than the column length, the data is inserted left-justified, and the remainder of the column is blank-filled.
9. To insert data into a character data type column, make sure the character set is the same for the column into which the data is being inserted and the value being inserted. However, if the value being inserted is an embedded variable array (that uses the default character set), it is automatically converted to the character set of the column into which it is being inserted.

Common rules on Format 4

1. If the table into which data is being inserted has a column of character data that does not use the default character set, data cannot be updated by row.

Common rules for Format 3 and Format 4

1. One or more embedded variable arrays should be specified in a clause other than the FOR clause. A failure to specify an embedded variable array can cause an error.
2. Specifying an embedded variable not in the array format in a clause other than the FOR clause can cause an error.
3. The number of elements in the embedded variable array or indicator variable array should be in the 1 to 4,096 range. An out of range value can cause an error. The number of elements should be greater than or equal to the maximum number

of rows specified in FOR:*embedded-variable*.

4. Because it includes embedded variable arrays and indicator variable arrays, the INSERT statement Format 3 cannot be preprocessed by the PREPARE statement. For dynamic execution of the INSERT statement Format 3, see *EXECUTE statement Format 2 (Execute an SQL statement using an array)*.
5. An INSERT statement using an array cannot be specified in a procedure.
6. If a warning-generating event occurs in any of the rows to be inserted, warning information is assigned to the SQLWARN flag of the SQL communications area.
7. If an error occurs in any of the rows to be inserted, the transaction is rolled back.
8. If the INSERT statement is executed on a table with a WITHOUT ROLLBACK specification, the timing for the release of row locking can vary depending on whether an index is defined. For details, see the rules on WITHOUT ROLLBACK in *CREATE TABLE (Define table)* in Chapter *3*.
9. Before inserting data into a shared table, the LOCK statement with respect to the table should be executed in the lock mode. An attempt to insert data into a shared table without executing the LOCK statement can cause an error. For details about updating a shared table, see the *HiRDB Version 9 Installation and Design Guide*. For the objects of locking during execution of the LOCK statement on a shared table, see the notes in *LOCK statement (Lock control on tables)* in Chapter *5*.

Rules on referential constraints

1. For rules on inserting referential constraints by embedded variable array or by row, see the rules on referential constraints in *INSERT statement Format 1 (Insert row)*.
2. For the insertion of rows into a referenced table, the referencing table is referenced to determine whether the value of the foreign key constituent column is included in the value of the primary key constituent column in the referenced table. The data guarantee level during the search through the referenced table assumes the share mode. For this reason, if during the insertion of rows into the referencing table, operations are performed on the referenced table by another transaction, the row insertion operation goes into a wait state until the transaction is settled.

Examples

1. Batch-insert data equivalent to 50 rows, assigned to an array variable in the C language, into an inventory table:

```
EXEC SQL BEGIN DECLARE SECTION;
  short   XINSERT_NUM;
  long    XPCODE[50];
  short   IPCODE[50];
```

```
  char    XPNAME[50][17];
  short   IPNAME[50];
EXEC SQL END DECLARE SECTION;
              :
Assign values to the elements of the variable array
              :
XINSERT_NUM = 50;
EXEC SQL FOR :XINSERT_NUM
    INSERT INTO STOCK(PCODE,PNAME)
      VALUES (:XPCODE:IPCODE,:XPNAME:IPNAME);
```

2. Insert the value of the entire row assigned to an array variable in the C language into the inventory table (STOCK) in a single operation covering 50 rows:

```
EXEC SQL BEGIN DECLARE SECTION;
  short   XINSERT_NUM;
  char    XROWS[50][31];
EXEC  SQL  END  DECLARE  SECTION;
              :
Assign values to the elements of the variable array
              :
XINSERT_NUM = 50;
EXEC  SQL  FOR :XINSERT_NUM
    INSERT INTO STOCK(ROW) VALUES(:XROWS);
```

OPEN statement Format 1 (Open cursor)

Function

Opens a cursor. The OPEN statement opens a cursor either declared in the DECLARE CURSOR statement or allocated by the ALLOCATE CURSOR statement, and positions the cursor before the first row of retrieval results so that the retrieval results can be fetched.

In Format 1, the OPEN instruction opens a cursor by using an embedded variable to assign a value to the ? parameter.

Privileges

Users with the SELECT privilege

To open a cursor, the user must have the SELECT privilege for all tables that are specified in the cursor declaration or in the ALLOCATE CURSOR statement.

Format 1: Opening a cursor (by assigning values to ? parameters using embedded variables)

```
OPEN {cursor-name | extended-cursor-name} [USING
:embedded-variable[,:embedded-variable] ...]
```

Operands

- *{cursor-name | extended-cursor-name}*

cursor-name

Specifies the name of the cursor to be opened.

extended-cursor-name

Specifies the extended cursor name that identifies the cursor to be opened.

For extended cursor names, see *2.27 Extended cursor name*.

- USING :*embedded-variable*[,:*embedded-variable*]

Specifies new embedded variables when the embedded variables specified in the SELECT statement of the DECLARE CURSOR statement are to be changed.

When values are assigned to the ? parameters specified in the SELECT statement preprocessed by the PREPARE statement, the embedded variables to which the values are assigned should be specified.

The values of the embedded variables specified in the SELECT statement of the DECLARE CURSOR statement or the values of the ? parameters remain in effect as SQL-runtime values until the cursor is closed. These values can be modified by closing the cursor and then reopening it.

The embedded variables specified in the USING clause replace in the order in which they are specified the embedded variables specified in the SELECT statement in the cursor declaration.

The embedded variables specified in the USING clause assign values to the ? parameters specified in the prepared SELECT statement in the order in which they are specified.

Notes

1. To reopen a cursor that is already open, it must be closed and then reopened.
2. For details about the trigger for closing a cursor by issuing a CLOSE statement internally, see *Common rule 1* in *CLOSE statement (Close cursor)* in this chapter.
3. When the FETCH statement is to be executed, the OPEN statement must be used to open a cursor and then the FETCH statement relative to that cursor must be executed.
4. When embedded variables are specified in the USING clause, values must have been assigned to them before the OPEN statement is executed.
5. If the embedded variable specified in the USING clause contains character data that uses the default character set, and that is different from the character set used for character data type items associated with the ? parameter or embedded variable specified in the SELECT statement, it is automatically converted to the character set used by the ? parameter or embedded variable specified in the SELECT statement.
6. When the OPEN statement is executed, HiRDB positions the cursor and performs lock control during lock of the associated FETCH statement.
7. A cursor name, similar to an embedded variable name, is effective only within a compile-unit module. Multiple SQLs relative to the same cursor cannot be used in multiple modules.
8. The USING clause cannot be specified in a procedure.
9. Multiple holdable cursors cannot be opened for a single table.

Example

Open cursor CR1 in order to fetch retrieval results from a stock table named STOCK:

```
OPEN CR1
```

OPEN statement Format 2 (Open cursor)

Function

Opens a cursor. The OPEN statement opens a cursor either declared in the DECLARE CURSOR statement or allocated by the ALLOCATE CURSOR statement, and positions the cursor before the first row of retrieval results so that the retrieval results can be fetched.

In Format 2, the OPEN instruction opens a cursor by using the SQL descriptor area to assign a value to the ? parameter.

Privileges

Users with the SELECT privilege

To open a cursor, the user must have the SELECT privilege for all tables that are included in the SELECT statement specified in the cursor declaration or in the ALLOCATE CURSOR statement.

Format 2: Opening a cursor (by assigning values to ? parameters using an SQL descriptor area)

```
OPEN cursor-name USING DESCRIPTOR [:]SQL-descriptor-area-name
                 [CHARACTER SET[:]character-set-descriptor-area-name]
```

Operands

- {*cursor-name* | *extended-cursor-name*}

cursor-name

Specifies the name of the cursor to be opened.

extended-cursor-name

Specifies the extended cursor name that identifies the cursor to be opened.

For extended cursor names, see *2.27 Extended cursor name*.

- [:]*SQL-descriptor-area-name*

SQL-descriptor-area-name

When using the SQL descriptor area to assign a value to a ? parameter specified in a SELECT statement preprocessed by the PREPARE statement, specifies the name of the SQL descriptor area that specifies the value to be assigned to the input ? parameter.

The variables specified in the SQLVAR array of the SQL descriptor area assign values to the ? parameters that are specified in the prepared SELECT statement in the order in which they are specified.

- [CHARACTER SET[:]*character-set-descriptor-area-name*]

character-set-descriptor-area-name

When using the SQL descriptor area to assign the value of a ? parameter specified in a SELECT statement preprocessed by the PREPARE statement, specifies the name of the character set descriptor area that specifies the character set name to be used for the value being assigned to the input ? parameter.

Notes

1. To reopen a cursor that is already open, it must be closed and then reopened.
2. For details about the trigger for closing a cursor by issuing a CLOSE statement internally, see *Common rule 1* in *CLOSE statement (Close cursor)* in this chapter.
3. When the FETCH statement is to be executed, the OPEN statement must be used to open a cursor and then the FETCH statement relative to that cursor must be executed.
4. Before executing the OPEN statement, the UAP must set the required information in the SQL descriptor area and the character set descriptor area. For SQL descriptor areas and character set descriptor areas, see the *HiRDB Version 9 UAP Development Guide*.
5. A cursor name, similar to an embedded variable name, is effective only within a compile-unit module. Multiple SQLs relative to the same cursor cannot be used in multiple modules.
6. Multiple holdable cursors cannot be opened for a single table.

Example

Open cursor CR2 in order to fetch retrieval results from a stock table; also specify information (SQLDA) for assigning values to the ? parameters specified in the SELECT statement preprocessed by the PREPARE statement:

```
OPEN CR2
  USING DESCRIPTOR SQLDA
```

PREPARE statement (Preprocess SQL)

Function

The PREPARE statement preprocesses so that the SQL statement given in *character-string* can be executed, and assigns either an SQL statement identifier or an extended statement name to the SQL statement. In addition, by specifying either the OUTPUT or INPUT operand, the user can also fetch the retrieval information or input/output information obtained by the DESCRIBE [OUTPUT] statement or the DESCRIBE INPUT statement.

Privileges

None.

Format

```
PREPARE {SQL-statement-identifier | extended-statement-name}
        FROM {'character-string'|:embedded-variable}
             [WITH {SQLNAME|[ALL] TYPE}
             [, {SQLNAME|[ALL] TYPE}] OPTION]
             [OUTPUT[:]SQL-descriptor-area-name
                    [[:]column-name-descriptor-area-name]
                TYPE[:]type-name-descriptor-area-name]]
                [CHARACTER_SET [:]character-set-descriptor-area-name]]
              [INPUT[:]SQL-descriptor-area-name
                    [[:]column-name-descriptor-area-name]
                [CHARACTER_SET [:]character-set-descriptor-area-name]]
```

Operands

- {*SQL-statement-identifier* | *extended-statement-name*}

SQL-statement-identifier

Specifies the name assigned to the SQL statement in order to identify the SQL that is to be prepared.

For details about SQL statement identifiers, see *1.1.7 Specification of names*.

HiRDB reserved words (other than SELECT and WITH) can be used as SQL statement identifiers. Even if an SQL statement identifier identical to a reserved word is used, it must not be enclosed in double quotation marks.

extended-statement-name

When a cursor is to be allocated by the ALLOCATE CURSOR statement, this operand specifies the extended statement name that was assigned to the SQL statement to identify the SQL statement to be preprocessed.

For extended statement names, see *2.26 Extended statement name*.

- {'*character-string*'|:*embedded-variable*}

character-string

Specifies directly as a character literal the character string representing the SQL to be preprocessed.

The SQL prefix or the SQL terminator cannot be specified in the character string to be preprocessed.

An apostrophe in the character literal specification of an SQL to be preprocessed must be specified as two apostrophes in succession.

The maximum length of an SQL to be preprocessed is 2,000,000 bytes. If an SQL of the embedded type is specified directly as a character literal, the maximum length is the maximum length of character literals in the host language.

embedded-variable

Specifies an embedded variable of the variable-length character type.

Character sets other than the default character set are prohibited.

- [WITH {SQLNAME|[ALL] TYPE} [, {SQLNAME|[ALL] TYPE}] OPTION]

SQLNAME

Specifies that column information on retrieval items and attribute names of a user-defined data type are to be received by specifying a Column Name Descriptor Area in a DESCRIBE or DESCRIBE TYPE statement. If a Column Name Descriptor Area name is specified in either the OUTPUT or INPUT clause, the SQLNAME operand can be omitted.

[ALL] TYPE

Specifies that type name information on retrieval items is to be received by specifying a Type Name Descriptor Area in a DESCRIBE statement.

The TYPE option can be omitted if a Type Name Descriptor Area name is specified in the OUTPUT clause. The ALL TYPE option should be specified if definition information of a user-defined type is to be received by the DESCRIBE TYPE statement. ALL TYPE cannot be omitted even when a Type Name Descriptor Area name is specified in the OUTPUT clause.

- [OUTPUT [:] *SQL-descriptor-area-name* [[:] *column-name-descriptor-area-name*]

  [TYPE [:] *type-name-descriptor-area-name*]]

  [CHARACTER_SET [:]*character-set-descriptor-area-name*]]

SQL-descriptor-area-name

Specifies the name of the SQL descriptor area that receives SQL retrieval item information (if the preprocessed SQL statement is a SELECT statement) or output ? parameter information (if the preprocessed SQL statement is a CALL statement).

For SQL descriptor areas, see the *HiRDB Version 9 UAP Development Guide*.

column-name-descriptor-area-name

Specifies either the name of a retrieval item or the Column Name Descriptor Area that receives a routine parameter name.

For Column Name Descriptor Areas, see the *HiRDB Version 9 UAP Development Guide*.

type-name-descriptor-area-name

Specifies the Type Name Descriptor Area that receives the user-defined type name for a retrieval item.

For Type Name Descriptor Areas, see the *HiRDB Version 9 UAP Development Guide*.

character-set-descriptor-area-name

Specifies the character set descriptor area into which the character set name for the retrieval item information (if the preprocessed SQL is a SELECT statement), or output ? parameter information (if the preprocessed SQL is a CALL statement) is to be stored.

For details about character set descriptor areas, see the *HiRDB Version 9 UAP Development Guide*.

- [INPUT [:] *SQL-descriptor-area-name* [[:] *column-name-descriptor-area-name*]]

  [CHARACTER_SET [:] *character-set-descriptor-area-name*]]

SQL-descriptor-area-name

Specifies the name of the SQL descriptor area that receives input ? parameter information.

For details about the SQL descriptor area, see the *HiRDB Version 9 UAP Development Guide*.

column-name-descriptor-area-name

Specifies the Column Name Descriptor Area that receives either the name of a retrieval item or the parameter name of a routine.

For details about Column Name Descriptor Areas, see the *HiRDB Version 9 UAP Development Guide*.

character-set-descriptor-area-name

Specifies the character set descriptor area into which the character set name for the input ? parameter information is to be stored.

For details about character set descriptor areas, see the *HiRDB Version 9 UAP Development Guide*.

Common rules

1. The type of an embedded variable is the following structure:

```
struct {
   long  xxxxxxx;       /* Effective length of SQL statement */
   char  yyyyyyy[n];    /* SQL statement storage area */
} zzzzzzz;
```

   The characters xxxxxxx indicate the effective length of the character string stored in character array yyyyyyy.

   $1 \leq$ (value of xxxxxxx) ≤ 2000000

   The effective length of a character string does not include the character 0 (zero) that indicates the end of the character string.

   n is any value.

2. SQLNAME cannot be specified more than once. Similarly, [ALL] TYPE cannot be specified more than once.
3. Before executing the PREPARE statement, the UAP should assign the number of SQLVAR flags (the SQLN area) in the SQL descriptor area.
4. Because SQLDATA and SQLIND are cleared when the DESCRIBE statement is executed, or when a PREPARE statement is executed in which INPUT or OUTPUT is specified, if you use the DESCRIBE statement, or if you use a PREPARE statement in which INPUT or OUTPUT is specified, a value must be assigned to SQLDATA or SQLIND after those statements are executed.
5. Specify a Column Name Descriptor Area only when the name of a retrieval item or the parameter name of a routine is to be received. Note that the parameter name of a routine can be received only when the ? parameter is specified by itself in an argument of the CALL statement. If a value expression including the ? parameter is specified, the length of the name of the Column Name Descriptor Area is 0.
6. Specify a Type Name Descriptor Area name only when the user-defined type name of the retrieval result is to be received.
7. Specify a character set descriptor area name only if the character set name is to be retrieved.

Notes

1. The results of preprocessing produced in a transaction remain in effect only

within that transaction. Therefore, any DESCRIBE, EXECUTE, OPEN, FETCH, or CLOSE statement that references the prepared SQL must be executed in the same transaction. However, if the preprocessed SQL is a holdable cursor, the following takes place:

- When the SQL is preprocessed in the transaction and committed:

  The preprocessing result remains valid until a DISCONNECT statement is executed.

- When the SQL is preprocessed in the transaction and rolled back:

  The preprocessing result is valid only within the transaction.

2. An SQL to be preprocessed by the PREPARE statement must be preprocessed in advance.

   The following SQLs can be preprocessed by the PREPARE statement:

   - Data manipulation SQLs:

     ASSIGN LIST statement (executed with EXECUTE statement)

     CALL statement (executed with EXECUTE statement)

     DELETE statement (executed with EXECUTE statement)

     Preparable dynamic DELETE statement: locating (executed by the EXECUTE statement)

     DROP LIST statement (executed with EXECUTE statement)

     INSERT statement (executed with EXECUTE statement)

     PURGE TABLE statement (executed with EXECUTE statement)

     SELECT statement (executed with OPEN, FETCH, and CLOSE statements)

     Single-row SELECT statement (executed by the EXECUTE statement)

     Dynamic SELECT statement (executed by the OPEN, FETCH, or CLOSE statement)

     UPDATE statement (executed with EXECUTE statement)

     Preparable dynamic UPDATE statement: locating (executed by the EXECUTE statement)

     Assignment statement (executed by the EXECUTE statement)

   - Control SQLs:

     LOCK TABLE statement (executed by EXECUTE statement)

     SET SESSION AUTHORIZATION statement (executed by the EXECUTE statement)

- Definition SQLs:

  ALTER INDEX, ALTER PROCEDURE, ALTER ROUTINE, ALTER TABLE, ALTER TRIGGER, COMMENT, CREATE AUDIT, CREATE CONNECTION SECURITY, CREATE FUNCTION, CREATE INDEX, CREATE PROCEDURE, CREATE SCHEMA, CREATE SEQUENCE, CREATE TABLE, CREATE TRIGGER, CREATE TYPE, CREATE VIEW, DROP AUDIT, DROP CONNECTION SECURITY, DROP DATA TYPE, DROP FUNCTION, DROP INDEX, DROP PROCEDURE, DROP SCHEMA, DROP SEQUENCE, DROP TABLE, DROP TRIGGER, DROP VIEW, GRANT, and REVOKE statements (all definition SQLs executed by the EXECUTE statement)

3. An SQL statement identifier, similar to an embedded variable name, is effective only within a compile-unit module. Multiple SQLs relative to the same SQL statement identifier cannot be used in multiple modules.
4. When the dynamic SELECT statement preprocessed by the PREPARE statement is being executed (after the OPEN statement has been executed and before the CLOSE statement is executed), any table specified in the FROM clause of the dynamic SELECT statement should not be updated by another SQL statement.
5. If the SQL statement identifier or extended statement name that is specified already identifies another SQL statement, the DEALLOCATE PREPARE statement is implicitly executed, and the previously identified SQL statement is nullified. After that, the specified SQL statement identifier or the extended statement name identify the SQL statement that was preprocessed by the PREPARE statement. However, if an error occurs in the implicitly executed DEALLOCATE PREPARE statement, the previously identified SQL statement remains unchanged.
6. If OUTPUT is specified in the PREPARE statement, the PREPARE statement is treated in the same way as the execution of the DESCRIBE [OUTPUT] statement. Similarly, if INPUT is specified in the PREPARE statement, the PREPARE statement is treated in the same way as the execution of the DESCRIBE [INPUT] statement. For details about the OUTPUT and INPUT options, see *DESCRIBE*.

Examples

1. Preprocess the SQL 'SELECT * FROM STOCK' provided in a character string for execution; assume that the SQL identifier assigned to the preprocessed SQL is named 'PRESQL':

```
PREPARE PRESQL FROM
  'SELECT * FROM STOCK'
```

2. Prepare the SQL character string specified in an embedded variable named XSQL; assume that the SQL identifier assigned to the preprocessed SQL is named 'PRESQL':

```
PREPARE PRESQL FROM :XSQL
```

PURGE TABLE statement (Delete all rows)

Function

The PURGE TABLE statement deletes all rows in a specified base table.

Privileges

A user who has the DELETE privilege for a table can delete rows from that table.

Format

```
PURGE TABLE[authorization-identifier.]table-identifier
            [{WITH ROLLBACK|NO WAIT}]
```

Operands

- [*authorization-identifier*.]*table-identifier*

authorization-identifier

Specifies the authorization identifier of the user who owns the table.

MASTER cannot be specified as an authorization identifier. For an explanation of omitting the authorization identifier, see *1.1.8 Qualifying a name*.

table-identifier

Specifies the name of the base table from which all rows are to be deleted.

- [{WITH ROLLBACK|NO WAIT}]

If neither the WITH ROLLBACK option nor the NO WAIT option is specified and the table from which all rows are to be deleted is being used by a transaction issued by another user, the system executes the PURGE TABLE statement after that transaction has terminated.

When USE is specified for the pd_check_pending operand in the system definition and a referencing table that is referencing a table from which all rows are to be deleted is being used by another user transaction, the referencing table is set to check pending status once the transaction terminates.

WITH ROLLBACK

Specifies that if the table from which all rows are to be deleted is being used by another user, the system is to cancel and invalidate the current transaction.

When USE is specified for the pd_check_pending operand in the system definition and a referencing table that is referencing a table from which all rows are to be deleted is being used by another user transaction, the transaction is cancelled and invalidated.

NO WAIT

Specifies that if the table from which all rows are to be deleted is being used by another user, HiRDB is to invalidate the current SQL without canceling the transaction.

When USE is specified for the pd_check_pending operand in the system definition and a referencing table that is referencing a table from which all rows are to be deleted is being used by another user transaction, the SQL statement is invalidated without canceling the transaction.

Common rules

1. When executed normally, the PURGE TABLE statement is committed as soon as its processing has been completed.
2. If the user LOB RDAREA that stores a LOB column or LOB attribute is in the frozen update status, the PURGE TABLE statement cannot be executed on a table containing the LOB column or the LOB attribute (an attempt to execute this statement causes an already frozen error).
3. The PURGE TABLE statement cannot be executed on falsification prevented tables.
4. If the PURGE TABLE statement is executed on a shared table, a locking equivalent to the LOCK statement with an EXCLUSIVE specification is applied to the shared table. For HiRDB/Parallel Server, the locking is applied on all back-end servers.

Rules related to the check pending status

1. When USE is specified for the pd_check_pending operand in the system definition and the operation-target table is a referenced table, the referencing table that references the operation-target table is set to check pending status.
2. If the operation-target table is in check pending status, the status is released. However, if either of the conditions listed below is satisfied, the check pending status in the dictionary tables is not released. In such a case, use the integrity check utility to release the check pending status in the dictionary tables.
   - NOUSE is specified for the pd_check_pending operand in the system definition.
   - The inner replica facility is being used.

Notes

1. The PURGE TABLE statement cannot be specified for a view table.
2. The PURGE TABLE statement cannot be specified from an X/Open-compliant UAP running under OLTP. When calling a procedure from a UAP running under OLTP, you cannot execute procedures using the PURGE TABLE statement.
3. The PURGE TABLE statement cannot be executed during trigger action.

4. If USE is specified for the pd_check_pending operand in the system definition, or the specification is omitted, no commands or utilities can be executed at the same time as the PURGE TABLE statement. For details see the description of the pd_check_pending operand in the manual *HiRDB Version 9 System Definition*.

Example

Delete all rows from stock table STOCK:

```
PURGE TABLE STOCK
```

Single-row SELECT statement (Retrieve one row)

Function

The single-row SELECT statement retrieves table data by fetching only one row of data without using a cursor.

Although the single-row SELECT statement has the same operands as the SELECT clause with a query specification, unlike the SELECT clause as a statement in a query specification it does not operate on sets.

The single-row SELECT statement includes the INTO clause that specifies the area for receiving the retrieved results.

Privileges

Same as the SELECT clause as a statement in a query specification

Format: Fetching up to one row of data into specified embedded variables

```
SELECT [{ALL|DISTINCT}] {selection-expression
                                [, selection-expression]...|*}
  [INTO {:embedded-variable[:indicator-variable]
        |[statement-label.]SQL-variable-name
        |[[authorization-identifier.]routine-identifier.]
            SQL-parameter-name
        |[statement-label.]SQL-variable-name..attribute-name
                [..attribute-name]...
        |[[authorization-identifier.]routine-identifier.]
            SQL-parameter-name
                ..attribute-name[..attribute-name]...}
      [, {:embedded-variable[:indicator-variable]
         |[statement-label.]SQL-variable-name
         |[[authorization-identifier.]routine-identifier.]
             SQL-parameter-name
         |[statement-label.]SQL-variable-name..attribute-name
                 [..attribute-name]...
         |[[authorization-identifier.]routine-identifier.]
             SQL-parameter-name
                 ..attribute-name[..attribute-name]...} ]...]
  (Table-Expression)
   FROM table-reference [, table-reference]...
       [WHERE search-condition]
       [GROUP BY value-expression [, value-expression]...]
       [HAVING search-condition]
  (Lock-option)
```

```
[[{WITH {SHARE|EXCLUSIVE}LOCK
   |WITHOUT LOCK [{WAIT|NOWAIT}]}]]
  [{WITH ROLLBCK|NO WAIT}]
    [FOR UPDATE [OF column-name[, column-name]...][NOWAIT]]
```

Operands

For the SELECT clause, see *2.3 Query specification*; for table expressions, see *2.5 Table expressions*; for the lock option, see *2.19 Lock option*.

- INTO clause

The INTO clause must always be specified when the SELECT statement is coded, either by itself or directly in a UAP or procedure.

However, the INTO clause cannot be specified in any of the following locations:

- A SELECT statement in an SQL statement preprocessed by the PREPARE statement
- A SELECT statement in an SQL statement preprocessed/executed by the EXECUTE IMMEDIATE statement
- A SELECT statement in a cursor declaration

- {:*embedded-variable*[:*indicator-variable*]
  |[*statement-label* .]*SQL-variable-name*
  |[[*authorization-identifier* .]*routine-identifier* .]*SQL-parameter-name*
  |[*statement-label* .]*SQL-variable-name*..*attribute-name*[..*attribute-name*]...
  |[[*authorization-identifier* .]*routine-identifier* .]*SQL-parameter-name*
  ..*attribute-name*[..*attribute-name*]...}
  [, {:*embedded-variable*[:*indicator-variable*]
  |[*statement-label* .]*SQL-variable-name*
  |[[*authorization-identifier* .]*routine-identifier* .]*SQL-parameter-name*
  |[*statement-label* .]*SQL-variable-name*..*attribute-name*
  [..*attribute-name*]...
  |[[*authorization-identifier* .]*routine-identifier* .]*SQL-parameter-name*
  ..*attribute-name*[..*attribute-name*]...}]...

embedded-variable

Specifies an embedded variable into which a column value in the row is to be read.

indicator-variable

Specifies when the column value to be read into the embedded variable may be the null value.

[*statement-label*.]*SQL-variable-name*

[[*authorization-identifier*.]*routine-identifier*.]*SQL-parameter-name*

Specifies either an SQL variable or an SQL parameter that is to receive the value of a column in a procedure. For details about specification values in a Java procedure, see *JDBC drivers or SQLJ* in the *HiRDB Version 9 UAP Development Guide*.

If an authorization identifier is specified in the SQL procedure statement of a public procedure definition, specify upper-case `PUBLIC` enclosed in double quotation marks (`"`) as the authorization identifier.

[*statement-label*.]*SQL-variable-name*..*attribute-name*[..*attribute-name*]...

[[*authorization-identifier*.] *routine-identifier*.]*SQL-parameter-name* *attribute-name*[..*attribute-name*]...

These operands are specified to receive the values of attributes in a column.

If an authorization identifier is specified in the SQL procedure statement of a public procedure definition, specify upper-case `PUBLIC` enclosed in double quotation marks (`"`) as the authorization identifier.

- `[FOR UPDATE [OF` *column-name*`[,` *column-name*`]...][NOWAIT]]`

`FOR UPDATE[OF` *column-name*`[,` *column-name*`]...]` clause is called the `FOR UPDATE` clause.

`FOR UPDATE [OF` *column-name*`[,` *column-name*`]...]`

Must be specified in order to update, delete, or add a row using data retrieved from a single-line `SELECT` statement. Omit this operand for a table that is retrieved from a single-line `SELECT` statement if there are no rows to be updated, deleted, or added. If no lock option is specified in the SQL statement, the lock option is determined by the value specified in `PDISLLVL` or the value specified for the data guarantee level specified in SQL-compile-option. However, if `YES` is indicated for `PDFORUPDATEEXLOCK` or if `FOR UPDATE EXCLUSIVE` is specified after the data guarantee level in SQL-compile-option, the lock option for such a cursor is assumed to be `WITH EXCLUSIVE LOCK`. For details, see the *HiRDB Version 9 UAP Development Guide*.

`[OF` *column-name*`[,` *column-name*`]...]`

In general, this clause is optional.

Specifies columns that are not specified in the selection expression of a single-line

SELECT statement. A column can be specified only once.

When a column name is specified, instead of using the column name that was specified in AS *column-name*, specify the column of the table that was specified in the FROM clause of the outermost query specification.

[NOWAIT]

The behavior is the same as if the FOR UPDATE clause was specified and the WITH EXCLUSIVE LOCK NO WAIT lock option was set. However, if *Lock-option* is specified, NOWAIT cannot be specified.

For details about operations when the WITH EXCLUSIVE LOCK NO WAIT lock option is specified, see *2.19 Lock option*.

Common rules

1. When retrieval results are limited to no more than one row, the single-row SELECT statement can be used to retrieve the data without using a cursor but by specifying the INTO clause. If the retrieval results include more than one row, the single-row SELECT statement cannot be used.
   - The number of retrieval result columns and the number of embedded variables specified in the INTO clause must agree. If they do not, the W warning flag is set in SQLWARN3 in the SQL Communications Area.
   - The data type of each embedded variable specified in the INTO clause must be either the same as the data type of the corresponding column or a data type that can be converted into that data type.
   - If the embedded variable is character data type that uses the default character set, and the retrieval result is character data type that uses a different character set than the embedded variable, it is automatically converted to the character set used for the embedded variable.
   - If the data fetched into a fixed-length character string (including a national character string or a mixed character string) embedded variable is shorter than the length of the retrieval item, the data is left-justified in the embedded variable and trailing blanks are added.
   - If the value of a retrieval results column is the null, the value of the corresponding embedded variable is unpredictable.
   - If the value of a retrieval results column is the null, an indicator variable must be specified.
2. If there are no rows to be fetched, the system returns the following return codes:
   - Return code 100 to SQLCODE in the SQL Communications Area
   - Return code 100 to the SQLCODE variable

- Return code '02000' to the SQLSTATE variable

3. UNION [ALL], EXCEPT [ALL], the ORDER BY clause, and the LIMIT clause can be specified according to the rules given in *2.1.1 Cursor specification: Format 1* and *2.3 Query specification*.

   If UNION [ALL] and EXCEPT [ALL] are specified, the INTO clause should be specified only once following the first occurrence of the SELECT clause.

4. The FOR UPDATE clause cannot be specified if any of the following are specified in the single-line SELECT statement or the lock option:

   (a) UNION [ALL], or EXCEPT [ALL]

   (b) A table specified in the FROM clause of the outermost query specification is specified in the FROM clause of a subquery

   (c) Joined tables specified in the outermost query specification

   (d) A derived table in a FROM clause specified in the outermost query specification

   (e) SELECT DISTINCT specified in the outermost query specification

   (f) A GROUP BY clause specified in the outermost query specification

   (g) A HAVING clause specified in the outermost query specification

   (h) A set function on the outermost query specification

   (i) The window function on the outermost query specification

   (j) Any of the following view tables in the FROM clause of the outermost query specification:

   - A view table defined by specifying any of items (a) to (i) above in a view definition statement
   - View tables that are defined by specifying a value expression, other than a column specification in the SELECT clause of the outermost query specification, in a view definition statement
   - A view table defined with the READ ONLY option specified in the view definition statement

   (k) WITHOUT LOCK NOWAIT

Examples

See the section on *DECLARE CURSOR Format 1 (Declare cursor)* for examples.

Dynamic SELECT statement Format 1 (Retrieve dynamically)

Function

The dynamic SELECT statement is used for the following purposes:

- For retrieving data from one or more tables
- For executing the SELECT statement dynamically
- For preparing by the PREPARE statement (the DECLARE CURSOR statement is used to declare a cursor, and the cursor is used to fetch the retrieval results on a row-by-row basis)

Privileges

See *2.1.1 Cursor specification: Format 1*.

Format

```
(Cursor-Specification-Format-1)
 (Query-Expression)
  (Query-Specification)
   {SELECT [ {ALL|DISTINCT}] {selection-expression
               [, selection-expression] . . .
                  |*}
   (Table-Expression)
   FROM table-reference [, table-reference] . . .
        [WHERE search-condition]
        [GROUP BY value-expression [, value-expression] . . .]
        [HAVING search-condition]
   | query-expression }
   |derived-query-expression UNION ALL
{query-specification|(derived-query-expression)}]
[ORDER BY {column-specification|sort-item-specification-number}
                [{ASC|DESC}]
            [, {column-specification|sort-item-specification-number}
                [{ASC|DESC}]] . . .]
    [LIMIT {[offset, ]{row_count | ALL}
        | {row_count |ALL} [OFFSET offset]}]
(Lock-Option)
 [[{WITH {SHARE|EXCLUSIVE} LOCK
    |WITHOUT LOCK [{WAIT|NOWAIT} ] } ]
 [ {WITH ROLLBACK|NO WAIT} ]]
 [FOR {UPDATE [OF column-name [, column-name] . . .] [NOWAIT]|READ
ONLY} ]
 [UNTIL DISCONNECT]
```

Operands

For the following items, see the indicated sections in this manual:

- *Cursor-Specification-Format-1*: *2.1.1 Cursor specification: Format 1*
- *Query-Expression*: *2.2 Query expressions*
- *Query-specification*: *2.3 Query specification*
- *Table-Expression*: *2.5 Table expressions*
- *Search-condition*: *2.7 Search conditions*
- *Lock-Option*: *2.19 Lock option*

■ FOR{UPDATE[OF *column-name* [, *column-name*]...] [NOWAIT]|READ ONLY}

FOR UPDATE [OF *column-name* [, *column-name*]...] is referred to as a FOR UPDATE clause

FOR UPDATE

This operand is specified when a table is being searched using a cursor and when rows in the table are updated or deleted using the cursor, and, in addition, when rows in the table are to be updated, added, or deleted either using another cursor or by directly specifying a search condition.

When the FOR UPDATE clause is omitted, the current cursor cannot be used to update, add, or delete rows from the table being retrieved.

The FOR UPDATE clause cannot be specified if either the cursor specification or the lock option contains any of the following specifications:

1. UNION[ALL] or EXCEPT[ALL]
2. A table specified in the FROM clause of the outermost query specification in the FROM clause of a subquery
3. Joined tables in an outermost query specification
4. SELECT DISTINCT in an outermost query specification
5. A table derived from a FROM clause in the outermost query specification
6. The GROUP BY clause in an outermost query specification
7. The HAVING clause in an outermost query specification
8. A set function in an outermost query specification
9. The window function in an outermost query specification

10. Any of the following view tables in the FROM clause of an outermost query specification:

    • A view table defined by specifying any of items (1) to (9) above in a view definition statement.

    • View tables that are defined by specifying a value expression other than a column specification in the SELECT clause of the outermost query specification in a view definition statement.

    • A view table defined by specifying the READ ONLY option in the view definition statement.

11. WITHOUT LOCK NOWAIT

12. A query specification name specified in the FROM clause of the outermost query specification in the query expression body in which a WITH clause is specified

OF *column-name* [, *column-name*]...

If a table is being searched using a cursor and when only searched rows using the cursor are to be updated, this operand specifies the column to be updated.

Columns that are not specified in the selection expression of the SELECT statement can also be specified in *column-name*. Each column specified in *column-name* must be distinct.

If a table is being searched using a cursor and rows in the table are not updated or deleted using that cursor or another cursor, and rows on which a cursor is not used are not updated, deleted, or added, this operand should not be specified.

The column name that is specified should specify a column in the table specified in the FROM clause of the outermost query specification, rather than the column name specified in AS *column-name*.

[NOWAIT]

Produces the same behavior as if the FOR UPDATE clause was specified and WITH EXCLUSIVE LOCK NO WAIT was specified as the lock option. However, if the lock option is specified, NOWAIT cannot be specified.

For details about operations when WITH EXCLUSIVE LOCK NO WAIT is specified as the lock option, see *2.19 Lock option*.

FOR READ ONLY

This operand is specified when performing an update by specifying another cursor or a direct search condition during a retrieval using a cursor. Specify FOR READ ONLY so that any update performed during the retrieval process does not affect the results of retrieval.

Specifying the FOR READ ONLY clause is subject to the following restrictions:

(a) Scalar operations, function calls, and component specifications that produce any of the following data types in the results cannot be specified in a selection expression:

- BLOB
- BINARY with a maximum length of 32,001 bytes or greater
- BOOLEAN
- Abstract data type

(b) Only column specifications can be specified as an output BLOB value with a WRITE specification.

(c) The GET_JAVA_STORED_ROUTINE_SOURCE specification cannot be specified.

- [UNTIL DISCONNECT]

Specifies that a holdable cursor is to be used. For details about holdable cursors, see the *HiRDB Version 9 UAP Development Guide*.

The following rules apply to holdable cursors:

1. A holdable cursor cannot be used in the following cases:
   - When a column of the abstract data type using a plug-in is specified
   - When a function call using a plug-in is specified
   - A query for a named derived table derived by specifying a function call using a plug-in
2. Definition SQL statements cannot be executed while a holdable cursor is open. If the holdable cursor is closed, execution of a definition SQL statement invalidates any preprocessing that is using the holdable cursor.
3. If, after an OPEN statement is executed for a SELECT statement using a holdable cursor, a PURGE TABLE statement is executed for a table used in the SELECT statement, the cursor is placed into closed status.
4. If, after an OPEN statement is executed for a SELECT statement using a holdable cursor and before a DISCONNECT is performed, another user issues a definition SQL statement for a table used in the SELECT statement, the definition SQL statement is placed into lock-wait status. Similarly, if, during the period when preprocessing relative to a SELECT statement using a holdable cursor is still in effect, another user issues a definition SQL statement for a table that is being used in the SELECT statement, the definition SQL statement is placed into lock-wait status.

Rule related to referential constraints

1. A holdable cursor that is used to retrieve a table in which a foreign key is defined is closed when the table being retrieved goes into check pending status.

Notes

1. Because specification of the `FOR UPDATE` clause causes a work table to be created, this operand should be omitted if no rows are to be updated, added, or deleted from the current table using a cursor.
2. By applying the work table creation suppression feature of the update SQL statement in the SQL optimization option and using the index key-value no-lock facility, you can update, add, or delete rows while using a cursor for which neither `FOR UPDATE` nor `FOR UPDATE OF` is specified.

Examples

For examples, see *DECLARE CURSOR Format 1 (Declare cursor)* and *ALLOCATE CURSOR statement Format 1 (Allocate a statement cursor)*.

Dynamic SELECT statement Format 2 (Retrieve dynamically)

Function

The dynamic SELECT statement uses a list for retrieving data from a table.

Privileges

The owner of a list can use that list to retrieve data from tables.

Format 2: Retrieving table data using a list

```
(Cursor-Specification-Format-2)
  SELECT {{value-expression|WRITE
specification|GET_JAVA_STORED_ROUTINE_SOURCE specification}
       [[AS]column-name]
          [, {value-expression|WRITE
specification|GET_JAVA_STORED_ROUTINE_SOURCE specification}
          [[AS]column-name]]...
       |*}
    FROM LIST list-name
(Lock-Option)
 [ {WITH {SHARE|EXCLUSIVE} LOCK
      |WITHOUT LOCK [ {WAIT|NOWAIT} ]}]
 [{WITH ROLLBACK|NO WAIT} ]
```

Operands

For cursor specification, see *2.1.2 Cursor specification: Format 2*; for the lock option, see *2.19 Lock option*.

- [*Lock-Option*]

Specifies that the base table to be searched is to be locked when retrieval is performed using a list.

Common rules

1. If a row that existed in the base table when the list was created is not found during retrieval processing, SQL code +110 is returned and retrieval processing continues.
2. The same user cannot manipulate a list by connecting to HiRDB concurrently in multiple sessions.

Notes

1. Rows in the base table that are deleted after the list is created cannot be retrieved during retrieval processing.

2. Rows in the base table that are updated after the list is created can be retrieved.
3. If rows are deleted and then rows are inserted into the base table after the list has been created, sometimes only inserted rows are retrieved.
4. Between the time an SQL for searching a table via a list is preprocessed by the `PREPARE` statement and the time the `OPEN` statement is executed, an `ASSIGN LIST` statement that specifies the same list name must not be executed.

