HITACHI
Inspire the Next

Hitachi Advanced Database
SQL Reference

3000-6-504-H0(E)

Notices

■ Relevant program products

P-9W62-C411 Hitachi Advanced Data Binder version 05-01 (for Red Hat(R) Enterprise Linux(R) Server 6 (64-bit x86_64) and Red Hat(R) Enterprise Linux(R) Server 7 (64-bit x86_64))

P-9W62-C311 Hitachi Advanced Data Binder Client version 05-01 (for Red Hat(R) Enterprise Linux(R) Server 6 (64-bit x86_64) and Red Hat(R) Enterprise Linux(R) Server 7 (64-bit x86_64))

P-2462-C114 Hitachi Advanced Data Binder Client version 05-01 (for Windows 7, Windows 8.1, Windows 10, Windows Server 2008 R2, Windows Server 2012, Windows Server 2012 R2, and Windows Server 2016)

This manual can be used for products other than the products shown above. For details, see the *Release Notes*.

Hitachi Advanced Data Binder is the product name of Hitachi Advanced Database in Japan.

■ Trademarks

HITACHI, HA Monitor, HiRDB, Job Management Partner 1 and JP1 are either trademarks or registered trademarks of Hitachi, Ltd. in Japan and other countries.

Access is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries.

AMD is a trademark of Advanced Micro Devices, Inc.

Excel is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries.

Intel is a trademark of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

Microsoft is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries.

MSDN is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Red Hat is a registered trademark of Red Hat, Inc. in the United States and other countries.

Red Hat Enterprise Linux is a registered trademark of Red Hat, Inc. in the United States and other countries.

UNIX is a trademark of The Open Group.

Visual Studio is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries.

Windows is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries.

Windows Server is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries.

Other company and product names mentioned in this document may be the trademarks of their respective owners.

1. This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org/)

2. This product includes cryptographic software written by Eric Young (eay@cryptsoft.com).

3. This product includes software written by Tim Hudson (tjh@cryptsoft.com).

4. This product uses OpenSSL Toolkit software in accordance with the OpenSSL License and Original SSLeay License, which are described as follows.

```
LICENSE ISSUES
==============
The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.
OpenSSL License
---------------
/* ====================================================================
* Copyright (c) 1998-2011 The OpenSSL Project. All rights reserved.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
*
* 1. Redistributions of source code must retain the above copyright
* notice, this list of conditions and the following disclaimer.
*
* 2. Redistributions in binary form must reproduce the above copyright
* notice, this list of conditions and the following disclaimer in
* the documentation and/or other materials provided with the
* distribution.
*
* 3. All advertising materials mentioning features or use of this
* software must display the following acknowledgment:
* "This product includes software developed by the OpenSSL Project
* for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
*
* 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
* endorse or promote products derived from this software without
* prior written permission. For written permission, please contact
* openssl-core@openssl.org.
*
* 5. Products derived from this software may not be called "OpenSSL"
* nor may "OpenSSL" appear in their names without prior written
* permission of the OpenSSL Project.
*
* 6. Redistributions of any form whatsoever must retain the following
* acknowledgment:
* "This product includes software developed by the OpenSSL Project
* for use in the OpenSSL Toolkit (http://www.openssl.org/)"
```

*
* THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
* EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
* OF THE POSSIBILITY OF SUCH DAMAGE.
* ==
*
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com). This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*
*/
Original SSLeay License

/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
* All rights reserved.
*
* This package is an SSL implementation written
* by Eric Young (eay@cryptsoft.com).
* The implementation was written so as to conform with Netscapes SSL.
*
* This library is free for commercial and non-commercial use as long as
* the following conditions are aheared to. The following conditions
* apply to all code found in this distribution, be it the RC4, RSA,
* lhash, DES, etc., code; not just the SSL code. The SSL documentation
* included with this distribution is covered by the same copyright terms
* except that the holder is Tim Hudson (tjh@cryptsoft.com).
*
* Copyright remains Eric Young's, and as such any Copyright notices in
* the code are not to be removed.
* If this package is used in a product, Eric Young should be given attribution
* as the author of the parts of the library used.
* This can be in the form of a textual message at program startup or
* in documentation (online or textual) provided with the package.

*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the copyright
* notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
* notice, this list of conditions and the following disclaimer in the
* documentation and/or other materials provided with the distribution.
* 3. All advertising materials mentioning features or use of this software
* must display the following acknowledgement:
* "This product includes cryptographic software written by
* Eric Young (eay@cryptsoft.com)"
* The word 'cryptographic' can be left out if the rouines from the library
* being used are not cryptographic related :-).
* 4. If you include any Windows specific code (or a derivative thereof) from
* the apps directory (application code) you must include an acknowledgement:
* "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
*
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* The licence and distribution terms for any publically available version or
* derivative of this code cannot be changed. i.e. this code cannot simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/
■Double precision SIMD-oriented Fast Mersenne Twister (dSFMT)
Copyright (c) 2007, 2008, 2009 Mutsuo Saito, Makoto Matsumoto
and Hiroshima University.
Copyright (c) 2011, 2002 Mutsuo Saito, Makoto Matsumoto, Hiroshima
University and The University of Tokyo.
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

* Neither the name of the Hiroshima University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

■ Microsoft product screen shots

Microsoft product screen shots reprinted with permission from Microsoft Corporation.

■ Microsoft product name abbreviations

This manual uses the following abbreviations for Microsoft product names:

Abbreviation			Full name or meaning
Windows	Windows 7	Windows 7 x86	Microsoft$^{(R)}$ Windows$^{(R)}$ 7 Professional (32-bit)
			Microsoft$^{(R)}$ Windows$^{(R)}$ 7 Enterprise (32-bit)
			Microsoft$^{(R)}$ Windows$^{(R)}$ 7 Ultimate (32-bit)
		Windows 7 x64	Microsoft$^{(R)}$ Windows$^{(R)}$ 7 Professional (64-bit)
			Microsoft$^{(R)}$ Windows$^{(R)}$ 7 Enterprise (64-bit)
			Microsoft$^{(R)}$ Windows$^{(R)}$ 7 Ultimate (64-bit)
	Windows 8.1	Windows 8.1 x86	Windows$^{(R)}$ 8.1 Pro (32-bit)
			Windows$^{(R)}$ 8.1 Enterprise (32-bit)

Abbreviation			Full name or meaning
		Windows 8.1 x64	Windows$^{(R)}$ 8.1 Pro (64-bit)
			Windows$^{(R)}$ 8.1 Enterprise (64-bit)
	Windows 10	Windows 10 x86	Windows$^{(R)}$ 10 Pro (32-bit)
			Windows$^{(R)}$ 10 Enterprise (32-bit)
		Windows 10 x64	Windows$^{(R)}$ 10 Pro (64-bit)
			Windows$^{(R)}$ 10 Enterprise (64-bit)
	Windows Server 2008 R2		Microsoft$^{(R)}$ Windows Server$^{(R)}$ 2008 R2 Standard
			Microsoft$^{(R)}$ Windows Server$^{(R)}$ 2008 R2 Enterprise
			Microsoft$^{(R)}$ Windows Server$^{(R)}$ 2008 R2 Datacenter
	Windows Server 2012		Microsoft$^{(R)}$ Windows Server$^{(R)}$ 2012 Standard
			Microsoft$^{(R)}$ Windows Server$^{(R)}$ 2012 Datacenter
	Windows Server 2012 R2		Microsoft$^{(R)}$ Windows Server$^{(R)}$ 2012 R2 Standard
			Microsoft$^{(R)}$ Windows Server$^{(R)}$ 2012 R2 Datacenter
	Windows Server 2016		Microsoft$^{(R)}$ Windows Server$^{(R)}$ 2016 Standard
			Microsoft$^{(R)}$ Windows Server$^{(R)}$ 2016 Datacenter

■ Restrictions

Information in this document is subject to change without notice and does not represent a commitment on the part of Hitachi. The software described in this manual is furnished according to a license agreement with Hitachi. The license agreement contains all of the terms and conditions governing your use of the software and documentation, including all warranty rights, limitations of liability, and disclaimers of warranty.

Material contained in this document may describe Hitachi products not available or features not available in your country.

No part of this material may be reproduced in any form or by any means without permission in writing from the publisher.

■ Issued

Apr. 2020

■ Copyright

All Rights Reserved. Copyright (C) 2012, 2020, Hitachi, Ltd.

Preface

This manual describes the SQL syntax used for manipulating databases in Hitachi Advanced Database.

Note that, in this manual, and in the information output by the product (messages, command output results, and so on), *HADB* is often used in place of *Hitachi Advanced Database*.