UPDATE statement Format 1 (Update data)

Function

The UPDATE statement updates the values of columns in the rows that satisfy specified search conditions or in the row indicated by a cursor.

Privileges

A user who has the UPDATE privilege for a table can update the row values in that table.

However, if a subquery is specified in the search condition, the user needs the SELECT privilege for the table for which the subquery is specified.

Format 1: Updating rows in a table on a column-by-column basis

```
UPDATE [authorization-identifier.]table-identifier
          [IN (RDAREA-name-specification)][[AS] correlation-name]
          [SQL-optimization-specification-for-used-index]
          (SET {update-object = update-value
             |(update-object,update-object[,update-object]) = row-subquery}
                [,{update-object = update-value
                   |(update-object,update-object[,update-object]...)
                      = row-subquery}]...
           |ADD repetition-column-name [{subscript|*}]
                  = {ARRAY [element-value[,element-value]...]
                    |?-parameter|:embedded-variable[:indicator-variable]}
                   [,repetition-column-name [{subscript|*}]
                        = {ARRAY [element-value[,element-value]...]
                          |?-parameter|:embedded-variable
                              [:indicator-variable]}]...
           |DELETE repetition-column-name [{subscript|*}]
                   [,repetition-column-name [{subscript|*}]]...}
[WHERE {search-condition|CURRENT OF {cursor-name|
extended-cursor-name}}]
[WITH ROLLBACK]

update-object ::= {column-name|component-specification
                  |column-name [{subscript|*}]}
```

Operands

- [*authorization-identifier*.]*table-identifier*

Specifies the table that is to be updated.

The following rules apply:

1. In read-only view tables, rows cannot be updated. For details about read-only view tables, see *Common rules* under *CREATE [PUBLIC] VIEW (Define view, define public view)* in Chapter 3.
2. If updating of a column in a view table is specified, the UPDATE statement updates the base table associated with that column of the view table.
3. The table name is valid throughout the entire UPDATE statement.
4. A name that is the same as the table name used in the FROM clause of the subquery cannot be specified if the column of the table to be updated is specified in an external reference in a subquery specified in the SET or ADD clause, and a value expression with any of the following attributes is specified in a selection expression in the subquery:
   - BLOB
   - BINARY type with a maximum length of 32,001 bytes or greater
   - Repetition column
   - Abstract data type

   However, if a view table is specified in the table name in the FROM clause of the subquery, all table names specified in a derived query expression in the view table definition are subject to this operation.

authorization-identifier

Specifies the authorization identifier of the owner of the table.

MASTER cannot be specified as an authorization identifier.

table-identifier

Specifies the name of the table that is to be updated.

■ [IN (*RDAREA-name-specification*)]

IN

Specifies the RDAREA to access.

RDAREA-name-specification ::= *value-specification*

Of the RDAREAs that contain the table specified in the table identifier, specify the name of the RDAREA to access as a *value-specification* of type VARCHAR, CHAR, MVARCHAR, or MCHAR. If multiple RDAREA names are specified, separate them with a comma (,). RDAREA names must be unique; an error occurs if duplicate RDAREA names are specified. For details about what characters are allowed in RDAREA names in *value-specification*, see *1.1.7 Specification of names*. Note also that leading and trailing whitespace is ignored in RDAREA names specified in *value-specification*. If the RDAREA name is enclosed in

double quotation marks (`"`), only whitespace outside the double quotation marks is ignored.

If *cursor-name* or *extended-cursor-name* is specified, specify the same set of RDAREAs (in any order) as the RDAREAs specified in the cursor declared in the cursor declaration. An error occurs if they are not specified.

If specifying an RDAREA that uses the inner replica facility, specify the original RDAREA name. To target the replica RDAREA, use the change current database command (`pddbchg` command), or the `PDDBACCS` operand in the client environment definition, to switch the RDAREA to access to the replica RDAREA.

- [`AS`] *correlation-name*

Specify this operand when a correlation name is to be used for the table being updated.

The scope of *correlation-name* is the entire `UPDATE` statement. The table identifier to be updated has no scope.

- *SQL-optimization-specification-on-used-index*

For details about SQL optimization specifications on used indexes, see *2.24 SQL optimization specification*.

- `SET` *update-object* = *update-value*

Specifies a table identifier when the value of a column or the value of an attribute of an abstract data type is to be updated.

column-name

Specifies the name of a column to be updated.

component-specification

Specifies the attribute of the abstract data type being updated.

column-name [{*subscript*|*}]

column-name

Specifies a repetition column whose elements are to be updated.

[{*subscript*|*}]

Specifies the position of the element that is to be updated; to update the last element, specify an asterisk (*).

If there are no elements in the repetition column that is to be updated, specifying an asterisk is meaningless.

update-value

Specifies any of the following items to be used as the paired column's new

value:

- Column name
- Component specification
- Literal
- Value expression (including an arithmetic or concatenation operation)
- Scalar subquery
- `USER`
- `CURRENT_DATE`
- `CURRENT_TIME`
- `CURRENT_TIMESTAMP[(p)]`
- `NULL` (null value)
- `DEFAULT` (Represents the predefined value for the column to be updated.)
- Embedded variable (and indicator variables)
- `?` parameter
- SQL variables or SQL parameters
- `ARRAY` [*element-value* [, *element-value*] ...][#]

  #: None of the following can be specified as the value of an element:

  - A column other than a repetition column
  - A subscripted repetition column
  - A literal
  - A value expression (including arithmetic and concatenation operations)
  - A scalar subquery
  - `USER`
  - `CURRENT_DATE`
  - `CURRENT_TIME`
  - `CURRENT_TIMESTAMP[(p)]`
  - `NULL` (represents the null value)
  - `DEFAULT`

- Embedded variables (and indicator variables)
- ? parameter
- An SQL variable or SQL parameter

Rules on update values

1. When a column name is specified as the updated value, the name of a column with either the same data type as the column or attribute being updated or a data type that can be converted to that data type must be specified.
2. When a scalar subquery is specified as the updated value, the data type of the column obtained as a result of the scalar subquery must be the same as that of the column or attribute to be updated or a data type that can be converted to it.
3. If the data to be updated is character data, it must use the same character set as the updated value. However, if an embedded variable (that uses the default character set), ? parameter, or string constant is specified as the updated value, it will automatically be converted to the character set used for the data being updated.
4. When an unsubscripted repetition column is specified as the update object, the name of an unsubscripted repetition column must be specified as the column name of the update value or in the selection expression for the scalar subquery.
5. When a subscripted repetition column is specified as the update object, the name of an unsubscripted repetition column cannot be specified as the column name of the update value or in the selection expression for the scalar subquery.
6. Embedded variables and indicator variables cannot be specified in UPDATE statements or procedures that are preprocessed by the PREPARE statement. In a procedure, either an SQL variable or an SQL parameter must be used.
7. When an embedded variable (indicator variable), ? parameter, SQL variable, or SQL parameter is specified, the data type of the embedded variable (for a ? parameter, the embedded variable that is specified to assign a value to it), SQL variable, or SQL parameter should be the data type of the column or attribute being updated or a data type that can be converted to that data type.

   If the column to be updated is a repetition column, the embedded variable (indicator variable) or ? parameter that holds the update value should have a repetition structure.
8. When an indicator variable with a negative value is specified, the value of the embedded variable is interpreted to be the null value and the null value is set in the corresponding column. When the value of a specified indicator

variable is 0 or positive, the value of the embedded variable is assigned to the corresponding column.

A ? parameter can be specified only in an UPDATE statement that is preprocessed by the PREPARE statement.

9. The value to be assigned to the ? parameter is specified in an embedded variable in the USING clause of the EXECUTE statement that corresponds to the PREPARE statement that prepares the UPDATE statement.
10. The update value must have a data type that can be converted to or compared with the column to be updated.

    However, if the column to be updated is of the national character data type and a character string literal is specified as the update value, the character string literal will be treated as a national character string literal. When a character string literal is treated as a national character string literal, only the character data length is checked; the character codes are not checked.
11. When the column to be updated is an unsubscripted repetition column, none of the following can be specified as the update value: a literal, value expression, USER, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP[(p)], SQL variable, or SQL parameter.
12. ARRAY [*element-value* [, *element-value*] ...] can be specified only if the column being updated is a repetition column.
13. A maximum of 30,000 element values can be specified in ARRAY [*element-value* [, *element-value*] ...], provided that the number does not exceed the maximum permissible number of elements for the column being updated.
14. DEFAULT, specified as an update value, takes the following predefined values:

    • If the DEFAULT clause is specified for the column to be updated, a specified predefined value takes effect.

    • If DEFAULT is not specified but WITH DEFAULT is specified, the predefined value for WITH DEFAULT takes effect.

    • If neither the DEFAULT clause nor WITH DEFAULT is specified, NULL becomes the predefined value.

Rules on element values

1. Embedded variables and indicator variables cannot be specified in an UPDATE statement preprocessed by a PREPARE statement or in a procedure. An SQL variable or parameter should be used in a procedure.
2. When an embedded variable (indicator variable), ? parameter, SQL variable,

or SQL parameter is specified, the data type of the embedded variable (for a ? parameter, the embedded variable that is specified to assign a value to it), SQL variable, or SQL parameter should be the data type of the column or attribute being updated or a data type that can be converted to that data type. In addition, the embedded variable (indicator variable) or ? parameter should have a simple structure.

3. If an indicator variable is specified and its value is negative, the value of the embedded variable is interpreted to be NULL, and the NULL value is assigned to the corresponding column. If the value of the indicator variable is either 0 or positive, the value of the embedded variable will be assigned to the corresponding column.
4. The ? parameter can be specified only in an UPDATE statement that is preprocessed by a PREPARE statement. The value to be assigned to the ? parameter is specified in an embedded variable in a USING clause in the EXECUTE statement for the PREPARE statement.
5. The value of an element should be of that can be converted into or compared with the data type of the column to be updated. However, if the column to be updated is of the national character data type and a character string constant is specified as the update value, the character string constant is treated as a national character string constant, in which case its character codes are not checked and only the length of the character data is checked.
6. When a scalar subquery is specified as an element value, an unsubscripted repetition column cannot be specified in the selection expression for the scalar subquery.
7. If DEFAULT is specified as the value of an element, the predefined value is the null value.

Rules on updating a column of the BLOB type or the BINARY type with a defined length of 32,001 bytes or greater, using concatenation operations

1. A column specification can be specified in the first operand of the concatenation operation. In the second term, an embedded variable, a ? parameter, an SQL variable, and an SQL parameter can be specified.
2. When specifying a concatenation operation of the BLOB type or the BINARY type with a defined length of 32,001 bytes or greater as an update value, be sure to specify the same column as the target of updating in the first operand of the concatenation operation.
3. A concatenation operation cannot be specified as the result of a concatenation operation.
4. The only data type that can be concatenated with a BLOB type is another BLOB type. Numeric data, character data, national character data, or mixed

character data cannot be concatenated.

5. The only data type that can be concatenated with the BINARY type is another BINARY type. Numeric data, character data, national character data, or mixed character data cannot be concatenated.
6. The results of a concatenation operation allow the null value, irrespective of whether the value of the first or second operand is subject to NOT NULL constraints.
7. A concatenation operation producing the actual length of the result exceeding the maximum length for the BLOB type or the BINARY type (2,147,483,647 bytes) results in an error.
8. If any of the following definitions is in the table to be updated, an update using concatenation operations can cause an error:
   - An UPDATE trigger is defined for the table to be updated
   - The column to be updated using concatenation operations is specified in a search condition with a check constraint.

- SET (*update-object* , *update-object*[, *update-object*] ...) = *row-subquery*

Specifies updating objects in order to update the values of multiple columns with the results of a row subquery. At least two update objects must be specified.

column-name

Specifies the name of a column that is to be updated.

component-specification

Specifies the attribute of an abstract data type that is to be updated.

column-name [{*subscript*|*}]

column-name

Specifies a repetition column whose elements are to be updated.

[{*subscript*|*}]

Specifies the position of the element that is to be updated; to update the last element, specify an asterisk (*).

If there are no elements in the repetition column that is to be updated, specifying an asterisk is meaningless.

row-subquery

Specifies a row subquery for retrieving the data to be updated. For details about row subqueries, see *2.4 Subqueries*.

The following rules apply to row subqueries:

1. The number of updates and the number of selection expressions in the row subquery should be the same.
2. The data type of the column to be obtained as the result of a row subquery should be the same as the data type of the column or attribute to be updated, or it must be an equivalent convertible data type.
3. If the data to be updated is character data, it must use the same character set as the column obtained from the row subquery.
4. When an unsubscripted repetition column is specified as an object of the update, specify the column name of an unsubscripted repetition column for the column in the selection expression in the row subquery.
5. When a subscripted repetition column is specified as an object of the update, do not specify the column name of an unsubscripted repetition column for the column in the selection expression in the row subquery.

Rules for the SET clause

1. When elements of a repetition column are being updated by specifying a subscript in the SET clause, only one update can be specified in the same SET clause relative to the same element in the same column.
2. When a repetition column is updated without specifying a subscript in the SET clause, the repetition column cannot be updated by specifying a subscript in the same SET clause.
3. When a repetition column is updated by specifying the asterisk (*) as a subscript, the repetition column cannot be updated in the same SET clause.

- ADD *repetition-column-name* [{*subscript* |*}] = {ARRAY [*element-value* [, *element-value*] ...] | *?-parameter* |: *embedded-variable* [: *indicator-variable*]}

Specifies addition of one or more elements to a repetition column.

repetition-column-name [{*subscript* |*}]

repetition-column-name

Specifies the repetition column to which elements are to be added.

[{*subscript* | *}]

Specifies as the subscript the position at which the elements are to be added. The asterisk (*) is specified when the added elements will be the last elements in the column.

ARRAY [*element-value* [, *element-value*] ...]

Specifies the element values that are to be added. The following items can be specified as element values:

- Column names (other than the names of repetition columns)
- Subscripted repetition columns
- Literals
- Value expressions (including arithmetic and concatenation operations)
- Scalar subquery
- `USER`
- `CURRENT_DATE`
- `CURRENT_TIME`
- `CURRENT_TIMESTAMP[(p)]`
- `NULL` (representing the `C` value)
- `DEFAULT`
- Embedded variables (and indicator variables)
- `?` parameters
- SQL variables or parameters

For rules governing the value of an element, see *Rules on element values* for `SET` *update-object* = *update-value*.

?-parameter

The data type of a `?` parameter should be the data type of the column being updated or a data type that can be converted to that data type. The structure of the `?` parameter should be the repetition structure.

A `?` parameter can be specified only in an `UPDATE` statement that is preprocessed by the `PREPARE` statement. The value to be assigned to the `?` parameter can be specified in the embedded variable in a `USING` clause in the `EXECUTE` statement for the `PREPARE` statement that performed preprocessing.

:*embedded-variable* [`:` *indicator-variable*]

Embedded variables and indicator variables cannot be specified in an `UPDATE` statement preprocessed by the `PREPARE` statement or in a procedure.

The data type of an embedded variable (indicator variable) should be the data type of the column to be updated or a data type that can be converted to that data type. The structure of the embedded variable (indicator variable) should be the repetition structure.

If an indicator variable is specified and its value is negative, the value of the embedded variable is interpreted to be the null value, and the null value is assigned to the corresponding column. If the value of the indicator variable is

either 0 or positive, the value of the embedded variable will be assigned to the corresponding column.

Rules for the ADD clause

1. Only one addition of elements to a column can be specified in one ADD clause.
2. When elements are added, it is important to ensure that the total number of elements in the column after the addition does not exceed the maximum permissible number of elements for the column.
3. A maximum of 30,000 values can be specified as element values in ARRAY [*element-value* [, *element-value*] ...].
4. Elements that are located following elements added by specifying a subscript are moved back. If a number greater by at least 2 than the number of elements in the repetition column to which elements are added is specified, null values are added until the number of elements in the repetition column is (specified elements - 1), and the new element values are added at the trailing end.
5. If a multi-column index is defined using the column to which elements are added, the same number of elements must be added to each of the repetition columns comprising the index and at the same element positions using a single ADD clause.
6. When an element that uses a cursor is added by specifying CURRENT OF *cursor-name* in the WHERE clause, column names or value expressions cannot be specified as element values in the ADD clause.
7. If DEFAULT is specified as the value of an element, the predefined value is the null value.

- DELETE *repetition-column-name* [{*subscript* | *}]

Specifies deletion of elements from a repetition column.

repetition-column-name [{*subscript* | *}]

repetition-column-name

Specifies the repetition column from which an element is to be deleted.

[{*subscript* | *}]

Specifies as the subscript the position of the element that is to be deleted. The asterisk (*) is specified when the last element in the column is to be deleted.

Rules for the DELETE clause

1. When a subscript is specified in the DELETE clause, only one column element can be specified for deletion per DELETE clause.

2. The subscript value cannot be greater than the number of elements in the column.
3. When the asterisk is specified, no other elements in the column can be specified for deletion simultaneously in the same DELETE clause.
4. Elements that follow a deleted element move up in order.
5. If a multi-column index is defined using the column from which elements are deleted, the same number of elements must be deleted from each of the repetition columns comprising the index at the same element positions using a single DELETE clause.

- [WHERE {*search-condition*|CURRENT OF *cursor-name*|*extended-cursor-name*}]

When this clause is omitted, all rows in the specified table are updated.

search-condition

Specifies criteria for selecting the rows to be updated.

The UPDATE statement updates all rows that satisfy the specified search condition.

cursor-name

Specifies the name of the cursor to be used to indicate the row to be updated.

A cursor name cannot be specified if the UPDATE statement is preprocessed by the PREPARE statement.

The cursor specified in *cursor-name* should be one that was declared in the cursor declaration. The name of the column to be updated using the UPDATE statement must be specified in the FOR UPDATE clause of the cursor declaration.

When a cursor name is specified, the cursor must be specified as updatable in the cursor declaration. For details about updatable cursors, see item 4 under *Common rules* in *DECLARE CURSOR Format 1 (Declare cursor)*.

The specified cursor must be opened by the OPEN statement and positioned by the FETCH statement at the row to be updated.

The position of the specified cursor remains unchanged after the UPDATE statement has executed. To update a row that follows the updated row, the cursor must be moved by executing a FETCH statement for the cursor.

extended-cursor-name

Specifies an extended cursor name that identifies the cursor that points to the row to be updated.

An extended cursor name cannot be specified if it is preprocessed by the PREPARE statement.

The extended cursor name that identifies the cursor allocated by the ALLOCATE

CURSOR statement should be specified. However, a result set cursor cannot be specified.

If *extended-cursor-name* is specified, an extended cursor must be specified for queries in which the FOR UPDATE clause is specified. For details about the FOR UPDATE clause, see the FOR UPDATE clause under *Operands* in *Dynamic SELECT statement Format 1 (Retrieve dynamically)* in this chapter.

The cursor identified by the extended cursor name must be open and also must be positioned on the row to be updated.

The position of the cursor identified by *extended-cursor-name* remains unchanged after the execution of the UPDATE statement. If a row after the updated row is to be updated, the FETCH statement should be executed on the cursor to move it.

For extended cursor names, see *2.27 Extended cursor name*.

- [WITH ROLLBACK]

Specifies that if the table to be updated is being used by another user, the transaction is to be cancelled and invalidated.

When the WITH ROLLBACK option is omitted and the table to be updated is being used by another user, the system executes the UPDATE statement after the transaction issued by the other user has terminated.

Common rules

1. When a column of the INTEGER or SMALLINT type is updated with fixed-point or floating-point data, any decimal places (to the right of the decimal point) are truncated before the update action. Similarly, when a column of the DECIMAL type is updated, any digits below the scaling for the column are truncated before the update action.
2. Character data, BLOB data, or BINARY data with a length greater than or equal to the length that was determined when the table was defined cannot be input as a value of the column to be updated.
3. Numeric data outside the defined range cannot be input as an update column value.
4. If the data that updates a fixed-length character string column (including a national character string or a mixed character string) is shorter than the length of the column, the data is inserted left-justified and the column is filled with trailing blanks.
5. When updating a column or attribute of the BLOB data type, as an update value you can specify a column specification, a component specification, an embedded variable, a ? parameter, an SQL variable, an SQL parameter, a concatenation operation, the SUBSTR scalar variable, a function call, a subquery, or NULL.

6. When a BLOB data type column or an attribute is updated, HiRDB deletes the existing data after writing the new data into the database. Therefore, the LOB RDAREA for data updating requires sufficient free space for writing the new data. An RDAREA full error may result if sufficient free space cannot be allocated.
7. When a column or an attribute of an abstract data type is updated, an embedded variable or a ? parameter cannot be specified as the update value.
8. When a column or an attribute of an abstract data type is updated, an abstract data type value cannot be specified as the update value, including a BLOB attribute for which no LOB attributes storage RDAREA name was specified at the time of table definition.
9. When elements are updated in or deleted from a column and a subscript is specified that is greater than the current number of elements in the column, or the asterisk is specified when the number of elements is 0, there will be no elements to be updated or deleted, in which case the specified updating or deletion action for the column is ignored. In such a case, W is set in the SQLWARN7 variable of the SQLCA.
10. Only one SET, ADD, or DELETE clause can be specified per SQL statement.
11. A maximum of 30,000 items can be specified in each SET, ADD, or DELETE clause.
12. SET, ADD, and DELETE clauses are executed from left to right in the specified order.
13. If the user LOB RDAREA that stores a LOB column or LOB attribute is in the frozen update status, the LOB column or LOB attribute cannot be updated (an attempt to update it causes an already frozen error).
14. The UPDATE statement cannot be executed on falsification prevented tables. However, updatable columns can be updated.

    The following table indicates the updatability of column values when UPDATE ONLY FROM NULL is specified for a falsification-prevented table:

Column value before update	Column value after update	Updatability
Null value	Null value	Y
Null value	Non-null value	Y
Non-null value	Null value	N
Non-null value	Non-null value#	N

Legend:

Y: Updatable.

N: Not updatable.

Note

For a repetition column, only updating by column from the null value (a value for which the number of elements is 0) without a subscript specification.

#: Includes the same value as the pre-update value.

15. If the index constituent column of a table with a `WITHOUT ROLLBACK` specification is the update object column, updating cannot be executed if the update value for the index constituent column is other than the pre-update value. For details, see the rules on `WITHOUT ROLLBACK` in *CREATE TABLE (Define table)* in Chapter *3*.
16. If the table to be updated is a shared table and if an index is defined for the column to be updated, the `LOCK` statement should be executed in the lock mode on the shared table before updating. An attempt to update without executing the `LOCK` statement the value of a column on which a shared table is defined can cause an error. However, the `LOCK` statement need not be executed if the value of the column for which an index is defined is not changed. For details about updating a shared table, see the *HiRDB Version 9 Installation and Design Guide*. For objects of locking for the execution of the `LOCK` statement on a shared table, see the notes in *LOCK statement (Lock control on tables)*.
17. If the table to be updated meets all of the following conditions and if columns in the table are to be updated by `DEFAULT`, the `LOCK` statement in the locking mode should be executed on the shared table before updating; an attempt to update the table without executing the `LOCK` statement can cause an error:
    - The table to be updated is a shared table.
    - The column to be updated is of the `timestamp` data type.
    - When the table was defined, `CURRENT_TIMESTAMP USING BES` was specified as the default.
18. If a cursor name or an extended cursor name is specified in *search-condition*, a subquery in which the table to be updated is specified in a `FROM` clause cannot be specified in the `SET` clause.
19. If *RDAREA-name-specification* is specified, you cannot use an index in which the number of partitions is different from the number of partitions of the table. When defining an index for queries that specify *RDAREA-name-specification*, define an index that has the same number of partitions as the number of partitions of the table.

Rules on referential constraints

1. For the rules on updating the primary key of a referenced table and the foreign key

of a referencing table, see the explanation of referencing actions in *CREATE TABLE (Define table)* in Chapter *3*.

2. For the updating of values of foreign key constituent columns in a referencing table, the referencing table is searched to determine whether the updating value is included in the value of the primary key constituent column in the referenced table. The data guarantee level during the search through the referenced table assumes the share mode. For this reason, if during the updating of the referencing table, operations are performed on the referenced table by another transaction, the updating action goes into a wait state until the transaction is settled.
3. For the deletion of a rows in a referenced table for which constraint operations are defined in `RESTRICT`, the referencing table is searched to determine whether the value of the primary key constituent column in the rows to be deleted is included in the value of a foreign key constituent column in the referencing table. The data guarantee level during the search through the referencing table assumes the share mode. For this reason, if, during the deletion of rows in the referenced table for which constraint operations are defined in `RESTRICT`, operations are performed on the referencing table by another transaction, the row deletion action goes into a wait state until the transaction is settled.
4. If any combination of the conditions listed below occurs, data incompatibility can occur between the referenced table and the referencing table subject to referential constraints. Such incompatibility can also occur regardless of whether the constraint operation is `RESTRICT` or `CASCADE`. For rules on referential constraints, see the *HiRDB Version 9 Installation and Design Guide*.
   - The transaction that deletes rows in the referencing table is different from the transaction that updates or deletes rows in the referenced table.
   - The above two transactions are executed simultaneously.
   - The value of the primary key constituent column deleted from the referencing table is the same as the value of the foreign key constituent column that is either updated in or deleted from the referenced table.
   - Either the transaction that deletes rows from the referencing table is committed or the transaction that updates or deletes rows in the referenced table is rolled back.

Note

A cursor name, similar to an embedded variable name, is effective only within a compile-unit module. Multiple SQLs relative to the same cursor cannot be used in multiple modules.

Examples

1. In a stock table named `STOCK`, change to 100 the stock quantity (`SQTY`) column

for any row whose product code (PCODE) column is 302S:

```
UPDATE STOCK
  SET SQTY = 100
  WHERE PCODE = '302S'
```

2. In the product code (PCODE) column in the stock table (STOCK), apply a 20% discount to the unit prices (PRICE) of products that end with the letter 'S':

```
UPDATE STOCK
  SET PRICE = PRICE*0.8
  WHERE PCODE LIKE'%S'
```

3. Update the unit price (PRICE) and stock quantity (SQTY) columns in the stock table (STOCK) using values that have been read into embedded variables:

```
UPDATE STOCK
  SET PRICE=:XPRICE,SQTY=:XSQTY
```

4. Find the product whose product code (PCODE) column value in the STOCK table is 302S and change its stock quantity (SQTY) to the stock quantity (XQTY) of the product whose product code (PCODE) column value is 302S in stock table 2 (STOCK2); the STOCK2 table has the same column definition as the STOCK table:

```
UPDATE STOCK
   SET SQTY=
       (SELECT SQTY FROM STOCK2 WHERE PCODE='302S')
   WHERE PCODE='302S'
```

5. Change the stock quantity (SQTY) and the unit price (PRICE) of the product whose product code (PCODE) column value in the STOCK table is 302S to the stock quantity (XQTY) and unit price (PRICE) of the product whose product code (PCODE) column value is 302S in stock table 2 (STOCK2); the STOCK2 table has the same column definition as the STOCK table:

```
UPDATE STOCK
   SET (PRICE,SQTY)=
       (SELECT PRICE,SQTY FROM STOCK2 WHERE PCODE='302S')
  WHERE PCODE='302S'
```

UPDATE statement Format 2 (Update data)

Function

Updates row by row the values of the table rows that satisfy the specified search condition or are indicated by a cursor.

Privileges

A user who has the UPDATE privilege for a table can update the row values in that table.

However, if a subquery is specified in the search condition, the user needs the SELECT privilege for the table for which the subquery is specified.

Format 2: Updating rows in a table with the FIX specification on a row-by-row basis

```
UPDATE [authorization-identifier.]table-identifier
       [IN (RDAREA-name-specification)][[AS] correlation-name]
       [used-index-SQL-optimization-specification]
             SET ROW= row-update-value
        [SQL-optimization-specification-for-used-index]
        [WHERE {search-condition|CURRENT OF
{cursor-name|extended-cursor-name}}]
        [WITH ROLLBACK]
```

Operands

All operands other than ROW and the operand rules for them are the same as for Format 1 of the UPDATE statement. The following rules apply to specifying ROW:

1. Row-by-row updating can be specified only on a base table with the FIX attribute. The operand ROW refers to an entire row. When ROW is specified, HiRDB treats the entire row as one set of data. The data type of the data used for updating should be the ROW type, regardless of the data types of the individual columns. (Variables corresponding to CHAR(*n*) (where *n* is the row length) or structures of the same length can be specified as ROW-type data; if a structure is specified, the structure should not contain any boundary alignment blanks.) The data length should be equal to the row length (the sum of the data lengths of the columns).
2. The platform on which the UAP runs and the platform on which the HiRDB server runs should have the same endian. The ROW option cannot be used between different endians. For example, if ROW is used in a Windows edition UAP, the HiRDB server should also use the Windows edition of the same endian.

- SET clause

Specifies that column values are to be updated.

- SET ROW=*row-update-value*

ROW

Specifies that data is to be updated on a row-by-row basis.

row-update-value

Any of the following items can be specified as the row update values corresponding to ROW:

- Embedded variables (and indicator variables)
- ? parameters
- SQL variables or SQL parameters

Common rules

1. Because the UPDATE statement by row updates the values of all columns, the UPDATE statement by row cannot be executed on a falsification-prevented table in which at least one column is not updatable.
2. The UPDATE statement cannot be executed by row for tables that contain columns of character data that do not use the default character set.
3. If a table with a WITHOUT ROLLBACK specification for which an index is defined is specified in *table-identifier*, updating cannot be executed if the update value for the index constituent column is other than the pre-update value. For details, see the rules on WITHOUT ROLLBACK in *CREATE TABLE (Define table)* in Chapter *3*.
4. If the table to be updated is a shared table and if an index is defined for one of the columns, the LOCK statement should be executed in the lock mode on the shared table before updating. An attempt to update the shared table by row (ROW specification) without executing the LOCK statement can cause an error. However, the LOCK statement need not be executed if the value of the column for which an index is defined is not changed. For details about updating a shared table, see the *HiRDB Version 9 Installation and Design Guide*. For objects of locking for the execution of the LOCK statement on a shared table, see the notes in *LOCK statement (Lock control on tables)*.
5. If *RDAREA-name-specification* is specified, you cannot use an index in which the number of partitions is different from the number of partitions of the table. When defining an index for queries that specify *RDAREA-name-specification*, define the index that has the same number of partitions as the number of partitions of the table.

Rules on referential constraints

1. For the rules on row-by-row updating of rows in a referenced table or referencing

table, see the explanation of referencing actions in *CREATE TABLE (Define table)* in Chapter *3*.

2. For the updating of rows in a referenced table for which constraint operations are defined in RESTRICT, the referencing table is searched to determine whether the updating value is included in the value of a foreign key constituent column in the referencing table. The data guarantee level during the search through the referencing table assumes the share mode. For this reason, if, during the deletion of rows in the referenced table for which constraint operations are defined in RESTRICT, operations are performed on the referencing table by another transaction, the row deletion action goes into a wait state until the transaction is settled.
3. If any combination of the following conditions occurs, data incompatibility can occur between the referenced table and the referencing table subject to referential constraints. Such incompatibility can also occur regardless of whether the constraint operation is RESTRICT or CASCADE. For rules on referential constraints, see the *HiRDB Version 9 Installation and Design Guide*.
   - The transaction that deletes rows in the referencing table is different from the transaction that updates or deletes rows in the referenced table.
   - The above two transactions are executed simultaneously.
   - The value of the primary key constituent column deleted from the referencing table is the same as the value of the foreign key constituent column that is either updated in or deleted from the referenced table.
   - Either the transaction that deletes rows from the referencing table is committed or the transaction that updates or deletes rows in the referenced table is rolled back.
4. If a combination of the following conditions occurs, a deadlock can occur between the transaction that manipulates the referenced table and the transaction that manipulates the referencing table. A deadlock can also occur if the constraint operation is either RESTRICT or CASCADE. For details about a deadlock between referenced and referencing tables, see the *HiRDB Version 9 Installation and Design Guide*.
   - The transaction that deletes the rows in the referencing table is different from the transaction that updates or deletes the rows in the referenced table.
   - The above two transactions are executed simultaneously.
   - The value of the foreign key constituent column updated in the referencing table is the same as the value of the primary key constituent column that is deleted from the referenced table.

Notes

1. If the data type of the table columns to be updated is DECIMAL, or a national character string, the system checks the contents of the applicable row update values.
2. A cursor name, similar to an embedded variable name, is effective only within a compile-unit module. Multiple SQLs relative to the same cursor cannot be used in multiple modules.
3. When performing retrieval or updating by row (ROW specification), the date data type portion of the embedded variable, the SQL variable, or the SQL parameter with respect to ROW is 4 bytes long, and is specified in an X'*YYYYMMDD*' format.

   When receiving date data in a predefined character string representation using a by-row (ROW specification) interface, when defining a column, define it as CHAR(10) rather than a date data type.

   For date operations, use the DATE scalar function, to be specified after the data is converted into the date data type.
4. When performing retrieval or updating by row (ROW specification), the time data type portion of the embedded variable, the SQL variable, or the SQL parameter with respect to ROW is 3 bytes long, and is specified in an X'*hhmmss*' format.

   When receiving time data in a predefined character string representation using a by-row (ROW specification) interface, when defining a column, define it as CHAR(8) rather than a time data type. For time operations, use the TIME scalar function, to be specified after the data is converted into the time data type.
5. For performing a retrieval or updating by row (ROW specification), the time stamp data type portion of the embedded variable, SQL variable, or SQL parameter with respect to ROW is (7 + *p*/2) byte long, and it should be in the X'*YYYYMMDDhhmmss*[*nn*. . . *n*]' format.

   When receiving time stamp data in a predefined character string representation using a by-row (ROW specification) interface, when defining a column, define it as CHAR with a length of 19, 22, 24, or 26 bytes rather than as a time stamp data type column.

Example

In a stock table named STOCK, update in a single operation the data in the row specified by cursor CR1 with the contents of embedded variable XROW:

```
UPDATE STOCK
  SET ROW = :XROW
  WHERE CURRENT OF CR1
```

UPDATE statement Format 3, Format 4 (Update row using an array)

Function

Multiple update operations can be performed by specifying embedded variables in an array format.

Format 3

In a given table, updates the values of rows meeting specified search conditions multiple times by column.

Format 4

In a table with a FIX specification, updates the values of rows meeting specified search conditions multiple times by column.

Privileges

A user who has the UPDATE privilege for a table can update the column values in that table.

However, if a subquery is specified in the search condition, the user needs the SELECT privilege for the table for which the subquery is specified.

Format 3: Specifying an embedded variable array to update rows in a table multiple times by column

```
FOR : embedded-variable
UPDATE [authorization-identifier.]table-identifier
       [IN (RDAREA-name-specification)][[AS] correlation-name]
       [used-index-SQL-optimization-specification]
       SET {update-object = update-value
           |(update-object, update-object[, update-object]) = row-subquery}
           [, {update-object = update-value
              |(update-object, update-object[, update-object]...) = row-subquery}]...
[WHERE search-condition]
[WITH ROLLBACK]

update-object:: = {column-name|component-specification | column-name [{subscript|*}]}
```

Format 4: Specifying an embedded variable array to update rows in a table with a FIX specification multiple times by row

```
FOR :embedded-variable
UPDATE [authorization-identifier.]table-identifier
       [IN (RDAREA-name-specification)][ [AS] correlation-name]
       [used-index-SQL-optimization-specification]
       SET ROW = row-update-value
      [WHERE search-condition]
      [WITH ROLLBACK]
```

Operands

See Format 1 for details about the operands and operand rules other than FOR, IN (*RDAREA-name-specification*), the SET clause, and search conditions.

Format 3 operands and operand rules

- SET clause

Updates the value of a column.

update-value

The following items can be specified in *update-value* as a column value in update value:

- : *embedded-variable-array* [: *indicator-variable-array*]
- Column name
- Literal
- Value expressions (including arithmetic and concatenation operations)
- Scalar subquery
- USER
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP[(*p*)]
- NULL (represents the null value)
- DEFAULT (represents the default for the column to be updated)
- ? parameter

Embedded variables not in the array format cannot be specified in *update-value*.

Format 4 operands and operand rules

Specifying the ROW operand is subject to the following rules:

1. Updating by row can be used only on base tables of the FIX attribute. ROW refers to an entire row. Specifying ROW causes HiRDB to treat the entire row as a single data item and update it with data from one area. The data type of the data to be updated should be the ROW type, irrespective of the data types of columns. For the ROW type, a variable corresponding to CHAR (*n*) [where *n* is the row length] or a structure of the same length can be specified, provided that the structure does not contain a boundary alignment gap. The data length should be equal to the row length (the sum of the data lengths of the columns).
2. The platform on which the UAP runs and the platform on which the HiRDB server runs should have the same endian. The ROW option cannot be used between different endians. For example, if ROW is used in a Windows edition UAP, the HiRDB server should also use the Windows edition of the same endian.

- SET clause

Updates the value of a column.

- SET ROW = *row-update-value*

ROW

Specify this option when updating data by row.

row-update-value

The following items can be specified as row update values corresponding to ROW:

: *embedded-variable-array* [: *indicator-variable-array*]

Operands and operand rules common to Formats 3 and 4

- FOR :*embedded-variable*

Specifies the embedded variable in which the number of times update operations are performed using an embedded variable array is assigned. An embedded variable of the SMALLINT type should be specified. The allowable range is from 1 to 4,096, no greater than the number of elements in the embedded variable array or in the indicator variable array. Zero and negative values are not allowed. An out-of-range value can produce a run-time error.

embedded-variable-array

This is the embedded variable declared in the array format. Specify an array variable to specify an update value other than the NULL value. Values to be used for updating rows should be assigned to the elements of the variable array. If a value to be used for updating contains the NULL value, both *embedded-variable-array* and *indicator-variable-array* should be specified.

indicator-variable-array

This is the indicator variable declared in the array format. Values indicating whether the values of the elements in *embedded-variable-array* are the NULL value should be assigned to the corresponding elements in *indicator-variable-array*. For values that can be assigned, see *1.6.5 Setting a value for an indicator variable*.

- [IN (*RDAREA-name-specification*)]

If *RDAREA-name-specification* is for an embedded variable, it must be an embedded variable array. Specifying an embedded variable that is not an array causes an error. See Format 1 for details about other operand rules.

- [WHERE *search-condition*]

The default is to update all rows in the specified table.

search-condition

Specifies the criteria by which the rows to be updated are selected. All rows that meet the search condition are updated. If an embedded variable is used in *search-condition*, an embedded variable not in the array format cannot be specified.

Common rules

Format 3 rules

1. The data type of *embedded-variable-array* should be the data type of the corresponding column or a convertible data type.
2. The UPDATE statement using an array cannot handle the BLOB type, the BINARY type with a maximum length of 32,001 bytes or greater, or the abstract data type.
3. The UPDATE statement using an array cannot update more than one element in a repetition column.
4. When a column of INTEGER or SMALLINT data type is updated with fixed-point or floating-point data, any fractional part (below the decimal point) is truncated before updating. Also, when fixed-point data is inserted into a DECIMAL type column, any digits below the scaling for the column are truncated.
5. Character data longer than the length that was specified when the table was defined or BINARY data cannot be entered as a value of the column to be updated.
6. Numeric data outside the range of the definition area cannot be entered as a value of the column to be updated.
7. If the data used to update a column of a fixed-length character string (including national character strings and mixed character strings) is shorter than the column length, the data is inserted left-justified, and the remainder of the column is

blank-filled.

8. The SET clause can be specified only once per SQL statement.
9. A maximum of 30,000 items can be specified in a SET clause.
10. The UPDATE statement cannot be executed on falsification-prevented tables. Updatable columns, however, can be updated.

    The following table indicates the updatability of column values when UPDATE ONLY FROM NULL is specified for a falsification-prevented table:

Column value before update	Column value after update	Updatability
Null value	Null value	Y
Null value	Non-null value	Y
Non-null value	Null value	N
Non-null value	Non-null value#	N

Legend:

Y: Updatable.

N: Not updatable.

Note

For a repetition column, only updating by column from the null value (a value for which the number of elements is 0) without a subscript specification.

In a falsification-prevented table, an error may result if, of the rows that meet specified search conditions, the column for which UPDATE ONLY FROM NULL contains rows with non-null values.

#: Includes the same value as the pre-update value.

11. An error occurs if WITHOUT ROLLBACK is specified for the table containing the columns to be updated and an index is defined for the column to be updated.
12. If the index constituent column of a table with a WITHOUT ROLLBACK specification is the update object column, updating cannot be executed if the update value for the index constituent column is other than the pre-update value. For details, see the rules on WITHOUT ROLLBACK in *CREATE TABLE (Define table)* in Chapter *3*.
13. If the table to be updated is a shared table and if an index is defined for the column to be updated, the LOCK statement should be executed in the lock mode on the shared table before updating. An attempt to update without executing the LOCK

statement the value of a column for which a shared table is defined can cause an error. However, the LOCK statement need not be executed if the value of the column for which an index is defined is not changed. For details about updating a shared table, see the *HiRDB Version 9 Installation and Design Guide*. For objects of locking for the execution of the LOCK statement for a shared table, see the notes in *LOCK statement (Lock control on tables)*.

14. If the table to be updated meets all of the following conditions and if columns in the table are to be updated by DEFAULT, the LOCK statement in the locking mode should be executed on the shared table before updating; an attempt to update the table without executing the LOCK statement can cause an error:
    - The table to be updated is a shared table.
    - The column to be updated is of the timestamp data type.
    - When the table was defined, CURRENT_TIMESTAMP USING BES was specified as the default.

Format 4 rules

1. If a table with a WITHOUT ROLLBACK specification for which an index is defined is specified in *table-identifier*, updating cannot be executed if the update value for the index constituent column is other than the pre-update value. For details, see the rules on WITHOUT ROLLBACK in *CREATE TABLE (Define table)* in Chapter *3*.
2. If the table to be updated is a shared table and if an index is defined for one of the columns, the LOCK statement should be executed in the lock mode on the shared table before updating. An attempt to update the shared table by row (ROW specification) without executing the LOCK statement can cause an error. However, the LOCK statement need not be executed if the value of the column for which an index is defined is not changed. For details about updating a shared table, see the *HiRDB Version 9 Installation and Design Guide*. For objects of locking for the execution of the LOCK statement on a shared table, see the notes in *LOCK statement (Lock control on tables)*.
3. Because the UPDATE statement by row updates the values of all columns, the UPDATE statement by row cannot be executed on a falsification-prevented table that contains even a single non-updatable column.
4. The UPDATE statement cannot be executed by row for tables that contain columns of character data that do not use the default character set.

Rules common to Formats 3 and 4

1. One or more variable arrays should be specified in a clause other than the FOR clause. A failure to specify a variable array can cause an error.
2. Specifying an embedded variable not in the array format in a clause other than the

FOR clause can cause an error.

3. The number of elements in the embedded variable array or the indicator variable array should be in the 1 to 4,096 range. Specifying an out-of-range value can cause an error. Such a number should be greater than or equal to the maximum number of elements specified in FOR :*embedded-variable*.
4. The elements that are updated in one updating operation in an embedded variable array or indicator variable array are elements that have the same element numbers.
5. Evaluation is performed sequentially from the first element of an array.
6. If more than one updating operation is performed, the object of updating is an update object after the previous updating using array elements was performed.
7. The total number of updated rows, including any rows updated in duplicate, is assigned to the SQLERRD[2] area of the SQL communications area.
8. Because it includes embedded variable arrays and indicator variable arrays, UPDATE statement Format 3 cannot be preprocessed by the PREPARE statement. For details about dynamic execution, see *EXECUTE statement Format 2 (Execute an SQL statement using an array)*.
9. An UPDATE statement using an array cannot be used in a procedure.
10. If an event that requires warning occurs in any of the rows to be updated, warning information is assigned to the SQLWARN flag of the SQL communications area.
11. If an error occurs in any of the rows to be updated, the transaction is rolled back.
12. The UPDATE statement cannot be specified for a falsification-prevented table. If a falsification-prevented table is defined, the rules on Formats 3 and 4 must be observed.
13. If *RDAREA-name-specification* is specified, you cannot use an index in which the number of partitions is different from the number of partitions of the table. When defining an index for queries that specify *RDAREA-name-specification*, define an index that has the same number of partitions as the number of partitions of the table.

Rules on referential constraints

1. For the rules on updating the primary key of a referenced table and the foreign key of a referencing table, or updating a referenced table or referencing table row by row, see the explanation of referencing actions in *CREATE TABLE (Define table)* in Chapter *3*.
2. For the updating of values of a foreign key constituent column in a referenced table for which constraint operations are defined in RESTRICT, the referencing table is searched to determine whether the updating value is included in the value

of a foreign key constituent column in the referencing table. The data guarantee level during the search through the referencing table assumes the share mode. For this reason, if during the deletion of rows in the referenced table for which constraint operations are defined in RESTRICT, operations are performed on the referencing table by another transaction, the row deletion action goes into a wait state until the transaction is settled.

3. If any combination of the following conditions occurs, data incompatibility can occur between the referenced table and the referencing table subject to referential constraints; such incompatibility can also occur regardless of whether the constraint operation is RESTRICT or CASCADE:
   - The transaction that deletes rows in the referencing table is different from the transaction that updates or deletes rows in the referenced table.
   - The above two transactions are executed simultaneously.
   - The value of the primary key constituent column deleted from the referencing table is the same as the value of the foreign key constituent column that is either updated in or deleted from the referenced table.
   - Either the transaction that deletes rows from the referencing table is committed or the transaction that updates or deletes rows in the referenced table is rolled back.
4. If a combination of the following conditions occurs, a deadlock can occur between the transaction that manipulates the referenced table and the transaction that manipulates the referencing table. A deadlock can also occur if the constraint operation is either RESTRICT or CASCADE.
   - The transaction that updates the rows in the referencing table is different from the transaction that updates the rows in the referenced table.
   - The above two transactions are executed simultaneously.
   - The value of the foreign key constituent column updated in the referencing table is the same as the value of the primary key constituent column that is deleted from the referenced table.

Examples

1. The values of the product code (PCODE) and quantity in stock (SQTY), which are stored in an array variable written in C, are listed in the tables below. The values of the quantity in stock (SQTY) are to be updated using the UPDATE statement Format 1.

   Table 4-7: Product code and inventory level stored in the table (before updating)

Product code	Pre-update inventory level
'101M'	40

Product code	Pre-update inventory level
'101L'	70
'201M'	15
'202M'	28
'302S'	7

Table 4-8: Product code and inventory level (assigned to the embedded variable array) in the row to be updated

Product code	Updated inventory level
'101M'	35
'101L'	62
'201M'	13
'202M'	10
'302S'	6

Table 4-9: Product code and inventory level stored in the table (after update)

Product code	Updated inventory level
'101M'	35
'101L'	62
'201M'	13
'202M'	10
'302S'	6

```
EXEC SQL BEGIN DECLARE SECTION;
      short   XUPDATE_NUM;
      char    XPCODE[5][5];
      short   IPCODE[5];
      long    XSQTY[5];
      short   ISQTY[5];
EXEC SQL END DECLARE SECTION;
    . . . Assign values to elements in the array variables . . .
        Assign{'101M','101L','201M','202M','302S'} to XPCODE
         Assign{35,62,13,10,6} to XSQTY
XUPDATE_NUM = 5;
```

```
EXEC SQL FOR :XUPDATE_NUM
UPDATE STOCK SET SQTY = :XSQTY:ISQTY
WHERE PCODE = :XPCODE:IPCODE;
```

2. Use `UPDATE` statement Format 2 to update an entire row in the inventory table (`STOCK`) by value of the product code (`PCODE`) assigned to an array variable in the C language, with the contents on the embedded variable array (`XROW`) in batch.

```
XUPDATE_NUM = 5;
EXEC SQL FOR :XUPDATE_NUM
UPDATE STOCK SET ROW = :XROW
WHERE PCODE = :XPCODE:IPCODE;
```

Preparable dynamic UPDATE statement: locating Format 1 (Update data using a preprocessable cursor)

Function

Updates a specified column in the row pointed to by the cursor in the table. This statement is used when updating is to be executed by the EXECUTE statement after preprocessing by a PREPARE statement or when preprocessing and execution are to be performed in a single operation using the EXECUTE IMMEDIATE statement.

Privileges

A user who has the UPDATE privilege for a table can update the column values in that table.

Format 1: Updating rows in a table by column, using a (preprocessable) cursor

```
UPDATE [[authorization-identifier.]table-identifier
       [IN (RDAREA-name-specification)][[AS] correlation-name]
       [used-index-SQL-optimization-specification] ]
       {SET {update-object = update-value
           | (update-object, update-object[, update-object]) = row-subquery}
           [, {update-object = update-value
              | (update-object, update-object[, update-object]...) = row-subquery} ]...
        | ADD repetition-column-name [{subscript | *}]
              = {ARRAY [element-value[, element-value]...]
                | ?-parameter}
             [, repetition-column-name [{subscript | * } ]
                  = {ARRAY [element-value [, element-value]...]
                    | ?-parameter}]...
        | DELETE repetition-column-name [{subscript | * }]
             [, repetition-column-name [ {subscript | * } ] ]...}
WHERE CURRENT OF GLOBAL cursor-name
[WITH ROLLBACK]

update-object:: = {column-name|component-specification|column-name [{subscript | * } ] }
```

Operands

For operands and operand rules other than *update-value* in the SET clause, ARRAY [*element-value* [, *element-value*] ...], and WHERE CURRENT OF GLOBAL *cursor-name*, see *UPDATE statement Format 1 (Update data)* in this chapter.

- SET clause

update-value

The following items can be specified in *update-value*:

- Column name
- Component specification
- Literal
- Value expressions (including arithmetic and concatenation operations)
- Scalar subquery
- `USER`
- `CURRENT_DATE`
- `CURRENT_TIME`
- `CURRENT_TIMESTAMP[(`*p*`)]`
- `NULL` (represents the null value)
- `DEFAULT` (represents the default for the column to be updated)
- `?` parameter
- `ARRAY` [*element-value* [`,` *element-value*]...][#]

#: The following items can be specified in *element-value*:

- A column name other than a repetition column
- Subscripted repetition column
- Literal
- Value expressions (including arithmetic and concatenation operations)
- Scalar subquery
- `USER`
- `CURRENT_DATE`
- `CURRENT_TIME`
- `CURRENT_TIMESTAMP[(`*p*`)]`
- `NULL` (represents the null value)
- `DEFAULT`
- `?` parameter

Rules on update value

1. An update value that includes an embedded variable, an indicator variable, an SQL variable, or an SQL parameter cannot be specified.

2. Other update values are subject to the rules on update values of UPDATE statement Format 1.

Rules on element value

1. An element value that includes an embedded variable, an indicator variable, an SQL variable, or an SQL parameter cannot be specified.
2. Other element values are subject to the rules on element values of UPDATE statement Format 1.

ARRAY [*element-value* [, *element-value*]...]

The following items can be specified as *element-value*:

- A column name other than a repetition column
- Subscripted repetition column
- Literal
- Value expressions (including arithmetic and concatenation operations)
- Scalar subquery
- USER
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP [(*p*)]
- NULL (represents the null value)
- DEFAULT
- ? parameter

For the rules on element values, see *Rules on element values* under the SET clause for *UPDATE statement Format 1 (Update data)* in this chapter.

■ WHERE CURRENT OF GLOBAL *cursor-name*

GLOBAL

Specifies GLOBAL as the scope of *cursor-name*.

cursor-name

This operand specifies the name of the cursor that points to the row to be updated.

The cursor specified in *cursor-name* is one identified by the extended cursor name that was specified in the ALLOCATE CURSOR statement. However, a result set cursor cannot be specified.

The cursor specified in *cursor-name* at run time must be open and positioned on

the row to be updated.

The position of the cursor specified in *cursor-name* remains unchanged after the execution of the UPDATE statement. If a row after the updated row is to be updated, the FETCH statement should be executed on the cursor to move it.

If *extended-cursor-name* is specified, an extended cursor must be specified for queries that specify the FOR UPDATE clause. For details about the FOR UPDATE clause, see the FOR UPDATE clause under *Operands* in *Dynamic SELECT statement Format 1 (Retrieve dynamically)* in this chapter.

Common rules

1. After preprocessing is performed by the PREPARE statement, an EXECUTE statement is used to execute, or an EXECUTE IMMEDIATE statement is used to preprocess and execute in a single operation.
2. When omitting a table identifier, make sure that, before preprocessing is performed, the ALLOCATE CURSOR statement is used to allocate the cursor to the dynamic SELECT statement. In this operation, the table that is the object of retrieval specified in the dynamic SELECT statement to which the cursor is allocated is assumed. When specifying a table identifier, it is not necessary that the cursor be allocated to the dynamic SELECT statement before the preprocessing.
3. The common rules on UPDATE statement Format 1 are applicable to the other common rules.

Rules on referential constraints

1. Rules on referential constraints for UPDATE statement Format 1 apply.

Examples

1. Dynamically execute the SQL statement that updates the unit price (PRICE) for the row pointed to by the cursor (*cr* (*scope*: GLOBAL, *value*: CR1)) in the inventory table (STOCK) to a 10% discount.

```
Set any name as the embedded variable :sel
PREPARE GLOBAL :sel FROM 'SELECT * FROM STOCK FOR UPDATE'
Assign CR1 to the embedded variable cr
ALLOCATE GLOBAL :cr CURSOR FOR GLOBAL :sel
PREPARE PRE1 FOR
    'UPDATE SET PRICE = A value equal to a 10% discount on the unit price
WHERE CURRENT OF GLOBAL CR1'
OPEN GLOBAL :cr
FETCH GLOBAL :cr INTO Name of the variable into which columns are fetched
EXECUTE PRE1
CLOSE GLOBAL :cr
DEALLOCATE PREPARE GLOBAL :sel
```

Preparable dynamic UPDATE statement: locating Format 2 (Update data using a preprocessable cursor)

Function

Updates a specified column in the row pointed to by the cursor in the table with a FIX specification. This statement is used when updating is to be executed by the EXECUTE statement after preprocessing by a PREPARE statement or when preprocessing and execution are to be performed in a single operation using the EXECUTE IMMEDIATE statement.

Privileges

A user who has the UPDATE privilege for a table can update the column values in that table.

Format 2: Updating rows by row in a table with a FIX specification, using a (preprocessable) cursor

```
UPDATE [ [authorization-identifier.]table-identifier
      [IN (RDAREA-name-specification)][ [AS] correlation-name]
      [used-index-SQL-optimization-specification] ]
       SET ROW = row-update-value
      WHERE CURRENT OF GLOBAL cursor-name
      [WITH ROLLBACK]
```

Operands

For operands other than *row-update-value* in the SET clause and WHERE CURRENT OF GLOBAL *cursor-name*, and for operand rules, see *UPDATE statement Format 2*.

For details about the WHERE CURRENT OF GLOBAL *cursor-name* operand and about operand rules, see *Preparable dynamic UPDATE statement: locating Format 1*.

- SET clause

row-update-value

The following item can be specified as a row update value associated with ROW:

- ? parameter

Common rules

1. After preprocessing is performed by the PREPARE statement, an EXECUTE statement is used to execute, or an EXECUTE IMMEDIATE statement is used to preprocess and execute in a single operation.

2. When omitting a table identifier, make sure that, before preprocessing is performed, the `ALLOCATE CURSOR` statement is used to allocate the cursor to the dynamic `SELECT` statement. In this operation, the table that is the object of retrieval specified in the dynamic `SELECT` statement to which the cursor is allocated is assumed. When specifying a table identifier, it is not necessary that the cursor be allocated to the dynamic `SELECT` statement before the preprocessing.
3. The common rules on `UPDATE` statement Format 2 are applicable to the other common rules.

Rules on referential constraints

1. Rules on referential constraints for `UPDATE` statement Format 2 apply.

Notes

1. See the notes on `UPDATE` statement Format 2.

Example

1. Dynamically execute the SQL statement that updates in a single operation the data in the row pointed to by the cursor (*cr* (*value*: `CR1`) in the inventory table (`STOCK`) with the contents of the embedded variable (`XROW`).

```
PREPARE GLOBAL :sel FOR 'SELECT * FROM STOCK FOR UPDATE'
Assigns CR1 to the embedded variable cr
ALLOCATE GLOBAL :cr CURSOR FOR GLOBAL :sel
PREPARE PRE1 FOR
    'UPDATE SET ROW = ? WHERE CURRENT OF GLOBAL CR1'
OPEN GLOBAL :cr
FETCH GLOBAL :cr INTO Name of the variable into which columns are fetched
EXECUTE PRE1 USING :XROW
CLOSE GLOBAL :cr
DEALLOCATE PREPARE GLOBAL :sel
```

Assignment statement Format 1 (Assign a value to an SQL variable or SQL parameter)

Function

Assigns a value to an SQL variable or SQL parameter.

Privileges

None.

Format: Assigning a value to an SQL variable or an SQL parameter

```
SET assign-to = assignment-value

assign-to::= { [statement-label.]SQL-variable-name
             | [[authorization-identifier.]routine-identifier.]SQL-parameter-name
             | [statement-label.]SQL-variable-name..attribute-name[..attribute-name]...
             | [[authorization-identifier.]routine-identifier.]SQL-parameter-name..attribute-name
[..attribute-name]...}
assignment-value::= {value-expression|NULL|DEFAULT}
```

Operands

- *assign-to*::= { [*statement-label*.] *SQL-variable-name*

  | [[*authorization-identifier*.] *routine-identifier*.] *SQL-parameter-name*

  | [*statement-label*.] *SQL-variable-name*.. *attribute-name* [. . *attribute-name*]...

  | [[*authorization-identifier*.] *routine-identifier*.] *SQL-parameter-name* .. *attribute-name* [.. *attribute-name*]...}

  Specifies the SQL variable or SQL parameter into which a value is assigned, or the attribute name of an SQL variable or the attribute name of an SQL parameter.

  If *authorization-identifier* is specified in an SQL procedure statement in a public procedure definition or a public function definition, specify upper-case `PUBLIC` enclosed in double quotation marks (`"`) as the authorization identifier.

- *assignment-value*::= {*value-expression* | `NULL` | `DEFAULT`}

  Specifies the value to be assigned.

Common rules

1. In *assign-to*, an SQL parameter mode for which the input/output mode (parameter mode) specified in *SQL-routine* is `IN` cannot be specified. The SQL parameter

name of a function cannot be specified.

2. The data type of *assign-to* must be compatible with the data type of *assignment-value*.
3. If the data type of *assign-to* is different from that of *assignment-value*, a type conversion is performed. If the two data types are the same, the assignment value is directly assigned to *assign-to*.
4. If *assignment-value* is character data, *assign-to* and *assignment-value* must use the same character set. However, if *assignment-value* is a string constant and if *assign-to* and *assignment-value* use different character sets, *assignment-value* will automatically be converted to the character set of *assign-to*.
5. A subquery cannot be specified in a value expression specified in *assignment-value*.
6. IF `DEFAULT` is specified in *assignment-value*, the default for the SQL value to be assigned is assigned. For details about declaring the default for an SQL variable, see *Compound statement (Execute multiple statements)* in Chapter 7. If an SQL parameter is specified in *assign-to* and `DEFAULT` is specified in *assignment-value*, the null value is assigned to the SQL parameter for the target of assignment. If a column name qualified with an old or new value correlation name is specified in the assignment target for the assignment statement (format 1) specified in a trigger SQL statement, and `DEFAULT` is specified in *assignment-value*, the default for the column in the assignment target is assigned. However, if the default for the column qualified with the old and new value correlation name of the assignment target is `CURRENT_TIMESTAMP` for which `USING BES` is specified, `DEFAULT` cannot be specified in *assignment-value*. For details about the trigger SQL statement, see *CREATE TRIGGER (Define a trigger)* in Chapter 3.

Notes

1. Format 1 of the assignment statement can only be specified in an SQL routine. For specifying an assignment statement in a routine other than an SQL routine, specify Format 2 of the assignment statement.

Assignment statement Format 2 (Assign a value to an embedded variable or a ? parameter)

Function

Assigns a value to an embedded variable or a ? parameter.

Privileges

None.

Format: Embedded-variable, or assigning to a ? parameter

```
SET  assign-to = assignment-value
assign-to::= {:embedded-variable [:indicator-variable] | ?-parameter}
assignment-value::= {:embedded-variable [:indicator-variable] AS data-type
        | ?-parameter AS data-type
        | LENGTH (value-expression)
        | SUBSTR (value-expression-1, value-expression-2[, value-expression-3])
        |POSITION(value-expression-1 IN value-expression-2 [FROM value-expression-3])}
```

Operands

- *assign-to*::= {:*embedded-variable* [:*indicator-variable*] | *?-parameter*}

Specifies the embedded variable or the ? parameter into which a value is to be assigned.

- *assignment-value*::= {:*embedded-variable* [:*indicator-variable*] AS *data-type*

  | *?-parameter* AS *data-type*

  | LENGTH (*value-expression*)

  | SUBSTR (*value-expression-1*, *value-expression-2* [, *value-expression-3*])

  | POSITION (*value-expression-1* IN *value-expression-2* [FROM *value-expression-3*]) }

Specifies the value to be assigned.

:*embedded-variable* [:*indicator-variable*] AS *data-type*

The only data types that can be specified are the BLOB and the BINARY types. In the AS clause, specify the data type of the embedded variable. An error can occur if, in this operation, the actual length of the data (for a locator, the actual length of the data allocated to the locator) given by *embedded-variable* is greater than the maximum length of the data type specified in the AS clause.

?-parameter AS *data-type*

The only data types that can be specified are the BLOB and the BINARY types. In the AS clause, specify the data type of *?-parameter*. An error can occur if, in this operation, the actual length of the data (for a locator, the actual length of the data allocated to the locator) given by *?-parameter* is greater than the maximum length of the data type specified in the AS clause.

LENGTH (*value-expression*)

SUBSTR (*value-expression-1*, *value-expression-2*, [*value-expression-3*])

POSITION (*value-expression-1* IN *value-expression-2* [FROM *value-expression-3*])

The only data types that can be specified in *value-expression* for the scalar function LENGTH, in *value-expression-1* for the scalar function SUBSTR, or in *value-expression-2* for the scalar function POSITION are the BLOB and the BINARY types. The only items that can be specified are *embedded-variable* and *?-parameter* For other specification methods, see the rules on the individual scalar functions.

Common rules

1. The data type of the assignment target must be compatible with the data type of the value being assigned.
2. If the data type of the assignment target is different from that of the value being assigned, a type conversion is performed. If they are of the same data type, the assigned value is directly assigned to the assignment target.
3. The following items cannot be specified in a value expression specified as an assignment value:
   - Column specification
   - Component specification
   - Scalar subquery

Notes

1. Format 2 of the assignment statement cannot be specified in an SQL routine. For specifying an assignment statement in an SQL routine, Format 1 of the assignment statement should be used.

Examples

Assign a part of the BLOB data allocated to the embedded variable (XLOC) of the BLOB locator to an embedded variable (XDATA) of the BLOB type.

```
SET :XDATA = SUBSTR(:XLOC AS BLOB(1M), 100, 1024)
```

Chapter

5. Control SQL

This chapter explains the syntax and structure of the control SQL.

General rules

Types and functions of the control SQL

The control SQL connects a UAP to HiRDB, disconnects a UAP from HiRDB, and performs lock control on tables.

The following table lists the types and functions of the control SQL.

Table 5-1: Types and functions of the control SQL

Type	Function
`CALL COMMAND` statement (Execute command or utility)	Executes a HiRDB command or utility.
`COMMIT` statement (Terminate transaction normally)	Terminates the current transaction normally, sets synchronization points, generates one unit of commitment, and puts into effect the databases updates performed by the transaction.
`CONNECT` statement (Connect UAP to HiRDB)	Passes the authorization identifier and password to HiRDB, enabling the UAP to use HiRDB.
`DISCONNECT` statement (Disconnect UAP from HiRDB)	Terminates the current transaction normally, sets synchronization points, generates one unit of commitment, and disconnects the UAP from HiRDB.
`LOCK` statement (Lock control on tables)	Performs exclusive lock on specified tables.
`ROLLBACK` statement (Cancel transaction)	Cancels the current transaction and nullifies the database updating performed by the transaction.
`SET SESSION AUTHORIZATION` statement (Change connected user)	Changes a connected user by reporting an authorization identifier and password to HiRDB.

Notes on specifying the control SQL

The transaction is the unit of logical work; it is the basic unit by which HiRDB recovers data and performs concurrent execution of UAPs. To improve the system's concurrent execution capability, the amount of time required by a transaction should be as short as possible. Especially when large quantities of data are updated, the duration of lock control, the number of resources subject to lock control, and the amount of log data generated must be taken into consideration. In some cases, it may be necessary to divide a transaction into several subunits.

Notes on X/open-compliant UAPs running under OLTP

The following SQL statements cannot be used in X/Open-compliant UAPs running under OLTP:

- `COMMIT` statement
- `CONNECT` statement
- `DISCONNECT` statement
- `ROLLBACK` statement

CALL COMMAND statement (Execute command or utility)

Function

Executes a HiRDB command or utility and obtains the execution results (standard output, standard error output, and the return value).

Privileges

Any user whose authorization identifier is specified in the system common definition `pd_sql_command_exec_users` operand.

Format

```
  CALL COMMAND {:embedded-variable-1|?-parameter-1|literal-1}
       [WITH {:embedded-variable-2|?-parameter-2|literal-2}[,
{:embedded-variable-2|?-parameter-2|literal-2}]...]
       [INPUT {:embedded-variable-3|?-parameter-3|literal-3}]
       [OUTPUT TO {:embedded-variable-4 :indicator-variable-1|?-parameter-4}]
       [ERROR TO {:embedded-variable-5 :indicator-variable-2|?-parameter-5}]
       [RETURN CODE TO {:embedded-variable-6|?-parameter-6}]
       [ENVIRONMENT {:embedded-variable-7|?-parameter-7|literal-4}]
       [SERVER {:embedded-variable-8|?-parameter-8|literal-5}]
```

Operands

- `:`*embedded-variable-1*|*?-parameter-1*|*literal-1*

Specifies the embedded variable, `?` parameter, or literal containing the name of the command or utility to be executed. The name of the command or utility cannot be specified using an absolute or relative path name. The data type of the embedded variable or `?` parameter must be a variable-length string with a maximum length of 30 bytes. Alternatively, if `UTF16` is specified as the character set of the embedded variable or `?` parameter, it must be a variable-length character string with a maximum length of 60 bytes. A literal must be no more than 30 bytes long.

- `WITH {:`*embedded-variable-2*|*?-parameter-2*|*literal-2*`}[,`
  `{:`*embedded-variable-2*|*?-parameter-2*|*literal-2*`}]...`

Specifies embedded variables, `?` parameters, or literals containing arguments to be passed to the command or utility. If there is a long list of arguments that cannot be specified in a single embedded variable, `?` parameter, or literal, they can be specified using multiple embedded variables, `?` parameters, or literals. In this case, the strings are concatenated in the order in which they are specified. To pass more than one argument, separate them with semicolons. If the argument itself is a semicolon, specify two consecutive semicolons. The data type of an embedded variable or `?` parameter must be a variable-length string with a maximum length of 32,000 bytes. The length

of a literal can be no more than 32,000 bytes. If an argument contains a path it must be an absolute path.

- `INPUT` { `:`*embedded-variable-3* | *?-parameter-3* | *literal-3* }

Specifies an embedded variable, `?` parameter, or literal that contains the contents of the standard input to be passed to the command or utility to execute. The data type of an embedded variable or `?` parameter must be a variable-length string with a maximum length of 32,000 bytes. The length of a literal can be no more than 32,000 bytes. For a command or utility that requires password input, the `INPUT` clause cannot be used to pass the password.

- `OUTPUT TO` { `:`*embedded-variable-4* `:`*indicator-variable-1* | *?-parameter-4* }

Specifies an embedded variable or a `?` parameter to be used to store the contents of the standard output of the command or utility being executed. The data type of the embedded variable or `?` parameter must be a binary data string (`BLOB`) with a maximum length of two gigabytes. If the standard output is longer than the maximum length of the embedded variable or `?` parameter, the information from the beginning of the output up to the maximum length will be stored, and any further information will be truncated. In this case, an indicator variable will be used to store the length of the standard output, indicating that the output was truncated. Always make sure to specify an indicator variable whenever embedded variables are used.

- `ERROR TO` { `:`*embedded-variable-5* `:`*indicator-variable-2* | *?-parameter-5* }

Specifies an embedded variable or `?` parameter to be used to store the contents of the standard error output of the command or utility being executed. The data type of the embedded variable and `?` parameter must be a binary data string (`BLOB`) with a maximum length of two gigabytes. If the standard error output is longer than the maximum length of the embedded variable or `?` parameter, the information from the beginning of the output up to the maximum length will be stored, and any further information will be truncated. In this case, an indicator variable will be used to store the length of the standard error, indicating that the output was truncated. Always make sure to specify an indicator variable whenever embedded variables are used.

- `RETURN CODE TO` { `:`*embedded-variable-6* | *?-parameter-6* }

Specifies the embedded variable or `?` parameter to be used to store the return value of the command or utility being executed. The data type of the embedded variable or `?` parameter must be an integer type.

- `ENVIRONMENT` { `:`*embedded-variable-7* | *?-parameter-7* | *literal-4* }

Specifies the embedded variable, `?` parameter, or literal that will contain the client environment definition when the command or utility is executed. The data type of an embedded variable or `?` parameter must be a variable-length string with a maximum length of 32,000 bytes. The length of a literal can be no more than 32,000 bytes.

A client environment definition is specified in the format

client-environment-definition-name = *value*. To specify multiple client environment definitions, separate them with semicolons. If the argument itself is a semicolon, specify two consecutive semicolons.

The default client environment definition for the environment in which the command or utility is to be executed inherits the client environment definition (PDDIR, PDCONFPATH, and so on) set when HiRDB was started. It also inherits the client environment definition that is set using the putenv format in the system definition. For details about the client environment definitions that you can specify, see the *HiRDB Version 9 UAP Development Guide*.

- SERVER { :*embedded-variable-8* | *?-parameter-8* | *literal-5* }

Specifies an embedded variable, ? parameter, or literal that contains the name of the server on which the command or utility is to be executed. The server name refers to the server name specified in the -s option of the pdstart operand in the system common definition. The data type of an embedded variable or ? parameter must be a variable-length string with a maximum length of eight bytes. However, if UTF16 is specified as the character set of the embedded variable or ? parameter, it must be a variable-length string with a maximum length of 16 bytes. The maximum length of a literal is eight bytes. To specify the System Manager Unit in a HiRDB/Parallel Server environment, specify MGR. If the SERVER clause is omitted, the MGR server name is assumed in a HiRDB/Parallel Server environment, and the SDS server name is assumed in HiRDB/Single Server environment.

Common rules

1. If a transaction is initiated by the command or utility that was executed, it will be independent from the transaction that executed the SQL.
2. A lock-release wait or deadlock may occur between the transaction that executed the SQL and the transaction initiated by the command or utility that was executed.
3. When a command or utility that requests a response is executed, passing the contents of standard input may cause the system to become unresponsive if the command or utility cannot be properly terminated.

   Example:

   Execute a backup (pdfbkup) of the HiRDB file system (/hirdb/ios/db0):

   When the pdfbkup command is run from the command line, it requests user input of G to continue the backup or T to confirm its termination, followed by a carriage return (CR).

```
  % pdfbkup /hirdb/ios/db0 /hirdb/ios/db0.backup
 1605756 19:27:29 SQA2        KFPI21514-Q HiRDB file system
area
  /hirdb/ios/db0 backup to /hirdb/ios/db0.backup.
[G:continue, T:terminate]
```

In the following CALL COMMAND statement, the system becomes unresponsive when the standard input, which does not contain a carriage return, is passed to the pdfbkup command.

```
  EXEC SQL CALL COMMAND 'pdfbkup' WITH '/hirdb/ios/db0;/
hirdb/ios/db0.backup' INPUT 'G';
```

In order to execute the pkfbkup command correctly, the standard input must include a newline in the INPUT clause as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
        char input_data[10];
EXEC SQL END DECLARE SECTION;
sprintf(input_data, "G\n");
EXEC SQL CALL COMMAND 'pdfbkup' WITH '/hirdb/ios/db0;
/hirdb/ios/db0.backup' INPUT :input_data;
```

If the system becomes unresponsive, use the pdcancel command to terminate the HiRDB process (FES for HiRDB/Parallel Server and SDS for HiRDB/Single Server) and the command or utility process.

Notes

1. If a timeout of the command or utility is required, specify the PDCALCMDWAITTIME operand in the client environment definition. If the operand times out during the execution of the command or utility, terminate the control command (pdcmdexec) process, and the executing command or utility process using the pdkill command (or the OS kill command in UNIX).
2. The command or utility may fail to execute depending on the limitations of the execution platform. The following table lists the information to obtain and where that information is stored if the execution of the command or utility fails.

   Table 5-2: Information to obtain and where it is stored if execution of the command or utility fails

Information to obtain	Where the information to obtain is stored
• System function name where the error occurred# • Error code#	*embedded-variable-5* or *?-parameter-5*
Exit code of the OS system function	*embedded-variable-6* or *?-parameter-6*

#

The error code and the system function where the error occurred are output in the format `func=`*aa....aa*`,errno=`*bb....bb* (where `errno` indicates an external integer variable representing the error state).

aa....aa: System function where the error occurred

bb....bb: Error code (empty if the error code cannot be obtained)

Corrective action

Investigate the error code in `errno.h` or in the appropriate OS user documentation, eliminate the cause of the error, and execute again.

One likely reason for an OS system function to fail to execute is that the arguments specified for the command or utility exceed the command line argument length limit defined in the OS. To resolve this issue, adjust the OS to increase the command line argument length limit.

Examples

1. Use the HiRDB system status display (`pdfls`) to obtain a list of HiRDB files in the HiRDB file system area (`/hirdb/ios/rdfiles`). Assign the execution result to the embedded variable (`:OUTPUT`) specified in the `OUTPUT TO` clause. This yields the same result as running `pdfls -H /hirdb/ios/rdfiles` from the command line.

```
  EXEC SQL BEGIN DECLARE SECTION;
          SQL TYPE IS BLOB(1M) OUTPUT;
  EXEC SQL END DECLARE SECTION;
 EXEC SQL CALL COMMAND 'pdfls' WITH '-H;/hirdb/ios/rdfiles'
OUTPUT TO :OUTPUT;
```

2. Use the RDAREA hold command (`pdhold`) to hold the RDAREAs (`RU01`, `RU02`, and `RU03`). Assign the execution result to the embedded variable (`:OUTPUT`) specified in the `OUTPUT TO` clause. This yields the same result as running `pdhold -r RU01,RU02,RU03` from the command line.

```
  EXEC SQL BEGIN DECLARE SECTION;
          SQL TYPE IS BLOB(1M) OUTPUT;
  EXEC SQL END DECLARE SECTION;
  EXEC SQL CALL COMMAND 'pdhold' WITH '-r;RU01,RU02,RU03'
OUTPUT TO :OUTPUT;
```

More than one literal can be specified in the `WITH` clause, as follows:

```
  EXEC SQL BEGIN DECLARE SECTION;
          SQL TYPE IS BLOB(1M) OUTPUT;
  EXEC SQL END DECLARE SECTION;
```

```
  EXEC SQL CALL COMMAND 'pdhold' WITH '-r;RU01',',RU02,RU03'
OUTPUT TO :OUTPUT;
```

A literal and an embedded variable can be specified in the `WITH` clause, as follows:

```
  EXEC SQL BEGIN DECLARE SECTION;
          SQL TYPE IS BLOB(1M) OUTPUT;
          char RDAREAS[100];
  EXEC SQL END DECLARE SECTION;
  sprintf(RDAREAS, "RU01,RU02,RU03");
  EXEC SQL CALL COMMAND 'pdhold' WITH '-r;',:RDAREAS OUTPUT
TO :OUTPUT;
```

COMMIT statement (Terminate transaction normally)

Function

The COMMIT statement terminates the current transaction normally, sets synchronization points, generates one unit of commitment, and puts into effect the database updates performed by the transaction.

Privileges

None.

Format

```
COMMIT [WORK] [RELEASE]
```

Operands

- WORK

This operand has no effect on the normal transaction termination function of the COMMIT statement. The WORK operand is supported for compatibility with JIS standards.

- RELEASE

Specifies that the UAP is to be disconnected from HiRDB after the transaction terminates normally.

Common rules

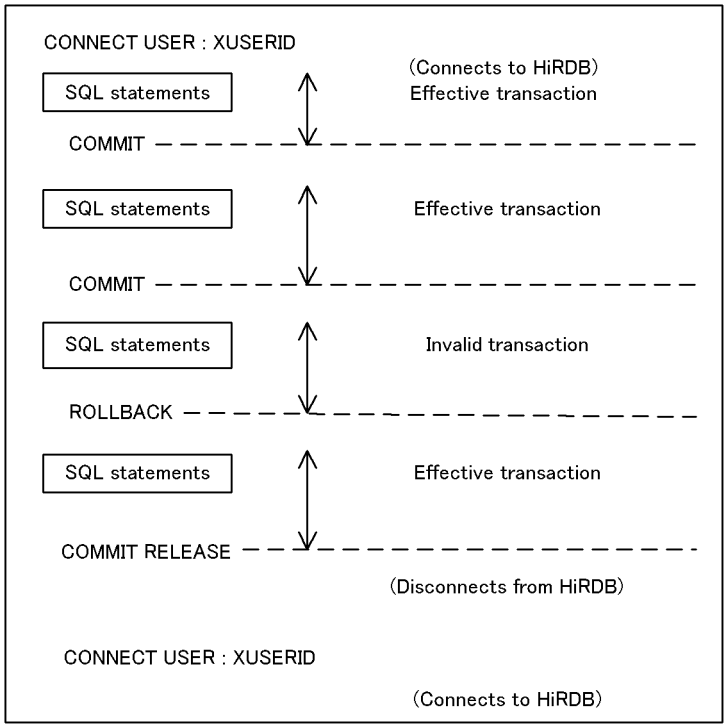
1. Executing a COMMIT statement closes all cursors that are open, except for holdable cursors.
2. All locked resources, except for those being used by holdable cursors, are released.
3. Executing the COMMIT statement nullifies all effective locators.

Notes

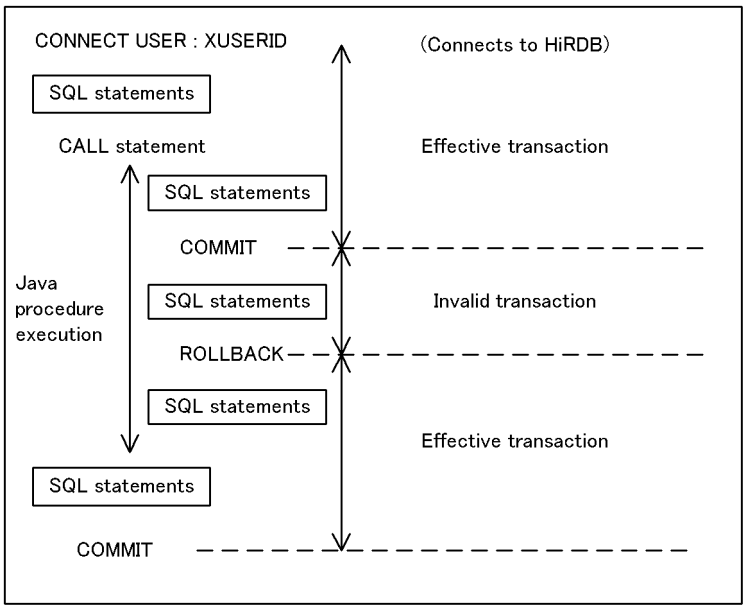
1. The COMMIT statement cannot be specified from an X/Open-compliant UAP executing under OLTP or from a Java procedure. If a procedure is called from a UAP running under OLTP, procedures using the COMMIT statement cannot be executed.
2. When executing the COMMIT statement from a procedure, you cannot specify the RELEASE operand.
3. The COMMIT statement cannot be executed during trigger action.

Examples

1. Enter a COMMIT statement with the RELEASE operand specified to terminate a transaction, then reconnect to HiRDB:



2. Enter a COMMIT statement to terminate a transaction either within a procedure or upon completion of the procedure:


CONNECT USER : XUSERID
(Connects to HiRDB)
SQL statements
CALL statement
Effective transaction
SQL statements
COMMIT
Java
procedure
execution
SQL statements
Invalid transaction
ROLLBACK
SQL statements
Effective transaction
SQL statements
COMMIT

CONNECT statement (Connect a UAP to HiRDB)

Function

The CONNECT statement passes the authorization identifier and password to HiRDB, enabling the UAP to use HiRDB. This operation is referred to as *connecting a UAP to HiRDB*.

Privileges

User with the DBA or CONNECT privilege

A user with the DBA or CONNECT privilege can connect a UAP to HiRDB.

Format

```
CONNECT [{USER:embedded-variable-1[USER:embedded-variable-2]
          | :embedded-variable-1[IDENTIFIED BY :embedded-variable-2]}]
```

Operands

embedded-variable-1

Specifies an embedded variable whose value is an authorization identifier.

If the UAP is written in C, this variable's data type must be a fixed-length character string of no more than 31 bytes that terminates with the null value. If the character string does not end with the null value, the character string whose length is (*area length* - 1) is used as the authorization identifier.

If the UAP is written in COBOL, this operand is of the fixed-length character string data type with a length of 30 bytes or less. The character string does not need to end with the null value.

Character sets other than the default character set cannot be specified.

If the CONNECT statement is specified but all operands after USER are omitted, the UAP is connected to HiRDB using the value set in the PDUSER environment variable as the authorization identifier.

When *embedded-variable-1* is specified as a case-sensitive variable, it must be enclosed in double quotation marks (").

embedded-variable-2

Specifies an embedded variable whose value is a password.

If the UAP is written in C, this variable's data type must be a fixed-length character string of no more than 31 bytes that terminates with the null value. If the character string does not end with the null value, the character string whose length is (*area length* - 1) is used as the password.

If the UAP is written in COBOL or OOCOBOL, this operand is of the fixed-length character string data type with a length of 30 bytes or less. The character string does not need to end with the null value.

Character sets other than the default character set cannot be specified.

For a user who does not have a password, omit the `USING` clause and the `IDENTIFIED BY` clause or specify a character string in the embedded variable.

Common rule

A UAP cannot be reconnected to HiRDB before it has been disconnected from HiRDB with the `DISCONNECT` statement.

Note

The `CONNECT` statement cannot be specified from an X/Open-compliant UAP running under OLTP.

Example

Connect a UAP to HiRDB by passing to HiRDB the user's authorization identifier (embedded variable `USER1`) and password (embedded variable `PSWD1`):