■ Intended readers

This manual is intended for:

- Application developers
- System engineers who design and set up HADB systems, and system administrators

Readers of this manual must have:

- A basic knowledge of SQL
- A basic knowledge of Java programming and a basic knowledge of JDBC (if you plan to create application programs in Java)
- A basic knowledge of programming in C or C++ (if you plan to create application programs in C or C++)
- A basic knowledge of ODBC (if you plan to create ODBC-compliant application programs)

■ Organization of this manual

This manual is organized into the following parts, chapters, and appendixes:

1. SELECT Statement Examples

   Chapter 1 explains, using examples, how to write `SELECT` statements with constructs such as predicates, set functions, `GROUP BY` clauses, and `HAVING` clauses. Read this chapter to understand how to write basic `SELECT` statements.

2. List of SQL Statements

   Chapter 2 provides a list of SQL statements that are supported by HADB, and explains how to read the SQL syntax specification format used in this manual.

3. Definition SQL

   Chapter 3 describes the functions, specification formats, and rules of definition SQL statements.

4. Data Manipulation SQL

   Chapter 4 describes the functions, specification formats, and rules of data manipulation SQL statements.

5. Control SQL

   Chapter 5 describes the functions, specification formats, and rules of control SQL statements.

6. SQL Basics

   Chapter 6 describes the basic elements of SQL, including rules for writing SQL statements, data types, and literals.

7. Constituent Elements

   Chapter 7 describes query expressions, query specifications, predicates, value expressions, set functions, and other elements that comprise SQL.

8. Scalar Functions

Chapter 8 describes the functions, specification formats, and rules of scalar functions.

A. SQL Reverse Lookup Reference

Appendix A explains SQL syntax organized by where it is used. This appendix provides a reverse lookup reference by which you can determine the SQL construct to use based on what you want to achieve.

B. List of Functions

Appendix B lists the functions supported by HADB and where each is used.

■ Related publications

This manual is part of a related set of manuals. The manuals in the set are listed below (with the manual numbers):

- *Hitachi Advanced Database Setup and Operation Guide* (3000-6-501(E))
- *Hitachi Advanced Database Application Development Guide* (3000-6-502(E))
- *Hitachi Advanced Database Command Reference* (3000-6-503(E))
- *Hitachi Advanced Database Messages* (3000-6-505(E))
- *HA Monitor Cluster Software Guide (for Linux$^{(R)}$ (x86) Systems)* (3000-9-201(E))
- *Job Management Partner 1 Version 10 Job Management Partner 1/Automatic Job Management System 3 System Design (Work Tasks) Guide* (3021-3-320(E))
- *JP1 Version 11 JP1/Base User's Guide* (3021-3-A01(E))

In references to Hitachi Advanced Database manuals, this manual uses *HADB* in place of *Hitachi Advanced Database*.

Example: *HADB Setup and Operation Guide*

In references to the HA Monitor manual, this manual uses *HA Monitor for Linux$^{(R)}$ (x86)* in place of *HA Monitor Cluster Software Guide (for Linux$^{(R)}$ (x86) Systems)*.

Example: *HA Monitor for Linux$^{(R)}$ (x86)*

In references to the Job Management Partner 1/Automatic Job Management System 3 manual, this manual uses *Job Management Partner 1/Automatic Job Management System 3 System Design (Work Tasks) Guide* in place of *Job Management Partner 1 Version 10 Job Management Partner 1/Automatic Job Management System 3 System Design (Work Tasks) Guide*.

Example: *Job Management Partner 1/Automatic Job Management System 3 System Design (Work Tasks) Guide*

In references to the JP1/Base manual, this manual uses *JP1/Base User's Guide* in place of *JP1 Version 11 JP1/Base User's Guide*.

Example: *JP1/Base User's Guide*

■ Conventions: Abbreviations for product names

This manual uses the following abbreviations for product names:

Abbreviation		Full name or meaning
HADB	HADB server	Hitachi Advanced Database
	HADB client	Hitachi Advanced Database Client
Linux	Linux	Linux$^{(R)}$
	Red Hat Enterprise Linux Server 6	Red Hat$^{(R)}$ Enterprise Linux$^{(R)}$ Server 6 (64-bit x86_64)
	Red Hat Enterprise Linux Server 6 (64-bit x86_64)	
	Red Hat Enterprise Linux Server 7	Red Hat$^{(R)}$ Enterprise Linux$^{(R)}$ Server 7 (64-bit x86_64)
	Red Hat Enterprise Linux Server 7 (64-bit x86_64)	
HDLM		Hitachi Dynamic Link Manager Software
JP1/AJS3		Job Management Partner 1/Automatic Job Management System 3
JP1/Audit		JP1/Audit Management - Manager
Red Hat Enterprise Linux Server 6 (64-bit x86_64)		Red Hat$^{(R)}$ Enterprise Linux$^{(R)}$ Server 6 (64-bit x86_64)
Red Hat Enterprise Linux Server 7 (64-bit x86_64)		Red Hat$^{(R)}$ Enterprise Linux$^{(R)}$ Server 7 (64-bit x86_64)

■ Conventions: Acronyms

This manual also uses the following acronyms:

Acronym	Full name or meaning
APD	Application Parameter Descriptor
API	Application Programming Interface
ARD	Application Row Descriptor
BI	Business Intelligence
BLOB	Binary Large Object
BNF	Backus-Naur Form
BOM	Byte Order Mark
CLI	Call Level Interface
CLOB	Character Large Object
CPU	Central Processing Unit
CSV	Character-Separated Values
DB	Database
DBMS	Database Management System
DMMP	Device Mapper Multipath
DNS	Domain Name System

Acronym	Full name or meaning
ER	Entity Relationship
HBA	Host Bus Adapter
ID	Identification number
IEF	Integrity Enhancement Facility
IP	Internet Protocol
IPD	Implementation Parameter Descriptor
IRD	Implementation Row Descriptor
JAR	Java Archive File
JDBC	Java Database Connectivity
JDK	Java Developer's Kit
JNDI	Java Naming and Directory Interface
JRE	Java Runtime Environment
JTA	Java Transaction API
LOB	Large Object
LRU	Least Recently Used
LV	Logical Volume
LVM	Logical Volume Manager
MSDN	Microsoft Developer Network
NFS	Network File System
NIC	Network Interface Card
NTP	Network Time Protocol
ODBC	Open Database Connectivity
OS	Operating System
PP	Program Product
RAID	Redundant Array of Independent Disks
RDBMS	Relational Database Management System
TLB	Translation Lookaside Buffer
URL	Uniform Resource Locator
VG	Volume Group
WWN	World Wide Name

■ 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.`

The table below shows the symbols used in this manual for explaining commands and operands, such as the operands used in server definitions.

Note that these symbols are used for explanatory purposes only; do not specify them in the actual operand or command.

Symbol	Meaning	Example
|	In syntax explanations, a vertical bar separates multiple items, and has the meaning of OR.	`adb_sql_text_out = {Y|N}` In this example, the vertical bar means that you can specify either `Y` or `N`.
[ ]	In syntax explanations, square brackets indicate that the enclosed item or items are optional.	`adbsql [-V]` In this example, the square brackets mean that you can specify `adbsql`, or you can specify `adbsql -V`.
{ }	In syntax explanations, curly brackets indicate that only one of the enclosed items is to be selected.	`adbcancel {--ALL|-u` *connection-ID*`}` In this example, the curly brackets mean that you can specify either `--ALL` or `-u` *connection-ID*.
...	In syntax explanations, an ellipsis (`...`) indicates that the immediately preceding item can be repeated as many times as necessary.	`adbbuff -n` *DB-area-name*`[,` *DB-area-name*`] ...` In this example, the ellipsis means that you can specify *DB-area-name* as many times as necessary.
{{ }}	In syntax explanations, double curly brackets indicate that the enclosed items can be repeated as a single unit.	`{{adbinitdbarea -n` *data-DB-area-name*`}}` In this example, the double curly brackets mean that you can specify `adbinitdbarea -n` *data-DB-area-name* as many times as necessary.
X (underline)	In syntax explanations, underlined characters indicate a default value.	`adb_import_errmsg_lv = {0|1}` In this example, the underline means that the value `0` is assumed by HADB when the operand is omitted.

Symbol	Meaning	Example
~	A swung dash indicates that the text following it explains the properties of the specified value.	`adb_sys_max_users` = *maximum-number-of-concurrent-connections* ~ <integer> ((`1` to `1024`)) <<`10`>> In this example, the text following the swung dash means that you can specify an integer in the range from `1` to `1024`. If the operand is not specified, the value `10` is assumed by HADB.
< >	Single angle brackets explain the data type of the specified value.	
(( ))	Double parentheses indicate the scope of the specified value.	
<< >>	Double angle brackets indicate a default value.	

■ Conventions: Path names

- `$INSTDIR` is used to indicate the server directory path (for installation).
- `$ADBDIR` is used to indicate the server directory path (for operation).
- `$DBDIR` is used to indicate the DB directory path.
- `%ADBCLTDIR%` (for a Windows HADB client) or `$ADBCLTDIR` (for a Linux HADB client) is used to indicate the client directory path.
- `%ADBODBTRCPATH%` is used to indicate the folder path where HADB's ODBC driver trace files are stored.

■ Conventions: Symbols used in mathematical formulas

The following table explains special symbols used by this manual in mathematical formulas:

Symbol	Meaning
↑↑	Round up the result to the next integer. Example: The result of ↑34 ÷ 3↑ is 12.
↓↓	Discard digits following the decimal point. Example: The result of ↓34 ÷ 3↓ is 11.
MAX	Select the largest value as the result. Example: The result of MAX(3 × 6, 4 + 7) is 18.
MIN	Select the smallest value as the result. Example: The result of MIN(3 × 6, 4 + 7) is 11.

■ Conventions: Syntax elements

Syntax element notation	Meaning
<path name>	The following characters can be used in path names: • In Linux Alphanumeric characters, hash mark (`#`), hyphen (`-`), forward slash (`/`), at mark (`@`), and underscore (`_`) • In Windows Alphanumeric characters, hash mark (`#`), hyphen (`-`), forward slash (`/`), at mark (`@`), underscore (`_`), backslash (`\`), and colon (`:`) Note, however, that the characters that can be used might differ depending on the operating system.

Syntax element notation	Meaning
<OS path name>	For an OS path name, all characters that can be used in a path name in the operating system can be used. For details about available characters, see the documentation for the operating system you are using.
<character string>	Any character string can be specified.

<integer suffixed by the unit>	Specify the value in a format consisting of a numeric character (in the range from `0` to `9`) followed by a unit (`MB` (megabyte), `GB` (gigabyte), or `TB` (terabyte)). Do not enter a space between the numeric character and the unit.
	• Examples of correct specification
	`1024MB`
	`512GB`
	`32TB`
	• Example of specification that causes an error
	`512 GB`

■ Conventions: KB, MB, GB, TB, PB, and EB

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.
- 1 PB (petabyte) is $1{,}024^5$ bytes.
- 1 EB (exabyte) is $1{,}024^6$ 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*.

■ HADB database language acknowledgements

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

- JIS X 3005 Family of Standards: Information Technology - Database Languages - SQL
- ISO/IEC 9075: Information Technology - Database Languages - SQL

Note:

JIS: Japanese Industrial Standard
ISO: International Organization for Standardization
IEC: International Electrotechnical Commission

Contents

1 SELECT Statement Examples

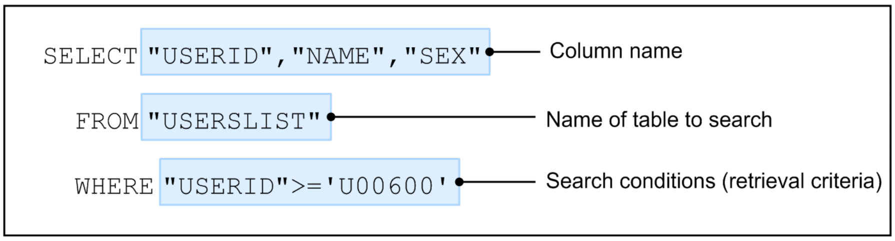
This chapter explains, through the use of examples, how to write `SELECT` statements.

Section 1.1 explains the basics of writing a `SELECT` statement. The remaining sections, starting with section 1.2, give examples illustrating how to write `SELECT` statements.

1.1 Basic syntax and rules for writing SELECT statements

This section describes the basic syntax and rules for writing a SELECT statement.

1.1.1 Basic syntax for writing a SELECT statement



```
SELECT "USERID","NAME","SEX"
  FROM "USERSLIST"
  WHERE "USERID">='U00600'
```

Column name:

Specify the column from which search results are retrieved (the column to display). Multiple column names can be specified.

Name of table to search:

In the FROM clause, specify the table to be searched. Multiple table names can be specified.

Search conditions (retrieval criteria):

In the WHERE clause, specify the search conditions to narrow down the retrieval data. You can use AND and OR to connect multiple search conditions in a WHERE clause.

Example: WHERE "USERID">='U00600' AND "SEX"='M'

If you execute the following SELECT statement, the retrieved results will be as shown below.

```
SELECT "USERID","NAME"
    FROM "USERSLIST"
        WHERE "USERID">='U00600'
```

- Configuration of table to be searched (customer table USERSLIST)

User ID	Customer name	Sex
(USERID)	(NAME)	(SEX)
U00555	Mike Johnson	M
U00358	Nancy White	F
U00212	Maria Gomez	F
U00687	Taro Tanaka	M
U00869	Bob Clinton	M

Note: Column names are shown in parentheses.

- Retrieval results

Customer ID	Customer name
(USERID)	(NAME)
U00687	Taro Tanaka
U00869	Bob Clinton

Retrieves data for customers whose USERID is U00600 or higher.

Note

For details about the syntax of search conditions, FROM clauses, and WHERE clauses, see the following:

- FROM clauses: 7.5.1 Specification format and rules for FROM clauses
- WHERE clauses: 7.6.1 Specification format for WHERE clauses
- Search conditions: 7.18.1 Specification format and rules for search conditions

1.1.2 Basic rules for writing a SELECT statement

The basic rules for writing a SELECT statement are as follows:

- We recommend that table names and column names specified in the SELECT statement be enclosed in double quotation marks ("). Enclosing a table or column name in double quotation marks allows you to specify the same name as an SQL reserved word, and eliminates the need to rewrite the SQL statement if a reserved word with that same name is added in the future.

  In addition, if a name is not enclosed in double quotation marks, any lowercase letters are treated as uppercase. For example, if you specify name, it is treated as NAME.
- Enclose CHAR type and VARCHAR type character string data in single quotation marks (').

  Example: WHERE "NAME"='Taro Tanaka'
- For type DATE data, enter dates in the following manner.

  Example 1: WHERE "PUR-DATE">=DATE'2011-09-06'

  Example 2: WHERE "PUR-DATE">=DATE'2011/09/06'

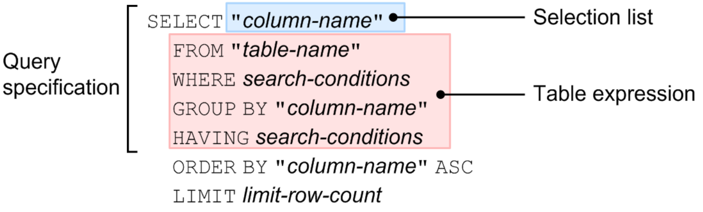
  The date format in Example 1 is used in the examples of SELECT statements given in this chapter.
- INTEGER type numeric data is not enclosed in single quotation marks (').

  Example: WHERE "PUR-NUM"=10

1.1.3 Relationship between SELECT statement syntax and its constituent elements

This subsection describes how the syntax of the SELECT statement is broken down into its constituent elements. The following figure shows the relationship between the SELECT statement syntax and its constituent elements.

Figure 1-1: Relationship between SELECT statement syntax and its constituent elements



```
SELECT "column-name"
  FROM "table-name"
  WHERE search-conditions
  GROUP BY "column-name"
  HAVING search-conditions
  ORDER BY "column-name" ASC
  LIMIT limit-row-count
```

The following describes each constituent element.

Query specification:

The *query specification* is the part of the statement that specifies the search conditions, the table to be searched, and the column from which retrieval results are to be extracted.

Selection list:

The *selection list* specifies items to be extracted as retrieval results. It is typically a column name, but set functions can also be specified.

Table expression:

The `FROM` clause, `WHERE` clause, `GROUP BY` clause, and `HAVING` clause are referred to collectively as *table expressions*.

ORDER BY clause

Specify this when you want the retrieval results sorted in ascending or descending order. For examples, see 1.3 Sorting retrieval results (ORDER BY clause).

LIMIT clause

Specify this when you want to set an upper limit on the number of rows in the retrieval results. For examples, see 1.4 Specifying the maximum number of rows of retrieval results (LIMIT clause).

Note

For details about the syntax of a query specification, selection list, or table expression, see the following.

- Query specification: 7.2.1 Specification format and rules for query specifications
- Selection list: (c) Selection list in (2) Explanation of specification format in 7.2.1 Specification format and rules for query specifications
- Table expression: 7.4.1 Specification format and rules for table expressions

1.1.4 Notes on reading sections 1.2 through the end of the chapter

- The remaining sections in this chapter give examples of how to write `SELECT` statements. Where multiple examples are presented, we start with a basic example and then progress through applied examples.
- For readability considerations, the order of rows in the retrieval results in our examples might differ from the order of rows in actual retrieval results.

1.2 Retrieving all the rows from a table

1.2.1 Example: Retrieve customer information for all customers

Retrieve all rows from the customer table (USERSLIST) and display the results. The customer table consists of columns for customer ID (USERID), name (NAME), and sex (SEX).

Table to search

■ USERSLIST

USERID	NAME	SEX
U00555	Mike Johnson	M
U00358	Nancy White	F
U00212	Maria Gomez	F
U00687	Taro Tanaka	M
U00869	Bob Clinton	M

Specification example

```
SELECT "USERID","NAME","SEX"
    FROM "USERSLIST"
```

Retrieval results

USERID	NAME	SEX
U00555	Mike Johnson	M
U00358	Nancy White	F
U00212	Maria Gomez	F
U00687	Taro Tanaka	M
U00869	Bob Clinton	M

Note

To retrieve all columns of a table, you can specify an asterisk (*) instead of the column names. The following is an example.