```
CONNECT USER :USER1 USING :PSWD1
```

DISCONNECT statement (Disconnect a UAP from HiRDB)

Function

The DISCONNECT statement terminates the current transaction normally, sets synchronization points, generates one unit of commitment, and disconnects the UAP from HiRDB.

Privileges

None.

Format

```
DISCONNECT
```

Common rules

1. The DISCONNECT statement disconnects the UAP from HiRDB after HiRDB has executed the COMMIT statement. During this process, all holdable cursors are also closed.
2. If the UAP terminates before the DISCONNECT statement is executed, the DISCONNECT statement (in the case of normal termination) is assumed after HiRDB executes the ROLLBACK statement.

Note

The DISCONNECT statement cannot be specified from an X/Open-compliant UAP running under OLTP.

Example

Disconnect a UAP from HiRDB:

```
DISCONNECT
```

LOCK statement (Lock control on tables)

Function

The LOCK statement performs lock control on specified tables. The LOCK statement can reduce the overhead that results from performing lock control in units of rows or key values when there are many rows on which HiRDB performs lock control automatically, when there are many rows from which the table is accessed for performing lock controls in units larger than the key-value unit, or when there are many key values.

Privileges

A user who has the SELECT privilege to a table can lock the table in the shared mode.

A user who has the INSERT, UPDATE, or DELETE privilege to a table can lock the table in the lock mode.

Format

```
LOCK TABLE [authorization-identifier.]table-identifier
            [,[authorization-identifier.]table-identifier]...
            [IN {SHARE | EXCLUSIVE} MODE]
            [UNTIL DISCONNECT]
            [{WITH ROLLBACK | NO WAIT | NOWAIT} ]
```

Operands

- [*authorization-identifier*.]*table-identifier*

  [,[*authorization-identifier*.]*table-identifier*]...

authorization-identifier

Specifies the authorization identifier of the owner of a table on which lock control is to be performed.

table-identifier

Specifies the name of the table that is to be subject to lock control.

If a view table is specified in *table-identifier*, the base table underlying the view table is locked. In this case, a lock is not performed on the view table.

A maximum of 64 table names can be specified; the same table name can be specified more than once.

- [IN SHARE MODE]

Specifies that the data in the specified tables to be subject to lock control can be referenced but not updated by other users (shared mode). During access to a table after

a LOCK statement in which this operand is specified has been issued, the number of rows in the shared mode and the number of locked resources for key values are reduced.

- [IN EXCLUSIVE MODE]

Specifies that the data in the specified tables to be subject to lock control cannot be referenced or updated by other users (exclusive mode). The tables can still be retrieved by another user if the user specifies the WITHOUT LOCK NOWAIT lock option. During access to a table after a LOCK statement in which this operand is specified has been issued, the number of rows in the shared mode, the number of key values, the number of rows in the locked mode, and the number of locked resources for key values are reduced.

- [UNTIL DISCONNECT]

This option must be specified when table data is to be locked until it is processed by the DISCONNECT statement. The default is to lock the table data until the transaction terminates.

- {with rollback | NO WAIT | NOWAIT}

Specifies that when locking competes with another user, an error is to be received without waiting for resolution of the contention. If this operand is omitted and locking competes, HiRDB waits either until the lock is released or until amount of time specified in the system-defined pd_lck_wait_timeout operand has elapsed.

WITH ROLLBACK

If the table subject to lock is being used by another user, this operand is specified when canceling and nullifying the transaction.

NO WAIT

Specifies that when a table subject to this lock control is being used by another user, the current SQL statement is to be nullified but the transaction is not to be cancelled.

NOWAIT

The same as specifying NO WAIT.

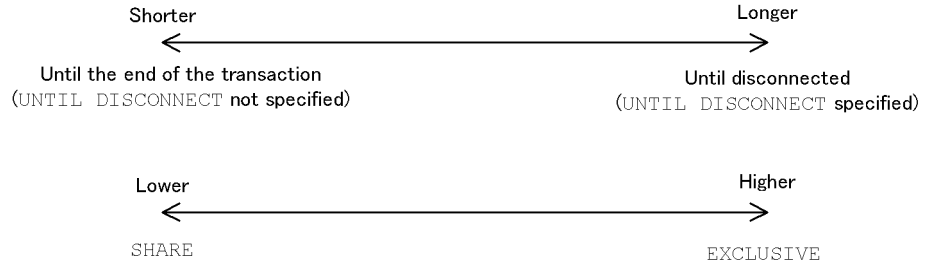
Common rules

1. If the base table that are subject to lock are duplicated, they are subject to duplicate elimination. The maximum number of base tables that can be locked by one LOCK statement is 64.
2. Data dictionary tables are not eligible for lock control.
3. Locking data or executing the LOCK statement causes the locking period to increase and the locking level to become higher. Therefore, even if the LOCK

statement is executed with a specification to reduce the locking period or lower the locking level, the locking period or level will remain the same.

The following figure illustrates increasing and decreasing the locking level and the length of the locking period.

Figure 5-1: Increasing and decreasing of locking period and level



4. If a PURGE TABLE statement is executed after a LOCK statement has been issued, the lock level changes to EXCLUSIVE. However, if the transaction has terminated before the PURGE TABLE statement executes, the lock level does not change to EXCLUSIVE even though the PURGE TABLE statement executes.
5. If a table locked by an UNTIL DISCONNECT specification is deleted by the DROP TABLE statement, the locking of the table is automatically reset by the HiRDB system.

Notes

1. Because lock control is usually performed by HiRDB, the LOCK statement should be issued only for purposes of changing the unit of lock control.
2. When UNTIL DISCONNECT is specified, the duration of locking depends on the specification of PDXAMODE in the client environment definition. For details about PDXAMODE and locking control, see the *HiRDB Version 9 UAP Development Guide*.
3. If the LOCK statement is executed on a shared table in the lock mode (in EXCLUSIVE MODE), the following RDAREAs are also subject to locking in addition to the shared table:
   - The RDAREA that stores the shared table
   - If an index is defined for the shared table, the RDAREA that stores the index

   For details about locking shared tables, see the *HiRDB Version 9 Installation and Design Guide*.

Example

Impose shared-mode lock control on a table named STOCK in order to retrieve all information from the table in a single transaction:

```
LOCK TABLE STOCK IN SHARE MODE
```

ROLLBACK statement (Cancel transaction)

Function

The ROLLBACK statement cancels the current transaction and nullifies the database updating performed by the transaction.

Privileges

None.

Format

```
ROLLBACK [WORK] [RELEASE]
```

Operands

- WORK

This operand has no effect on the transaction cancellation function of the ROLLBACK statement. The WORK operand is supported for compatibility with JIS standards.

- RELEASE

Specifies that the UAP is to be disconnected from the HiRDB after the transaction has been canceled and database updates have been nullified.

Common rules

1. The ROLLBACK statement closes all cursors that are currently open.
2. The ROLLBACK statement resets any lock controls that were put into effect during the current transaction. However, lock controls set by a LOCK TABLE statement with UNTIL DISCONNECT specified are not reset.
3. Definition SQL statements are not subject to rollback.
4. Preprocessing of a dynamic SELECT statement with holdable cursor specification is nullified if it was performed during the current transaction. In other cases, the preprocessing remains effective.
5. Executing the ROLLBACK statement nullifies all effective locators.

Notes

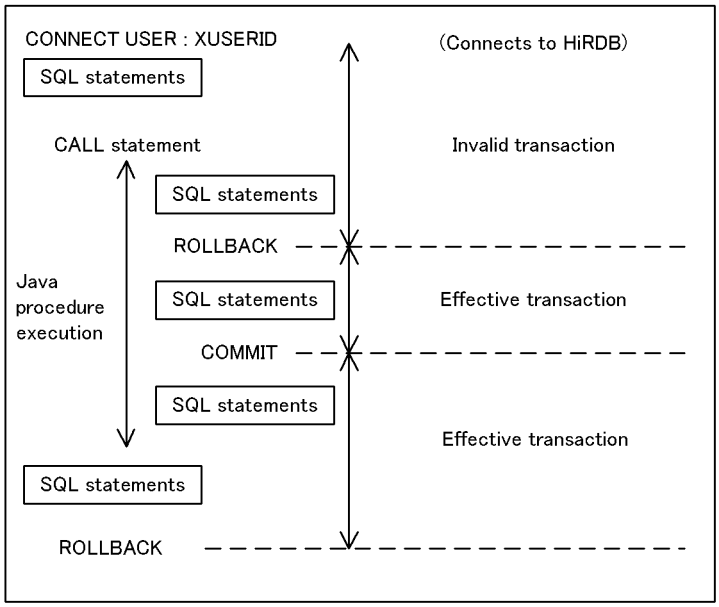
1. The ROLLBACK statement cannot be specified from an X/Open compliant UAP running under OLTP. Similarly, if a procedure is called from a UAP running under OLTP, procedures using the ROLLBACK statement cannot be executed.
2. When executing the ROLLBACK statement from a procedure, you cannot specify the RELEASE operand.

3. The ROLLBACK statement cannot be executed during trigger action.

Examples

1. Cancel the current transaction:
   ROLLBACK

2. Cancel the transaction either within a procedure of after the procedure has terminated:



SET SESSION AUTHORIZATION statement (Change connected user)

Function

The SET SESSION AUTHORIZATION statement posts an authorization identifier and a password to HiRDB to make a change in the users who are connected.

Privileges

Users with the DBA or CONNECT privilege

These users can change connected users.

Format

```
SET SESSION AUTHORIZATION {:embedded variable-1|?-parameter-1}
    [{USING|IDENTIFIED BY} {: embedded variable-2|?-parameter-2 }]
```

Operands

- *embedded-variable-1*|*?-parameter-1*

Specifies an embedded variable or ? parameter that has an authorization identifier as its value.

If the UAP is written in C, specify a fixed-length character string of up to 31 bytes, the final character of which is the null value. If the character string does not terminate with the null value, the character string occupying (*area length* - 1) is used as the authorization identifier.

If the UAP is written in COBOL, specify a fixed-length character string of up to 30 bytes; this character string need not terminate with the null value.

Character sets other than the default character set cannot be specified in *embedded-variable-1*. If character set UTF16 is specified in *?-parameter-1*, specify its data type as a fixed-length string of no more than 60 bytes.

If the character strings are to be case sensitive, enclose them in double quotation marks (").

- *embedded-variable-2* | *?-parameter-2*

Specifies an embedded variable or ? parameter that contains a password as its value.

If the UAP is written in C, specify a fixed-length character string of up to 31 bytes, the final character of which is the null value. If the character string does not terminate with the null value, the character string occupying (*area length* - 1) is used as the password.

If the UAP is written in COBOL, specify a fixed-length character string of up to 30 bytes; this character string need not terminate with the null value.

Character sets other than the default character set cannot be specified in *embedded-variable-2*. If character set UTF16 is specified in *?-parameter-2*, specify its data type as a fixed-length string of no more than 60 bytes.

Common rules

1. The user can be changed only if the statement is executed before the transaction is started or it is executed first in the transaction.
2. All results preprocessed by the PREPARE statement are nullified.
3. The user cannot be changed if a holdable cursor is open.
4. The user cannot be changed if a lock specified with UNTIL DISCONNECT is applied by a LOCK TABLE statement.
5. When specifying SET SESSION AUTHORIZATION in an X/Open compliant UAP running under OLTP, specify this statement so that it is executed immediately after a tx_begin or xa_begin function call.
6. If SET SESSION AUTHORIZATION of the dynamic SQL statement is executed, the statement is committed upon completion of the processing.
7. This statement cannot be executed within a stored procedure.

Chapter
6. Embedded Language Syntax

This chapter explains the syntax and structure of the embedded language.

General rules

The embedded language is an SQL that is used in conjunction with a program SQL for creating an embedded UAP, declaring embedded variables, and declaring processing by means of return codes.

The following table lists the types and functions of the embedded language.

Table 6-1: Types and functions of the embedded language

Type	Function
BEGIN DECLARE SECTION (Declare beginning of embedded SQL)	Indicates the beginning of an embedded variables declaration section that specifies the embedded variables and indicator variables used in the SQL.
END DECLARE SECTION (Declare end of embedded SQL)	Indicates the end of an embedded variables declaration section.
ALLOCATE CONNECTION HANDLE (Allocate connection handle)	Allocates a connection handle to be used by a UAP in an environment where a multi-connection function is used.
FREE CONNECTION HANDLE (Release connection handle)	Releases a connection handle that was allocated by ALLOCATE CONNECTION HANDLE.
DECLARE CONNECTION HANDLE SET (Declare connection handle to be used)	Declares the connection handle to be used by a UAP in an environment where a multi-connection function is used.
DECLARE CONNECTION HANDLE UNSET (Reset all connection handles being used)	Resets all declarations of connection handle usage specified in DECLARE CONNECTION HANDLE SET statements prior to this statement.
GET CONNECTION HANDLE (Get connection handles)	Allocates the connection handle to be used by a UAP when the multi-connection facility is used under the X/Open XA interface environment.
COPY (Include library text)	Includes (copies) a source library text into the source program.
GET DIAGNOSTICS (Retrieve diagnostic information)	If the SQL statement that was executed immediately before is CREATE PROCEDURE, CREATE FUNCTION, CREATE TYPE, ALTER PROCEDURE, ALTER ROUTINE, ALTER TRIGGER, CREATE TRIGGER, a CALL statement, a dynamic SELECT statement with a WITH clause specification, or a cursor declaration, the statement acquires relevant error information and diagnostic information from the diagnostic area.
COMMAND EXECUTE (Execute commands from UAP)	Executes HiRDB and OS commands from within a UAP.
SQL prefix	Indicates the beginning of an SQL.
SQL terminator	Indicates the end of an SQL.

Type	Function
WHENEVER (Declare embedded exception)	Declares UAP processing, based on the return code set by HiRDB in the SQL Communications Area after an SQL has executed.
SQLCODE variable	Receives a return code returned by HiRDB after an SQL has executed.
SQLSTATE variable	Receives a return code returned by HiRDB after an SQL has executed.
PDCNCTHDL-type variable declaration	Declares the handle that has the connection information to be used in an environment where a multi-connection function is used.
INSTALL JAR	Registers a JAR file in a HiRDB server.
REPLACE JAR	Re-registers a JAR file in a HiRDB server.
REMOVE JAR	Deletes a JAR file from a HiRDB server.
INSTALL CLIB	Install a new C library file on a HiRDB server.
REPLACE CLIB	Replace a C library file on a HiRDB server.
REMOVE CLIB	Removes a C library file from a HiRDB server.
DECLARE AUDIT INFO SET	Set account information for applications that access the HiRDB server and other user connection information.

BEGIN DECLARE SECTION (Declare beginning of embedded SQL)

Function

The BEGIN DECLARE SECTION declares the beginning of an embedded SQL declare section. Embedded variables and indicator variables used in the SQL must be specified in the embedded SQL declare section.

Format

```
BEGIN DECLARE SECTION
```

Common rules

1. The end of an embedded SQL declare section is denoted by specifying an END DECLARE SECTION (declaring the end of an embedded SQL).
2. All embedded variables and indicator variables used in the SQL must be declared in the embedded SQL declare section.
3. Any number of embedded SQL declare sections (including no embedded SQL begin declare sections) can be specified in an embedded UAP.
4. Only the declaration of variables can be specified in an embedded SQL declare section. However, embedded SQL declare sections that do not contain any declarations of variables can be specified.

Examples

Declare embedded variables used in an SQL:

C language

```
EXEC SQL BEGIN DECLARE SECTION;
   char XPCODE[5];
   char XPNAME[21];
   char XCOLOR[11];
   long XPRICE;
   long XSQUANTITY;
EXEC SQL END DECLARE SECTION;
```

COBOL language

```
   EXEC SQL
      BEGIN DECLARE SECTION
   END-EXEC.
77 XPCODE     PIC X(4).
77 XPNAME     PIC X(20).
77 XCOLOR     PIC X(10).
77 XPRICE     PIC S9(9) COMP.
77 XSQUANTITY PIC S9(9) COMP.
```

```
EXEC SQL
    END DECLARE SECTION
END-EXEC.
```

END DECLARE SECTION (Declare end of embedded SQL)

Function

The `END DECLARE SECTION` declares the end of an embedded SQL declare section.

Format

```
END DECLARE SECTION
```

Common rules

1. The beginning of an embedded SQL declare section is denoted by specifying a `BEGIN DECLARE SECTION` (declaring the beginning of an embedded SQL).
2. All embedded variables and indicator variables used in the SQL should be declared in the embedded SQL declare section.

Examples

See the section on the *BEGIN DECLARE SECTION (Declare beginning of embedded SQL)* for examples.

ALLOCATE CONNECTION HANDLE (Allocate connection handle)

Function

ALLOCATE CONNECTION HANDLE allocates a connection handle to be used by a UAP in an environment where a multi-connection function is used.

Format

```
ALLOCATE CONNECTION HANDLE :PDCNCTHDL-type-variable,
                           :return-code-receiving-variable
                           [, {:connection-PDHOST-variable,
                               :connection-PDNAMEPORT-variable
                           |:environment-variable-group-name-variable}]
```

Operands

- *PDCNCTHDL-type-variable*

Specifies an embedded variable that was declared as a PDCNCTHDL-type.

- *return-code-receiving-variable*

Specifies an embedded variable that was declared as an INT type.

The following values are returned to the return code receiving variable:

C language

Normal allocation:

```
p_rdb_RC_NORM
```

Invalid connection handle value:

```
p_rdb_RC_ERRPARM
```

Insufficient memory:

```
p_rdb_RC_MEMERR
```

These values are defined in the pdberrno.h file.

COBOL language

Normal allocation:

```
P-RDB-RC-NORM
```

Invalid connection handle value:

```
P-RDB-RC-ERRPARM
```

Insufficient memory:

```
P-RDB-RC-MEMERR
```

These values are defined in the PDBSQLCAMTH.CBL file.

- :*connection-PDHOST-variable*

Specifies the destination host name using an embedded variable declared as CHARACTER type (area length of 511 bytes). Specify the host name in the format specified in PDHOST in the client environment definition.

Character sets other than the default character set cannot be specified.

- :*connection-PDNAMEPORT-variable*

Specifies the port number to connect to as an embedded variable declared as SMALLINT type. Specify the port number in the format specified in PDNAMEPORT specified in the client environment definition.

- :*environment-variable-group-name-variable*

Specifies an embedded variable that is declared as a CHAR type (area length of 256 bytes).

Character sets other than the default character set cannot be specified.

In the UNIX environment, specify the file name of the regular file in which the environment variable is coded in terms of an absolute path name (a maximum of 256 bytes including the null character).

In a Windows environment, specify the group name registered using the environment variable registration tool (a maximum of 31 bytes including the null character), or the absolute path name of the environment variable group file name (a maximum of 256 bytes including the null character). All environment variable group file name specifications are assumed to begin with *drive-name*:\. Environment variable group files in a Windows environment are based on the Windows ini file specification. In a Windows environment, even when specifying environment variable group files using long path names that include spaces, do not enclose the path name in double quotation marks (").

For details about environment variable groups, see the *HiRDB Version 9 UAP Development Guide*.

Common rules

1. ALLOCATE CONNECTION HANDLE should be issued before the CONNECT statement.
2. The embedded variables to be used must be declared in the embedded SQL declare section.
3. If allocation of a connection handle fails, the system sets the error code in the return code-receiving variable.

4. The connection PDHOST variable and the connection PDNAMEPORT variable should be declared either together or not at all. If these variables are omitted, the system connects to a database on the basis of a value specified in the client environment definition.
5. A null character must be used at the end of the host name specified in the connection PDHOST variable.

Note

1. Connection handles that have been allocated are not released when a DISCONNECT statement is issued. To release a connection handle, a FREE CONNECTION HANDLE statement must be issued.
2. When setting a value for the connection PDHOST variable and environment variable group name variable in COBOL, a null character must be used at the end of the value.
3. When specifying the connection PDHOST variable and connection PDNAMEPORT variable, if a high-speed connection or an FES host direct connection is specified in the client environment definition, the connection to the HiRDB server will be made using the connection type specified in the client environment definition. For details about how to select the connection type, see *Environment variables and connection types for HiRDB servers* in the *HiRDB Version 9 UAP Development Guide*.

Example

1. The following are examples of using a PDCNCTHDL type variable:

C language

```
EXEC SQL BEGIN DECLARE SECTION;
  PDCNCTHDL   CnctHdl;
  long        AlchdlRtn;
EXEC SQL END DECLARE SECTION;
EXEC SQL ALLOCATE CONNECTION HANDLE :CnctHdl,
                                    :AlchdlRtn;
```

COBOL language

```
DATA DIVISION.
WORKING-STORAGE SECTION.
  EXEC SQL
    BEGIN DECLARE SECTION
  END-EXEC.
01 CNCTHDL     SQL TYPE IS PDCNCTHDL.
01 ALCHDLRTN   PIC S9(9) COMP.
  EXEC SQL
    END DECLARE SECTION
  END-EXEC.
        :
```

```
PROCEDURE DIVISION.
        :
  EXEC SQL
    ALLOCATE CONNECTION HANDLE :CNCTHDL,
                               :ALCHDLRTN;
  END-EXEC.
```

2. The following are examples of using a connection PDHOST variable and a connection PDNAMEPORT variable:

C language

```
EXEC SQL BEGIN DECLARE SECTION;
  PDCNCTHDL   CnctHdl;
  long        AlchdlRtn;
  char        CnctHost[31];
  short       CnctPort;
EXEC SQL END DECLARE SECTION;
strcpy(CnctHost,"HOST01");
EXEC SQL ALLOCATE CONNECTION HANDLE :CnctHdl,
                                    :AlchdlRtn,
                                    :CnctHost,
                                    :CnctPort;
```

COBOL language

```
DATA DIVISION.
WORKING-STORAGE SECTION.
  EXEC SQL
    BEGIN DECLARE SECTION
  END-EXEC.
01 CNCTHDL     SQL TYPE IS PDCNCTHDL.
01 ALCHDLRTN   PIC S9(9) COMP.
01 CNCTHOST    PIC X(31).
01 CNCTPORT    PIC S9(4) COMP.
  EXEC SQL
    END DECLARE SECTION
  END-EXEC.
        :
PROCEDURE DIVISION.
        :
  MOVE 'HOST01' & X'00' TO CNCTHOST.
  EXEC SQL
    ALLOCATE CONNECTION HANDLE :CNCTHDL,
                               :ALCHDLRTN,
                               :CNCTHOST,
                               :CNCTPORT;
  END-EXEC.
```

3. The following are examples of using an environment variable group name:

C language

```
EXEC SQL BEGIN DECLARE SECTION;
  PDCNCTHDL   CnctHdl;
  long        AlchdlRtn;
  char        GroupName[31];
EXEC SQL END DECLARE SECTION;
strcpy(GroupName,"HRD01");
EXEC SQL ALLOCATE CONNECTION HANDLE :CnctHdl,
                                    :AlchdlRtn,
                                    :GroupName;
```

COBOL language

```
DATA DIVISION.
WORKING-STORAGE SECTION.
  EXEC SQL
    BEGIN DECLARE SECTION
  END-EXEC.
01 CNCTHDL     SQL TYPE IS PDCNCTHDL.
01 ALCHDLRTN   PIC S9(9) COMP.
01 GROUPNAME   PIC X(31).
  EXEC SQL
    END DECLARE SECTION
  END-EXEC.
        :
PROCEDURE DIVISION.
        :
  MOVE 'HRD01' & X'00' TO GROUPNAME.
  EXEC SQL
    ALLOCATE CONNECTION HANDLE :CNCTHDL,
                               :ALCHDLRTN,
                               :GROUPNAME
  END-EXEC.
```

FREE CONNECTION HANDLE (Release connection handle)

Function

FREE CONNECTION HANDLE releases a connection handle that was allocated by ALLOCATE CONNECTION HANDLE.

Format

```
FREE CONNECTION HANDLE :PDCNCTHDL-type-variable
                       :return-code-receiving-variable
```

Operands

- *PDCNCTHDL-type-variable*

Specifies the PDCNCTHDL-type variable that was specified in ALLOCATE CONNECTION HANDLE.

- *return-code-receiving-variable*

Specifies the return code receiving variable that was specified in ALLOCATE CONNECTION HANDLE.

The following values are returned to the return code receiving variable:

C language

Normal allocation:

```
p_rdb_RC_NORM
```

Invalid connection handle value:

```
p_rdb_RC_ERRPARM
```

Connection handle being used:

```
p_rdb_RC_SIMERR
```

These values are defined in the pdberrno.h file.

COBOL language

Normal allocation:

```
P-RDB-RC-NORM
```

Invalid connection handle value:

```
P-RDB-RC-ERRPARM
```

Connection handle being used:

```
P-RDB-RC-SIMERR
```

These values are defined in the SQLCAMTH.CBL file.

Common rules

1. FREE CONNECTION HANDLE should be issued after a DISCONNECT statement.
2. The embedded variables to be used must have been declared in the embedded SQL declare section.
3. If release of a connection handle fails, the system sets the error code in the return code receiving variable.

Note

Connection handles that have been reserved are not released when a DISCONNECT statement is issued. To release a connection handle, a FREE CONNECTION HANDLE statement must be issued.

Example

Release the connection handle that was allocated in the example shown in the section on ALLOCATE CONNECTION HANDLE:

C language

```
EXEC SQL BEGIN DECLARE SECTION;
  PDCNCTHDL   CnctHdl;
  long        FrchdlRtn;
EXEC SQL END DECLARE SECTION;
EXEC SQL FREE CONNECTION HANDLE :CnctHdl,
                                :FrchdlRtn;
```

COBOL language

```
DATA DIVISION.
WORKING-STORAGE SECTION.
  EXEC SQL
    BEGIN DECLARE SECTION
  END-EXEC.
01 CNCTHDL     SQL TYPE IS PDCNCTHDL.
01 FRCHDLRTN   PIC S9(9) COMP.
  EXEC SQL
    END DECLARE SECTION
  END-EXEC.
        :
PROCEDURE DIVISION.
        :
  EXEC SQL
    FREE CONNECTION HANDLE :CNCTHDL,
                           :FRCHDLRTN;
  END-EXEC.
```

DECLARE CONNECTION HANDLE SET (Declare connection handle to be used)

Function

In an environment that uses the multi-connection facility, DECLARE CONNECTION HANDLE SET declares the connection handle that is used by SQL statements in a UAP or by the SQL Communications Area.

Format

```
DECLARE CONNECTION HANDLE SET :PDCNCTHDL-type-variable
```

Operands

- *PDCNCTHDL-type-variable*

Specifies the embedded variable that was declared as a PDCNCTHDL-type variable.

Common rules

1. The scope of the connection handle specified in DECLARE CONNECTION HANDLE SET depends on the location in the source program where the connection handle appears. The variable for the connection handle specified in a DECLARE CONNECTION HANDLE SET is effective on all SQL statements until another DECLARE CONNECTION HANDLE SET or DECLARE CONNECTION HANDLE UNSET appears.
2. The connection handle to be used can be declared as many times as necessary in the same UAP.

Notes

1. For the C language, any SQL statement that was issued before DECLARE CONNECTION HANDLE SET was coded is processed under the assumption that the SQL statement was issued in the single-connection environment.

   For the COBOL language, UAPs using the multi-connection facility cannot code an SQL statement before DECLARE CONNECTION HANDLE SET is coded.
2. What is declared in a DECLARE CONNECTION HANDLE SET statement is the name of a variable for a connection handle; it is not the value itself.
3. Before referencing the SQL communication area storing the results of execution of an SQL statement using the multi-connection facility, DECLARE CONNECTION HANDLE SET must be executed using the module.

Example

Declare a connection handle whose PDCNCTHDL-type variable is hCnct:

C language

```
EXEC SQL BEGIN DECLARE SECTION;
    PDCNCTHDL CnctHdl;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE CONNECTION HANDLE SET :CnctHdl;
```

COBOL language

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC
    01 CNCTHDL SQL TYPE IS PDCNCTHDL.
EXEC SQL END DECLARE SECTION END-EXEC

EXEC SQL DECLARE CONNECTION HANDLE SET :CNCTHDL END-EXEC
```

DECLARE CONNECTION HANDLE UNSET (Reset all connection handles being used)

Function

DECLARE CONNECTION HANDLE UNSET resets all declarations of connection handle usage specified in DECLARE CONNECTION HANDLE SET statements prior to this statement.

Format

```
DECLARE CONNECTION HANDLE UNSET
```

Common rules

1. The scope of the connection handles specified in a DECLARE CONNECTION HANDLE UNSET statement depends on the location in the source program where the connection handles appear. Releasing connection handles by issuing DECLARE CONNECTION HANDLE UNSET causes use of all connection handles by all SQL statements to be discontinued until another DECLARE CONNECTION HANDLE SET appears.
2. DECLARE CONNECTION HANDLE UNSET cannot be used in the COBOL language.

Note

The system does not process a multi-connection function for SQL statements that are issued after a DECLARE CONNECTION HANDLE UNSET has been issued.

Example

Release all connection handles that have been declared:

```
DECLARE CONNECTION HANDLE UNSET;
```

GET CONNECTION HANDLE (Get connection handle)

Function

When the multi-connection facility is used under the X/Open XA interface environment, GET CONNECTION HANDLE allocates the connection handle to be used by a UAP.

Format

```
GET CONNECTION HANDLE:PDCNCTHDL-type-variable,
                     :return-code-receiving-variable,
                     :environment-variable-group-identifier-variable
```

Operands

- : *PDCNCTHDL-type-variable*

Specifies an embedded variable that was declared as a PDCNCTHDL-type.

- : *return-code-receiving-variable*

Specifies an embedded variable that was declared as an INT type.

The following values are returned to the return code receiving variable:

C language

Normal receipt:

```
p_rdb_RC-NORM
```

Error:

Error code associated with the particular error

These values are defined in the pdberrno.h file.

COBOL language

Normal receipt:

```
P-RDB-RC-NORM
```

These values are defined in the SQLCAMTH.CBL file.

- : *environment-variable-group-identifier-variable*

Specifies an embedded variable that is declared as CHAR type (five bytes, fixed).

Character sets other than the default character set cannot be specified.

The specified value should be the environment variable group identifier (five bytes, fixed, including null characters) that is specified in the character string for the

xa_open function.

Common rules

1. The variables must be declared in the embedded variables declaration section.
2. If acquisition of the connection handle fails, the resulting error code is set in the return code receiving variable.

Notes

1. When an SQL statement that uses the connection handle acquired in this section is issued, the connection handle to be used must be declared by specifying DECLARE CONNECTION HANDLE SET.
2. For the COBOL language, when assigning a value to the environment variable group identifier, the last character in the assigned value should be the null character.

Example

Allocate a connection handle with a PDCNCTHDL-type variable CnctHdl, return code receiving variable GetchdlRtn, and environment variable group identifier variable GroupName:

C language

```
EXEC SQL BEGIN DECLARE SECTION;
  PDCNCTHDL   CnctHdl;
  long        GetchdlRtn;
  char        GroupId[5];
EXEC SQL END DECLARE SECTION;
strcpy(GroupId,"HR01") ;
EXEC SQL GET CONNECTION HANDLE  :CnctHdl,
                                :GetchdlRtn,
                                :GroupId;
```

COBOL language

```
DATA DIVISION.
WORKING-STORAGE SECTION.
  EXEC SQL
    BEGIN DECLARE SECTION
  END-EXEC.
01 CNCTHDL     SQL TYPE IS PDCNCTHDL.
01 GETCHDLRTN  PIC S9(9) COMP.
01 GROUPID     PIC X(5).
  EXEC SQL
    END DECLARE SECTION
  END-EXEC.
        :
PROCEDURE DIVISION.
```

```
      :
MOVE 'HR01' & X'00' TO GROUPID.
EXEC SQL
  GET CONNECTION HANDLE :CNCTHDL,
                        :GETCHDLRTN,
                        :GROUPID;
END-EXEC.
```

Additional notes

If HiRDB does not support the multi-connection facility, SQL statements cannot be executed at that HiRDB using multiple connection handles from one transaction. If such a transaction is executed, the HiRDB server aborts with the Pac2354 code and the transaction is rolled back. Therefore, SQL statements that use multiple connection handles cannot be executed simultaneously within a single transaction.

For different copies of HiRDB that are connected to different sites, multiple connection handles can be used within a single transaction.

Following is an example of an incorrect specification:

HiRDB 1 registered in both RM01 and RM02

```
tx_begin()
strcpy(grpnm, "RM01")
GET CONNECTION HANDLE;
hCnct,:rc,:grpnm;
DECLARE CONNECTION HANDLE SET :hCnct;
```

SQL execution

```
strcpy(grpnm, "RM02")
GET CONNECTION HANDLE :hCnct,:rc,:grpnm;
DECLARE CONNECTION HANDLE SET :hCnct;
```

SQL execution

```
tx_commit()
```

COPY (Include library text)

Function

COPY includes (copies) a source library text into the source program.

Format

COPY *source-text-name*

Operands

source-text-name

Specifies the name of the file (exclusive of the suffix) that contains either library text or a header file, as a character string of 30 characters or less.

Common rules

1. COPY must be enclosed between an SQL prefix and an SQL terminator.

   The SQL terminator must be the last item of coding through the end of the row.

2. Directories in which library text and header files are cataloged are searched in the following sequence; the directories should therefore be stored in one of the files listed below:

   Order in which directories are searched

   - Directories cataloged by means of an environment variable (PDCBLLIB in the case of COBOL or OOCOBOL; PDCLIB in the case of C or C++)
   - Current directory

   Order in which files are searched (for COBOL and OOCOBOL)

   - Suffixed files stored by PDCBLFIX environment variable
   - *file-name.cbl*
   - *file-name.CBL*
   - *file-name.cob*

   Order in which files are searched (for C and C++)

   - *file-name.h*

3. In SQL, COPY cannot be specified in a text included by COPY.
4. With COBOL or OOCOBOL, the library text must be specified in the fixed-format reference format.

Notes

A source library text that contains embedded variables or SQL statements should not be included using the `COPY` or `INCLUDE` statement of COBOL or OOCOBOL.

The `#include` instruction of C or C++ must not be used to include a file that contains SQL statements.

Example

Include the source library text with file name `SAMPLE` into the source program:

COBOL language

```
EXEC SQL
   COPY SAMPLE
END-EXEC.
```

GET DIAGNOSTICS (Retrieve diagnostic information)

Function

If the SQL statement that was executed immediately before is any of the following, this statement acquires relevant error information and diagnostic information from the diagnostic area:

- Definition SQL statement
- Data manipulation SQL statement
- Control SQL statement
- Embedded language grammar (`INSTALL JAR`, `REPLACE JAR`, `REMOVE JAR`, `INSTALL CLIB`, `REPLACE CLIB`, `REMOVE CLIB`)
- Routine control SQL

When recreating an SQL object in a routine, information on the successfully recreated routine is included in this diagnostic information.

Format

```
GET DIAGNOSTICS
    {:embedded-variable=statement-information-item-name
        [,:embedded-variable=statement-information-item-name]...
     |EXCEPTION condition-number
        :embedded-variable=each-item-of-condition-name
            [,:embedded-variable=each-item-of-condition-name]...}

statement-information-item-name::={NUMBER | MORE}
each-item-of-condition-name::={RETURNED_SQLCODE
                              |ERROR_POSITION
                              |ERROR_SQL_NO
                              |ERROR_SQL
                              |ROUTINE_TYPE
                              |ROUTINE_SCHEMA
                              |ROUTINE_NAME
                              |TRIGGER_SCHEMA
                              |TRIGGER_NAME
                              |CONSTRAINT_SCHEMA
                              |CONSTRAINT_NAME
                              |MESSAGE_TEXT
                              |QUERY_NAME
                              |CONDITION_IDENTIFIER}
```

Operands

- *statement-information-item-name*

NUMBER

This operand is specified to obtain the number of diagnostic information items in the diagnostics area. Specify a data type of SMALLINT.

MORE

This operand is specified to obtain information about whether there is more diagnostic information than can be stored in the diagnostics area.

Specify a data type of CHAR with a length of one byte.

If all of the diagnostic information has been stored in the diagnostics area, this will be set to N, otherwise it will be set to Y.

- *condition-number*

Specifies in an embedded variable the ordinal number of diagnostic information to be obtained.

- *each-item-of-condition-name*

RETURNED_SQLCODE

This operand is specified to obtain the value of the return code (SQLCODE). The INTEGER data type must be specified.

ERROR_POSITION

This operand is specified when acquiring the position of the error in the SQL statement if a syntax error occurs. The data type must be specified as INTEGER. If an error other than a syntax error has occurred, the value 0 is set.

ERROR_SQL_NO

This operand is specified to acquire a number indicating the type of the error-containing SQL procedure statement in the routine when one of the following SQL statements is executed:

- CREATE PROCEDURE
- CREATE FUNCTION
- CREATE TYPE
- CREATE TRIGGER
- ALTER PROCEDURE
- ALTER ROUTINE
- ALTER TRIGGER

- `CALL` statement
- Data manipulation SQL statement that induces a trigger (only trigger action-related information can be acquired)

The number indicates the position of the SQL statement in the procedure, beginning with `0`. The data type must be specified as `INTEGER`.

`ERROR_SQL`

This operand is specified to obtain a character string that indicates the type of the SQL diagnostic information.

Specify a data type of `VARCHAR` with a length of 32 bytes (64 bytes if character set name `UTF16` is specified).

The following table lists the character strings that are set.