Specification example

```
SELECT * FROM "USERSLIST"
```

Retrieval results

USERID	NAME	SEX
U00555	Mike Johnson	M
U00358	Nancy White	F
U00212	Maria Gomez	F
U00687	Taro Tanaka	M
U00869	Bob Clinton	M

1.3 Sorting retrieval results (ORDER BY clause)

Use the ORDER BY clause to sort retrieval results in ascending or descending order. The specification format of the ORDER BY clause is as follows.

Specification format

```
SELECT "column-name" FROM "table-name"
    WHERE search-condition
    ORDER BY "column-name" ASC
```

ORDER BY "*column-name*" ASC:

Specify the column to be sorted on in *column-name*. Specify ASC to sort the retrieval results in ascending order, or DESC to sort them in descending order.

> **Note**
>
> You can also specify a sort key that is not a column name in the ORDER BY clause. For details about the syntax of the ORDER BY clause, see 7.24 Sort specification list.

1.3.1 Example 1: Sort retrieval results by customer ID

Sort all of the data in the customer table (USERSLIST) by customer ID (USERID). The customer table consists of columns for customer ID (USERID), name (NAME), and sex (SEX).

Table to search

■ USERSLIST

USERID	NAME	SEX
U00555	Mike Johnson	M
U00358	Nancy White	F
U00212	Maria Gomez	F
U00687	Taro Tanaka	M
U00869	Bob Clinton	M

Specification example

```
SELECT "USERID","NAME","SEX"
    FROM "USERSLIST"
    ORDER BY "USERID" ASC
```

Retrieval results

USERID	NAME	SEX
U00212	Maria Gomez	F
U00358	Nancy White	F
U00555	Mike Johnson	M
U00687	Taro Tanaka	M
U00869	Bob Clinton	M

Retrieval results are sorted by customer ID.

Note

The name of the column to be sorted on is specified in the ORDER BY clause. In this example, we are sorting by customer ID, so we specify USERID in the ORDER BY clause.

1.3.2 Example 2: Sort retrieval results by date of purchase and customer ID

Sort all of the data in the sales history table (SALESLIST) by date of purchase (PUR-DATE). In cases where the date of purchase is the same, order by customer ID (USERID). The sales history table consists of columns for customer ID (USERID), product code (PUR-CODE), quantity purchased (PUR-NUM), and date of purchase (PUR-DATE).

Table to search

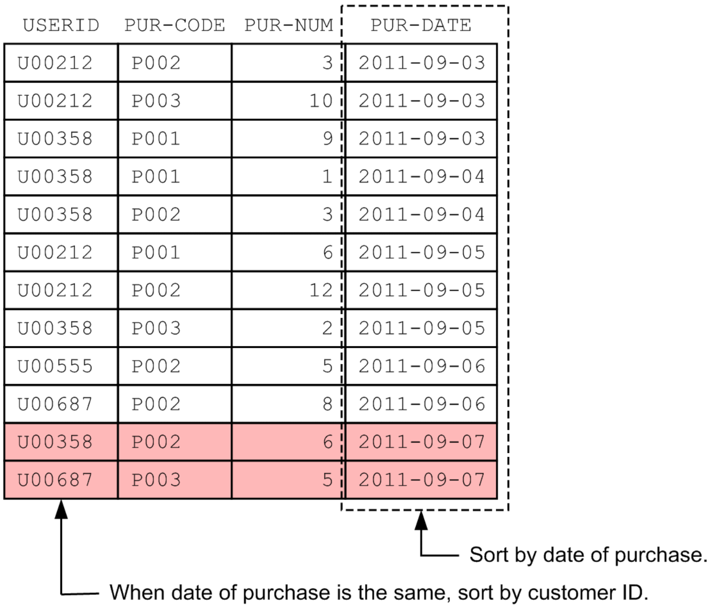
■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00358	P001	1	2011-09-04
U00555	P002	5	2011-09-06
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00358	P002	6	2011-09-07
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07
U00212	P001	6	2011-09-05
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04

Specification example

```
SELECT "USERID","PUR-CODE","PUR-NUM","PUR-DATE"
    FROM "SALESLIST"
    ORDER BY "PUR-DATE" ASC,"USERID" ASC
```

Retrieval results



USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00212	P003	10	2011-09-03
U00358	P001	9	2011-09-03
U00358	P001	1	2011-09-04
U00358	P002	3	2011-09-04
U00212	P001	6	2011-09-05
U00212	P002	12	2011-09-05
U00358	P003	2	2011-09-05
U00555	P002	5	2011-09-06
U00687	P002	8	2011-09-06
U00358	P002	6	2011-09-07
U00687	P003	5	2011-09-07

Sort by date of purchase.

When date of purchase is the same, sort by customer ID.

Note

You can specify multiple columns in the `ORDER BY` clause. The column that is specified first is given highest priority in the ordering. In this example, results are first ordered by date of purchase (`PUR-DATE`), and then ordered by customer ID (`USERID`) in cases where the date of purchase is the same.

1.4 Specifying the maximum number of rows of retrieval results (LIMIT clause)

Use the LIMIT clause to specify the maximum number of rows in the retrieval results. The specification format of the LIMIT clause is as follows.

Specification format

```
SELECT "column-name" FROM "table-name"
    WHERE search-condition
    LIMIT row-count
```

LIMIT *row-count*:

Specify the maximum number of rows allowed in the retrieval results in row-count.

Note

In addition to the maximum number of rows to be returned (row-count), you can also specify in the LIMIT clause the offset of the first row to be returned (offset). The offset option will be omitted in these examples. For details about the syntax of the LIMIT clause, see 7.9.1 Specification format and rules for LIMIT clauses.

1.4.1 Example: Specify the maximum number of rows in the retrieval results

Search the sales history table (SALESLIST) and display the top three results ordered by quantity purchased (PUR-NUM).

Table to search

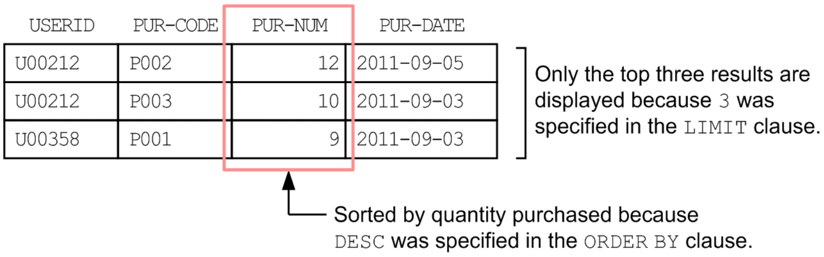
■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00358	P001	1	2011-09-04
U00555	P002	5	2011-09-06
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00358	P002	6	2011-09-07
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07
U00212	P001	6	2011-09-05
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04

Specification example

```
SELECT "USERID","PUR-CODE","PUR-NUM","PUR-DATE"
    FROM "SALESLIST"
    ORDER BY "PUR-NUM" DESC
    LIMIT 3
```

Retrieval results



USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	12	2011-09-05
U00212	P003	10	2011-09-03
U00358	P001	9	2011-09-03

Only the top three results are displayed because 3 was specified in the LIMIT clause.

Sorted by quantity purchased because DESC was specified in the ORDER BY clause.

1.5 Retrieving data with search conditions specified

Specify search conditions (retrieval criteria) in the WHERE clause to narrow down the rows to be retrieved. The specification format of the WHERE clause is as follows.

Specification format

- To specify only one search condition

```
SELECT "column-name" FROM "table-name"
    WHERE search-condition
```

- To specify two or more search conditions

```
SELECT "column-name" FROM "table-name"
    WHERE search-condition-1 AND search-condition-2 ...
```

or

```
SELECT "column-name" FROM "table-name"
    WHERE search-condition-1 OR search-condition-2 ...
```

To specify multiple search conditions in the WHERE clause, connect them using AND or OR. You can specify a mix of ANDs and ORs.

WHERE *search-condition-1* AND *search-condition-2*:

Rows that satisfy both *search-condition-1* and *search-condition-2* will be retrieved.

WHERE *search-condition-1* OR *search-condition-2*:

Rows that satisfy either *search-condition-1* or *search-condition-2* will be retrieved.

Note

For details about the syntax of WHERE clauses or search conditions, see the following.

- WHERE clause: 7.6.1 Specification format for WHERE clauses
- Search conditions: 7.18.1 Specification format and rules for search conditions

1.5.1 Example 1: Retrieve data conditioned on date of purchase

Retrieve the customer ID (USERID), product code (PUR-CODE), and date of purchase (PUR-DATE) from the sales history table (SALESLIST) for customers who purchased a product on September 6, 2011 or later.

Table to search

■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00358	P001	1	2011-09-04
U00555	P002	5	2011-09-06
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00358	P002	6	2011-09-07
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07
U00212	P001	6	2011-09-05
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04

Specification example

```
SELECT "USERID","PUR-CODE","PUR-DATE"
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2011-09-06'
```

Retrieval results

USERID	PUR-CODE	PUR-DATE
U00555	P002	2011-09-06
U00687	P002	2011-09-06
U00358	P002	2011-09-07
U00687	P003	2011-09-07

Retrieves data for September 6, 2011 or later.

Note

- When specifying search conditions in the `WHERE` clause, you can use the comparison operators listed below. The following table lists the comparison operators and their meanings.

  Table 1-1: Comparison operators and their meanings

No.	Comparison operator	Meaning
1	=	equal to
2	<>, !=, or ^=	not equal to
3	<	less than
4	<=	less than or equal to
5	>	greater than
6	>=	greater than or equal to

- If the value specified in the conditional expression is a `CHAR` type or `VARCHAR` type character string, enclose the value in single quotation marks (`'`).

  Example: `WHERE "NAME"='Taro Tanaka'`

- If the value specified in the conditional expression is a date of type `DATE`, specify it in the following manner.
  Example: `WHERE "PUR-DATE">=DATE'2011-09-06'`

1.5.2 Example 2: Retrieve data conditioned on date of purchase and product code

Retrieve the customer ID (`USERID`), product code (`PUR-CODE`), and date of purchase (`PUR-DATE`) from the sales history table (`SALESLIST`) for customers who purchased a product whose product code is P002 on September 6, 2011 or later.

Table to search

■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00358	P001	1	2011-09-04
U00555	P002	5	2011-09-06
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00358	P002	6	2011-09-07
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07
U00212	P001	6	2011-09-05
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04

Specification example

```
SELECT "USERID","PUR-CODE","PUR-DATE"
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2011-09-06'
        AND "PUR-CODE"='P002'
```

Retrieval results

USERID	PUR-CODE	PUR-DATE
U00555	P002	2011-09-06
U00687	P002	2011-09-06
U00358	P002	2011-09-07

Retrieves data for September 6, 2011 or later.

Retrieves data for product code `P002`.

Note

The `WHERE` clause specifies the following two search conditions connected by `AND`.

- Purchase of a product on or after September 6, 2011

- Purchase of a product whose product code is P002

1.5.3 Example 3: Retrieve data conditioned on date of purchase and two product codes

Retrieve the customer ID (`USERID`), product code (`PUR-CODE`), and date of purchase (`PUR-DATE`) from the sales history table (`SALESLIST`) for customers who purchased a product whose product code is P001 or P003 on September 4, 2011 or later.

Table to search

■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00358	P001	1	2011-09-04
U00555	P002	5	2011-09-06
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00358	P002	6	2011-09-07
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07
U00212	P001	6	2011-09-05
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04

Specification example

```
SELECT "USERID","PUR-CODE","PUR-DATE"
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2011-09-04'
        AND ("PUR-CODE"='P001' OR "PUR-CODE"='P003')
```

Retrieval results

USERID	PUR-CODE	PUR-DATE
U00358	P001	2011-09-04
U00358	P003	2011-09-05
U00212	P001	2011-09-05
U00687	P003	2011-09-07

Retrieves data for September 4, 2011 or later.

Retrieves data for product code `P001` or `P003`.

Note

If both `AND` and `OR` are specified, `AND` is evaluated first. To change the priority of evaluation, specify `( )` as in the specification example above.

1.6 Retrieving data with a search range specified (BETWEEN predicate)

The `BETWEEN` predicate is used to specify a search range. The specification format of the `BETWEEN` predicate is as follows.

Specification format

```
SELECT "column-name" FROM "table-name"
    WHERE "column-name" BETWEEN value-1 AND value-2
```

column-name:

Specify the column that is being narrowed down by the search range.

`BETWEEN` *value-1* `AND` *value-2***:**

Specify the lower limit of the search range in *value-1*. Specify the upper limit of the search range in *value-2*.

```
Example: WHERE C1 BETWEEN 10 AND 20
```

In this example, the search range includes rows where the value of column `C1` is between `10` and `20` (including both `10` and `20`).

Note

For details about the syntax of the `BETWEEN` predicate, see 7.19.1 BETWEEN predicate.

1.6.1 Example 1: Retrieve customers who purchased products during a period

Retrieve the customer ID (`USERID`), product code (`PUR-CODE`), and date of purchase (`PUR-DATE`) from the sales history table (`SALESLIST`) for customers who purchased products between September 4, 2011 and September 5, 2011.

Table to search

■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00358	P001	1	2011-09-04
U00555	P002	5	2011-09-06
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00358	P002	6	2011-09-07
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07
U00212	P001	6	2011-09-05
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04

Specification example

```
SELECT "USERID","PUR-CODE","PUR-DATE"
    FROM "SALESLIST"
       WHERE "PUR-DATE" BETWEEN DATE'2011-09-04' AND DATE'2011-09-05'
```

Retrieval results

USERID	PUR-CODE	PUR-DATE
U00358	P001	2011-09-04
U00358	P002	2011-09-04
U00212	P002	2011-09-05
U00212	P001	2011-09-05
U00358	P003	2011-09-05

Note

A BETWEEN predicate could instead be rewritten using AND conditions. For example, the SELECT statement below, which uses an AND condition, gives the same retrieval results as the specification example above, which uses a BETWEEN predicate.

For details about AND conditions, see 1.5 Retrieving data with search conditions specified.

```
SELECT "USERID","PUR-CODE","PUR-DATE"
    FROM "SALESLIST"
       WHERE "PUR-DATE">=DATE'2011-09-04'
       AND "PUR-DATE"<=DATE'2011-09-05'
```

1.6.2 Example 2: Retrieve customers who purchased products outside of a period

Retrieve the customer ID (USERID), product code (PUR-CODE), and date of purchase (PUR-DATE) from the sales history table (SALESLIST) for customers who purchased products outside of the period September 4, 2011 and September 5, 2011.

Table to search

■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00358	P001	1	2011-09-04
U00555	P002	5	2011-09-06
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00358	P002	6	2011-09-07
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07
U00212	P001	6	2011-09-05
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04

Specification example

```
SELECT "USERID","PUR-CODE","PUR-DATE"
    FROM "SALESLIST"
       WHERE "PUR-DATE" NOT BETWEEN DATE'2011-09-04' AND DATE'2011-09-05'
```

Retrieval results

USERID	PUR-CODE	PUR-DATE
U00212	P002	2011-09-03
U00212	P003	2011-09-03
U00358	P001	2011-09-03
U00555	P002	2011-09-06
U00687	P002	2011-09-06
U00687	P003	2011-09-07
U00358	P002	2011-09-07

Note

If NOT is specified, the search will target values that do not satisfy the conditional expression immediately following the NOT. If you specify NOT BETWEEN DATE'2011-09-04' AND DATE'2011-09-05', as in the specification example above, the retrieval criteria will exclude September 4, 2011 through September 5, 2011.

1.7 Retrieving data that meets one of multiple conditions (IN predicate)

Use the `IN` predicate if you want to specify multiple conditions (values) and retrieve data that match any one of them. The specification format of the `IN` predicate is as follows.

Specification format

```
SELECT "column-name" FROM "table-name"
    WHERE "column-name" IN (value-1,value-2,...)
```

column-name:

Specify the column to use for narrowing down the retrieval.

`IN` (*value-1*, *value-2*, ...):

Specify the values to be retrieved. Rows that match any of the values specified here will be retrieved.

Note

For details about the syntax of the `IN` predicate, see 7.19.3 IN predicate.

1.7.1 Example 1: Retrieve customers who purchased product code P001 or P003

Retrieve the customer ID (`USERID`), product code (`PUR-CODE`), and date of purchase (`PUR-DATE`) from the sales history table (`SALESLIST`) of customers who purchased products with product code P001 or P003 on or after September 5, 2011.

Table to search

■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00358	P001	1	2011-09-04
U00555	P002	5	2011-09-06
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00358	P002	6	2011-09-07
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07
U00212	P001	6	2011-09-05
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04

Specification example

```
SELECT "USERID","PUR-CODE","PUR-DATE"
    FROM "SALESLIST"
       WHERE "PUR-CODE" IN ('P001','P003')
       AND "PUR-DATE">=DATE'2011-09-05'
```

Retrieval results

USERID	PUR-CODE	PUR-DATE
U00212	P001	2011-09-05
U00358	P003	2011-09-05
U00687	P003	2011-09-07

Note

An IN predicate could be rewritten using OR conditions. For example, the SELECT statement below, which uses an OR condition, gives the same retrieval results as the specification example above, which uses an IN predicate.