Table 6-2: Character strings that are set in ERROR_SQL

Classification (SQL type)		Whether character string is set				Character string that is set
		Routine		Other		
		Define	Execute	Pre-process	Execute	
Definition SQL	ALTER INDEX	N	N	Y	Y	ALTER INDEX
	ALTER PROCEDURE	N	N	Y	Y	ALTER PROCEDURE
	ALTER ROUTINE	N	N	Y	Y	ALTER ROUTINE
	ALTER TABLE	N	N	Y	Y	ALTER TABLE
	ALTER TRIGGER	N	N	Y	Y	ALTER TRIGGER
	`COMMENT` (annotate a table)	N	N	Y	Y	COMMENT TABLE
	`COMMENT` (annotate a column)	N	N	Y	Y	COMMENT COLUMN
	CREATE AUDIT	N	N	Y	Y	CREATE AUDIT
	CREATE CONNECTION SECURITY	N	N	Y	Y	CREATE CONNECTION SECURITY
	CREATE FUNCTION	N	N	Y	Y	CREATE FUNCTION
	CREATE PUBLIC FUNCTION	N	N	Y	Y	CREATE FUNCTION

Classification (SQL type)		Whether character string is set				Character string that is set
		Routine		Other		
		Define	Execute	Pre-process	Execute	
	CREATE INDEX	N	N	Y	Y	CREATE INDEX
	CREATE PROCEDURE	N	N	Y	Y	CREATE PROCEDURE
	CREATE PUBLIC PROCEDURE	N	N	Y	Y	CREATE PROCEDURE
	CREATE SCHEMA	N	N	Y	Y	CREATE SCHEMA
	CREATE SEQUENCE	N	N	Y	Y	CREATE SEQUENCE
	CREATE TABLE	N	N	Y	Y	CREATE TABLE
	CREATE TRIGGER	N	N	Y	Y	CREATE TRIGGER
	CREATE TYPE	N	N	Y	Y	CREATE TYPE
	CREATE VIEW	N	N	Y	Y	CREATE VIEW
	CREATE PUBLIC VIEW	N	N	Y	Y	CREATE VIEW
	DROP AUDIT	N	N	Y	Y	DROP AUDIT
	DROP CONNECTION SECURITY	N	N	Y	Y	DROP CONNECTION SECURITY
	DROP DATA TYPE	N	N	Y	Y	DROP DATA TYPE
	DROP FUNCTION	N	N	Y	Y	DROP FUNCTION
	DROP PUBLIC FUNCTION	N	N	Y	Y	DROP FUNCTION
	DROP INDEX	N	N	Y	Y	DROP INDEX
	DROP PROCEDURE	N	N	Y	Y	DROP PROCEDURE
	DROP PUBLIC PROCEDURE	N	N	Y	Y	DROP PROCEDURE
	DROP SCHEMA	N	N	Y	Y	DROP SCHEMA

Classification (SQL type)		Whether character string is set				Character string that is set
		Routine		Other		
		Define	Execute	Pre-process	Execute	
	DROP SEQUENCE	N	N	Y	Y	DROP SEQUENCE
	DROP TABLE	N	N	Y	Y	DROP TABLE
	DROP TRIGGER	N	N	Y	Y	DROP TRIGGER
	DROP VIEW	N	N	Y	Y	DROP VIEW
	DROP PUBLIC VIEW	N	N	Y	Y	DROP VIEW
	GRANT CONNECT	N	N	Y	Y	GRANT CONNECT
	GRANT DBA	N	N	Y	Y	GRANT DBA
	GRANT RDAREA	N	N	Y	Y	GRANT RDAREA
	GRANT SCHEMA	N	N	Y	Y	GRANT SCHEMA
	`GRANT` access privileges	N	N	Y	Y	GRANT ACCESS
	GRANT AUDIT	N	N	Y	Y	GRANT AUDIT
	REVOKE CONNECT	N	N	Y	Y	REVOKE CONNECT
	REVOKE DBA	N	N	Y	Y	REVOKE DBA
	REVOKE RDAREA	N	N	Y	Y	REVOKE RDAREA
	REVOKE SCHEMA	N	N	Y	Y	REVOKE SCHEMA
	`REVOKE` access privileges	N	N	Y	Y	REVOKE ACCESS
Data manipulation SQL	`ALLOCATE` CURSOR statement	N	N	N	Y	ALLOCATE CURSOR
	`ASSIGN LIST` statement	N	N	Y	Y	ASSIGN LIST
	`CALL` statement	Y	Y	Y	Y	CALL
	`CLOSE` statement	Y	Y	N	N	CLOSE

Classification (SQL type)		Whether character string is set				Character string that is set
		Routine		Other		
		Define	Execute	Pre-process	Execute	
		N	N	Y	Y	(See the SQL types specified in the cursor)
	DEALLOCATE PREPARE statement	N	N	N	Y	--
	DECLARE CURSOR	Y	N	N	N	DECLARE CURSOR
		N	N	Y	Y	(See the SQL types for the dynamic SELECT statement)
	DELETE statement	Y	Y	Y	Y	DELETE
	Preparable dynamic DELETE statement: positioning	N	N	Y	Y	DELETE
	DESCRIBE statement	N	N	Y	Y	(See the SQL types specified in the DESCRIBE statement)
	DESCRIBE CURSOR statement	N	N	Y	Y	(See the SQL types specified in the DESCRIBE CURSOR statement)
	DESCRIBE TYPE statement	N	N	Y	Y	(See the SQL types specified in the DESCRIBE TYPE statement)
	DROP LIST statement	N	N	Y	Y	DROP LIST
	EXECUTE statement	N	N	Y	Y	(See the SQL types executed by the EXECUTE statement)
	EXECUTE IMMEDIATE statement	N	N	Y	Y	(See the SQL types executed by the EXECUTE IMMEDIATE statement)
	FETCH statement	Y	Y	N	N	FETCH

Classification (SQL type)		Whether character string is set				Character string that is set
		Routine		Other		
		Define	Execute	Pre-process	Execute	
		N	N	Y	Y	(See the SQL types specified in the cursor)
	FREE LOCATOR statement	N	N	Y	Y	FREE LOCATOR
	INSERT statement	Y	Y	Y	Y	INSERT
	OPEN statement	Y	Y	N	N	OPEN
		N	N	Y	Y	(See the SQL types specified in the cursor)
	PREPARE statement	Y	Y	Y	Y	(See the SQL types specified in the PREPARE statement)
	PURGE TABLE statement	Y	Y	Y	Y	PURGE TABLE
	Single-line SELECT statement	Y	Y	Y	Y	SELECT
	Dynamic SELECT statement	N	N	Y	Y	SELECT
	UPDATE statement	Y	Y	Y	Y	UPDATE
	Preparable dynamic UPDATE statement: positioning	N	N	Y	Y	UPDATE
	Assignment statement	Y	Y	Y	Y	SET
Control SQL	CALL COMMAND statement	Y	Y	Y	Y	CALL COMMAND
	COMMIT statement	Y	Y	Y	Y	COMMIT
	CONNECT statement	N	N	N	N	--
	DISCONNECT statement	N	N	N	N	--
	LOCK statement	Y	Y	Y	Y	LOCK TABLE
	ROLLBACK statement	Y	Y	Y	Y	ROLLBACK
	SET SESSION AUTHORIZATION statement	N	N	Y	Y	SET SESSION AUTHORIZATION

Classification (SQL type)		Whether character string is set				Character string that is set
		Routine		Other		
		Define	Execute	Pre-process	Execute	
Embedded language grammar	BEGIN DECLARE SECTION	N	N	N	N	--
	END DECLARE SECTION	N	N	N	N	(Not applicable)
	ALLOCATE CONNECTION HANDLE	N	N	N	N	--
	FREE CONNECTION HANDLE	N	N	N	N	--
	DECLARE CONNECTION HANDLE SET	N	N	N	N	(Not applicable)
	DECLARE CONNECTION HANDLE UNSET	N	N	N	N	(Not applicable)
	GET CONNECTION HANDLE	N	N	N	N	--
	COPY	N	N	N	N	(Not applicable)
	GET DIAGNOSTICS	N	N	N	N	--
	COMMAND EXECUTE	N	N	N	N	--
	SQL prefix	N	N	N	N	(Not applicable)
	SQL terminator	N	N	N	N	(Not applicable)
	WHENEVER	N	N	N	N	--
	`SQLCODE` variable	N	N	N	N	(Not applicable)
	`SQLSTATE` variable	N	N	N	N	(Not applicable)
	`PDCNCTHDL` type variable declaration	N	N	N	N	(Not applicable)
	INSTALL JAR	N	N	Y	Y	INSTALL JAR
	REPLACE JAR	N	N	Y	Y	REPLACE JAR
	REMOVE JAR	N	N	Y	Y	REMOVE JAR
	INSTALL CLIB	N	N	Y	Y	INSTALL CLIB

Classification (SQL type)		Whether character string is set				Character string that is set
		Routine		Other		
		Define	Execute	Pre-process	Execute	
	REPLACE CLIB	N	N	Y	Y	REPLACE CLIB
	REMOVE CLIB	N	N	Y	Y	REMOVE CLIB
	DECLARE AUDIT INFO SET	N	N	N	N	(Not applicable)
Routine control SQL	SQL variable declaration	Y	Y	N	N	DECLARE
	Cursor declaration	Y	N	N	N	DECLARE CURSOR
		Y	Y	N	N	SELECT
	Condition declaration	Y	N	N	N	DECLARE CONDITION
	Handler declaration	Y	N	N	N	DECLARE HANDLER
	SQL procedure statement	Y	Y	N	N	(See the SQL types specified in the SQL procedure statement)
	Compound statement	Y	Y	N	N	BEGIN
	`IF` statement	Y	Y	N	N	IF
	`LEAVE` statement	Y	Y	N	N	LEAVE
	`RETURN` statement	Y	Y	N	N	RETURN
	`WHILE` statement	Y	Y	N	N	WHILE
	`FOR` statement	Y	Y	N	N	FOR
	`WRITE` LINE statement	Y	Y	N	N	WRITE LINE
	`SIGNAL` statement	Y	Y	N	N	SIGNAL
	`RESIGNAL` statement	Y	Y	N	N	RESIGNAL
Other than the above		Y	Y	Y	Y	Single-byte space

Legend

Y: Character string in the *Whether character string is set* columns is set.

N: Character string is not set.

--: No character string to set.

Parentheses indicate a comment.

ROUTINE_TYPE

This operand is specified to acquire the type of the error-containing function or procedure. Specify a data type of CHAR with a length of one byte (or two bytes if character set name UTF16 is specified). The character P is set for a procedure, and the character F is set for a function.

ROUTINE_SCHEMA

This operand is specified to acquire the authorization identifier of the error-containing function or procedure. Specify a data type of VARCHAR with a length of 30 bytes (or 60 bytes if character set name UTF16 is specified).

In case of the following errors, the authorization identifier is set to PUBLIC.

- Error when deleting a public function or public procedure definition
- Error when executing a CALL statement to a public procedure

ROUTINE_NAME

This operand is specified to acquire the identifier of the error-containing function or procedure. Specify a data type of VARCHAR with a length of 30 bytes (or 60 bytes if character set name UTF16 is specified).

TRIGGER_SCHEMA

This operand is specified to acquire the authorization identifier of the error-containing trigger. Specify a data type of VARCHAR with a length of 30 bytes (or 60 bytes if character set name UTF16 is specified).

TRIGGER_NAME

This operand is specified to obtain the authorization identifier of the error-containing trigger. Specify a data type of VARCHAR with a length of 30 bytes (or 60 bytes if character set name UTF16 is specified).

CONSTRAINT_SCHEMA

This operand is specified to obtain the authorization identifier of the error-containing constraint if the error was caused by a check constraint violation or referential constraint violation. Specify a data type of VARCHAR with a length of 30 bytes (or 60 bytes if character set name UTF16 is specified).

CONSTRAINT_NAME

This operand is specified to obtain the name of the error-containing constraint if the error was caused by a check constraint violation or referential constraint violation. Specify a data type of VARCHAR with a length of 30 bytes (or 60 bytes if character set name UTF16 is specified).

MESSAGE_TEXT

This operand is specified to obtain the message text. Specify a data type of VARCHAR with a length of 254 bytes (or 508 bytes if character set name UTF16 is specified).

QUERY_NAME

This operand is specified to obtain the query name of the query specification when an error occurs during execution of a dynamic SELECT statement with a WITH clause specified or a cursor declaration. Specify a data type of VARCHAR with a length of 30 bytes (or 60 bytes if character set name UTF16 is specified).

CONDITION_IDENTIFIER

This operand is specified to acquire the SIGNAL statement, the condition name specified in the RESIGNAL statement, or the SQLSTATE value that was executed in an SQL procedure or in a trigger. Specify a data type of VARCHAR with a length of 30 bytes (or 60 bytes if character set name UTF16 is specified).

If an SQLSTATE value is specified, 'SQLSTATE:*XXXXX*' (where *XXXXX* denotes the specified SQLSTATE value) is set.

Common rules

1. GET DIAGNOSTICS cannot be executed dynamically.
2. Only the results of the most recently executed SQL statement can be obtained. For details about which SQL statements return results, see *Table 6-2*.
3. The data type of the embedded variable in which a condition number is specified should be SMALLINT.
4. The embedded variable for receiving statement-information-item or each-item-of-condition should be of the same data type as the respective item.
5. A value less than 0 or a value greater than the number of errors in the diagnostics area cannot be specified in a condition number.
6. In the case of errors that occurred before SQL analysis, it may not be possible to obtain diagnostic information with GET DIAGNOSTICS. If GET DIAGNOSTICS is executed under such a circumstance, 0 errors are returned. Such error information must be obtained by referring to the SQLCA that was in effect at the time the errors occurred.
7. If no information is provided in the explanation of the condition details name of an SQL, a one-byte blank or 0 is set.

8. With respect to errors that may occur during the execution of a function, `GET DIAGNOSTICS` may fail to acquire any of the following types of diagnostic information:
   - `ROUTINE_TYPE`
   - `ROUTINE_SCHEMA`
   - `ROUTINE_NAME`

Example

Obtain the following diagnostics information for a previously executed `CREATE FUNCTION` statement:

- Return code (embedded variable `XSQLCODE` with `INTEGER` data type)
- Error location in the SQL statement in the event of a syntax error (embedded variable `XPOSITION` with `INTEGER` data type) piece
- Number indicating position of an error-generating SQL procedure statement in the routine (embedded variable `XSQL_NO` with `INTEGER` data type)
- Character string indicating the error-generating SQL procedure statement in the routine (embedded variable `XSQL` with `VARCHAR(32)` data type)
- Message text (embedded variable `XMESSAGE` with `VARCHAR(254)` data type)

```
CREATE FUNCTION NEXTDAY(INDATE DATE, DAYOFWEEK CHAR(3))
  RETURNS DATE
  BEGIN
   DECLARE SDOW, TDOW INTEGER;
   SET TDOW=(CASE, DAYOFWEEK WHEN 'SUN' THEN 0
      WHEN 'MON' THEN 1
      WHEN 'TUE' THEN 2
      WHEN 'WED' THEN 3
      WHEN 'THU' THEN 4
      WHEN 'FRI' THEN 5
      ELSE 6 END);
   SET SDOW=MOD(DAYS(INDATE),7);
   RETURN  (INDATE + (CASE WHEN TDOW>SDOW THEN TDOW-SDOW
      ELSE 7+TDOW-SDOW END) DAYS);
  END
GET DIAGNOSTICS EXCEPTION 1
    :XSQLCODE=RETURN_SQLCODE,
    :XPOSITION=ERROR_POSITION,
    :XSQL_NO=ERROR_SQL_NO,
    :XSQL=ERROR_SQL,
    :XMESSAGE=MESSAGE_TEXT
```

COMMAND EXECUTE (Execute commands from a UAP)

Function

COMMAND EXECUTE executes HiRDB and OS commands from within a UAP.

COMMAND EXECUTE cannot be executed if HiRDB Control Manager - Agent has not been installed in the HiRDB server. HiRDB Control Manager - Agent is required because it executes commands.

Format

```
COMMAND EXECUTE:command-line-variable,
                :return-code-receiving-variable,
                :execution-results-receiving-area-length-variable,
                :execution-results-length-receiving-variable,
                :execution-results-receiving-variable,
                :executed-command-return-code-receiving-variable,
                :environment-variable-group-name-variable
```

Operands

- : *command-line-variable*

The command line containing the command to be executed at the HiRDB server is set in the command line variable.

Specify an embedded variable declared as a CHAR type (maximum area length is 30,000 bytes). The command line must end with a null character.

Character sets other than the default character set cannot be specified.

Multiple commands should not be specified in the command line variable. If multiple commands are specified, the integrity of the operation cannot be guaranteed.

- : *return-code-receiving-variable*

A return code from execution of COMMAND EXECUTE is set in the return code receiving variable. Specify an embedded variable declared as an INT type.

Following are the values that can be set in the return code receiving variable; in the event of an error, detailed information is set in the execution results receiving variable:

p_rdb_RC_NORM

Command executed normally at the HiRDB server.

p_rdb_RC_ERRPARM

There was an invalid argument.

p_rdb_RC_PROTO

A communication error occurred.

p_rdb_RC_NOTF

No environment variable group was found.

p_rdb_RC_TIMEOUT

Timeout occurred.

p_rdb_RC_SQLERR

Some other error occurred.

- : *execution-results-receiving-area-length-variable*

The area length of the execution results receiving variable is set in the execution results receiving area length variable. Specify an embedded variable declared as an INT type.

The execution results receiving area length should not exceed 2 GB.

- : *execution-results-length-receiving-variable*

The output length for the execution results receiving variable is set in the execution results length receiving variable. Specify an embedded variable declared as an INT type.

- : *execution-results-receiving-variable*

The address of the area allocated for receiving execution results is set in the execution results receiving variable. Specify an embedded variable declared as a PDOUTBUF type.

After COMMAND EXECUTE has executed, the following value is assigned to the execution results receiving variable, with any additional trailing data (specified value of *execution-results-receiving-area-length-variable* - 1) being truncated. A one-byte null character is set at the end of the execution results.

p_rdb_RC_NORM set in the return code receiving variable:

The results line (standard output and standard error output) of executing the command at the HiRDB server is set.

Anything other than p_rdb_RC_NORM set in the return code receiving variable:

A detailed message associated with the error code is set.

- : *executed-command-return-code-receiving-variable*

The return code from the command line executed at the HiRDB server is set in the executed command return code receiving variable. Specify an embedded variable declared as an INT type.

A valid value is set in the executed command return code receiving variable only when COMMAND EXECUTE terminates normally (p_rdb_RC_NORM is set in the executed command return code receiving variable).

If the executed command does not output information to the standard output device or to the standard error output device, the value 0 is assigned to the execution command return code-receiving variable.

- : *environment-variable-group-name-variable*

UNIX edition

Specify as an absolute path name the name (1 to 256 bytes, including NULL characters) of the normal file in which the client environment definition is defined.

Windows edition

Specify a group name (1 to 31 bytes, including NULL characters) that was registered using the HiRDB client environment variable registration tool.

Specify an embedded variable declared as a CHAR type (maximum area length of 256 bytes).

Character sets other than the default character set cannot be specified.

If environment variable groups are not used, set the NULL character in the first byte.

For details about environment variable groups, see the *HiRDB Version 9 UAP Development Guide*.

Common rules

1. All embedded variables that are used must be declared in the embedded variables declaration section.
2. To use COMMAND EXECUTE, the following client environment definition statements must be specified:
   - PDASTHOST (host name of HiRDB Control Manager - Agent)
   - PDASTPORT (port number of HiRDB Control Manager - Agent)
   - PDSYSTEMID (HiRDB identifier of HiRDB Control Manager - Agent)
   - PDASTUSER (authorization identifier for executing commands on the HiRDB server)

   For details about client environment variable definitions, see the *HiRDB Version 9 UAP Development Guide*.
3. COMMAND EXECUTE can be executed while the UAP is connected to HiRDB. However, because control does not return to the UAP until the command terminates, take care so that deadlock does not occur.

4. Do not specify a command that requires a response. HiRDB Control Manager - Agent does not accept requests for entry of a response, so such a command terminates in an error.
5. Any command input file to be executed should be set up at the HiRDB server in advance.
6. Even if execution of the command takes a very long time, control does not return to the UAP until the command terminates. You can specify PDCMDWAITTIME in the client environment definition so that the HiRDB client does not have to wait for too long a time.

   If the client does time out, use the OS's `kill` command (`pdkill` command in the Windows edition) to cancel the HiRDB Control Manager - Agent process or the command being executed.
7. Note that if execution of the command takes a very long time during connection to the HiRDB server, specification of PDSWAITTIME and PDSWATCHTIME in the client environment definition can cause the HiRDB server to detect the timeout and terminate the connection.

Notes

1. During execution of COMMAND EXECUTE, only the following client environment definition settings are in effect:

   PDCLTPATH, PDASTHOST, PDASTUSER, PDUSER, PDASTPORT, PDCMDWAITTIME, PDCMDTRACE, PDSYSTEMID, PDCLTAPNAME
2. For the COBOL language, COMMAND EXECUTE cannot be executed.

Example

Execute the `pdls` command from a UAP at a HiRDB/Single Server:

```
EXEC SQL BEGIN DECLARE SECTION;
   char          CmdLine[30000];
   int           ReturnCode;
   int           OutBufLen;
   int           OutDataLen;
   int           DataLength;
   PDOUTBUF      OutBuf ;
   int           CmdRetCode;
   char          EnvGroup[256];
EXEC SQL END DECLARE SECTION;

strcpy(CmdLine,"c:\HiRDB_S\bin\pdls -d trn");
OutBuf = malloc(30000) ;
OutBufLen = 30000 ;
EnvGroup[0] = '\0';
```

```
EXEC SQL COMMAND EXECUTE    :CmdLine,
                            :ReturnCode,
                            :OutBufLen,
                            :DataLength,
                            :OutBuf,
                            :CmdRetCode,
                            :EnvGroup ;

if (ReturnCode == p_rdb_RC_NORM)
{
      if (CmdRetCode == 0)
      {
          printf("%sSuccessful execution\n",CmdLine) ;
          printf("Execution results:%s\n", OutBuf) ;
      }
      else
      {
          printf("%sExecution failed\n",CmdLine) ;
          printf("Execution results:%s\n", OutBuf) ;
      }
}
else
{
          printf("ReturnCode=%d\n",ReturnCode) ;
          printf("Error details:%s\n",OutBuf) ;
}
```

SQL prefix

Function

The SQL prefix indicates the beginning of an SQL.

Format

```
EXEC SQL
```

Common rules

1. Each SQL must be enclosed between an SQL prefix and an SQL terminator.
2. See the *HiRDB Version 9 UAP Development Guide* for the SQL coding rules.

Examples

Code the `OPEN` statement:

C language

```
EXEC SQL
   OPEN CR1;
```

COBOL language

```
EXEC SQL
   OPEN CR1
END-EXEC.
```

SQL terminator

Function

The SQL terminator indicates the end of an SQL.

Format

C language
```
;
```

COBOL language
```
END-EXEC
```

Common rules

1. Each SQL must be enclosed between an SQL prefix and an SQL terminator.
2. See the *HiRDB Version 9 UAP Development Guide* for the SQL coding rules.

Examples

See the section on the *SQL prefix* for examples.

WHENEVER (Declare embedded exception)

Function

WHENEVER declares processing of a UAP, based on the return code (SQLCODE) set in the SQL Communications Area by HiRDB after an SQL has executed.

Format

```
WHENEVER
    {SQLERROR | SQLWARNING | NOT FOUND}
    {CONTINUE {GO TO | GOTO} [:]host-identifier
     |[DO] PERFORM [:]host-identifier
     |DO {break | continue | 'command-statement' }}
```

Operands

- {SQLERROR | SQLWARNING}

SQLERROR

Specifies the processing to be performed when the SQL does not execute normally due to human error or a HiRDB problem (i.e., when negative values are returned to SQLCODE in the SQL Communications Area and to the SQLCODE variable).

SQLWARNING

Specifies the processing to be performed when the SQL executed normally but a condition requiring a warning to the user was detected (when W is returned to the SQLWARN0 area of the SQL Communications Area or when a positive value other than 100 is returned to the SQLCODE area and the SQLCODE variable of the SQL Communications Area).

- NOT FOUND

Specifies the processing to be performed when there are no more rows to be retrieved during table retrieval (i.e., when the value 100 is returned to SQLCODE in the SQL Communications Area, the value 100 is returned to the SQLCODE variable, and the value 02000 is returned to the SQLSTATE variable).

- {GO TO | GOTO}

Specifies the branching address when execution of the UAP is to branch; following are the host identifiers:

- Label (in the case of C language)
- Section or paragraph name (in the case of COBOL)

- [DO] PERFORM [:] *host-identifier*

Specifies a procedure that is to be executed; following are the applicable procedure specifications:

- Function name (in the case of C language)
- Section or paragraph name (in the case of COBOL)

Object methods cannot be specified.

- `DO` {`break` | `continue` | '*command-statement*'}

This statement either causes branching of the execution of the UAP or executes an arbitrary command statement. The `continue` command and '*command-statement*' can be used in the C and C++ languages.

`DO break`

Executes the `break` statement.

`DO continue`

Executes the `continue` statement.

`DO '`*command-statement*`'`

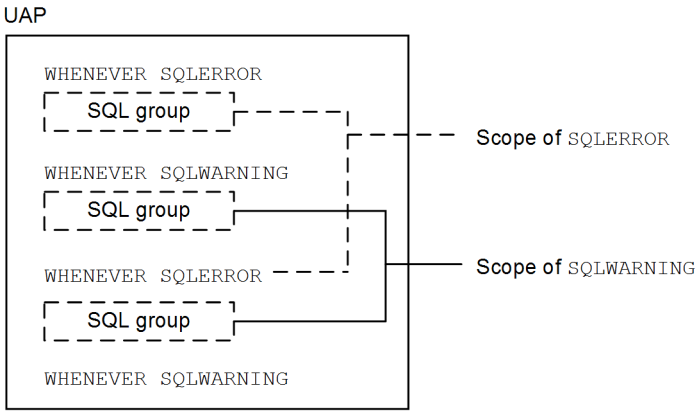
Executes any command statement of the host language, coded as a character string (for example, a function call specifying an argument).

Common rules

1. If there is no `WHENEVER` specification, `CONTINUE` is assumed for all return codes.
2. `WHENEVER` can be specified multiple times in a UAP.
3. SQLs other than the `ROLLBACK` statement cannot be executed in the `SQLERROR` status.
4. After a procedure has executed, control returns to the instruction following the SQL at which the exception-handling event occurred.
5. The scope of the processing specified in an embedded exception declaration depends on its location in the source program; for example, a processing action specification in an embedded exception declaration is effective for all SQL execution results that occur in the source program until another embedded exception declaration specifying the processing of the same singular condition is encountered.

   The following figure shows the scope of processing specified in embedded exception declarations.

Figure 6-1: Scope of processing specified in WHENEVER declarations



6. In the event of overlapping scopes for SQLERROR, SQLWARNING, and NOT FOUND, the SQLs are processed according to the following priorities:

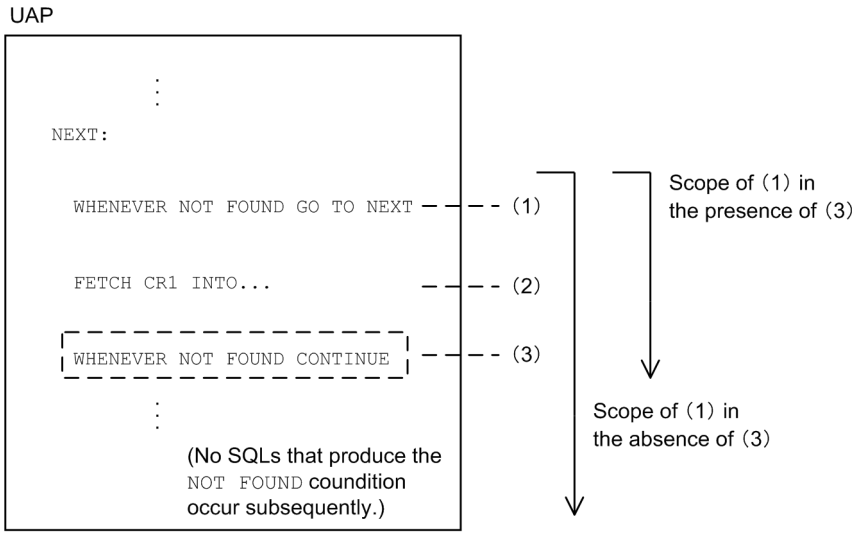
   SQLERROR → NOT FOUND → SQLWARNING

Notes

1. If the need for the action declared in a WHENEVER statement ceases during processing of the SQL, a WHENEVER statement with the CONTINUE option (for continuing the processing) must be inserted at a program location where the statement will be a moot point in order provide for the same exception-handling event (SQLERROR, etc.).

   The following figure shows an example of coding a WHENEVER statement.

Figure 6-2: Example of coding a WHENEVER statement (1)



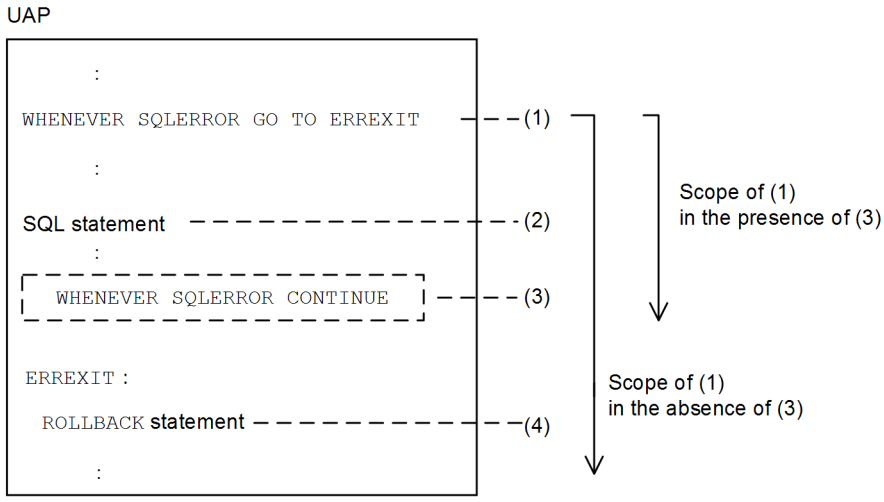
Explanation

If there are no SQLs that produce the NOT FOUND condition in the SQL statements following (3), any processing that deals with the NOT FOUND condition is not required in the SQL statements following (3). Therefore, in WHENEVER statements following (3), the processing is CONTINUE.

2. If the ROLLBACK statement is executed at a branching address because of an execution error in the SQL, care must be taken that an endless loop is not created by a WHENEVER statement with the SQLERROR specification.

   The following figure shows an example of coding a WHENEVER statement.

Figure 6-3: Example of coding a WHENEVER statement (2)



Explanation:

If the WHENEVER statement in (3) is not specified, the WHENEVER statement in (1) will be applied to the ROLLBACK statement in (4). In this case, an infinite loop results if an error occurs during rollback processing. To provide for this contingency, the WHENEVER statement in (3) (which specifies CONTINUE) is specified before the ROLLBACK statement is executed.

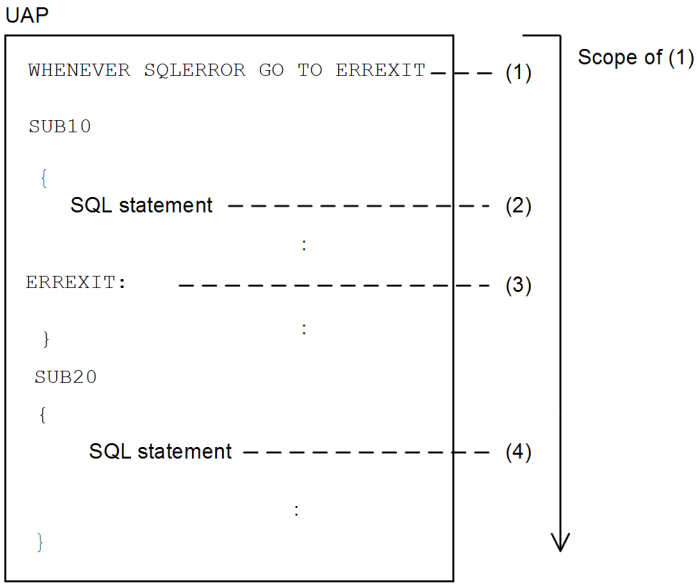
3. The scope of an embedded exception declaration extends to multiple functions in the UAP.

   The destination for the GOTO statement specified in the WHENEVER statement should be specified in the same function as the SQL statement. If the destination is specified out of the function, a compilation error can result.

   A separate WHENEVER statement must be specified, as necessary, for each function.

   The following figure shows an example of coding a WHENEVER statement.

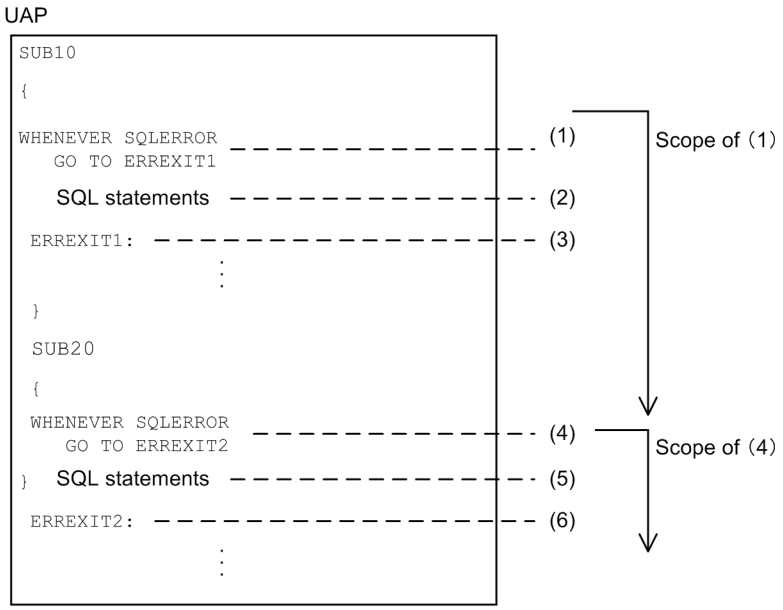
Figure 6-4: Example of coding a WHENEVER statement (3)



Explanation:

If an SQL error (SQLERROR) occurs at (2), control branches to (3) because destination (3) specified in (1) is in the same function as (2). On the other hand, if an SQL error occurs at (4), a compile-time error results because destination (3), specified in (1), is not in the same function as (4).

Figure 6-5: Example of coding a WHENEVER statement (4)



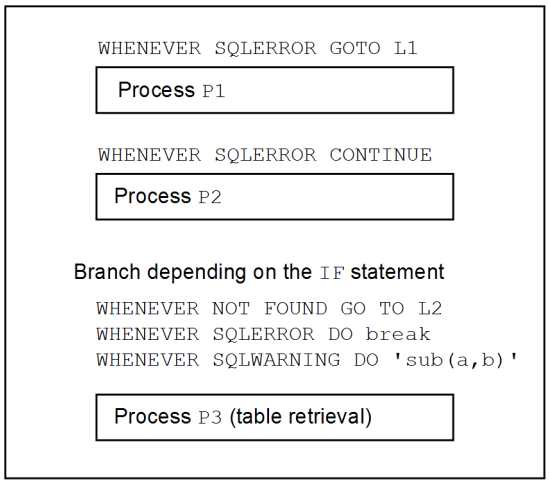
Explanation

Branching is successful because destination (3) that is taken if an SQL error (SQLERROR) occurs at (2) and destination (6) that is taken if an SQL error occurs at (5) are in the same function in their respective SQL statements.

4. The WHENEVER statement should not be declared before FREE CONNECTION HANDLE.

Examples

- If a negative value is returned to SQLCODE after the SQL in process P1 is executed, branch to L1.
- If a negative value is returned to SQLCODE after the SQL in process P2 is executed, cancel control by the WHENEVER statement in order to use an IF statement to determine a branching address.
- If there are no more rows to be retrieved in the table search in process P3, branch to L2.
- Executes the break statement when a negative value is returned to SQLCODE during the retrieval of a table in the process P3.
- Calls a function if a condition about which a warning is to be issued to users occurs during the retrieval of a table in the process P3.


WHENEVER SQLERROR GOTO L1
Process P1
WHENEVER SQLERROR CONTINUE
Process P2
Branch depending on the IF statement
WHENEVER NOT FOUND GO TO L2
WHENEVER SQLERROR DO break
WHENEVER SQLWARNING DO 'sub(a,b)'
Process P3 (table retrieval)

SQLCODE variable

When an SQL is executed, HiRDB sets a return code (SQLCODE). The value returned to the SQLCODE variable is the same as the contents of SQLCODE in the SQL Communications Area.

Because the system includes the necessary declaration statement in the source program during preprocessing, the SQLCODE variable does not need to be declared in the UAP. In C language, the data type of the SQLCODE variable is declared as a signed long int; in COBOL, it is declared as S9(9) COMPUTATIONAL.

To reference the SQLCODE variable, the variable name SQLCODE must be specified.

In an environment the uses the multi-connection facility, the connection handle used by SQLCODE must be declared with DECLARE CONNECTION HANDLE SET.

SQLSTATE variable

When an SQL is executed, HiRDB sets a return code (SQLSTATE). The SQLSTATE variable is a 5-digit character string composed of a 2-digit class and a 3-digit subclass.

Because the system includes the necessary declaration statement in the source program during preprocessing, the SQLSTATE variable does not need to be declared in the UAP. In C language, the data type of the SQLSTATE variable is declared as char[5].; in COBOL, it is declared as PIC X(5).

To reference the SQLSTATE variable, the variable name SQLSTATE must be specified.

In an environment that uses the multi-connection facility, the connection handle used by SQLSTATE must be declared with DECLARE CONNECTION HANDLE SET.

The value to which SQLSTATE is set depends on the settings of the client environment variable PDSTANDARDSQLSTATE and of pd_standard_sqlstate in the system common definition. For details about SQLSTATE, see the manual *HiRDB Version 9 Messages*.

The meaning of the classes and their relationships to subclasses are shown below.

Class	Subclass	Condition
00	000	Normal termination
01	*nnn*	Normal termination (with warning)
02	000	No data
40	*nnn*	Abnormal termination (transaction was rolled back)
R2	000	No data (in searches using a list, however, data sometimes exists in the row that is returned when the list is created)
mm	*nnn*	Abnormal termination

Notes

The meanings of *mm* and *nnn* are as follows.

mm: The SQLSTATE class is set as described in the manual *HiRDB Version 9 Messages*.

nnn: The SQLSTATE subclass is set as described in the manual *HiRDB Version 9 Messages*.

However, *mm* and *nnn* may vary depending on the value of the HiRDB facility extensions.

PDCNCTHDL type variable declaration

Function

The PDCNCTHDL-type variable declaration declares the connection handle-type variable to be used in an environment where a multi-connection function is used.

Format 1: C language

```
PDCNCTHDL connection-handle-variable-name
```

Operand

- *connection-handle-variable-name*

Specifies the connection handle that is to be used.

Common rules

The PDCNCTHDL-type variable declaration is used when an SQL statement is executed in an environment where a multi-connection function is used.

Notes

1. PDCNCTHDL-type variables are declared in the embedded SQL declaration section.
2. A PDCNCTHDL-type variable cannot be declared as an array.

Example

Declare a PDCNCTHDL-type variable named CnctHdl:

```
EXEC SQL BEGIN DECLARE SECTION;
    PDCNCTHDL    CnctHdl;
EXEC SQL END DECLARE SECTION;
```

Format 2: COBOL language

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC
01 connection-handle-variable-name SQL TYPE IS PDCNCTHDL.
EXEC SQL END DECLARE SECTION END-EXEC
```

Operands

- *connection-handle-variable-name*

Specifies the name of the connection handle to be used.

Common rule

1. Declare PDCNCTHDL-type variables when executing the SQL statement in an environment using the multi-connection facility.

Notes

1. PDCNCTHDL-type variables are declared in the embedded SQL declaration section.
2. PDCNCTHDL-type variables cannot use the OCCURS clause.

Example

Declare a PDCNCTHDL-type variable named CONNECTHDL.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC
  01 CONNECTHDL SQL TYPE IS PDCNCTHDL.
EXEC SQL END DECLARE SECTION END-EXEC
```

INSTALL JAR (Register JAR file)

Function

INSTALL JAR registers a JAR file at the HiRDB server. The file is registered in the JAR file storage directory associated with the authorization identifier that is connected.

Format

```
INSTALL JAR{:embedded-variable|'character-string'}
```

Operands

- {:*embedded-variable*|'*character-string*'}

Specifies as an absolute or relative path name for the JAR file that is to be registered.

:*embedded-variable*

Specifies a VARCHAR-type embedded variable that contains the name that is to be registered as its value.

Character sets other than the default character set cannot be specified.

'*character-string*'

Specifies as a character string literal a name for the JAR file that is to be registered.

Common rules

1. You must be connected to the HiRDB server to execute INSTALL JAR.
2. Any error code is returned to SQLCODE.
3. A JAR file at a different server machine cannot be specified.
4. Wildcards cannot be used.
5. If a JAR file with the specified name is already registered, an error results; the existing file is not overwritten.
6. INSTALL JAR should be executed before any transaction is started.

Example

Register a JAR file named c:\work\sampleproc.jar by setting its name in an embedded variable:

```
EXEC SQL BEGIN DECLARE SECTION ;
struct {
   short  len ;
   char   str[256] ;
```

```
        } filename ;
        EXEC SQL END DECLARE SECTION ;
        EXEC SQL CONNECT ;
        strcpy(filename.str,"c:\work\sampleproc.jar") ;
        filename.len = strlen(filename.str) ;
        EXEC SQL INSTALL JAR :filename ;
```

REPLACE JAR (Re-register JAR file)

Function

REPLACE JAR re-registers a JAR file at the HiRDB server. The file is re-registered in the JAR file storage directory associated with the authorization identifier that is connected.

Format

```
REPLACE JAR{:embedded-variable|'character-string'}
```

Operands

- {:*embedded-variable*|'*character-string*'}

Specifies as an absolute or relative path name a name for the JAR file that is to be re-registered.

:*embedded-variable*

Specifies a VARCHAR-type embedded variable that contains the name that is to be re-registered as its value.

Character sets other than the default character set cannot be specified.

'*character-string*'

Specifies as a character string literal a name for the JAR file that is to be re-registered.

Common rules

1. You must be connected to the HiRDB server to execute REPLACE JAR.
2. Any error code is returned to SQLCODE.
3. A JAR file at a different server machine cannot be specified.
4. Wildcards cannot be used.
5. If a JAR file with the specified name is already registered, an error results; the existing file is not overwritten.
6. REPLACE JAR should be executed before any transaction is started.

Example

Re-register a JAR file named c:\work\sampleproc.jar by setting its name in an embedded variable:

```
EXEC SQL BEGIN DECLARE SECTION ;
struct {
    short  len ;
```

```
    char   str[256] ;
} filename ;
EXEC SQL END DECLARE SECTION ;
EXEC SQL CONNECT ;
strcpy(filename.str,"c:\work\sampleproc.jar") ;
filename.len = strlen(filename.str) ;
EXEC SQL REPLACE JAR :filename ;
```

REMOVE JAR (Remove JAR file)

Function

REMOVE JAR removes a JAR file from the HiRDB server. The file is removed from the JAR file storage directory associated with the authorization identifier that is connected.

Format

```
REMOVE JAR{:embedded-variable|'character-string'}
```

Operands

- {:*embedded-variable*|'*character-string*'}

Specifies the name of the JAR file that is to be removed. The JAR file cannot be specified with an absolute or relative path name.

:*embedded-variable*

Specifies a VARCHAR-type embedded variable that contains the name of the JAR file that is to be removed as its value.

Character sets other than the default character set cannot be specified.

'*character-string*'

Specifies as a character string literal the name of the JAR file that is to be removed.

Common rules

1. You must be connected to the HiRDB server to execute REMOVE JAR.
2. Any error code is returned to SQLCODE.
3. A JAR file at a different server machine cannot be specified.
4. Wildcards cannot be used.
5. REMOVE JAR should be executed before any transaction is started.

Example

Remove the JAR file named sampleproc.jar by setting its name in an embedded variable:

```
EXEC SQL BEGIN DECLARE SECTION ;
struct {
    short  len ;
    char   str[256] ;
} filename ;
```

```
EXEC SQL END DECLARE SECTION ;
EXEC SQL CONNECT ;
strcpy(filename.str,"sampleproc.jar") ;
filename.len = strlen(filename.str) ;
EXEC SQL REMOVE JAR :filename ;
```

INSTALL CLIB (Install external C library file)

Function

In order to execute an external C stored routine on the machine that the UAP is executed on, INSTALL CLIB installs a new external C library file on the HiRDB server on which the external C stored routine is implemented.

Format

```
INSTALL CLIB {:embedded-variable|'character-string'}
```

Operands

- {:*embedded-variable*|'*character-string*'}

Specifies the name of the external C library file to be installed. Specify an absolute path name or a relative path name for the name of the external C library file.