For details about OR conditions, see 1.5 Retrieving data with search conditions specified.

```
SELECT "USERID","PUR-CODE","PUR-DATE"
    FROM "SALESLIST"
       WHERE ("PUR-CODE"='P001' OR "PUR-CODE"='P003')
       AND "PUR-DATE">=DATE'2011-09-05'
```

1.7.2 Example 2: Retrieve customers who purchased products except for a specific customer

Retrieve the customer ID (USERID), product code (PUR-CODE), and quantity purchased (PUR-NUM) from the sales history table (SALESLIST). At this time, skip retrieval for customers whose customer IDs (USERID) are U00212 and U00358.

Table to search

■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00358	P001	1	2011-09-04
U00555	P002	5	2011-09-06
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00358	P002	6	2011-09-07
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07
U00212	P001	6	2011-09-05
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04

Specification example

```
SELECT "USERID","PUR-CODE","PUR-NUM"
    FROM "SALESLIST"
       WHERE "USERID" NOT IN ('U00212','U00358')
```

Retrieval results

USERID	PUR-CODE	PUR-NUM
U00555	P002	5
U00687	P002	8
U00687	P003	5

Note

If `NOT` is specified, the search will return values that do not satisfy the conditional expression immediately following the `NOT`. If you specify `NOT IN ('U00212','U00358')`, as in the specification example above, the retrieval criteria will exclude customer IDs U00212 and U00358.

1.8 Retrieving data that contains a specific character string (LIKE predicate)

Use the `LIKE` predicate to retrieve data that contains a specific character string. The specification format of the `LIKE` predicate is as follows.

Specification format

```
SELECT "column-name" FROM "table-name"
    WHERE "column-name" LIKE 'pattern-character-string'
```

column name:

Specify the column to use for narrowing down the retrieval.

> **Note**
>
> You can also specify expressions other than column names. For details about the syntax of the `LIKE` predicate, see 7.19.4 LIKE predicate.

`LIKE '`*pattern-character-string*`'`:

Specify the pattern character string to search for. The main pattern character strings (wildcards) are the following:

- `%`
  This denotes any character string of zero or more characters. If you specify `'ACT%'`, it will match the character strings such as `ACT`, `ACTOR`, and `ACTION`.
- _ (underscore)
  This denotes any single character. If you specify `'_I_'`, it will match the character strings such as `BIT`, `HIT`, and `KIT`.

> **Note**
>
> - For details about the syntax of pattern character strings, see 7.19.4 LIKE predicate.
> - You can also specify `ESCAPE` in the `LIKE` predicate. For details, see 7.19.4 LIKE predicate.

1.8.1 Example 1: Retrieve customers whose name begins with M

Retrieve from the customer table (`USERSLIST`) the customer ID (`USERID`), name (`NAME`), and sex (`SEX`) of customers whose name begins with M.

Table to search

■ `USERSLIST`

USERID	NAME	SEX
U00555	Mike Johnson	M
U00358	Nancy White	F
U00212	Maria Gomez	F
U00687	Taro Tanaka	M
U00869	Bob Clinton	M

Specification example

```
SELECT "USERID","NAME","SEX"
    FROM "USERSLIST"
        WHERE "NAME" LIKE 'M%'
```

Retrieval results

USERID	NAME	SEX
U00212	Maria Gomez	F
U00555	Mike Johnson	M

1.8.2 Example 2: Retrieve customers whose name does not begin with M

Retrieve from the customer table (`USERSLIST`) the customer ID (`USERID`), name (`NAME`), and sex (`SEX`) of female customers whose name does not begin with M.

Table to search

■ USERSLIST

USERID	NAME	SEX
U00555	Mike Johnson	M
U00358	Nancy White	F
U00212	Maria Gomez	F
U00687	Taro Tanaka	M
U00869	Bob Clinton	M

Specification example

```
SELECT "USERID","NAME","SEX"
    FROM "USERSLIST"
        WHERE "NAME" NOT LIKE 'M%'
        AND "SEX"='F'
```

Retrieval results

USERID	NAME	SEX
U00358	Nancy White	F

Note

If `NOT` is specified, the search will return values that do not satisfy the conditional expression immediately following the `NOT`. If you specify `NOT LIKE 'M%'`, as in the specification example above, the retrieval criteria will exclude character strings that begin with M.

1.9 Retrieving data with multiple tables specified (table join)

If the data to be retrieved is distributed across multiple tables, perform the retrieval by associating columns that contain the same information. This is called a *table join*. As an example, we describe a table join of the sales history table (SALESLIST) and customer table (USERSLIST).

Example:

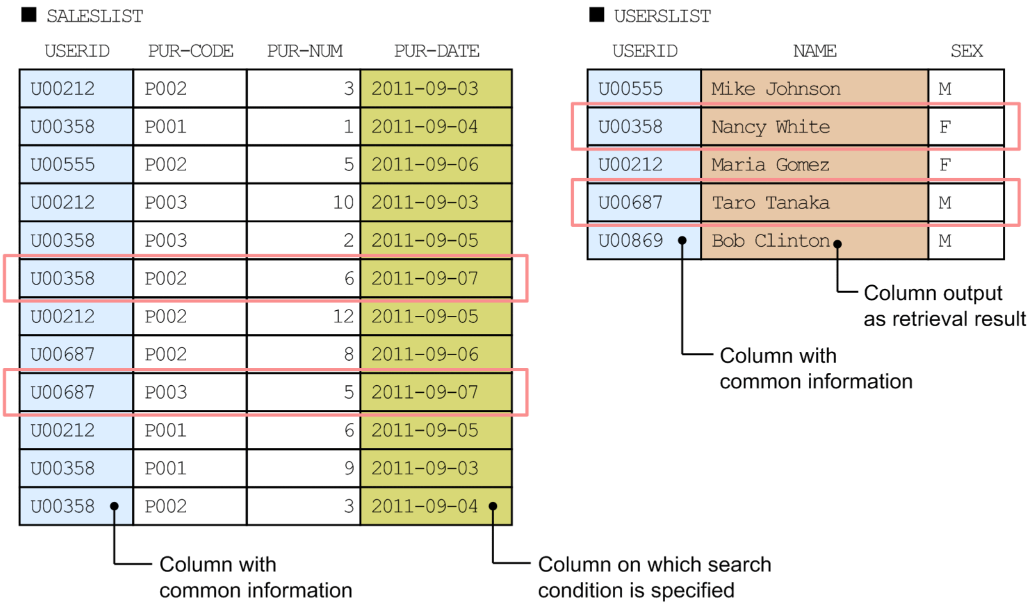
The following retrieves the name (NAME) of customers who purchased a product on September 7, 2011 from the sales history table (SALESLIST) and customer table (USERSLIST).

SELECT statement specification

```
SELECT "NAME"
    FROM "SALESLIST","USERSLIST"
        WHERE "PUR-DATE"=DATE'2011-09-07'
        AND "SALESLIST"."USERID"="USERSLIST"."USERID"
```

Description

The date of purchase (PUR-DATE) information specified in the search condition is located in the sales history table (SALESLIST), while the name (NAME) information to be output as the retrieval result is located in the customer table (USERSLIST). In this case, we join the SALESLIST and USERSLIST tables using the customer ID column (USERID), which is common to both tables.



■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00358	P001	1	2011-09-04
U00555	P002	5	2011-09-06
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00358	P002	6	2011-09-07
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07
U00212	P001	6	2011-09-05
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04

■ USERSLIST

USERID	NAME	SEX
U00555	Mike Johnson	M
U00358	Nancy White	F
U00212	Maria Gomez	F
U00687	Taro Tanaka	M
U00869	Bob Clinton	M

Column with common information
Column on which search condition is specified
Column output as retrieval result
Column with common information

Retrieval results

Nancy White
Taro Tanaka

1.9.1 Example 1: Retrieve customer purchases from the customer table and sales history table (1 of 3)

Retrieve the customer ID (USERID), name (NAME), product code (PUR-CODE), and date of purchase (PUR-DATE) of customers who purchased products on or after September 6, 2011 from the sales history table (SALESLIST) and customer table (USERSLIST).

Table to search

■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00358	P001	1	2011-09-04
U00555	P002	5	2011-09-06
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00358	P002	6	2011-09-07
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07
U00212	P001	6	2011-09-05
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04

■ USERSLIST

USERID	NAME	SEX
U00555	Mike Johnson	M
U00358	Nancy White	F
U00212	Maria Gomez	F
U00687	Taro Tanaka	M
U00869	Bob Clinton	M

Specification example

```
SELECT "SALESLIST"."USERID","NAME","PUR-CODE","PUR-DATE"
    FROM "SALESLIST","USERSLIST"
        WHERE "PUR-DATE">=DATE'2011-09-06'
        AND "SALESLIST"."USERID"="USERSLIST"."USERID"
```

Retrieval results

USERID	NAME	PUR-CODE	PUR-DATE
U00555	Mike Johnson	P002	2011-09-06
U00687	Taro Tanaka	P002	2011-09-06
U00358	Nancy White	P002	2011-09-07
U00687	Taro Tanaka	P003	2011-09-07

Note

- Note that if both tables include columns with the same name, these columns are identified by using a specification in the "*table-name*"."*column-name*" format. In this example, the USERID column applies. Therefore, the "SALESLIST"."USERID" and "USERSLIST"."USERID" specifications are used for identification.
- In the FROM clause, specify all the tables to be searched.
- Specify the conditional expression AND "SALESLIST"."USERID"="USERSLIST"."USERID" in order to join the tables based on the value of the customer ID column (USERID) as the key.