:*embedded-variable*

Specifies the name of the external C library file as a VARCHAR type embedded variable.

Character sets other than the default character set cannot be specified.

'*character-string*'

Specifies the name of the external C library file as a character string literal.

Common rules

1. Before you execute INSTALL CLIB, the computer you are using must be connected to a HiRDB server.
2. Error codes are returned in SQLCODE.
3. External C library files on other servers cannot be specified.
4. Wildcards cannot be used.
5. If an external C library file with the same name is already installed, executing INSTALL CLIB results in an error and the existing external C library file is not overwritten.
6. INSTALL CLIB must be executed before the start of a transaction.
7. The external C library file must be created on the same platform on which it is to be installed.

Example

Specify and install an external C library file (`c:\work\sampleproc.dll`) using an embedded variable.

```
EXEC SQL BEGIN DECLARE SECTION ;
struct {
    short  len ;
    char   str[256] ;
} filename ;
EXEC SQL END DECLARE SECTION ;
EXEC SQL CONNECT ;
strcpy(filename.str,"c:\\work\\sampleproc.dll") ;
filename.len = strlen(filename.str) ;
EXEC SQL INSTALL CLIB :filename ;
```

REPLACE CLIB (Replace external C library file)

Function

In order to execute an external C stored routine on the machine that the UAP is executed on, REPLACE CLIB replaces an existing external C library file on the HiRDB server on which the external C stored routine is implemented.

Format

```
REPLACE CLIB {:embedded-variable|'character-string'}
```

Operands

- {:*embedded-variable*|'*character-string*'}

Specifies the name of the external C library file to be replaced. Specify an absolute path name or a relative path name for the name of the external C library file.

:*embedded-variable*

Specifies the name of the external C library file as a VARCHAR type embedded variable.

Character sets other than the default character set cannot be specified.

'*character-string*'

Specifies the name of the external C library file as a character string literal.

Common rules

1. Before you execute REPLACE CLIB, the computer you are using must be connected to a HiRDB server.
2. Error codes are returned in SQLCODE.
3. External C library files on other servers cannot be specified.
4. Wildcards cannot be used.
5. If an external C library file with the same name is already installed, it is overwritten.
6. The external C library file will be installed on the HiRDB server if it is not already installed.
7. REPLACE CLIB must be executed before the start of a transaction.
8. The external C library file must be created on the same platform on which it is to

be replaced.

9. A C library file cannot be replaced during execution of a transaction.

Example

Specify and replace an external C library file (`c:\work\sampleproc.dll`) using an embedded variable.

```
EXEC SQL BEGIN DECLARE SECTION ;
struct {
    short  len ;
    char   str[256] ;
} filename ;
EXEC SQL END DECLARE SECTION ;
EXEC SQL CONNECT ;
strcpy(filename.str,"c:\\work\\sampleproc.dll") ;
filename.len = strlen(filename.str) ;
EXEC SQL REPLACE CLIB :filename ;
```

REMOVE CLIB (Remove external C library file)

Function

REMOVE CLIB removes an external C library file used to implement an external C stored routine that has been installed on a HiRDB server.

Format

```
REMOVE CLIB {:embedded-variable|'character-string'}
```

Operands

- {:*embedded-variable*|'*character-string*'}

Specifies the name of the external C library file to remove. Specify only the file name, without a directory path.

:*embedded-variable*

Specifies the name of the external C library file as a VARCHAR type embedded variable.

Character sets other than the default character set cannot be specified.

'*character-string*'

Specifies the name of the external C library file as character string literal.

Common rules

1. Before executing REMOVE CLIB, the computer you are using must be connected to a HiRDB server.
2. Error codes are returned in SQLCODE.
3. Wildcards cannot be used.
4. REMOVE CLIB must be executed before the start of a transaction.
5. A C library file cannot be removed during execution of a transaction.

Example

Specify and remove an external C library file (sampleproc.dll) using an embedded variable.

```
EXEC SQL BEGIN DECLARE SECTION ;
struct {
    short  len ;
```

```
    char   str[256] ;
} filename ;
EXEC SQL END DECLARE SECTION ;
EXEC SQL CONNECT ;
strcpy(filename.str,"sampleproc.dll") ;
filename.len = strlen(filename.str) ;
EXEC SQL REMOVE CLIB:filename ;
```

DECLARE AUDIT INFO SET (Set user connection information)

Function

DECLARE AUDIT INFO SET specifies, for applications that access a HiRDB server, account information and other user connection information. The specified user connection information remains in effect until it is terminated, and is output to the HiRDB server's audit trail when the SQL code is executed.

Format

```
DECLARE AUDIT INFO SET POS=:embedded-variable,
                       INF=:embedded-variable[:indicator-variable]
```

Operands

- POS=:*embedded-variable*

Specifies the number of additional user information items that are to be set as INTEGER type embedded variables. The numbers corresponding to each additional user information item are as follows:

- 1: Additional user information item 1
- 2: Additional user information item 2
- 3: Additional user information item 3

- INF=:*embedded-variable*[:*indicator-variable*]

Specifies the user connection information as an embedded variable declared as a VARCHAR type (area length of 100 bytes or less). Specify the null value to remove information that has already been set. Additional user information item 1 through additional user information item 3 is included in the user connection information, and all are output in the audit trail.

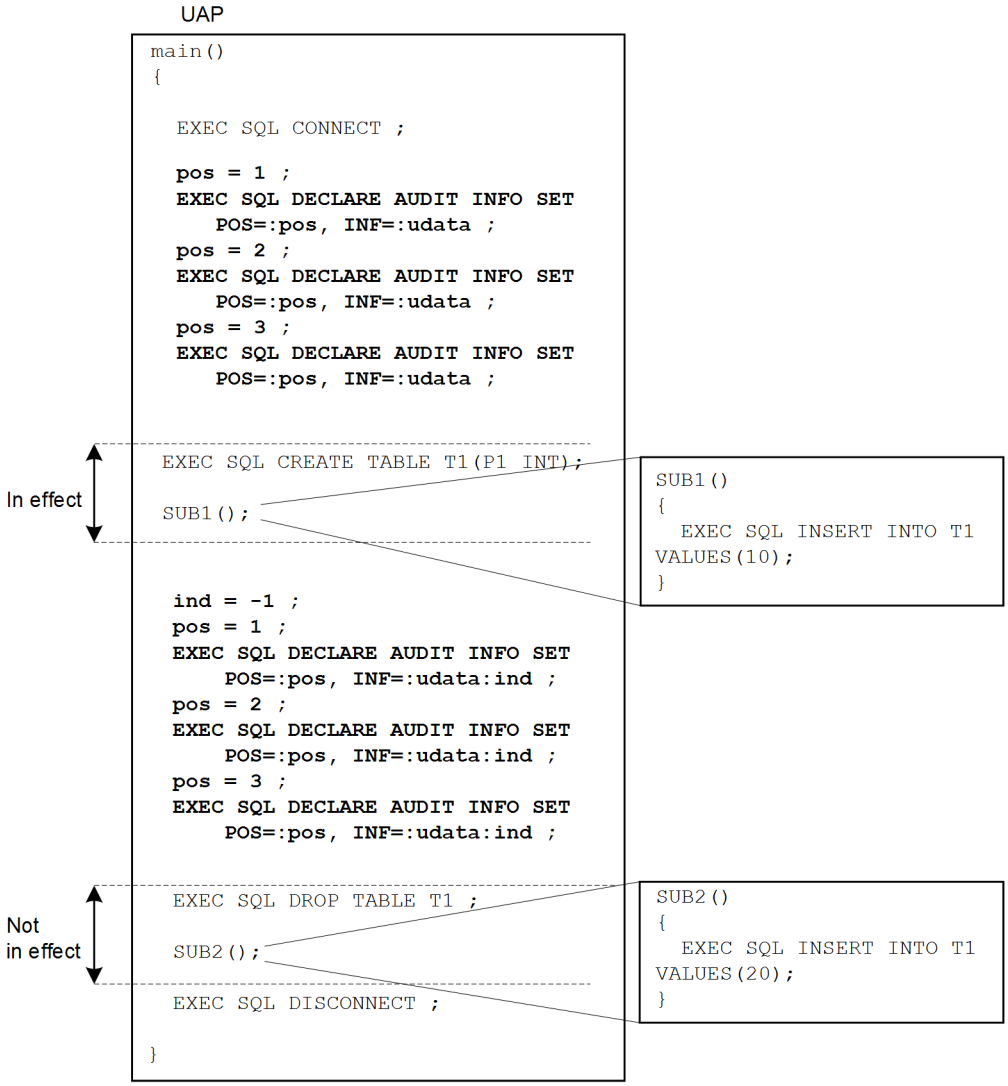
Character sets other than the default character set cannot be specified.

Common rules

1. The embedded variable to be used must be declared in the embedded variable declaration section.
2. Error codes are returned in SQLCODE.
3. If DECLARE AUDIT INFO SET is re-executed when user connection information is already set, the existing user connection information is updated.
4. The user connection information that is set is in effect until terminated by setting

a null value in DECLARE AUDIT INFO SET. The figure below shows the effective scope of the user connection information that is set in DECLARE AUDIT INFO SET.

Figure 6-6: Effective scope of user connection information set in DECLARE AUDIT INFO SET



5. If user connection information is set in DECLARE AUDIT INFO SET before the CONNECT statement is executed, that information is carried over for the CONNECT.
6. If DISCONNECT is executed while user connection information is set, the user connection information is not cleared. To clear it, specify the null value in INF

and execute DECLARE AUDIT INFO SET.

7. If there are multiple connections, the user connection information must be set separately for each connection handle.

Note

1. When the CONNECT statement is executed, user connection information is not output to the audit trail.

Examples

Example 1: Set user connection information (additional user information item 1 through 3) that is specified in an embedded variable.

```
EXEC SQL BEGIN DECLARE SECTION ;
SQL TYPE IS VARCHAR(100) udata ;
long pos ;
EXEC SQL END DECLARE SECTION ;

strcpy(udata.str,"userA") ;
udata.len = strlen(udata.str) ;
pos = 1;
EXEC SQL DECLARE AUDIT INFO SET POS=:pos, INF=:udata ;
strcpy(udata.str," user data B ") ;
udata.len = strlen(udata.str) ;
pos = 2;
EXEC SQL DECLARE AUDIT INFO SET POS=:pos, INF=:udata ;
strcpy(udata.str," user data C ") ;
udata.len = strlen(udata.str) ;
pos = 3;
EXEC SQL DECLARE AUDIT INFO SET POS=:pos, INF=:udata ;
```

Example 2: Clear the contents of the user connection information (additional user information items 1 through 3).

```
EXEC SQL BEGIN DECLARE SECTION ;
SQL TYPE IS VARCHAR(100) udata ;
long pos ;
short ind ;
EXEC SQL END DECLARE SECTION ;

ind = -1;
pos = 1;
EXEC SQL DECLARE AUDIT INFO SET POS=:pos, INF=:udata:ind ;
pos = 2;
EXEC SQL DECLARE AUDIT INFO SET POS=:pos, INF=:udata:ind ;
pos = 3;
```

```
EXEC SQL DECLARE AUDIT INFO SET POS=:pos, INF=:udata:ind ;
```

Chapter

7. Routine Control SQL

This chapter explains the syntax and structure of routine control SQL.

General rules

Routine control SQL statements can be specified in a routine definition SQL procedure statement.

The following table lists the types and functions of routine control SQL.

Table 7-1: Types and functions of routine control SQL

Type	Function
Assignment statement (Assign value)	Assigns a value to an SQL variable or an SQL parameter.#
Compound statement (Execute multiple statements)	Executes several SQL statements as a group, treating them as a single SQL statement.
`IF` statement (Execute by conditional branching)	Executes SQL statements that meet a given set of conditions.
`LEAVE` statement(Exit statement)	Exits from a compound statement or the `WHILE` statement and terminates the execution of the statement.
`RETURN` statement (Return function return value)	Returns a return value from a function.
`FOR` statement (Repeat a statement on rows)	Repeatedly executes the SQL statement on rows in the table.
`WHILE` statement (Repeat statements)	Executes repeatedly a set of SQL statements.
`WRITE LINE` statement (Output character string to a file)	Outputs a character string to a file.
`SIGNAL` statement (Signal an error)	Signals an error.
`RESIGNAL` statement (Resignal an error)	Resignals an error.

#: For details about the assignment statement in an SQL procedure statement, see *Assignment statement Format 1 (Assign a value to an SQL variable or SQL parameter)*.

In addition to the routine control SQL statements shown in *Table 7-1*, the following SQL statements, which cannot be specified in a function body, can be specified as SQL procedure statements in a routine:

- `CALL` statement (Procedure call)
- `CLOSE` statement (Close cursor)
- `DECLARE CURSOR` (Declare cursor)
- `DELETE` statement (Delete row)

- `FETCH` statement (Fetch data)
- `INSERT` statement (Insert row)
- `OPEN` statement (Open cursor)
- `PURGE TABLE` statement (Delete all rows)
- 1-row `SELECT` statement (Retrieve one row)
- `UPDATE` statement (Update data)
- `COMMIT` statement (Normal termination of transaction)
- `LOCK` statement (Lock table)
- `ROLLBACK` statement (Cancel transaction)

Compound statement (Execute multiple statements)

Function

This compound statement executes a group of SQL statements as a single, compound SQL statement.

Format

[*starting-label*:]
BEGIN
[{*SQL-variable-declaration*;|*cursor-declaration*;|*condition-declaration*;|*handler-declaration*;}]...
[*SQL-procedure-statement*;]...

END[*end-label*]
SQL-variable-declaration::=DECLARE *SQL-variable-name*[,
SQL-variable-name]...*data-type* [DEFAULT *clause*]
DEFAULT *clause*::= DEFAULT [*default-value*]
condition-declaration::=DECLARE *condition-name* CONDITION
[*FOR-SQLCODE-value*]
handler-declaration::=DECLARE *handler-type*
HANDLER FOR *condition-value*[, *condition-value*]... *handler-action*
handler-type::={CONTINUE|EXIT}
condition-value::={SQLERROR|NOT FOUND|*condition-name*|*SQLCODE-value*}
handler-action::=*SQL-procedure-statement*
SQLCODE-value::=SQLCODE [VALUE] *integer-literal*

Operands

- [*starting-label*]

Specifies the statement label for a compound statement.

- BEGIN

Specifies the beginning of a compound statement.

- *SQL-variable-declaration*::=DECLARE *SQL-variable-name*[,
  SQL-variable-name]...*data-type* [DEFAULT *clause*]

Declares the SQL variables that are used in the compound statement. If an SQL variable is allocated, the default value for the SQL variable is assigned as an initial value. The default for the SQL variable is specified in the DEFAULT clause. If the DEFAULT column is omitted, the default for the SQL variable is the null value.

SQL-variable-name

Specifies the name of the SQL variable being declared.

data-type

Specifies the data type of the SQL variable being declared.

Rules on SQL variable declarations

1. The name of the SQL variable declared in *SQL-variable-declaration* must be distinct from any parameter names used in the routine.
2. The name of the SQL variable declared in *SQL-variable-declaration* that is directly included in the same compound statement cannot be specified in duplicate.
3. The SQL variable declared in a compound statement is allocated at the beginning of the compound statement and is released at the end.
4. The scope of an SQL variable is inside the compound statement in which it is declared and in the handler action for a handler declaration that is declared in the same compound statement. If a compound statement is specified in an SQL procedure statement in the compound statement, the SQL variable also remains in effect in the inner compound statement.
5. If a compound statement is specified in an SQL procedure statement in a compound statement, and if the SQL variable declared in the outer compound statement is identical to the name of the SQL variable declared in the inner compound statement, the SQL variable declared inside remains in effect in the inner compound statement. When the inner compound statement terminates, the SQL variable declared outside takes effect.
6. `BOOLEAN` cannot be specified as a data type for the SQL variable declared in *SQL-variable-declaration*.

`DEFAULT` *clause*`::= DEFAULT [`*default-value*`]`

For rules on the `DEFAULT` clause, see the rules on the `DEFAULT` clause in *CREATE TABLE (Define table)*.

The following operands cannot be specified in *default-value*:
`CURRENT_TIMESTAMP[(`*fractional-second-precision*`)] USING BES` and
`CURRENT TIMESTAMP[(`*fractional-second-precision*`)] USING BES`.

- *cursor-declaration*

Declares the cursor to be used in the compound statement.

Rules on cursor declarations

1. The name of the cursor declared in *cursor-declaration* that is directly included in the same compound statement cannot be specified in duplicate.
2. The cursor declared in a compound statement is allocated at the beginning of the compound statement and is released at the end. However, cursors that

were declared by specifying `WITH RETURN` are not released.

3. The scope of a cursor is inside the compound statement in which it is declared and in the handler action for a handler declaration that is declared in the same compound statement. If a compound statement is specified in an SQL procedure statement in the compound statement, the cursor also remains in effect in the inner compound statement.
4. If a compound statement is specified in an SQL procedure statement in a compound statement, and if the cursor declared in the outer compound statement is identical to the name of the cursor declared in the inner compound statement, the cursor declared inside remains in effect in the inner compound statement. When the inner compound statement terminates, the cursor declared outside takes effect.

- *condition-declaration* `::=DECLARE` *condition-name* `CONDITION` [*FOR-SQLCODE-value*]

Declares a handler declaration, a `SIGNAL` statement, or the condition name to be used in the `RESIGNAL` statement, and the associated value of `SQLCODE`.

condition-name

Specifies the name of the condition to be declared.

FOR-SQLCODE-value

Specifies the value of `SQLCODE` to be associated with the condition being declared.

Rules on condition declarations

1. The same condition name cannot be specified in duplicate in another condition declaration that is directly included in the same compound statement.
2. The scope of a condition name is inside the compound statement in which it is declared and in the handler action for a handler declaration that is declared in the same compound statement. If a compound statement is specified in an SQL procedure statement in the compound statement, the condition name also remains in effect in the inner compound statement.
3. If a compound statement is specified in an SQL procedure statement in a compound statement, and if the condition name declared in the outer compound statement is identical to the condition name declared in the inner compound statement, condition name cannot be declared in the inner compound statement.
4. When specifying more than one condition declaration that is directly included in the same compound statement, the value of `SQLCODE` cannot be specified.

5. When declaring the condition name to be used in the SIGNAL or RESIGNAL statement, omit the option *FOR-SQLCODE-value*; if this is specified, an error may occur.

- *handler-declaration*::=DECLARE *handler-type* HANDLER FOR *condition-value*[, *condition-value*]... *handler-action*

Declares the handler that performs exception processing in the compound statement.

When the condition name of the SQLCODE value, SIGNAL statement, or RESIGNAL statement in the results of execution of the SQL statement in the compound statement matches the condition value specified in the handler declaration, the handler receives control and executes the *handler-action*.

handler-type::={CONTINUE|EXIT}

CONTINUE

Upon execution of *handler-action*, transfers control to the SQL procedure statement following the SQL procedure statement in which the exception occurred. If the SQL procedure statement in which the exception occurred is an IF or WHILE statement of a routine control SQL statement, control is transferred to the SQL procedure statement following END IF or END WHILE[*end-label*].

EXIT

After the execution of *handler-action*, transfers control to the end of the compound statement in which the handler declaration was specified.

condition-value::={SQLERROR|NOT FOUND|*condition-name*|*SQLCODE-value*}

Specifies the condition under which the handler takes effect.

SQLERROR

This option is specified when calling the handler if SQLERROR occurs. The condition SQLERROR corresponds to the case in which SQLCODE < 0.

NOT FOUND

This option is specified to call the handler when NOT FOUND occurs. The NOT FOUND option corresponds with SQLCODE = 100.

condition-name

Specifies the condition name for the condition under which a handler is called.

The *condition-name* operand must be defined in the condition declaration and must include the handler declaration in its scope.

If an SQLCODE value corresponding to *condition-name* is defined in the condition declaration, the handler is called when the SQLCODE matches the value. If an

SQLCODE value corresponding to *condition-name* is not defined in the condition declaration, the handler is called only when the SIGNAL or RESIGNAL statement specifying the condition name is executed.

SQLCODE-value

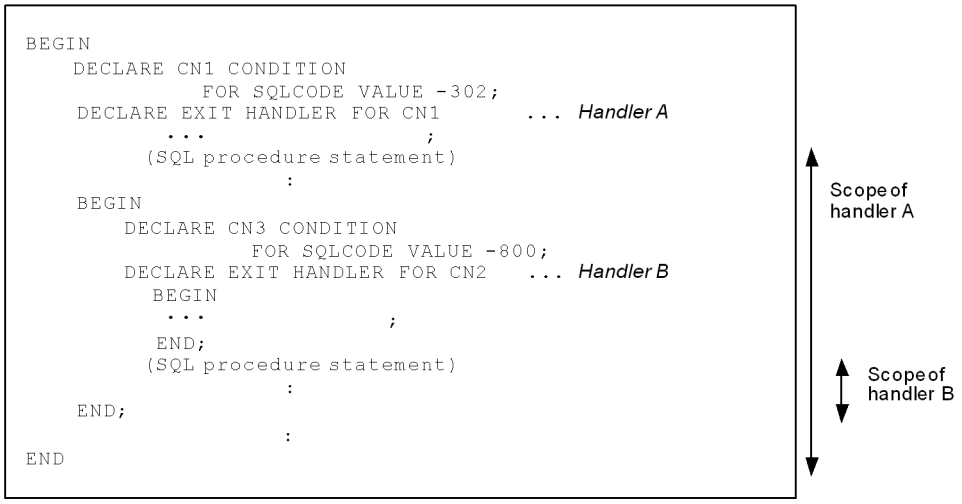
Specifies the value of SQLCODE that indicates the condition under which the handler is called.

handler-action::=*SQL-procedure-statement*

Specifies the SQL statement to be executed when the handler is called.

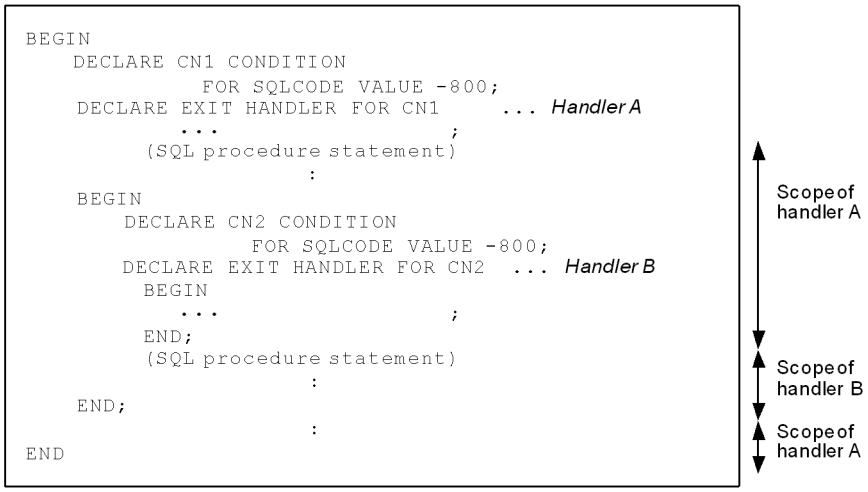
Rules on handler declarations

1. The scope of a handler is the SQL procedure statement in the compound statement in which the handler is declared. If a compound statement is specified in the SQL procedure statement in the compound statement, the handler is effective in the entire inner compound statement. However, any SQL procedure statement in the handler declaration in the compound statement in which the handler is declared is nullified. An example is shown as follows:



2. If either SQLERROR or NOT FOUND is specified in *condition-value* in the handler declaration, *SQLCODE-value* or *condition-name* cannot be specified at the same time.
3. In a handler declaration, the same condition value cannot be specified in duplicate. Similarly, a condition name indicating the same SQLCODE as *SQLCODE-value* cannot be specified.

4. A condition value indicating the same condition cannot be specified in another handler declaration that is included in the same compound statement.
5. A handler declaration that specified either SQLERROR or NOT FOUND in *condition-value* is called a general handler declaration; all other handler declarations are called special handler declarations. If a general handler declaration and a special handler declaration specifying a condition value that indicates the same SQL execution status (abnormal termination, normal termination with warning, or no data) are defined in the same compound statement, only the special handler declaration takes effect on the SQLCODE value that was specified in the special handler declaration.
6. If a compound statement is specified as an SQL procedure statement in a compound statement, and if Handler A declared in the outer compound statement and Handler B declared in the inner compound statement specify the same SQLCODE or condition name, the inner Handler B prevails in the inner compound statement. Upon termination of the inner compound statement, the outer Handler A takes effect again. An example of this is shown in the following:



7. If the handler action does not terminate normally (SQLCODE does not equal 0), and if there is another handler that meets the condition, that handler is called.
8. The value is SQLCODE = 0 immediately after the commencement of handler action.

- *SQL-procedure-statement*

Specifies the SQL procedure statement to be executed in the compound statement.

- `END [`*end-label*`]`

Specifies the end of the compound statement. In *end-label*, specify a statement label.

- *SQLCODE-value*`::=SQLCODE[VALUE]` *integer-literal*

Specifies the value of `SQLCODE` with an integer literal.

The value `0`, which indicates normal termination, cannot be specified as a value of `SQLCODE`. The following table lists the integer literals specified in the value of `SQLCODE`.

Table 7-2: Integer literals specified in SQLCODE value

Execution status of SQL statement	Value of SQLCODE
Normal termination (with warning)	> 0 (≠ 100, 110)
No data	100
Abnormal termination	< 0

The following table lists the messages corresponding to `SQLCODE` that can appear in HiRDB.

Table 7-3: Messages corresponding to SQLCODE that can appear in HiRDB

SQLCODE	Corresponding message ID
-*yyy*	`KFPA11`*yyy*
`-1`*yyy*	`KFPA19`*yyy*
`-3`*yyy*	`KFPA18`*yyy*
yyy	`KFPA12`*yyy*
`+3`*yyy*	`KFPA13`*yyy*

Common rules

1. If a compound statement is specified in the outermost SQL procedure statement and a begin label is omitted, the routine identifier for that routine is assumed to be the statement label. If a compound statement is specified in an SQL procedure statement in a compound statement, and a begin label for the inner compound statement is omitted, it is assumed that there is no statement label.
2. When specifying an end label, specify a statement label with the same name as the begin label.
3. The scope of a statement label is from the beginning of the compound statement

in which the statement label is specified at the end of the compound statement. A statement label identical to a statement label for other statements or group variable names included in the compound statement cannot be specified. If a handler declaration is included in the compound statement, that handler declaration is exempted from this rule. The following code provides an example of where statement labels of the same name can or cannot be specified:

```
AAA: BEGIN  ..........................................1
  DECLARE CN1 CONDITION FOR SQLCODE VALUE -800;
  DECLARE EXIT HANDLER FOR CN1
  AAA: BEGIN  ..........................................2
      :
  END AAA;
  AAA: BEGIN  ..........................................3
    DECLARE CN2 CONDITION FOR SQLCODE VALUE -800;
    DECLARE EXIT HANDLER FOR CN2
      :
  END AAA;
    :
END AAA
```

Explanation

Although the statement label in line 2 is the same name as line 1, it can be specified because it is in the handler declaration.

The statement label in line 3 is the same name as in line 1; it cannot be specified because it is not in the handler declaration.

4. Specified SQL procedure statements are executed in the order in which they are specified.
5. If an error occurs during the execution of an SQL procedure statement, the transaction is nullified only if the error is implicitly subject to a rollback. Any error occurring during execution of an SQL procedure statement for a trigger action is always implicitly subject to a rollback.
6. If an error occurs during execution of an SQL procedure statement, the error is handled according to the following rules:
   - If an error without implicit rollback occurs:

     If there is a handler meeting the conditions, the exception processing of the handler is executed. If a condition-meeting handler does not exist, the execution of the SQL routine is terminated at that time, and an error is returned. Any subsequent SQL procedure statements are not executed.
   - If an error with an implicit rollback occurs:

     Even if there is a condition-meeting handler, exception processing is not executed. Execution of the SQL routine or the trigger at that time is

terminated, and an error is returned. Any subsequent SQL procedure statements are not executed.

7. The maximum number of nesting levels for compound statements and the FOR statement is 255.
8. Any of the following names specified in *SQL-variable-name*, *cursor-name*, or *condition-name* must be enclosed in double quotation marks (").
   - CONDITION
   - EXIT
   - HANDLER

Note

1. Compound statements can be specified in an SQL routine or a trigger.

Examples

1. Defines a procedure (PROC1) that applies a 30% discount on the unit price of a product for which the quantity indicated in the inventory table (STOCK) is 1, 000 or greater, deletes the row if the result of the discount is 0, and in other cases applies a 10% discount to the unit price:

```
CREATE PROCEDURE PROC1(OUT OUTDATA INT)
BEGIN
  DECLARE CR1 CURSOR FOR SELECT QUANTITY FROM STOCK ;
  OPEN CR1 ;
  WHILE SQLCODE=0 DO
    FETCH CR1 INTO OUTDATA ;
    IF SQLCODE=0 THEN
      IF OUTDATA|>=1000 THEN
       UPDATE STOCK SET PRICE = (1-0.3)*PRICE WHERE CURRENT
OF CR1 ;
      ELSE IF OUTDATA=0 THEN
        DELETE FROM STOCK WHERE CURRENT OF CR1 ;
      ELSE
       UPDATE STOCK SET PRICE = (1-0.1)*PRICE WHERE CURRENT
OF CR1 ;
      END IF ;
    END IF ;
  END;
END
```

2. Defines a procedure (PROC2) that updates the quantity of a specified product code in the inventory table (STOCK):
   - The specified quantity is less than or equal to 0 (the condition illegal_value defined in the condition is TRUE):

The SIGNAL statement generates an error, and the exception processing sets a message in the output parameter. Execution of the SQL procedure statement terminates.

- No data with a specified product code is found (NOT FOUND):

  The exception processing sets a message in the output parameter. Execution of the SQL procedure statement terminates.

- Attempt to update a NOT NULL-constrained column with the NULL value (SQLCODE = -210):

  The exception processing sets a message in the output parameter. Execution of the SQL procedure statement continues.

```
CREATE PROCEDURE PROC2(IN UPCODE CHAR(4), IN UQUANTITY INT,
                       OUT MSG MVARCHAR(255))
BEGIN
  DECLARE PQUANTITY INT;
  DECLARE illegal_value CONDITION ;
  DECLARE EXIT HANDLER FOR illegal_value
    SET MSG=M'Invalid value as a quantity';
  DECLARE EXIT HANDLER FOR NOT FOUND
    SET MSG=M'Specified product code not found';
  DECLARE CONTINUE HANDLER FOR SQLCODE VALUE -210
   SET MSG=M'Attempt to update a NOT NULL-constrained column
with NULL
               Attempt ignored';
  SET MSG ='';
  IF UQUANTITY<0 THEN
    SIGNAL illegal_value;
  ELSE
   UPDATE STOCK SET SQUANTITY=UQUANTITY WHERE SPCODE=UPCODE;
    SET MSG=MSG||M'Processing complete'
    SELECT SQUANTITY INTO PQUANTITY FROM STOCK WHERE
SPCODE=UPCODE;
    SET MSG=MSG||M'Current quantity:'||NUMEDIT(PQUANTITY,
'<999999');
  END IF;
END
```

3. Defines an SQL procedure (PROC3) that registers new product data in the inventory table (STOCK).

   It is assumed that the product code column (PCODE) is the primary key. If the product code for the data to be inserted is a duplicate of the product code for previously registered data (SQLCODE = -803), the transaction is rolled back using the exception processing, and a message is set in the output parameter. Execution of the SQL procedure statement terminates.

```
CREATE PROCEDURE PROC3(IN  UPCODE CHAR(4), IN  UPNAME
```

```
NCHAR(8),
                        IN  UCOL   NCHAR(1), IN  UPRICE INT,
                        OUT MSG    MVARCHAR(255))
BEGIN
  DECLARE EXIT HANDLER FOR SQLCODE VALUE -803
    BEGIN
      ROLLBACK;
      SET MSG=M'Rollback performed due to duplicate key
violation';
    END;
  INSERT INTO STOCK VALUES(UPCODE, UPNAME, UCOL, UPRICE, 0);
  SET MSG=M'Registration complete.';
END
```

IF Statement (Execute by conditional branching)

Function

The IF statement executes SQL statements that meet a given set of conditions.

Format

```
IF search-condition THEN SQL-procedure-statement;
      [SQL-procedure-statement;]...
   [ELSEIF search-condition THEN SQL-procedure-statement;
      [SQL-procedure-statement;]...]
   [ELSE SQL-procedure-statement;[SQL-procedure-statement;]...]
END IF
```

Operands

- IF *search-condition* THEN *SQL-procedure-statement*;[*SQL-procedure-statement*;] . . .

search-condition

Specifies the conditions under which the SQL procedure statement specified in the THEN clause is executed.

SQL-procedure-statement

Specifies the SQL statement that is to be executed if the condition specified in the IF clause is met.

- [ELSEIF *search-condition* THEN *SQL-procedure-statement*;[*SQL-procedure-statement*;] . . .]

search-condition

Specifies the conditions under which the SQL procedure statement specified in the THEN clause is executed.

SQL-procedure-statement

Specifies the SQL statement that is to be executed if the condition specified in the IF clause is not met but the condition specified in the ELSEIF clause is met.

- [ELSE *SQL-procedure-statement*; [*SQL-procedure-statement*;] . . .]

SQL-procedure-statement

Specifies the SQL statement that is to be executed if the conditions specified in the IF and ELSEIF clauses are not met.

- END IF

Specifies the end of the `IF` statement.

Common rules

1. SQL procedure statements are executed in the order in which they are specified. If an error occurs during the execution of an SQL procedure statement, any subsequent SQL procedure statements will not be executed.
2. A subquery cannot be specified in a search condition.

Note

`IF` statements can be specified in an SQL routine.

LEAVE statement (Exit statement)

Function

The LEAVE statement exits from a compound statement, the WHILE statement, or the FOR statement, and terminates the execution of those statements.

Format

```
LEAVE [statement-label]
```

Operand

- *statement-label*

Specifies the statement label for the compound statement, WHILE statement, or FOR statement from which control exits, and the execution of which is to be terminated.

If the statement label is omitted, the execution of the innermost compound statement surrounding the LEAVE statement from which the statement label was omitted, the WHILE statement, or the FOR statement is terminated prematurely.

Common rules

1. For a statement label, specify the starting label of the statement (compound statement, WHILE statement, or FOR statement) that includes the LEAVE statement.
2. The LEAVE statement that causes control to leave the handler action cannot be specified in the handler action.

Note

1. LEAVE statements can be specified in an SQL routine.

RETURN statement (Return function return value)

Function

The RETURN statement returns a return value from a function.

Format

RETURN *return-value*
return-value::={*value-expression*|NULL}

Operands

- *return-value*::={*value-expression* | NULL}

Specifies a value expression or NULL as the return value.

Common rules

1. A return value is converted to the data type that was specified in the RETURNS clause when the function was defined.
2. The RETURN statement cannot be specified in a procedure.
3. A subquery cannot be specified in a value expression that is specified in *return-value*.

Note

RETURN statements can be specified in an SQL function.

WHILE statement (Repeat statements)

Function

The WHILE statement executes repeatedly a set of SQL statements.

Format

```
 [starting-label:]
WHILE search-condition DO
  SQL-procedure-statement;[SQL-procedure-statement;]...
END [WHILE] [termination-label]
```

Operands

- [*starting-label*]

Specifies the starting label for the WHILE statement.

- *search-condition*

Specifies the condition under which the SQL procedure statements are to be executed repeatedly.

- *SQL-procedure-statement*

Specifies the SQL procedure statements that are to be executed repeatedly.

- END [WHILE] [*termination-label*]

Specifies the end of the WHILE statement. Specify a statement label as a termination label.

The WHILE operand has the same effect whether or it is specified.

Common rules

1. When a termination label is being specified, care must be taken that it is a statement label that has the same name as the starting label.
2. The scope of a statement label is from the beginning to the end of the WHILE statement. A statement label identical to the statement label of other statements contained in the WHILE statement or identical to a loop variable name cannot be specified. However, if there is a handler declaration in the SQL procedure statement, the scope of the statement label excludes the handler declaration. The following shows examples in which statement labels of the same name can and cannot be specified:

```
AAA: WHILE X < 100 DO
  BEGIN  ..........................................1
    DECLARE CN1 CONDITION FOR SQLCODE VALUE -800;
    DECLARE EXIT HANDLER FOR CN1
```

```
    AAA: BEGIN  ......................................2
        :
    END AAA;
    AAA: BEGIN  ......................................3
      DECLARE CN2 CONDITION FOR SQLCODE VALUE -800;
      DECLARE EXIT HANDLER FOR CN2
        :
    END AAA;
    SET X=X+1;
  END
END WHILE AAA
```

Explanation

Although the statement label in line 2 is identical to the label in line 1, it can be specified because it is in a handler declaration.

The statement label in line 3 is identical to that in line 1; it cannot be specified because it is not in a handler declaration.

3. HiRDB evaluates the search condition, executes the SQL statements if the result is `TRUE`, and executes the SQL statements repeatedly until the search condition becomes `FALSE` or indefinite.
4. SQL procedure statements are executed in the order in which they are specified. If an error occurs during the execution of an SQL procedure statement, any subsequent SQL procedure statements will not be executed, and the execution of the `WHILE` statement is also terminated.
5. A subquery cannot be specified in a search condition.

Note

`WHILE` statements can be specified in an SQL routine.

FOR statement (Repeat a statement on rows)

Function

Repeats the execution of a given SQL statement with respect to rows in a table.

Format

```
[starting-label:]
FOR loop-variable-name AS
 [cursor-name CURSOR [WITH HOLD]FOR]
   cursor-specification-Format-1
 <lock-option>
  [[{WITH {SHARE|EXCLUSIVE}LOCK
    | WITHOUT LOCK [{WAIT|NOWAIT}]}]
  [{WITH ROLLBACK|NO WAIT}]]
    [FOR {UPDATE [OF column-name [, column-name]...] [NOWAIT]|READ ONLY }]
    [UNTIL DISCONNECT]
 DO
 SQL procedure-statement;[SQL procedure-statement;]...
END FOR [end-label]
```

Operands

- [*starting-label*]

  Specifies the statement label for the FOR statement.

- *loop-variable-name*

  Specifies the qualifier for the SQL variable that is implicitly declared.

  During the execution of the FOR statement, the SQL variable that takes a derived column name with a cursor specification is implicitly declared. The implicitly declared SQL variable can be qualified with a loop variable name.

  Loop variable names are subject to the rules on label names. Consequently, restrictions on label names are also applicable to loop variable names.

- *cursor-name*

  Specifies the name of a cursor.

  If *cursor-name* is omitted, HiRDB generates a specific cursor name. For details about cursor names, see *1.1.7 Specification of names*.

- [WITH HOLD]

  Specify this option when using a holdable cursor.

Because WITH HOLD is functionally the same as specifying UNTIL DISCONNECT, for an explanation of this statement, see *UNTIL DISCONNECT*. Also, there is no functional difference due to whether or not this specification is duplicated with UNTIL DISCONNECT.

For restrictions on the holdable cursor, see *DECLARE CURSOR Format 1(Declare cursor)*.

- *cursor-specification* Format 1

  Specifies the cursor that represents the contents of a query.

  For cursor specification, see *2.1.1 Cursor specification: Format 1*.

Rules on cursor-specification that is specified in a FOR statement

- An unnamed column cannot be specified in a derived column. When specifying an unnamed column as a derived column, specify the AS clause, and assign an alias to the derived column.
- A derived column with a duplicate derived column name cannot be specified.
- [*table-specification*.] ROW cannot be specified in a derived column.
- In a derived column, an unsubscripted repetition column without a flattening specification cannot be specified in the FROM clause.

If a derived column in the cursor specification format has been derived from any of the items listed below and AS *column-name* is omitted, that column becomes a nameless column. If the derived column is a scalar subquery, the derived column name depends on the derived column name of the selection expression of the scalar subquery.

- Scalar operation (including the window function)
- Function call
- Set function
- Literal
- USER
- CURRENT_DATE value function
- CURRENT_TIME value function
- CURRENT_TIMESTAMP value function
- Component specification
- GET_JAVA_STORED_ROUTINE_SOURCE specification
- WRITE specification

- SQL variable
- SQL parameter

- *lock-option*

  Specifies the lock mode when a query is made, and the action to be taken when another user has exclusive use of resources.

  For details about the lock option, see *2.19 Lock option*.

- `[FOR{UPDATE [OF` *column-name* `[, ` *column-name* `]...] [NOWAIT]|READ ONLY}]`

  `FOR UPDATE [OF` *column-name* `[, ` *column-name* `]...]` is referred to as the `FOR UPDATE` clause.

  `FOR UPDATE [OF` *column-name* `[, ` *column-name* `]...]` is referred to as the `FOR UPDATE` clause.

  For details about the `FOR UPDATE` clause and `FOR READ ONLY`, see *DECLARE CURSOR Format 1 (Declare cursor)* in Chapter *4*.

- `[UNTIL DISCONNECT]`

  Specify this option when using a holdable cursor.

  This specification is functionally identical to the `WITH HOLD` specification. Also, there is no functional difference due to whether or not this specification is duplicated with `WITH HOLD`.

  For details about restrictions on the holdable cursor, see *DECLARE CURSOR Format 1(Declare cursor)*.

- `SQL` *procedure-statement*

  Specifies the SQL procedure statement that is executed repeatedly.