1.9.2 Example 2: Retrieve customer purchases from the customer table and sales history table (2 of 3)

Retrieve the customer ID (USERID), name (NAME), sex (SEX), product code (PUR-CODE), and date of purchase (PUR-DATE) from the sales history table (SALESLIST) and customer table (USERSLIST) for customers who meet the following condition:

- Male customers who purchased a product on or after September 6, 2011

Table to search

■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00358	P001	1	2011-09-04
U00555	P002	5	2011-09-06
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00358	P002	6	2011-09-07
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07
U00212	P001	6	2011-09-05
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04

■ USERSLIST

USERID	NAME	SEX
U00555	Mike Johnson	M
U00358	Nancy White	F
U00212	Maria Gomez	F
U00687	Taro Tanaka	M
U00869	Bob Clinton	M

Specification example

```
SELECT "SALESLIST"."USERID","NAME","SEX","PUR-CODE","PUR-DATE"
    FROM "SALESLIST","USERSLIST"
        WHERE "PUR-DATE">=DATE'2011-09-06'
        AND "SEX"='M'
        AND "SALESLIST"."USERID"="USERSLIST"."USERID"
```

Retrieval results

USERID	NAME	SEX	PUR-CODE	PUR-DATE
U00555	Mike Johnson	M	P002	2011-09-06
U00687	Taro Tanaka	M	P002	2011-09-06
U00687	Taro Tanaka	M	P003	2011-09-07

1.9.3 Example 3: Retrieve customer purchases from the customer table and sales history table (3 of 3)

Retrieve the customer ID (USERID), name (NAME), sex (SEX), product code (PUR-CODE), and date of purchase (PUR-DATE) from the sales history table (SALESLIST) and customer table (USERSLIST) for customers who meet either of the following conditions:

- Male customers who purchased products on or after September 6, 2011
- Female customers who purchased products on or after September 5, 2011

Table to search

■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00358	P001	1	2011-09-04
U00555	P002	5	2011-09-06
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00358	P002	6	2011-09-07
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07
U00212	P001	6	2011-09-05
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04

■ USERSLIST

USERID	NAME	SEX
U00555	Mike Johnson	M
U00358	Nancy White	F
U00212	Maria Gomez	F
U00687	Taro Tanaka	M
U00869	Bob Clinton	M

Specification example

```
SELECT "SALESLIST"."USERID","NAME","SEX","PUR-CODE","PUR-DATE"
    FROM "SALESLIST","USERSLIST"
        WHERE (("PUR-DATE">=DATE'2011-09-06' AND "SEX"='M')
        OR ("PUR-DATE">=DATE'2011-09-05' AND "SEX"='F'))
        AND "SALESLIST"."USERID"="USERSLIST"."USERID"
```

Retrieval results

USERID	NAME	SEX	PUR-CODE	PUR-DATE
U00212	Maria Gomez	F	P001	2011-09-05
U00212	Maria Gomez	F	P002	2011-09-05
U00358	Nancy White	F	P003	2011-09-05
U00555	Mike Johnson	M	P002	2011-09-06
U00687	Taro Tanaka	M	P002	2011-09-06
U00358	Nancy White	F	P002	2011-09-07
U00687	Taro Tanaka	M	P003	2011-09-07

1.10 Eliminating duplication in retrieval results (SELECT DISTINCT)

Use SELECT DISTINCT to eliminate duplication in retrieval results. The specification format of SELECT DISTINCT is as follows.

Specification format

```
SELECT DISTINCT "column-name" FROM "table-name"
    WHERE search-condition
```

DISTINCT:

Specify this if you want to eliminate duplication in retrieval results.

> **Note**
>
> For details about the syntax of SELECT DISTINCT, see 7.2.1 Specification format and rules for query specifications.

1.10.1 Example: Retrieve customers who purchased products

Retrieve from the sales history table (SALESLIST) and customer table (USERSLIST) the customer ID (USERID) and name (NAME) of customers who purchased products on September 5, 2011.

Table to search

■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00358	P001	1	2011-09-04
U00555	P002	5	2011-09-06
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00358	P002	6	2011-09-07
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07
U00212	P001	6	2011-09-05
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04

■ USERSLIST

USERID	NAME	SEX
U00555	Mike Johnson	M
U00358	Nancy White	F
U00212	Maria Gomez	F
U00687	Taro Tanaka	M
U00869	Bob Clinton	M

Specification example

```
SELECT DISTINCT "SALESLIST"."USERID","NAME"
    FROM "SALESLIST","USERSLIST"
        WHERE "PUR-DATE"=DATE'2011-09-05'
        AND "SALESLIST"."USERID"="USERSLIST"."USERID"
```

Retrieval results

USERID	NAME
U00212	Maria Gomez
U00358	Nancy White

Note

If you do not specify `SELECT DISTINCT`, the retrieval results are as follows.

Specification example

```
SELECT "SALESLIST"."USERID","NAME"
    FROM "SALESLIST","USERSLIST"
        WHERE "PUR-DATE"=DATE'2011-09-05'
        AND "SALESLIST"."USERID"="USERSLIST"."USERID"
```

Retrieval results

USERID	NAME
U00212	Maria Gomez
U00212	Maria Gomez
U00358	Nancy White

1.11 Determining the number of retrieved data items (COUNT(*))

Use the set function COUNT(*) to determine the number of retrieved data items.

Note

For details about the syntax of COUNT(*), see 7.22.3 COUNT.

1.11.1 Example 1: Determine the total number of customers

Determine the total number of customers in the customer table (USERSLIST).

Table to search

■ USERSLIST

USERID	NAME	SEX
U00555	Mike Johnson	M
U00358	Nancy White	F
U00212	Maria Gomez	F
U00687	Taro Tanaka	M
U00869	Bob Clinton	M

Specification example

```
SELECT COUNT(*)
    FROM "USERSLIST"
```

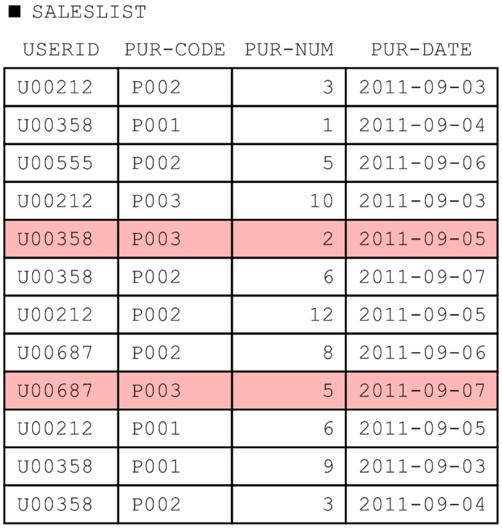
Retrieval results

COUNT(*)
5

1.11.2 Example 2: Determine the number of people who purchased a product

Determine the total number of people in the sales history table (SALESLIST) who purchased the product of product code P003 on or after September 5, 2011.

Table to search



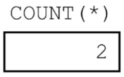
■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00358	P001	1	2011-09-04
U00555	P002	5	2011-09-06
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00358	P002	6	2011-09-07
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07
U00212	P001	6	2011-09-05
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04

Specification example

```
SELECT COUNT(*)
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2011-09-05'
        AND "PUR-CODE"='P003'
```

Retrieval results



COUNT(*)
2

1.12 Determining the maximum, minimum, average, or sum of the retrieved data (set functions)

Use the set functions MAX, MIN, AVG, and SUM to determine the maximum value, minimum value, average, or sum of the retrieved data.

Note

For details about the syntax of set functions, see 7.22 Set functions.

1.12.1 Example 1: Determine the maximum, minimum, and average quantities purchased

Determine the maximum, minimum, and average value of the quantity purchased (PUR-NUM) for product code P002 in the sales history table (SALESLIST).

Table to search

■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00358	P001	1	2011-09-04
U00555	P002	5	2011-09-06
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00358	P002	6	2011-09-07
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07
U00212	P001	6	2011-09-05
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04

Specification example

```
SELECT MAX("PUR-NUM"),MIN("PUR-NUM"),AVG("PUR-NUM")
    FROM "SALESLIST"
        WHERE "PUR-CODE"='P002'
```

Retrieval results

MAX(PUR-NUM)	MIN(PUR-NUM)	AVG(PUR-NUM)
12	3	6
Maximum value	Minimum value	Average value

1.12.2 Example 2: Determine the sum of quantities purchased)

Determine the sum of quantities purchased (PUR-NUM) on September 6, 2011 for product code P002 in the sales history table (SALESLIST).