Rules on the SQL procedure statement specified in the FOR statement

- An `OPEN`, `FETCH`, or `CLOSE` statement specifying the cursor in the `FOR` statement cannot be specified.
- A `LEAVE` statement with a loop variable name specification cannot be specified.
- If the cursor in the `FOR` statement is not a holdable cursor, the `COMMIT`, `ROLLBACK`, or `PURGE TABLE` statement cannot be specified. If a procedure specifying any of these statements is called by a `CALL` statement, a run-time error may occur.
- The specified SQL procedure statements are executed in the order in which they are specified. If an error occurs during the execution of an SQL procedure statement, any subsequent SQL procedure statement is not

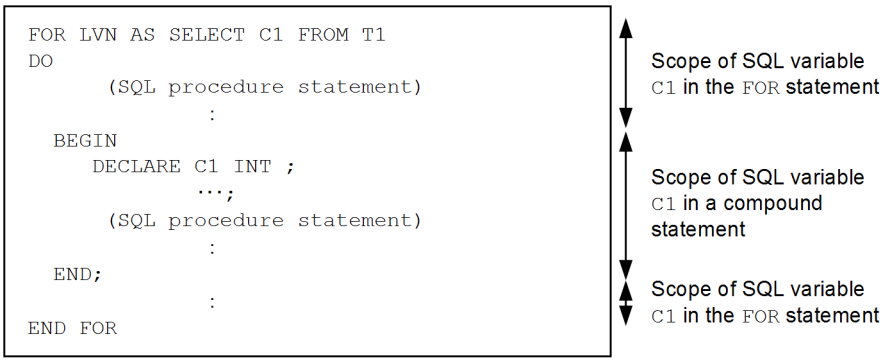
executed. The execution of the FOR statement also terminates.

- END FOR [*end-label*]

  Specifies the termination of the FOR statement. In *end-label*, a statement label is specified.

Common rules

1. When specifying *end-label*, specify a statement label with the same name as the starting label.
2. The scope of a statement label and that of a loop variable name is between the beginning and the end of the FOR statement. A statement label or a loop variable name that is identical to the statement label or loop variable name of other statements contained in the FOR statement cannot be specified. However, if the SQL procedure statement contains a handler declaration, such identical statement labels or loop variable names can still be specified in the handler declaration.
3. The SQL variable that is implicitly declared in the FOR statement or the cursor that is explicitly declared is allocated first in the FOR statement and is released last.
4. The scope of the SQL variable implicitly declared in the FOR statement, or the cursor that is explicitly declared, is within the FOR statement in which the SQL statement or the cursor is declared.
5. If a compound statement is declared in an SQL procedure statement in a FOR statement, or if a FOR statement is declared in an SQL procedure statement in a compound statement, and if the SQL variable implicitly declared in the FOR statement and the SQL variable declared in the compound statement have the same name, in the inner routine control SQL, the SQL variable declared in the inner routine control SQL statement takes effect. When the inner routine control SQL statement terminates, the SQL variable declared in the outer routine control SQL statement takes effect. An example is shown below:



6. If a FOR statement is specified in an SQL procedure statement in a FOR statement, and if the SQL variable implicitly declared in the outer FOR statement and the SQL variable implicitly declared in the inner FOR statement have the same name, in the inner FOR statement, the SQL variable declared in the inner FOR statement takes effect. When the inner FOR statement terminates, the SQL variable declared in the outer FOR statement takes effect.
7. If a compound statement is specified in an SQL procedure statement in a FOR statement, if a FOR statement is specified in an SQL statement in the compound statement, and if the cursor declared in the FOR statement and the cursor declared in the compound statement have the same name, the cursor declared in the inner routine control SQL statement takes effect. When the inner routine control SQL statement terminates, the cursor declared outside takes effect.
8. If a FOR statement is specified in an SQL procedure statement in a FOR statement, and if the cursor declared in the outer FOR statement and the cursor declared in the inner FOR statement have the same name, the cursor declared in the inner FOR statement takes effect. When the inner FOR statement terminates, the cursor declared in the outer FOR statement takes effect.
9. If an error occurs during the execution of an SQL procedure, the transaction is nullified only if the error is subject to an implicit rollback. If an error occurs during the execution of a trigger operation SQL procedure statement, the transaction is always implicitly rolled back.
10. The maximum allowable number of nesting levels for FOR statements and compound statements is 255.

Notes

1. A FOR statement can be specified in an SQL procedure or a trigger.

Examples

Of the data listed in table T1, assign the data with a column C1 (INT type) less than or equal to 100 to Table T2, and for other data, define a procedure (PROC1) that assigns the data to Table T3:

```
CREATE PROCEDURE PROC1 ()
FLBL :
FOR LVN AS SELECT C1, C2, C3 FROM T1 DO
 IF C1 <= 100 THEN
  INSERT INTO T2 VALUES(LVN.C1, LVN.C2, LVN.C3) ;
 ELSE
  INSERT INTO T3 VALUES(LVN.C1, LVN.C2, LVN.C3) ;
 END IF ;
END FOR FLBL
```

SQL statements equivalent to the FOR statements shown in the above example can be implemented in the following SQL statements:

```
CREATE PROCEDURE PROC1 ()
LVN :
BEGIN
  DECLARE C1, C2, C3 INT ;
  DECLARE FCN CURSOR FOR SELECT C1, C2, C3 FROM T1 ;
  DECLARE AT_END CHAR(1) DEFAULT 'N' ;
  OPEN FCN ;
  FLBL :
  WHILE AT_END != 'Y' DO
    FETCH FCN INTO C1, C2, C3 ;
    IF SQLCODE = 100 THEN
      SET AT_END = 'Y' ;
    ELSE
      IF C1 <= 100 THEN
        INSERT INTO T2 VALUES(LVN.C1, LVN.C2, LVN.C3) ;
      ELSE
        INSERT INTO T3 VALUES(LVN.C1, LVN.C2, LVN.C3) ;
      END IF ;
    END IF ;
  END WHILE FLBL ;
 CLOSE FCN ;
END LVN
```

WRITE LINE statement (Character string output to a file)

Function

Outputs the character string in a specified value expression to a file.

Format

```
WRITE LINE value-expression
```

Operands

- *value-expression*

Specifies the value expression to be output to the file.

The following rules apply to the value expression:

1. The following items can be specified in *value-expression*:
   - Literals
   - USER
   - SQL variables or SQL parameters
   - Concatenation operations
   - Scalar functions
   - Function calls
   - CASE expressions
   - CAST specifications
   - SQLERRM_OF_LAST_CONDITION
2. The data type of the value expression must be a character data type (CHAR or VARCHAR), a national character data type (NCHAR or NVARCHAR), or a mixed character data type (MCHAR or MVARCHAR).
3. If the data type of the result of the value expression is character data, the character set used to output the character data to a file is the character set specified in the value expression.
4. The value expression is output to the location specified by PDWRTLNPATH in the client environment definition. Use PDWRTLNFILSZ in the client environment definition to specify the maximum size of the file to which the value expression is to be output. For details about PDWRTLNPATH and PDWRTLNFILSZ, see the

manual *HiRDB Version 9 UAP Development Guide*.

5. The line-terminating code of the environment on the HiRDB server side is appended to the end of the character string in the value expression. The line-terminating code varies from one HiRDB server operating system to another. If the value of the value expression exceeds the size specified in the PDWRTLNCOMSZ client environment definition, for the line-terminating code on the HiRDB server side to be appended at the end, the trailing end of the value expression character expression is deleted so that the character string fits in the size specified in PDWRTLNCOMSZ, and the line-terminating code is appended. For details about PDWRTLNCOMSZ, see the *HiRDB Version 9 UAP Development Guide*.

   The following table lists the line-terminating codes that are appended on the HiRDB server side.

   Table 7-4: Line-terminating codes that are appended on the HiRDB server side

HiRDB server OS	Appended line-terminating code
HP-UX	NL(X'0a')
Solaris	NL(X'0a')
AIX	NL(X'0a')
Linux	NL(X'0a')
Windows	CR(X'0d')＋NL(X'0a')

6. If the value of the value expression is the null value, the line-terminating code is not output to the file.
7. A subquery cannot be specified in a value expression.

Common rules

1. The WRITE LINE statement can be specified in SQL procedures and triggers; it cannot be specified in an SQL function.
2. The WRITE LINE statement cannot be executed without specification of the PDWRTLNFILSZ client environment definition.

Example

Convert the SQL parameter fromdate of the date data type into a character string and output the results to a file:

```
CREATE PROCEDURE proc_1(IN fromdate DATE)
  BEGIN
    WRITE LINE 'fromdate='||char(fromdate);
  END;
```

SIGNAL statement (Signal error)

Function

Generates an error and signals it; clears any information that has been assigned to the diagnostic area up to that point.

Format

```
SIGNAL signal-value

signal-value::={SQLSTATE-value|condition-name}
SQLSTATE-value::=SQLSTATE [VALUE] character-string-literal
```

Operands

- *signal-value*::={*SQLSTATE-value*|*condition-name*}

Specifies the value to be returned to the UAP.

- *SQLSTATE-value*::=SQLSTATE [VALUE] *character-string-literal*

Specifies a value (combination of upper case characters A to Z, and numeric characters 0 to 9) that is valid as an SQLSTATE value, in 5 characters. Specify a value according to the following rules for SQLSTATE in HiRDB:

- The values 00, 01, 02, and R2 cannot be specified as an SQLSTATE class (the first two characters). These values are not error-indicating classes.
- SQLSTATE classes beginning with the characters 0 to 5, A to I, or R cannot be specified. These values are reserved by HiRDB.

If the SQLSTATE class fails to comply with these rules, a definition-time error may occur. For SQLSTATE values, see *SQLSTATE variable*.

- *condition-name*

Specifies the condition name that was declared in the condition declaration.

If *condition-name* is specified, the code R0000 indicating an error (abnormal termination without an implicit rollback) is set in the SQLSTATE.

If an SQLCODE value associated with *condition-name* is defined, an error may occur.

Common rules

1. Execution of the SIGNAL statement causes the value -1400 to be assigned to the SQLCODE.

2. Execution of the RESIGNAL statement does not cause an implicit rollback. However, if the statement is executed in the trigger, an implicit rollback ensues, by excluding the tables that are defined by specifying WITHOUT ROLLBACK. Tables that are defined by specifying WITHOUT ROLLBACK are not rolled back after completion of row updating (including additions and deletions) even when the RESIGNAL statement is executed in the trigger.
3. Execution of the SIGNAL statement clears any condition information items that were assigned to the diagnostic area before execution of the SIGNAL statement.

   In the statement information item NUMBER of the diagnostic area, the value *i* + 1, and the value 'N' is assigned to MORE.

   In the first (condition number 1) ERROR_SQL condition information item, the value 'REGIONAL' is assigned. If a condition name is specified in *signal-value*, a condition name is assigned to CONDITION_IDENTIFIER. If *SQLSTATE-value* is specified, 'SQLSTATE:*xxxxx*' (where *xxxxx* is the specified *SQLSTATE-value*) is assigned to CONDITION_IDENTIFIER.

Notes

1. The SIGNAL statement can be specified in SQL procedures and triggers.
2. Execution of the SIGNAL statement clears any diagnostic information that was set in the diagnostic area before execution. The RESIGNAL statement can be used to prevent the clearing of older, history diagnostic information.

Example

1. Define an SQL procedure (STOCK_UPDATE1) that updates the quantity of a specified product code in the inventory table (STOCK). If the quantity specified in the input parameter is a negative number, use the SIGNAL statement to generate an error, and assign an Invalid value as a quantity to the output parameter.

```
CREATE PROCEDURE STOCK_UPDATE1(IN UPCODE INT, IN UQUANTITY
INT,
                                OUT MSG NVARCHAR(50))
BEGIN
  DECLARE illegal_value CONDITION ;
  DECLARE EXIT HANDLER FOR illegal_value
    SET MSG=N'Invalid value as a quantity';
  DECLARE EXIT HANDLER FOR NOT FOUND
    SET MSG=N'Specified product code not found.';
  IF UQUANTITY<0 THEN
    SIGNAL illegal_value;
  ELSE
   UPDATE STOCK SET SQUANTITY=UQUANTITY WHERE SPCODE=UPCODE;
    SET MSG=N'Updating completed'.;
  END IF;
END
```

2. Before deleting rows in the inventory table (`STOCK`), use the `SIGNAL` statement to generate an error, and define a trigger (`SIGNALTRIG`) that suppresses the deletion of rows from the inventory table.:

```
CREATE TRIGGER SIGNALTRIG
  BEFORE DELETE ON STOCK
  SIGNAL SQLSTATE '99001'
```

RESIGNAL statement (Resignal error)

Function

Generates and signals an error, and adds diagnostic information to the diagnostic area.

Format

```
RESIGNAL [signal-value]

signal-value::={SQLSTATE-value|condition-name}
SQLSTATE-value::=SQLSTATE [VALUE] character-string-literal
```

Operands

- *signal-value*::={*SQLSTATE-value*|*condition-name*}

Specifies the value to be returned to the UAP.

- *SQLSTATE-value*::=SQLSTATE [VALUE] *character-string-literal*

Specifies a value (combination of upper case characters A to Z, and numeric characters 0 to 9) that is valid as an SQLSTATE value, in 5 characters. Specify a value according to the following rules for SQLSTATE in HiRDB.

- The values 00, 01, 02, and R2 cannot be specified as an SQLSTATE class (the first two characters). These values are not error-indicating classes.
- SQLSTATE classes beginning with the character 0 to 5, A to I, or R cannot be specified. These values are reserved by HiRDB.

If the SQLSTATE class fails to comply with these rules, a definition-time error may occur. For SQLSTATE values, see *SQLSTATE variable*.

- *condition-name*

Specifies the condition name that was declared in the condition declaration.

If *condition-name* is specified, the code 'R0000' indicating an error (abnormal termination without an implicit rollback) is set in the SQLSTATE.

If an SQLCODE value associated with *condition-name* is defined, an error may occur.

Common rules

1. If the SQL procedure statement that called the handler is not found before the RESIGNAL statement is executed, a runtime error may occur.
2. If *signal-value* is not specified, the code 'R0000' indicating an error (abnormal

termination without an implicit rollback) is set in the SQLSTATE.

3. Execution of the RESIGNAL statement causes the assignment of the following values in the SQLCODE:
   - With *signal-value* not specified:

     The SQLCODE is not changed. The value that was set when the SQL procedure statement called the handler is retained.
   - With *signal-value* specified:

     The value -1400 is assigned.
4. Execution of the RESIGNAL statement does not cause an implicit rollback. However, if the statement is executed in the trigger, an implicit rollback ensues, by excluding the tables that are defined by specifying WITHOUT ROLLBACK. Tables that are defined by specifying WITHOUT ROLLBACK are not rolled back after completion of row updating (including additions and deletions) even when the RESIGNAL statement is executed in the trigger.
5. Execution of the RESIGNAL statement causes the following types of information to be assigned to the diagnostic area:
   - With *signal-value* not specified:

     Diagnostic information is not updated.
   - With *signal-value* specified:

     In the statement information item NUMBER of the diagnostic area, the value *i* + 1 (the value before execution of the RESIGNAL statement is *i*), and the value N is assigned to MORE. The information that was assigned to the *i*-the condition information item in the diagnostic area (condition number *i*), is re-assigned to the *i* + 1st item (condition number *i* + 1). If the maximum number of condition information items is exceeded, the value Y is set in the statement information item MORE. In the first (condition number 1) ERROR_SQL condition information item, the value 'REGIONAL' is assigned. If a condition name is specified in *signal-value*, a condition name is assigned to CONDITION_IDENTIFIER. If *SQLSTATE-value* is specified, 'SQLSTATE:*xxxxx*' (where *xxxxx* is the specified *SQLSTATE-value*) is assigned to CONDITION_IDENTIFIER.

Note

1. The RESIGNAL statement can be specified in SQL procedures and triggers.

Example

For the inventory table (STOCK), specify an SQL procedure (STOCK_UPDATE2) that updates the quantity of a specified product code. If the specified quantity is less than

0, a matching product is not found, or the update process fails, the RESIGNAL statement generates an error, and respective SQLSTATE values are set.

```
  CREATE PROCEDURE STOCK_UPDATE2(IN UPCODE INT, IN UQUANTITY
INT)
  BEGIN
    DECLARE illegal_value CONDITION ;
    DECLARE EXIT HANDLER FOR illegal_value
      RESIGNAL SQLSTATE VALUE '66001';
    DECLARE EXIT HANDLER FOR NOT FOUND
      SIGNAL SQLSTATE VALUE '66002';
    DECLARE EXIT HANDLER FOR SQLERROR
      RESIGNAL SQLSTATE VALUE '66003';
    IF UQUANTITY<0 THEN
      SIGNAL illegal_value;
    ELSE
      UPDATE STOCK SET SQUANTITY=UQUANTITY WHERE SPCODE=UPCODE;
    END IF;
  END
```

Appendixes

A. Reserved Words
B. List of SQLs
C. Example Database

A. Reserved Words

This appendix contains the following sections:

- A.1 SQL reserved words
- A.2 HiRDB reserved words

A.1 SQL reserved words

SQL includes reserved words defined in ISO as *ISO 9075-1992 Database Language SQL* (*SQL 92*) and reserved words defined in JIS as *JIS X 3005-1990 Database Language SQL*. The reserved words used in HiRDB are based on the JIS standards.

Reserved words are stored as keywords that are used in SQL statements. Therefore, reserved words cannot be used as table or column names.

A reserved word that appears in an SQL statement must be enclosed in double quotation marks (`"`). When enclosed in double quotation marks, a reserved word can be used as any character string.

Tables *A-1* to *A-25* show SQL reserved words.

For the tables, the following legends apply:

Y: Reserved word

--: Not a reserved word

SQL92: ISO SQL 1992

SQL99: ISO SQL 1999

UNIFY: UNIFY2000

XDM/RD: XDM/RD E2

HiRDB (V6): HiRDB Version 6

HiRDB (V7): HiRDB Version 7

HiRDB (V8): HiRDB Version 8 or later

Table A-1: SQL reserved words (A)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
ABS	--	--	--	Y	Y	Y	Y
ABSOLUTE	Y	Y	--	--	Y	Y	Y
ACCESS	--	--	Y	--	Y	Y	Y

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
ACTION	Y	Y	--	Y	Y	Y	Y
ADD	Y	Y	Y	Y	Y	Y	Y
ADMIN	--	Y	--	--	--	--	--
AFTER	--	Y	--	--	Y	Y	Y
AGGREGATE	--	Y	--	--	--	--	--
ALIAS	--	Y	--	--	Y	Y	Y
ALL	Y	Y	Y	Y	Y	Y	Y
ALLOCATE	Y	Y	Y	Y	Y	Y	Y
ALTER	Y	Y	Y	Y	Y	Y	Y
AMOUNT	--	--	Y	--	Y	Y	Y
AND	Y	Y	Y	Y	Y	Y	Y
ANDNOT	--	--	--	Y	Y	Y	Y
ANSI	--	--	Y	--	Y	Y	Y
ANY	Y	Y	Y	Y	Y	Y	Y
ARE	Y	Y	--	--	Y	Y	Y
ARRAY	--	Y	--	Y	Y	Y	Y
AS	Y	Y	Y	Y	Y	Y	Y
ASC	Y	Y	Y	Y	Y	Y	Y
ASSERTION	Y	Y	--	--	Y	Y	Y
ASSIGN	--	--	--	Y	Y	Y	Y
ASYNC	--	--	--	--	Y	Y	Y
AT	Y	Y	Y	--	Y	Y	Y
AUTHORIZATION	Y	Y	Y	Y	Y	Y	Y
AUTO	--	--	Y	--	Y	Y	Y
AVG	Y	--	Y	Y	Y	Y	Y

Table A-2: SQL reserved words (B)

Reserved word	SQL92	SQL99	UNIFY	XDM/RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
BASE	--	--	Y	--	Y	Y	Y
BEFORE	--	Y	--	--	Y	Y	Y
BEGIN	Y	Y	Y	Y	Y	Y	Y
BETWEEN	Y	--	Y	Y	Y	Y	Y
BINARY	--	Y	Y	Y	Y	Y	Y
BIT	Y	Y	--	--	Y	Y	Y
BIT_AND_TEST	--	--	--	--	Y	Y	Y
BIT_LENGTH	Y	--	--	--	Y	Y	Y
BLOB	--	Y	--	Y	Y	Y	Y
BOOLEAN	--	Y	--	Y	Y	Y	Y
BOTH	Y	Y	--	Y	Y	Y	Y
BREADTH	--	Y	--	--	Y	Y	Y
BTREE	--	--	Y	--	Y	Y	Y
BUFFER	--	--	Y	--	Y	Y	Y
BY	Y	Y	Y	Y	Y	Y	Y
BYTE	--	--	Y	--	Y	Y	Y

Table A-3: SQL reserved words (C)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
CALL	--	Y	--	Y	Y	Y	Y
CASCADE	Y	Y	--	--	Y	Y	Y
CASCADED	Y	Y	--	--	--	--	--
CASE	Y	Y	--	Y	Y	Y	Y
CAST	Y	Y	--	Y	Y	Y	Y
CATALOG	Y	Y	--	--	Y	Y	Y

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
CHANGE	--	--	--	Y	Y	Y	Y
CHAR	Y	Y	Y	Y	Y	Y	Y
CHARACTER	Y	Y	Y	Y	Y	Y	Y
CHAR_LENGTH	Y	--	--	--	Y	Y	Y
CHARACTER_LENGTH	Y	--	--	--	Y	Y	Y
CHECK	Y	Y	Y	Y	Y	Y	Y
CLASS	--	Y	--	--	--	--	--
CLOB	--	Y	--	Y	--	--	--
CLOSE	Y	Y	Y	Y	Y	Y	Y
CLUSTER	--	--	--	Y	Y	Y	Y
COALESCE	Y	--	--	Y	Y	Y	Y
COLLATE	Y	Y	--	--	Y	Y	Y
COLLATION	Y	Y	--	--	Y	Y	Y
COLUMN	Y	Y	Y	Y	Y	Y	Y
COLUMNS	--	--	Y	--	Y	Y	Y
COMMENT	--	--	--	Y	Y	Y	Y
COMMIT	Y	Y	Y	Y	Y	Y	Y
COMPLETION	--	Y	--	--	Y	Y	Y
COMPRESSED	--	--	--	Y	--	--	--
CONDITION	--	Y	--	--	--	Y	Y
CONFIGURATION	--	--	Y	--	Y	Y	Y
CONNECT	Y	Y	Y	Y	Y	Y	Y
CONNECTION	Y	Y	--	--	Y	Y	Y
CONST	--	--	Y	--	Y	Y	Y
CONSTRAINT	Y	Y	--	Y	Y	Y	Y
CONSTRAINTS	Y	Y	--	--	Y	Y	Y

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
CONSTRUCTOR	--	Y	--	--	Y	Y	Y
CONTIGUOUS	--	--	Y	--	Y	Y	Y
CONTINUE	Y	Y	Y	--	Y	Y	Y
CONVERT	Y	--	--	--	Y	Y	Y
CORR	--	--	--	Y	--	--	--
CORRESPONDING	Y	Y	--	--	Y	Y	Y
COUNT	Y	--	Y	Y	Y	Y	Y
COUNT_FLOAT	--	--	--	--	--	Y	Y
COVAR_POP	--	--	--	Y	--	--	--
COVAR_SAMP	--	--	--	Y	--	--	--
CREATE	Y	Y	Y	Y	Y	Y	Y
CROSS	Y	Y	--	Y	Y	Y	Y
CUBE	--	Y	--	Y	--	--	--
CUME_DIST	--	--	--	Y	--	--	--
CURAID	--	--	Y	--	Y	Y	Y
CURRENT	Y	Y	Y	Y	Y	Y	Y
CURRENT_DATE	Y	Y	--	Y	Y	Y	Y
CURRENT_DEFAULT_TRANSFORM_GROUP	--	Y	--	--	--	--	--
CURRENT_PATH	--	Y	--	--	--	--	--
CURRENT_ROLL	--	Y	--	--	--	--	--
CURRENT_TIME	Y	Y	--	Y	Y	Y	Y
CURRENT_TIMESTAMP	Y	Y	--	Y	Y	Y	Y
CURRENT_TRANSFORM_GROUP_FOR_TYPE	--	Y	--	--	--	--	--
CURRENT_USER	Y	Y	--	Y	Y	Y	Y
CURSOR	Y	Y	Y	Y	Y	Y	Y

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
CYCLE	--	Y	--	--	Y	Y	Y

Table A-4: SQL reserved words (D)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
DATA	--	Y	Y	Y	Y	Y	Y
DATABASE	--	--	Y	--	Y	Y	Y
DATE	Y	Y	Y	Y	Y	Y	Y
DAY	Y	Y	--	Y	Y	Y	Y
DAYS	--	--	--	Y	Y	Y	Y
DBA	--	--	Y	Y	Y	Y	Y
DEALLOCATE	Y	Y	Y	--	Y	Y	Y
DEC	Y	Y	Y	Y	Y	Y	Y
DECIMAL	Y	Y	Y	Y	Y	Y	Y
DECLARE	Y	Y	Y	Y	Y	Y	Y
DEFAULT	Y	Y	Y	Y	Y	Y	Y
DEFER	--	--	Y	--	Y	Y	Y
DEFERRABLE	Y	Y	--	--	Y	Y	Y
DEFERRED	Y	Y	Y	--	Y	Y	Y
DELETE	Y	Y	Y	Y	Y	Y	Y
DEMOTING	--	--	Y	--	Y	Y	Y
DENSE_RANK	--	--	--	Y	--	--	--
DEPTH	--	Y	--	--	Y	Y	Y
DEREF	--	Y	--	--	--	--	--
DESC	Y	Y	Y	Y	Y	Y	Y
DESCRIBE	Y	Y	Y	Y	Y	Y	Y
DESCRIPTION	--	--	Y	--	--	--	--

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
DESCRIPTOR	Y	Y	Y	Y	Y	Y	Y
DESTROY	--	Y	--	--	--	--	--
DESTRUCTOR	--	Y	--	--	--	--	--
DETERMINISTIC	--	Y	--	--	--	--	--
DEVICE	--	--	Y	--	Y	Y	Y
DIAGNOSTICS	Y	Y	--	Y	Y	Y	Y
DICTIONARY	--	Y	--	--	Y	Y	Y
DIGITS	--	--	--	Y	Y	Y	Y
DIRECT	--	--	Y	--	Y	Y	Y
DISCONNECT	Y	Y	Y	Y	Y	Y	Y
DISPLAY	--	--	Y	--	--	--	--
DISTINCT	Y	Y	Y	Y	Y	Y	Y
DO	--	Y	--	Y	Y	Y	Y
DOMAIN	Y	Y	--	--	--	--	--
DOUBLE	Y	Y	Y	Y	Y	Y	Y
DOUBLE_PRECISION	--	--	Y	--	Y	Y	Y
DROP	Y	Y	Y	Y	Y	Y	Y
DYNAMIC	--	Y	--	--	--	--	--

Table A-5: SQL reserved words (E)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
EACH	--	Y	--	Y	Y	Y	Y
EDIT	--	--	Y	--	Y	Y	Y
ELSE	Y	Y	--	Y	Y	Y	Y
ELSEIF	--	Y	--	Y	Y	Y	Y
ENCRYPT	--	--	--	--	--	Y	Y

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
END	Y	Y	Y	Y	Y	Y	Y
END-EXEC	Y	Y	--	--	--	--	--
EQUALS	--	Y	--	--	Y	Y	Y
ESCAPE	Y	Y	Y	Y	Y	Y	Y
ESTIMATED	--	--	Y	--	Y	Y	Y
EVERY	--	Y	--	Y	--	--	--
EXCEPT	Y	Y	--	Y	Y	Y	Y
EXCEPTION	Y	Y	--	Y	Y	Y	Y
EXCLUSIVE	--	--	--	Y	Y	Y	Y
EXEC	Y	Y	Y	--	Y	Y	Y
EXECUTE	Y	Y	Y	Y	Y	Y	Y
EXISTS	Y	--	Y	Y	Y	Y	Y
EXIT	--	Y	--	--	--	Y	Y
EXTERN	--	--	Y	--	Y	Y	Y
EXTERNAL	Y	Y	--	--	Y	Y	Y
EXTRACT	Y	--	--	--	Y	Y	Y

Table A-6: SQL reserved words (F)

Reserved word	SQL92	SQL99	UNIFY	XDM/RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
FALSE	Y	Y	--	Y	Y	Y	Y
FETCH	Y	Y	Y	Y	Y	Y	Y
FILE	--	--	Y	--	Y	Y	Y
FILTER	--	--	--	Y	--	--	--
FIRST	Y	Y	--	--	Y	Y	Y
FIX	--	--	--	Y	Y	Y	Y
FIXED	--	--	Y	--	Y	Y	Y

Reserved word	SQL92	SQL99	UNIFY	XDM/RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
FLAT	--	--	--	Y	Y	Y	Y
FLOAT	Y	Y	Y	Y	Y	Y	Y
FOR	Y	Y	Y	Y	Y	Y	Y
FORCE	--	--	Y	Y	Y	Y	Y
FOREIGN	Y	Y	--	Y	Y	Y	Y
FOUND	Y	Y	Y	--	Y	Y	Y
FREE	--	Y	--	--	--	Y	Y
FROM	Y	Y	Y	Y	Y	Y	Y
FULL	Y	Y	--	Y	Y	Y	Y
FUNCTION	--	Y	--	Y	Y	Y	Y

Table A-7: SQL reserved words (G)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
GENERAL	--	Y	--	--	Y	Y	Y
GET	Y	Y	Y	Y	Y	Y	Y
GET_JAVA_STORED_ROUTINE_SOURCE	--	--	--	--	Y	Y	Y
GLOBAL	Y	Y	--	--	Y	Y	Y
GO	Y	Y	Y	--	Y	Y	Y
GOTO	Y	Y	Y	--	Y	Y	Y
GRANT	Y	Y	Y	Y	Y	Y	Y
GROUP	Y	Y	Y	Y	Y	Y	Y
GROUPING	--	Y	--	Y	--	--	--

Table A-8: SQL reserved words (H)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
HANDLER	--	Y	--	--	--	Y	Y

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
HASH	--	--	Y	--	Y	Y	Y
HAVING	Y	Y	Y	Y	Y	Y	Y
HELP	--	--	Y	--	Y	Y	Y
HEX	--	--	--	Y	Y	Y	Y
HOST	--	Y	--	--	--	--	--
HOUR	Y	Y	--	Y	Y	Y	Y
HOURS	--	--	--	Y	Y	Y	Y
HUGE	--	--	Y	--	Y	Y	Y

Table A-9: SQL reserved words (I)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
IDENTIFIED	--	--	--	Y	Y	Y	Y
IDENTITY	Y	Y	--	--	Y	Y	Y
IF	--	Y	--	Y	Y	Y	Y
IGNORE	--	Y	--	--	Y	Y	Y
IMMEDIATE	Y	Y	Y	Y	Y	Y	Y
IN	Y	Y	Y	Y	Y	Y	Y
INDEX	--	--	Y	Y	Y	Y	Y
INDICATOR	Y	Y	Y	Y	Y	Y	Y
INITIALIZE	--	Y	--	--	--	--	--
INITIALLY	Y	Y	--	--	Y	Y	Y
INNER	Y	Y	--	Y	Y	Y	Y
INOUT	--	Y	--	Y	Y	Y	Y
INPUT	Y	Y	Y	--	Y	Y	Y
INSENSITIVE	Y	--	--	--	Y	Y	Y
INSERT	Y	Y	Y	Y	Y	Y	Y

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
INT	Y	Y	Y	Y	Y	Y	Y
INTEGER	Y	Y	Y	Y	Y	Y	Y
INTERSECT	Y	Y	--	Y	Y	Y	Y
INTERVAL	Y	Y	--	Y	Y	Y	Y
INTO	Y	Y	Y	Y	Y	Y	Y
IS	Y	Y	Y	Y	Y	Y	Y
ISOLATION	Y	Y	--	Y	Y	Y	Y
IS_USER_CONTAINED_IN_HDS_GROUP	--	--	--	--	Y	Y	Y
ITERATE	--	Y	--	Y	--	--	--

Table A-10: SQL reserved words (J)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
JOIN	Y	Y	--	Y	Y	Y	Y

Table A-11: SQL reserved words (K)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
KEY	Y	Y	Y	Y	Y	Y	Y

Table A-12: SQL reserved words (L)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
LABEL	--	--	--	Y	--	--	--
LANGUAGE	Y	Y	Y	Y	Y	Y	Y
LARGE	--	Y	--	Y	Y	Y	Y
LAST	Y	Y	--	--	Y	Y	Y
LATERAL	--	Y	--	--	--	--	--
LEADING	Y	Y	Y	Y	Y	Y	Y

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
LEAVE	--	Y	--	Y	Y	Y	Y
LEFT	Y	Y	--	--	Y	Y	Y
LENGTH	--	--	Y	Y	Y	Y	Y
LESS	--	Y	--	--	Y	Y	Y
LEVEL	Y	Y	Y	Y	Y	Y	Y
LIKE	Y	Y	Y	Y	Y	Y	Y
LIMIT	--	Y	--	--	Y	Y	Y
LINES	--	--	Y	--	Y	Y	Y
LINK	--	--	Y	--	Y	Y	Y
LIST	--	--	--	Y	Y	Y	Y
LOCAL	Y	Y	--	--	Y	Y	Y
LOCALTIME	--	Y	--	--	--	--	--
LOCALTIMESTAMP	--	Y	--	--	--	--	--
LOCATOR	--	Y	--	--	--	Y	Y
LOCK	--	--	--	Y	Y	Y	Y
LOCKS	--	--	Y	--	Y	Y	Y
LOGID	--	--	Y	--	Y	Y	Y
LOGNAME	--	--	Y	--	Y	Y	Y
LONG	--	--	Y	Y	Y	Y	Y
LOOP	--	Y	--	Y	Y	Y	Y
LOWER	Y	--	--	Y	Y	Y	Y

Table A-13: SQL reserved words (M)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
MAP	--	Y	--	--	--	--	--
MATCH	Y	Y	--	--	Y	Y	Y

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
MAX	Y	--	Y	Y	Y	Y	Y
MAXUSAGES	--	--	--	--	Y	Y	Y
MCHAR	--	--	--	Y	Y	Y	Y
MICROSECOND	--	--	--	Y	--	--	--
MICROSECONDS	--	--	--	Y	--	--	--
MIN	Y	--	Y	Y	Y	Y	Y
MINUTE	Y	Y	--	Y	Y	Y	Y
MINUTES	--	--	--	Y	Y	Y	Y
MOD	--	--	--	Y	Y	Y	Y
MODE	--	--	Y	Y	Y	Y	Y
MODIFIES	--	Y	--	--	--	--	--
MODIFY	--	Y	--	--	Y	Y	Y
MODULE	Y	Y	Y	Y	Y	Y	Y
MONTH	Y	Y	--	Y	Y	Y	Y
MONTHS	--	--	--	Y	Y	Y	Y
MOVE	--	--	Y	--	Y	Y	Y
MVARCHAR	--	--	--	Y	Y	Y	Y

Table A-14: SQL reserved words (N)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
NAMES	Y	Y	--	--	Y	Y	Y
NATIONAL	Y	Y	--	Y	Y	Y	Y
NATURAL	Y	Y	--	--	Y	Y	Y
NCHAR	Y	Y	--	Y	Y	Y	Y
NCLOB	--	Y	--	--	--	--	--
NESTING	--	Y	--	--	--	--	--

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
NEW	--	Y	--	Y	Y	Y	Y
NEXT	Y	Y	--	--	Y	Y	Y
NO	Y	Y	--	Y	Y	Y	Y
NONE	--	Y	--	--	Y	Y	Y
NONLOCAL	--	--	--	Y	--	--	--
NOT	Y	Y	Y	Y	Y	Y	Y
NOWAIT	--	--	--	Y	Y	Y	Y
NULL	Y	Y	Y	Y	Y	Y	Y
NULLABLE	--	--	Y	--	Y	Y	Y
NULLIF	Y	--	--	Y	Y	Y	Y
NUMERIC	Y	Y	Y	Y	Y	Y	Y
NVARCHAR	--	--	--	Y	Y	Y	Y

Table A-15: SQL reserved words (O)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
OBJECT	--	Y	--	--	Y	Y	Y
OCTET_LENGTH	Y	--	--	--	Y	Y	Y
OF	Y	Y	Y	Y	Y	Y	Y
OFF	--	Y	--	--	Y	Y	Y
OFFSET	--	--	Y	--	Y	Y	Y
OID	--	--	--	--	Y	Y	Y
OLD	--	Y	--	Y	Y	Y	Y
ON	Y	Y	Y	Y	Y	Y	Y
ONLY	Y	Y	Y	Y	Y	Y	Y
OPEN	Y	Y	Y	Y	Y	Y	Y
OPERATION	--	Y	--	--	Y	Y	Y

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
OPERATORS	--	--	--	--	Y	Y	Y
OPTION	Y	Y	Y	Y	Y	Y	Y
OPTIMIZE	--	--	--	Y	Y	Y	Y
OR	Y	Y	Y	Y	Y	Y	Y
ORDER	Y	Y	Y	Y	Y	Y	Y
ORDINALITY	--	Y	--	--	--	--	--
OTHERS	--	--	--	--	Y	Y	Y
OUT	--	Y	--	Y	Y	Y	Y
OUTER	Y	Y	--	Y	Y	Y	Y
OUTPUT	Y	Y	Y	--	Y	Y	Y
OVER	--	-- (Changed to Y beginning in 2001)	--	Y	--	Y	Y
OVERFLOW	--	--	Y	--	Y	Y	Y
OVERLAPS	Y	--	--	--	--	--	--
OVERWRITE	--	--	Y	--	--	--	--
OWN	--	--	--	Y	Y	Y	Y
OWNER	--	--	Y	--	--	--	--

Table A-16: SQL reserved words (P)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
PAD	Y	Y	--	--	Y	Y	Y
PAGE	--	--	--	--	Y	Y	Y
PARAMETER	--	Y	--	--	--	--	--
PARAMETERS	--	Y	--	--	Y	Y	Y
PARTIAL	Y	Y	--	--	Y	Y	Y

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
PARTITION	--	--	--	Y	--	--	--
PARTITIONED	--	--	--	--	Y	Y	Y
PATH	--	Y	Y	--	--	--	--
PCTFREE	--	--	--	Y	Y	Y	Y
PENDANT	--	--	--	--	Y	Y	Y
PERCENT_RANK	--	--	--	Y	--	--	--
PERCENTILE_CONT	--	--	--	Y	--	--	--
PERCENTILE_DISC	--	--	--	Y	--	--	--
PIC	--	--	Y	--	Y	Y	Y
PICTURE	--	--	Y	Y	Y	Y	Y
POSITION	Y	--	--	--	Y	Y	Y
POSTFIX	--	Y	--	--	--	--	--
PREALLOCATED	--	--	Y	--	Y	Y	Y
PRECISION	Y	Y	Y	Y	Y	Y	Y
PREFERRED	--	--	Y	--	Y	Y	Y
PREFIX	--	Y	--	--	--	--	--
PREORDER	--	Y	--	--	Y	Y	Y
PREPARE	Y	Y	Y	Y	Y	Y	Y
PRESERVE	Y	Y	--	--	Y	Y	Y
PRIMARY	Y	Y	Y	Y	Y	Y	Y
PRIMLEGES	--	--	Y	--	--	--	--
PRIOR	Y	Y	--	--	Y	Y	Y
PRIVATE	--	--	Y	Y	Y	Y	Y
PRIVILEGES	Y	Y	--	Y	Y	Y	Y
PROCEDURE	Y	Y	Y	Y	Y	Y	Y
PROGRAM	--	--	--	Y	Y	Y	Y

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
PROTECTED	--	--	--	Y	Y	Y	Y
PUBLIC	Y	Y	Y	Y	Y	Y	Y
PURGE	--	--	--	Y	Y	Y	Y

Table A-17: SQL reserved words (R)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
RANDOM	--	--	--	--	Y	Y	Y
RANGE	--	--	--	Y	--	--	--
RANK	--	--	--	Y	--	--	--
RD	--	--	--	--	Y	Y	Y
RDAREA	--	--	--	Y	Y	Y	Y
RDNODE	--	--	--	Y	--	--	--
READ	Y	Y	Y	Y	Y	Y	Y
READS	--	Y	--	--	--	--	--
REAL	Y	Y	Y	Y	Y	Y	Y
RECOMPILE	--	--	--	--	Y	Y	Y
RECOVERABLE	--	--	Y	--	Y	Y	Y
RECOVERY	--	--	--	--	Y	Y	Y
RECURSIVE	--	Y	--	Y	Y	Y	Y
REDO	--	Y	--	--	--	--	--
REF	--	Y	--	--	Y	Y	Y
REFERENCES	Y	Y	Y	Y	Y	Y	Y
REFERENCING	--	Y	--	Y	Y	Y	Y
REGLIKE	--	--	Y	--	Y	Y	Y
REGR_AVGX	--	--	--	Y	--	--	--
REGR_AVGY	--	--	--	Y	--	--	--

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
REGR_COUNT	--	--	--	Y	--	--	--
REGR_INTERCEPT	--	--	--	Y	--	--	--
REGR_R2	--	--	--	Y	--	--	--
REGR_SLOPE	--	--	--	Y	--	--	--
REGR_SXX	--	--	--	Y	--	--	--
REGR_SXY	--	--	--	Y	--	--	--
REGR_SYY	--	--	--	Y	--	--	--
RELATIVE	Y	Y	--	--	Y	Y	Y
RELEASE	--	--	--	Y	Y	Y	Y
RELEASING	--	--	Y	--	Y	Y	Y
RENAME	--	--	Y	--	Y	Y	Y
REPEAT	--	Y	--	Y	--	--	--
RESERVED	--	--	--	Y	--	--	--
RESIGNAL	--	Y	--	--	Y	Y	Y
RESTART	--	--	Y	--	Y	Y	Y
RESTRICT	Y	Y	--	--	Y	Y	Y
RESULT	--	Y	--	--	--	--	--
RETURN	--	Y	--	Y	Y	Y	Y
RETURNS	--	Y	--	Y	Y	Y	Y
REVOKE	Y	Y	Y	Y	Y	Y	Y
RIGHT	Y	Y	--	--	Y	Y	Y
ROLE	--	Y	--	--	Y	Y	Y
ROLLBACK	Y	Y	Y	Y	Y	Y	Y
ROLLUP	--	Y	--	Y	--	--	--
ROOT	--	--	Y	--	Y	Y	Y
ROUTINE	--	Y	--	Y	Y	Y	Y

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
ROW	--	Y	--	Y	Y	Y	Y
ROW_NUMBER	--	--	--	Y	--	--	--
ROWID	--	--	Y	Y	Y	Y	Y
ROWS	Y	Y	--	Y	Y	Y	Y

Table A-18: SQL reserved words (S)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
SAVEPOINT	--	Y	--	--	Y	Y	Y
SCALE	--	--	Y	--	Y	Y	Y
SCAN	--	--	Y	--	Y	Y	Y
SCATTERED	--	--	Y	--	--	--	--
SCHEMA	Y	Y	Y	Y	Y	Y	Y
SCHEMAS	--	--	Y	--	Y	Y	Y
SCOPE	--	Y	--	--	Y	Y	Y
SCROLL	Y	Y	--	--	Y	Y	Y
SD	--	--	--	--	Y	Y	Y
SEARCH	--	Y	--	--	Y	Y	Y
SECOND	Y	Y	--	Y	Y	Y	Y
SECONDS	--	--	--	Y	Y	Y	Y
SECTION	Y	Y	Y	--	Y	Y	Y
SEGMENT	--	--	Y	--	Y	Y	Y
SELECT	Y	Y	Y	Y	Y	Y	Y
SENSITIVE	--	--	--	--	Y	Y	Y
SEPARATE	--	--	Y	--	Y	Y	Y
SEPARATOR	--	--	Y	--	Y	Y	Y
SEQUENCE	--	Y	--	--	Y	Y	Y

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
SESSION	Y	Y	--	--	Y	Y	Y
SESSION_USER	Y	Y	--	--	Y	Y	Y
SET	Y	Y	Y	Y	Y	Y	Y
SETS	--	Y	--	--	--	--	--
SFLIKE	--	--	--	--	Y	Y	Y
SHARE	--	--	--	Y	Y	Y	Y
SHLIKE	--	--	Y	--	--	--	--
SHORT	--	--	Y	--	Y	Y	Y
SIGN	--	--	Y	--	--	--	--
SIGNAL	--	Y	--	Y	Y	Y	Y
SIMILAR	--	--	--	--	Y	Y	Y
SIZE	Y	Y	Y	--	Y	Y	Y
SLOCK	--	--	Y	--	Y	Y	Y
SMALLFLT	--	--	--	Y	Y	Y	Y
SMALLINT	Y	Y	Y	Y	Y	Y	Y
SOME	Y	Y	Y	Y	Y	Y	Y
SPACE	Y	Y	--	--	Y	Y	Y
SPECIFIC	--	Y	--	Y	--	--	--
SPECIFICTYPE	--	Y	--	--	--	--	--
SPLIT	--	--	Y	--	Y	Y	Y
SQL	Y	Y	Y	--	Y	Y	Y
SQL_STANDARD	--	--	Y	--	Y	Y	Y
SQLCODE	Y	--	--	Y	Y	Y	Y
SQLCODE_OF_LAST_CONDITION	--	--	--	--	--	--	Y
SQLCODE_TYPE	--	--	Y	--	Y	Y	Y
SQLCOUNT	--	--	--	Y	Y	Y	Y

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
SQLDA	--	--	--	Y	Y	Y	Y
SQLERRM	--	--	--	Y	Y	Y	Y
SQLERRM_OF_LAST_CONDITION	--	--	--	--	--	--	Y
SQLERRMC	--	--	--	Y	Y	Y	Y
SQLERRML	--	--	--	Y	Y	Y	Y
SQLERROR	Y	--	Y	--	Y	Y	Y
SQLEXCEPTION	--	Y	--	--	Y	Y	Y
SQLNAME	--	--	--	Y	Y	Y	Y
SQLSTATE	Y	Y	--	Y	Y	Y	Y
SQLWARN	--	--	--	Y	Y	Y	Y
SQLWARNING	--	Y	Y	--	Y	Y	Y
START	--	Y	Y	--	Y	Y	Y
STATE	--	Y	--	--	--	--	--
STATEMENT	--	Y	--	--	--	--	--
STATIC	--	Y	Y	--	Y	Y	Y
STDDEV_POP	--	--	--	Y	--	--	--
STOP	--	--	Y	--	Y	Y	Y
STOPPING	--	--	--	Y	Y	Y	Y
STRUCTURE	--	Y	--	--	Y	Y	Y
SUBSTR	--	--	--	Y	Y	Y	Y
SUBSTRING	Y	--	--	--	Y	Y	Y
SUM	Y	--	Y	Y	Y	Y	Y
SUPPRESS	--	--	--	--	Y	Y	Y
SYNONYM	--	--	Y	--	Y	Y	Y
SYSTEM_USER	Y	Y	--	--	Y	Y	Y

Table A-19: SQL reserved words (T)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
TABLE	Y	Y	Y	Y	Y	Y	Y
TABLES	--	--	Y	--	--	--	--
TEMPORARY	Y	Y	--	--	Y	Y	Y
TERMINATE	--	Y	--	--	--	--	--
TEST	--	--	--	--	Y	Y	Y
TEXT	--	--	Y	--	Y	Y	Y
THAN	--	Y	--	--	--	--	--
THEN	Y	Y	--	Y	Y	Y	Y
THERE	--	--	--	--	Y	Y	Y
TIME	Y	Y	Y	Y	Y	Y	Y
TIMESTAMP	Y	Y	--	Y	Y	Y	Y
TIMESTAMP_FORMAT	--	--	--	--	--	Y	Y
TIMEZONE_HOUR	Y	Y	--	--	Y	Y	Y
TIMEZONE_MINUTE	Y	Y	--	--	Y	Y	Y
TO	Y	Y	Y	Y	Y	Y	Y
TRAILING	Y	Y	--	Y	Y	Y	Y
TRANSACTION	Y	Y	Y	--	Y	Y	Y
TRANSLATE	Y	--	--	--	Y	Y	Y
TRANSLATION	Y	Y	--	--	Y	Y	Y
TREAT	--	Y	--	--	Y	Y	Y
TRIGGER	--	Y	--	Y	Y	Y	Y
TRIM	Y	--	--	Y	Y	Y	Y
TRUE	Y	Y	--	Y	Y	Y	Y
TYPE	--	Y	Y	Y	Y	Y	Y

Table A-20: SQL reserved words (U)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
UAMT	--	--	Y	--	Y	Y	Y
UBINBUF	--	--	Y	--	Y	Y	Y
UCHAR	--	--	Y	--	Y	Y	Y
UDATE	--	--	Y	--	Y	Y	Y
UHAMT	--	--	Y	--	--	--	--
UHANT	--	--	--	--	Y	Y	Y
UHDATE	--	--	Y	--	Y	Y	Y
UNBOUNDED	--	--	--	Y	--	--	--
UNDER	--	Y	--	Y	Y	Y	Y
UNDO	--	Y	--	--	--	--	--
UNIFY_2000	--	--	Y	--	Y	Y	Y
UNION	Y	Y	Y	Y	Y	Y	Y
UNIONALL	--	--	--	--	Y	Y	Y
UNIQUE	Y	Y	Y	Y	Y	Y	Y
UNKNOWN	Y	Y	--	Y	Y	Y	Y
UNLIMITED	--	--	Y	--	Y	Y	Y
UNLOCK	--	--	Y	--	Y	Y	Y
UNTIL	--	Y	--	Y	Y	Y	Y
UNNEST	--	Y	--	--	--	--	--
UPDATE	Y	Y	Y	Y	Y	Y	Y
UPPER	Y	--	--	Y	Y	Y	Y
USAGE	Y	Y	Y	Y	Y	Y	Y
USE	--	--	Y	--	Y	Y	Y
USER	Y	Y	Y	Y	Y	Y	Y
USER_GROUP	--	--	--	Y	--	--	--

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
USER_LEVEL	--	--	--	Y	--	--	--
USING	Y	Y	Y	Y	Y	Y	Y
UTIME	--	--	Y	--	Y	Y	Y
UTXTBUF	--	--	Y	--	Y	Y	Y

Table A-21: SQL reserved words (V)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
VALUE	Y	Y	Y	Y	Y	Y	Y
VALUES	Y	Y	Y	Y	Y	Y	Y
VAR_POP	--	--	--	Y	--	--	--
VAR_SAMP	--	--	--	Y	--	--	--
VARCHAR	Y	Y	--	Y	Y	Y	Y
VARCHAR_FORMAT	--	--	--	--	--	Y	Y
VARIABLE	--	Y	--	--	Y	Y	Y
VARYING	Y	Y	--	Y	Y	Y	Y
VIEW	Y	Y	Y	Y	Y	Y	Y
VIRTUAL	--	--	--	--	Y	Y	Y
VISIBLE	--	--	--	--	Y	Y	Y
VOLATILE	--	--	Y	--	Y	Y	Y
VOLUME	--	--	Y	--	Y	Y	Y
VOLUMES	--	--	Y	--	Y	Y	Y

Table A-22: SQL reserved words (W)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
WAIT	--	--	--	Y	Y	Y	Y
WHEN	Y	Y	--	Y	Y	Y	Y

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
WHENEVER	Y	Y	Y	--	Y	Y	Y
WHERE	Y	Y	Y	Y	Y	Y	Y
WHILE	--	Y	--	Y	Y	Y	Y
WINDOW	--	--	--	Y	--	--	--
WITH	Y	Y	Y	Y	Y	Y	Y
WITHIN	--	--	--	Y	--	--	--
WITHOUT	--	Y	--	Y	Y	Y	Y
WORK	Y	Y	Y	Y	Y	Y	Y
WRITE	Y	Y	Y	--	Y	Y	Y

Table A-23: SQL reserved words (X)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
XLIKE	--	--	--	--	Y	Y	Y
XLOCK	--	--	Y	--	Y	Y	Y
XML	--	--	--	--	--	--	Y
XMLAGG	--	--	--	--	--	--	Y
XMLEXISTS	--	--	--	--	--	--	Y
XMLPARSE	--	--	--	--	--	--	Y
XMLQUERY	--	--	--	--	--	--	Y
XMLSERIALIZE	--	--	--	--	--	--	Y

Table A-24: SQL reserved words (Y)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
YEAR	Y	Y	--	Y	Y	Y	Y
YEARS	--	--	--	Y	Y	Y	Y

Table A-25: SQL reserved words (Z)

Reserved word	SQL92	SQL99	UNIFY	XDM/ RD	HiRDB (V6)	HiRDB (V7)	HiRDB (V8)
ZONE	--	Y	--	--	--	--	--

A.2 HiRDB reserved words

The reserved words used by HiRDB cannot be specified as names, such as authorization identifiers or table identifiers.

The following table lists the HiRDB reserved words.

Table A-26: HiRDB reserved words

Reserved word	Names that cannot be defined
ALL	Authorization identifier, table identifier, RDAREA name
HiRDB	Authorization identifier
MASTER#	Authorization identifier
PUBLIC	Authorization identifier
Names beginning with SQL or sql	Embedded variable, standard variable name, host identifier

#: Used in the following resource authorization identifiers provided by a data dictionary table or a plug-in:

- User defined type
- Index type
- Function

A.3 Reserved words that can be deleted using the SQL reserved word deletion facility

Tables *A-27* to *A-48* show the reserved words that can be deleted by using the SQL reserved word deletion facility, and the functionality that is disabled if a given reserved word is deleted.

The following legend is used in the tables:

--: There is no facility that is disabled when a reserved word is deleted.

Table A-27: Reserved words that can be deleted (A)

Reserved word	Disabled functionality
ABS	• Scalar function ABS

Reserved word	Disabled functionality
ALLOCATE	• LOB attribute with abstract data type • Plug-in
AMOUNT	--
ANDNOT	• Difference sets between lists
ANSI	--
ARRAY	• Repetition column
ASSERTION	--
ASYNC	--
AUTO	--

Table A-28: Reserved words that can be deleted (B)

Reserved word	Disabled functionality
BASE	--
BEGIN	• Compound statements of routine control SQL statements • User-defined function
BINARY	• Data types BINARY, BINARY LARGE OBJECT
BIT_AND_TEST	• Scalar function BIT_ADN_TEST
BLOB	• Data type BLOB
BOOLEAN	• Data type BOOLEAN
BOTH	--
BREADTH	--
BTREE	--
BUFFER	--
BYTE	--

Table A-29: Reserved words that can be deleted (C)

Reserved word	Disabled functionality
CALL	• Stored procedure • Execution of a command or utility

Reserved word	Disabled functionality
CASE	• CASE expression
CAST	• CAST specification
COALESCE	• CASE abbreviation
COLUMNS	--
COMPLETION	--
CONDITION	• Conditional declaration of a routine control SQL compound statement
CONFIGURATION	--
CONST	--
CONSTRAINT	• Referential constraint
CONTIGUOUS	--
CORRESPONDING	--
COUNT_FLOAT	• COUNT_FLOAT set function
CROSS	--
CURAID	--
CURRENT_DATE	• CURRENT_DATE value function • DEFAULT clause
CURRENT_TIME	• CURRENT_TIME value function • DEFAULT clause
CURRENT_TIMESTAMP	• CURRENT_TIMESTAMP value function • DEFAULT clause
CURRENT_USER	--
CYCLE	Sequence generator cycle option for the CREATE SEQUENCE statement

Table A-30: Reserved words that can be deleted (D)

Reserved word	Disabled functionality
DATA	• Deleting an abstract data type
DATABASE	--

Reserved word	Disabled functionality
DATE	● Data type DATE ● Scalar function DATE ● CURRENT_DATE value function ● DEFAULT clause
DAY	● Data type INTERVAL YEAR TO DAY ● Scalar function DAY ● Date operation
DAYS	● Scalar function DAYS ● Date operation
DEFER	--
DEMOTING	--
DEPTH	--
DEVICE	--
DIAGNOSTICS	● Embedded language syntax GET DIAGNOSTICS
DICTIONARY	--
DIGITS	● Scalar function DIGITS
DIRECT	--
DO	● Routine control SQL FOR statement, WHILE statement
DOUBLE_PRECISION	--

Table A-31: Reserved words that can be deleted (E)

Reserved word	Disabled functionality
EACH	● Trigger
EDIT	--
ELSE	● Routine control SQL IF statement
ELSEIF	● Routine control SQL IF statement
ENCRYPT	--
END	● CASE expression ● Routine control SQL compound statement, FOR statement, IF statement, WHILE statement
EQUALS	--

Reserved word	Disabled functionality
ESTIMATED	--
EXCEPTION	• Embedded language syntax GET DIAGNOSTICS
EXIT	• Handler declaration for routine control SQL compound statement
EXTERN	--
EXTRACT	--

Table A-32: Reserved words that can be deleted (F)

Reserved word	Disabled functionality
FALSE	• Boolen predicate IS FALSE
FIXED	--
FORCE	--
FREE	• Data manipulation SQL FREE LOCATOR statement
FULL	--
FUNCTION	• User-defined function • Plug-in • Narrowing of audit trail based on object name

Table A-33: Reserved words that can be deleted (G)

Reserved word	Disabled functionality
GENERAL	--
GET	• Embedded language syntax GET DIAGNOSTICS
GET_JAVA_STORED_ROUTINE_SOURCE	• Scalar function GET_JAVA_STORED_ROUTINE_SOURCE

Table A-34: Reserved words that can be deleted (H)

Reserved word	Disabled functionality
HANDLER	• Routine control SQL handler declaration
HELP	--
HEX	• Scalar function HEX

Reserved word	Disabled functionality
HOUR	● Data type INTERVAL HOUR TO SECOND ● Scalar function HOUR ● Time operation
HOURS	● Time operation
HUGE	--

Table A-35: Reserved words that can be deleted (I)

Reserved word	Disabled functionality
IF	● Routine control SQL IF statement
IGNORE	--
INNER	● Join table INNER JOIN
INOUT	● Stored procedure
INTERSECT	--
INTERVAL	● Data type INTERVAL HOUR TO SECOND, INTERVAL YEAR TO DAY ● Logging interval option for sequence generators (CREATE SEQUENCE statement)
ISOLATION	● SQL compile option ISOLATION
IS_USER_CONTAINED_IN_HDS_GROUP	● Scalar function IS_USER_CONTAINED_IN_HDS_GROUP

Table A-36: Reserved words that can be deleted (L)

Reserved word	Disabled functionality
LARGE	● Data type LARGE DECIMAL, BINARY LARGE OBJECT
LEADING	--
LEAVE	● Routine control SQL LEAVE statement
LENGTH	● Scalar function LENGTH ● SQL compile option SUBSTR LENGTH
LESS	--
LEVEL	● SQL compile options OPTIMIZE LEVEL, ADD OPTIMIZE LEVEL
LIMIT	● LIMIT clause

Reserved word	Disabled functionality
LINES	--
LINK	--
LOCATOR	--
LOCKS	--
LOGID	--
LOGNAME	--
LOOP	--
LOWER	• Scalar function LOWER

Table A-37: Reserved words that can be deleted (M)

Reserved word	Disabled functionality
MAXUSAGES	--
MINUTE	• Scalar function MINUTE • Time operation
MINUTES	• Time operation
MOD	• Scalar function MOD
MONTH	• Scalar function MONTH • Date operation
MONTHS	• Date operation
MOVE	--

Table A-38: Reserved words that can be deleted (N)

Reserved word	Disabled functionality
NATURAL	--
NEW	• Trigger
NOWAIT	• Lock option NO WAIT
NULLIF	• CASE abbreviation

Table A-39: Reserved words that can be deleted (O)

Reserved word	Disabled functionality
OFF	--
OID	--
OLD	● Trigger
ONLY	● Falsification-prevented table ● Read-only view table ● FOR READ ONLY
OPERATION	--
OPERATORS	--
OPTIMIZE	● SQL compile options OPTIMIZE LEVEL, ADD OPTIMIZE LEVEL
OTHERS	--
OUT	● Stored procedure
OUTER	● Join table OUTER JOIN
OVER	● Window function
OVERFLOW	--
OWN	--

Table A-40: Reserved words that can be deleted (P)

Reserved word	Disabled functionality
PARAMETERS	--
PENDANT	--
PIC	--
PICTURE	--
PREALLOCATED	--
PREFERRED	--
PREORDER	--
PRIVATE	● Abstract data type
PROTECTED	● Abstract data type

Reserved word	Disabled functionality
PURGE	• Changing table partitioning conditions • Definition SQL ALTER TABLE WITHOUT PURGE • Data manipulation SQL PURGE TABLE statement

Table A-41: Reserved words that can be deleted (R)

Reserved word	Disabled functionality
RANDOM	--
RD	--
READ	• Read-only view table • FOR READ ONLY
RECOMPILE	--
RECOVERABLE	--
RECURSIVE	--
REF	--
REFERENCING	• Trigger
REGLIKE	--
RELEASING	--
RESTART	--
RETURN	• User-defined function
RETURNS	• User-defined function
RIGHT	--
ROLE	--
ROOT	--
ROUTINE	• Compiling a routine
ROW	• Lock LOCK ROW by row • Trigger • (ROW specification) interface by row
ROWS	--

Table A-42: Reserved words that can be deleted (S)

Reserved word	Disabled functionality
SAVEPOINT	--
SCALE	--
SCAN	--
SCHEMAS	--
SCOPE	--
SD	--
SEARCH	--
SECOND	● Data INTERVAL HOUR TO SECOND ● Scalar function SECOND ● Time operation
SECONDS	● Time operation
SENSITIVE	--
SEPARATE	--
SEPARATOR	--
SEQUENCE	● CREATE SEQUENCE statement ● DROP SEQUENCE statement
SESSION_USER	--
SFLIKE	--
SHORT	--
SIGNAL	● Routine control SQL statement SIGNAL statement
SIMILAR	● SIMILAR predicate
SLOCK	--
SQL_STANDARD	--
SQLCODE_OF_LAST_CONDITION	● SQLCODE_OF_LAST_CONDITION value specification
SQLCODE_TYPE	--
SQLDA	--
SQLERRM	--

Reserved word	Disabled functionality
SQLERRM_OF_LAST_CONDITION	• SQLERRM_OF_LAST_CONDITION value specification
SQLERRMC	--
SQLERRML	--
SQLEXCEPTION	--
SQLWARN	--
STATIC	--
STOP	--
STOPPING	--
SUBSTR	• Scalar function SUBSTR • SQL compile option SUBSTR LENGTH
SYSTEM_USER	--

Table A-43: Reserved words that can be deleted (T)

Reserved word	Disabled functionality
TEST	--
TEXT	--
THEN	• CASE expression • Routine control SQL IF statement
THERE	--
TIME	• Data type TIME • Scalar function TIME • CURRENT_TIME value function • DEFAULT clause default value
TIMESTAMP	• Data type TIMESTAMP • Scalar function TIMESTAMP • CURRENT_TIMESTAMP value function • DEFAULT clause default value
TIMESTAMP_FORMAT	• Scalar function TIMESTAMP_FORMAT
TRAILING	--
TRANSACTION	--

Reserved word	Disabled functionality
TREAT	--
TRIGGER	● Trigger ● Narrowing of audit trail based on object name
TRIM	--
TRUE	● Boolean predicate IS TRUE
TYPE	● Abstract data type ● Plug-in ● Narrowing of audit trail based on object name

Table A-44: Reserved words that can be deleted (U)

Reserved word	Disabled functionality
UAMT	--
UBINBUF	--
UCHAR	--
UDATE	--
UHANT	--
UHDATE	--
UNDER	● Abstract data type
UNIFY_2000	--
UNIONALL	--
UNKNOWN	● Boolean predicate IS UNKNOWN
UNLIMITED	--
UNLOCK	--
UNTIL	● UNTIL DISCONNECT
UPPER	● Scalar function UPPER
USAGE	● FOR PUBLIC USAGE sequence generator (CREATE SEQUENCE statement)
USE	--
UTIME	--
UTXTBUF	--

Table A-45: Reserved words that can be deleted (V)

Reserved word	Disabled functionality
VALUE	• Scalar function VALUE • NEXT VALUE expression
VARCHAR_FORMAT	• Scalar function VARCHAR_FORMAT
VARIABLE	--
VARYING	• Data types CHARACTER VARYING, NATIONAL CHARACTER VARYING
VIRTUAL	--
VISIBLE	--
VOLATILE	--
VOLUME	--
VOLUMES	--

Table A-46: Reserved words that can be deleted (W)

Reserved word	Disabled functionality
WHEN	• CASE expression • Trigger
WHILE	• Falsification-prevented table • Routine control SQL WHILE statement

Table A-47: Reserved words that can be deleted (X)

Reserved word	Disabled functionality
XLOCK	--
XML	• XML type • XML constructor function
XMLAGG	• XMLAGG set function
XMLEXISTS	• XMLEXISTS predicate
XMLPARSE	• XMLPARSE function
XMLQUERY	• XMLQUERY function
XMLSERIALIZE	• XMLSERIALIZE function

Table A-48: Reserved words that can be deleted (Y)

Reserved word	Disabled functionality
YEAR	• Data type INTERVAL YEAR TO DAY • Scalar function YEAR • Date operation
YEARS	• Date operation

B. List of SQLs

The following table lists the SQL statements that can be used by type. The *Under OLTP* column indicates whether the SQL statement can be used in an X/Open-compliant UAP running under OLTP.

Table B-1: SQL statements (definition SQL)

Type	Function	Available SQL		
		C	COBOL	Under OLTP
ALTER INDEX (Change index definition)	Changes the name of an index.	Y	Y	N
ALTER PROCEDURE (Recreate SQL object of procedure)	Recreates an SQL object of a procedure.	Y	Y	N
ALTER ROUTINE (Recreate SQL objects for functions, procedures, and triggers)	Recreates SQL objects for functions, procedures, and triggers.	Y	Y	N
ALTER TABLE (Alter table definition)	Adds column to a base table. Changes the data type. Increases the maximum length of columns of the variable-length data type. Deletes empty base table columns. Changes the uniqueness constraint on empty base table cluster keys. Renames tables and columns.	Y	Y	N
ALTER TRIGGER (Recreate trigger SQL object)	Recreates a trigger SQL object.	Y	Y	N
COMMENT (Comment)	Adds a comment in a table or a column.	Y	Y	N
CREATE AUDIT (Define audit target event)	Defines the audit event to be recorded as an audit trail, and its target.	Y	Y	N
CREATE CONNECTION SECURITY (Define CONNECT security facility)	Defines security items related to the CONNECT security facility.	Y	Y	N
CREATE [PUBLIC] FUNCTION (Define function, define public function)	Defines a function or a public function.	Y	Y	N
CREATE INDEX (Define index)	Defines an index (ascending or descending order) on the columns in a base table.	Y	Y	N

Type	Function	Available SQL		
		C	COBOL	Under OLTP
CREATE [PUBLIC] PROCEDURE (Define procedure, define public procedure)	Defines a procedure or a public procedure.	Y	Y	N
CREATE SCHEMA (Define schema)	Defines a schema.	Y	Y	N
CREATE SEQUENCE (Define sequence generator)	Defines a sequence generator.	Y	Y	N
CREATE TABLE (Define base table)	Defines a base table.	Y	Y	N
CREATE TRIGGER (Define trigger)	Defines a trigger.	Y	Y	N
CREATE TYPE (Define type)	Defines an abstract data type.	Y	Y	N
CREATE [PUBLIC] VIEW (Define view, define public view)	Defines a view table or a public view table.	Y	Y	N
DROP AUDIT (Delete audit target event)	Deletes from the audit targets definitions for which the audit target event defined in CREATE AUDIT matches the contents.	Y	Y	N
DROP CONNECTION SECURITY (Delete CONNECT security facility)	Deletes security items related to the CONNECT security facility.	Y	Y	N
DROP DATA TYPE (Delete user-defined data type)	Deletes a user-defined data type.	Y	Y	N
DROP [PUBLIC] FUNCTION (Delete function, delete public function)	Deletes a function or a public function.	Y	Y	N
DROP INDEX (Delete index)	Deletes an index.	Y	Y	N
DROP [PUBLIC] PROCEDURE (Delete procedure, delete public procedure)	Deletes a procedure or a public procedure.	Y	Y	N
DROP SCHEMA (Delete schema)	Deletes a schema.	Y	Y	N
DROP SEQUENCE (Delete sequence generator)	Deletes a sequence generator.	Y	Y	N
DROP TABLE (Delete table)	Deletes a base table and any indexes, comments, access privilege, view tables, and triggers that are associated with it.	Y	Y	N
DROP TRIGGER (Delete trigger)	Deletes a trigger.	Y	Y	N

Type	Function	Available SQL		
		C	COBOL	Under OLTP
DROP [PUBLIC] VIEW (Delete view table, public view table)	Deletes a view table or a public view table.	Y	Y	N
GRANT AUDIT (Change auditor's password)	Changes the auditor's password.	Y	Y	N
GRANT CONNECT (Grant connect privilege)	Grants the connect privilege to a user.	Y	Y	N
GRANT DBA (Grant DBA privilege)	Grants the DBA privilege to a user.	Y	Y	N
GRANT RDAREA (Grant RDAREA utilization privilege)	Grants the RDAREA utilization privilege to a user.	Y	Y	N
GRANT SCHEMA (Grant schema definition privilege)	Grants the schema definition privilege to a user.	Y	Y	N
GRANT access privilege (Grant access privileges)	Grants access privilege to a user.	Y	Y	N
REVOKE CONNECT (Revoke connect privilege)	Revokes a user's connect privilege.	Y	Y	N
REVOKE DBA (Revoke DBA privilege)	Revokes a user's DBA privilege.	Y	Y	N
REVOKE RDAREA (Revoke RDAREA utilization privilege)	Revokes a user's RDAREA utilization privilege.	Y	Y	N
REVOKE SCHEMA (Revoke schema definition privilege)	Revokes a user's schema definition privilege.	Y	Y	N
REVOKE access privilege (Revoke access privileges)	Revokes a user's access privilege.	Y	Y	N

Y: Can be used.

N: Cannot be used.

Table B-2: SQL statements (data manipulation SQL)

Type	Function	Available SQL		
		C	COBOL	Under OLTP
ALLOCATE CURSOR statement (Allocate cursor)	Allocates a cursor to the SELECT statement preprocessed by the PREPARE statement or a group of result sets returned by a procedure.	Y	Y	Y
ASSIGN LIST statement (Create list)	Creates a list from a base table.	Y	Y	Y
CALL statement* (Call procedure)	Calls a procedure.	Y	Y	Y
CLOSE statement (Close cursor)	Closes a cursor.	Y	Y	Y
DEALLOCATE PREPARE statement (Nullify the preprocessing)	Releases the allocation of the SQL statement preprocessed by the PREPARE statement.	Y	Y	Y
DECLARE CURSOR (Declare cursor)	Declares a cursor to receive the results of retrieval by the SELECT statement, one row at a time, using the FETCH statement.	Y	Y	Y
DELETE statement (Delete row)	Deletes a row that satisfies a set of specified search conditions or a row that is indicated by the cursor.	Y	Y	Y
Preparable dynamic DELETE statement: locating (Delete row using a preprocessable cursor)	Deletes the row pointed to by a specified cursor, used to execute the statement dynamically.	Y	Y	Y
DESCRIBE statement (Receive retrieval information and I/O information)	Returns to the SQL descriptor area SQL retrieval information, output information, or input information that has been preprocessed by the PREPARE statement.	Y	Y	Y
DESCRIBE CURSOR statement (Receive cursor retrieval information)	Returns retrieval information on the cursor that references the result set returned by a procedure to the SQL descriptor area.	Y	Y	Y

Type	Function	Available SQL		
		C	COBOL	Under OLTP
DESCRIBE TYPE statement (Receive definition information on a user-defined data type)	Returns to the SQL descriptor area definition information (attribute data codes and data lengths) on a user-defined data type that is directly or indirectly contained in SQL retrieval item information that is preprocessed by a PREPARE statement.	Y	Y	Y
DROP LIST statement (Delete list)	Deletes a list.	Y	Y	Y
EXECUTE statement (Execute SQL)	Executes SQL preprocessed by the PREPARE statement.	Y	Y	Y
EXECUTE IMMEDIATE statement (Preprocess and execute SQL)	Preprocesses and executes the SQL specified in a character string.	Y	Y	Y
FETCH statement (Fetch data)	Advances to the next row the cursor indicating the row to be fetched and reads to the embedded variable specified by the INTO clause the value of a column in that row.	Y	Y	Y
FREE LOCATOR statement (Invalidate locator)	Nullifies the locator.	Y	Y	Y
INSERT statement (Inserting row)	Inserts a row into a table. Can insert one row by direct value specification. Can also insert one or more rows by using the SELECT statement.	Y	Y	Y
OPEN statement (Open cursor)	Opens a cursor. Positions the cursor declared in DECLARE CURSOR or allocated by ALLOCATE CURSOR at a position preceding the first row of the retrieval results so that retrieval results can be fetched.	Y	Y	Y
PREPARE statement (Preprocess SQL)	Preprocesses the SQL indicated by a character string and assigns a name (SQL statement identifier or extended statement name) to the SQL.	Y	Y	Y
PURGE TABLE statement (Delete all rows)	Deletes all rows in a base table.	Y	Y	N

Type	Function	Available SQL		
		C	COBOL	Under OLTP
One-row SELECT statement (Retrieve one row)	Searches table data. When only one row of data is to be fetched from a table, the one-row SELECT statement, which fetches data without using a cursor, must be specified.	Y	Y	Y
Dynamic SELECT statement (Dynamic retrieval)	Searches table data. The dynamic SELECT statement is preprocessed by the PREPARE statement. When retrieving data, either declare the cursor by using DECLARE CURSOR or allocate the cursor using the ALLOCATE CURSOR statement, and then use the cursor to fetch retrieval results row by row.	Y	Y	Y
UPDATE statement (Update data)	Updates the value of a specified column in a table row that meets a specified search condition or in the row indicated by the cursor.	Y	Y	Y
Preparable dynamic UPDATE statement: locating (Update data using a preprocessable cursor)	Updates the value of a specified column in the row pointed to by a specified cursor; this is used for dynamic execution.	Y	Y	Y
Assignment statement (Assign value)	Assigns a value.	Y	Y	Y

Y: Can be used.

N: Cannot be used.

Note

If a procedure is called under OLTP, and if the procedure contains PURGE TABLE, COMMIT, or ROLLBACK statements, the procedure cannot be executed.

Table B-3: SQL statements (control SQL)

Type	Function	Available SQL		
		C	COBOL	Under OLTP
CALL COMMAND statement (Execute command or utility)	Executes a HiRDB command or utility.	Y	Y	Y
COMMIT statement (Terminate transaction normally)	Terminates normally the current transaction, sets a synchronization point, and generates one unit of commitment. The transaction puts the contents of the updated database into effect.	Y	Y	N
CONNECT statement (Connect to HiRDB)	Posts an authorization identifier and password to HiRDB so that the UAP can use HiRDB.	Y	Y	N
DISCONNECT statement (Disconnect from HiRDB)	Terminates normally the current transaction, sets a synchronization point, and generates one unit of commitment. Afterwards, disconnects the UAP from HiRDB.	Y	Y	N
LOCK statement (Lock table)	Locks a specified table.	Y	Y	Y
ROLLBACK statement (Cancel transaction)	Cancels the current transaction and any database updates performed within the transaction.	Y	Y	N
SET SESSION AUTHORIZATION statement (Change executing user)	Changes the currently connected user.	Y	Y	Y

Y: Can be used.

N: Cannot be used.

Table B-4: SQL statements (embedded language)

Type	Function	Available SQL		
		C	COBOL	Under OLTP
BEGIN DECLARE SECTION (Embedded SQL begin declaration)	Indicates the beginning of an embedded variable declaration section. Embedded variables and indicator variables used in SQL are specified in an embedded variable declaration section.	Y	Y	Y
END DECLARE SECTION (Embedded SQL end declaration)	Indicates the end of an embedded variable declaration section.	Y	Y	Y
ALLOCATE CONNECTION HANDLE (Allocate connection handle)	Allocates a connection handle to be used by a UAP in an environment where a multi-connection function is used.	Y	Y	N
FREE CONNECTION HANDLE (Release connection handle)	Releases a connection handle that was allocated by ALLOCATE CONNECTION HANDLE.	Y	Y	N
DECLARE CONNECTION HANDLE SET (Declare connection handle to be used)	Declares the connection handle to be used by a UAP in an environment where a multi-connection function is used.	Y	Y	N
DECLARE CONNECTION HANDLE UNSET (Reset all connection handles being used)	Resets all declarations of connection handle usage specified in DECLARE CONNECTION HANDLE SET statements prior to this statement.	Y	N	N
GET CONNECTION HANDLE (Get connection handle)	When the multi-connection facility is used under the X/Open XA interface environment, allocates the connection handle to be used by the UAP.	Y	Y	N
COPY (Include library text)	Includes a library text into the source program.	N	Y	Y
GET DIAGNOSTICS (Retrieve diagnostic information)	If the preceding SQL statement is the CREATE PROCEDURE or CALL statement, obtains error information and diagnostic information from the diagnostics area.	N	N	N
COMMAND EXECUTE (Execute command from UAP)	Executes HiRDB and OS commands from within a UAP.	Y	N	N

Type	Function	Available SQL		
		C	COBOL	Under OLTP
SQL prefix	Indicates the beginning of an SQL.	Y	Y	Y
SQL terminator	Indicates the end of an SQL.	Y	Y	Y
WHENEVER (Embedded exception declaration)	Declares UAP processing by means of a return code that has been set in the SQL communication area by HiRDB after the SQL has executed.	Y	Y	Y
SQLCODE variable	Receives the return code issued by HiRDB following the execution of an SQL.	Y	Y	Y
SQLSTATE variable	Receives the return code issued by HiRDB following the execution of SQL.	Y	Y	Y
PDCNCTHDL-type variable declaration	Declares the handle that has the connection information to be used in an environment where a multi-connection function is used.	Y	N	N
INSTALL JAR (Register JAR file)	Installs a JAR file on the HiRDB server.	Y	N	N
REPLACE JAR (Re-register JAR file)	Installs a JAR file on the HiRDB server by overwriting it.	Y	N	N
REMOVE JAR (Delete JAR file)	Uninstalls a JAR file from the HiRDB server.	Y	N	N
DECLARE AUDIT INFO SET (Set user connection information)	Sets account information and other user connection information for applications that access a HiRDB server.	Y	Y	Y

Y: Can be used.

N: Cannot be used.

Table B-5: SQL statements (routine control SQL)

Type	Function	Available SQL		
		C	COBOL	Under OLTP
Compound statement (Execute multiple statements)	Executes a group of SQL statements as a single SQL statement.	Y	Y	N
Assignment statement (Assign value)	Assigns a value to an SQL variable or an SQL parameter.	Y	Y	N
IF statement (Execute by conditional branching)	Executes SQL statements under certain conditions.	Y	Y	N
RETURN statement (Return function return value)	Returns a return value from a function.	Y#1	Y#1	N
WHILE statement (Repeat a set of statements)	Executes repeatedly a set of SQL statements.	Y	Y	N
FOR statement (Repeat a statement on rows)	Repeatedly executes SQL statements on rows in a table.	Y#3	Y#3	N
LEAVE statement (Leave statement)	Exits from a compound statement or the WHILE statement and terminates the execution of the statement.	Y	Y	N
WRITE LINE statement (Output of a character string to a file)	Outputs the character string of a specified value expression to a file.	Y	Y	N
SIGNAL statement (Error signaling)	Generates and signals an error.	Y#2	Y#2	N
RESIGNAL statement (Error resignaling)	Generates and resignals an error.	Y#2	Y#2	N

Legend:

Y: This cannot be used directly in a UAP. However, it can be used to define SQL procedures, SQL functions, and trigger actions in CREATE PROCEDURE, CREATE FUNCTION, and CREATE TRIGGER.

N: Cannot be used.

Note

The following SQL statements, other than routine control SQL statements, can be specified in a procedure definition: CALL statement, CLOSE statement, DECLARE CURSOR statement, DELETE statement, FETCH statement, INSERT statement, OPEN statement, PURGE TABLE statement, single-row SELECT statement, UPDATE

statement, `COMMIT` statement, `LOCK` statement, and `ROLLBACK` statement. SQL statements other than routine control SQL statements cannot be used in a function.

#1: This cannot be used when defining an SQL procedure and a trigger action in `CREATE PROCEDURE` and `CREATE TRIGGER`.

#2: This cannot be used to define an SQL function in `CREATE FUNCTION`.

#3: This cannot be used in `CREATE FUNCTION`.

C. Example Database

The following figure shows an example of the basic table structure for the tables that are used as examples in the manual.

Figure C-1: Example of basic table structure

Table name: Inventory table (STOCK)

CHAR(4)	NCHAR(8)	NCHAR(1)	INTEGER	INTEGER
Product code (PCODE)	Product name (PNAME)	Color (COLOR)	Unit price (PRICE)	Inventory level (SQUANTITY)
101M	Blouse	White	35.00	85
101L	Blouse	Blue	35.00	62
201M	Polo shirt	White	36.40	29
202M	Polo shirt	Red	36.40	67
302S	Skirt	White	51.10	65
353M	Skirt	Green	47.60	56
353L	Skirt	Red	47.60	18
411M	Sweater	Blue	84.00	12
412M	Sweater	Red	84.00	22
591S	Socks	White	2.50	280
591M	Socks	Blue	2.50	90
591L	Socks	Red	2.50	300

Table name: Orders received (ORDER)

CHAR(6)	CHAR(5)	CHAR(4)	INTEGER	DATE	TIME
Order slip number (FNO)	Customer code (CCODE)	Product code (PCODE)	Quantity ordered (OQUANTITY)	Order received date (ODATE)	Order received time (OTIME)
026551	TT002	101M	10	1995-06-14	09:23:11
026552	TT002	591M	25	1995-06-14	09:23:11
026553	TH001	353M	8	1995-06-14	10:10:55
026554	TK001	411M	6	1995-06-14	10:15:47
026555	TA001	591M	30	1995-06-14	10:15:47
026556	TT002	202M	10	1995-06-14	11:48:09
026557	TZ001	411M	5	1995-06-14	13:02:00
026558	TZ001	412M	4	1995-06-14	13:02:00
026559	TH001	591M	80	1995-06-14	14:04:16
026560	TT001	591L	10	1995-06-14	15:31:20

Index

D

E

F

G

H

I

M

N

O

P

Q

R

T