Table to search

■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00358	P001	1	2011-09-04
U00555	P002	5	2011-09-06
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00358	P002	6	2011-09-07
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07
U00212	P001	6	2011-09-05
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04

Specification example

```
SELECT SUM("PUR-NUM")
    FROM "SALESLIST"
        WHERE "PUR-CODE"='P002'
        AND "PUR-DATE"=DATE'2011-09-06'
```

Retrieval results

SUM(PUR-NUM)
13

Sum

1.13 Aggregating retrieved data by group (GROUP BY clause, HAVING clause)

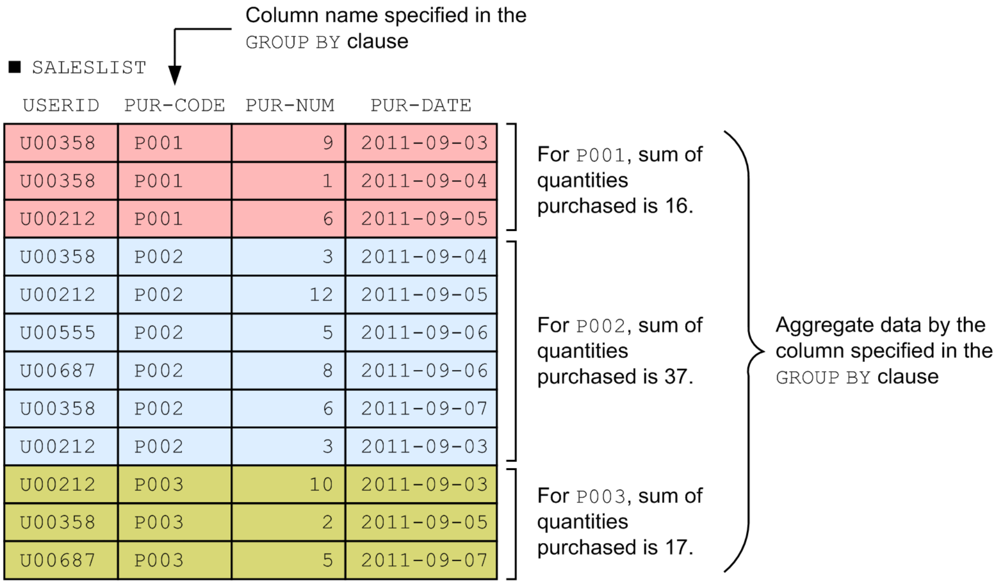
Use the GROUP BY clause to aggregate retrieved data by group. In the examples of the GROUP BY clause shown below, the sales history table (SALESLIST) is used.

Example:

The following determines the sum of the quantities purchased for each product code (PUR-CODE) in the sales history table (SALESLIST).

SELECT statement specification

```
  SELECT "PUR-CODE",SUM("PUR-NUM")
      FROM "SALESLIST"
      GROUP BY "PUR-CODE"
```



Column name specified in the GROUP BY clause

■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00358	P001	9	2011-09-03
U00358	P001	1	2011-09-04
U00212	P001	6	2011-09-05
U00358	P002	3	2011-09-04
U00212	P002	12	2011-09-05
U00555	P002	5	2011-09-06
U00687	P002	8	2011-09-06
U00358	P002	6	2011-09-07
U00212	P002	3	2011-09-03
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00687	P003	5	2011-09-07

For P001, sum of quantities purchased is 16.

For P002, sum of quantities purchased is 37.

For P003, sum of quantities purchased is 17.

Aggregate data by the column specified in the GROUP BY clause

Retrieval results

PUR-CODE	SUM(PUR-NUM)
P001	16
P002	37
P003	17

Sum of quantities purchased, by product code

The specification format of the GROUP BY clause and HAVING clause is as follows.

Specification format

```
SELECT "column-name" FROM "table-name"
    WHERE search-condition
    GROUP BY "column-name"
    HAVING search-condition
```

GROUP BY "*column-name*":

Specify the column by which the retrieved data is aggregated. For example, the following will aggregate the retrieved data by product code (PUR-CODE).

```
Example: GROUP BY "PUR-CODE"
```

`HAVING` *search-condition*:

You can specify search conditions to narrow down the retrieved data that was aggregated by groups in the `GROUP BY` clause. For a specification example, see 1.13.4 Example 4: Determine the quantity purchased for each product code (narrow down retrieval by specifying a HAVING clause).

Note

You can also specify a grouping specification that is not a column name in the `GROUP BY` clause. For details about the syntax of the `GROUP BY` clause and `HAVING` clause, see the following.

- `GROUP BY` clause: 7.7.1 Specification format and rules for GROUP BY clauses
- `HAVING` clause: 7.8.1 Specification format and rules for HAVING clauses

1.13.1 Example 1: Determine the number of purchases for each customer

Obtain from the sales history table (`SALESLIST`) a list of the number of purchases for each customer.

Table to search

■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P001	6	2011-09-05
U00212	P002	3	2011-09-03
U00212	P002	12	2011-09-05
U00212	P003	10	2011-09-03
U00358	P001	1	2011-09-04
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04
U00358	P002	6	2011-09-07
U00358	P003	2	2011-09-05
U00555	P002	5	2011-09-06
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07

Specification example

```
SELECT "USERID",COUNT(*)
    FROM "SALESLIST"
        GROUP BY "USERID"
```

Retrieval results

USERID	COUNT(*)
U00212	4
U00358	5
U00555	1
U00687	2

Note

The columns specified in the GROUP BY clause must match the columns specified between the SELECT statement and the FROM clause, or an SQL error results. In the example above, the USERID column is specified in both locations.

An example of an SQL statement that generates an error is given below.

Example of an SQL statement that generates an error

```
SELECT "USERID","PUR-CODE",COUNT(*)
    FROM "SALESLIST"
        GROUP BY "USERID"
```

Example of a correct SQL statement

```
SELECT "USERID","PUR-CODE",COUNT(*)
    FROM "SALESLIST"
        GROUP BY "USERID","PUR-CODE"
```

The SQL statement above obtains the number of purchases by customer (USERID) and product code (PUR-CODE). The retrieval results are as follows.

Retrieval results

USERID	PUR-CODE	COUNT(*)
U00212	P001	1
U00212	P002	2
U00212	P003	1
U00358	P001	2
U00358	P002	2
U00358	P003	1
U00555	P002	1
U00687	P002	1
U00687	P003	1

1.13.2 Example 2: Determine the number of sales for each product code

Determine the number of sales on or after September 5, 2011 for each product code (PUR-CODE) in the sales history table (SALESLIST).

Table to search

■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00358	P001	9	2011-09-03
U00212	P002	3	2011-09-03
U00212	P003	10	2011-09-03
U00358	P001	1	2011-09-04
U00358	P002	3	2011-09-04
U00212	P001	6	2011-09-05
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00555	P002	5	2011-09-06
U00358	P002	6	2011-09-07
U00358	P003	2	2011-09-05
U00687	P003	5	2011-09-07

Specification example

```
SELECT "PUR-CODE",COUNT(*)
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2011-09-05'
        GROUP BY "PUR-CODE"
```

Retrieval results

PUR-CODE	COUNT(PUR-CODE)
P001	1
P002	4
P003	2

1.13.3 Example 3: Determine the sum and average of the quantities purchased for each product code

Determine the sum and average of the quantities purchased on or after September 3, 2011 for each product code (PUR-CODE) in the sales history table (SALESLIST).

Table to search

■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00358	P001	9	2011-09-03
U00358	P001	1	2011-09-04
U00212	P001	6	2011-09-05
U00358	P002	3	2011-09-04
U00212	P002	12	2011-09-05
U00555	P002	5	2011-09-06
U00687	P002	8	2011-09-06
U00358	P002	6	2011-09-07
U00212	P002	3	2011-09-03
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00687	P003	5	2011-09-07

Specification example

```
SELECT "PUR-CODE",SUM("PUR-NUM"),AVG("PUR-NUM")
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2011-09-03'
        GROUP BY "PUR-CODE"
```

Retrieval results

PUR-CODE	SUM(PUR-NUM)	AVG(PUR-NUM)
P001	16	5
P002	37	6
P003	17	5
	Sums	Averages

1.13.4 Example 4: Determine the quantity purchased for each product code (narrow down retrieval by specifying a HAVING clause)

Determine the sum and average of the quantities purchased on or after September 3, 2011 for each product code (PUR-CODE) in the sales history table (SALESLIST).

In this case, we retrieve only those product codes where the quantities purchased is 20 or fewer.

Table to search

■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00358	P001	9	2011-09-03
U00358	P001	1	2011-09-04
U00212	P001	6	2011-09-05
U00358	P002	3	2011-09-04
U00212	P002	12	2011-09-05
U00555	P002	5	2011-09-06
U00687	P002	8	2011-09-06
U00358	P002	6	2011-09-07
U00212	P002	3	2011-09-03
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00687	P003	5	2011-09-07

Specification example

```
SELECT "PUR-CODE",SUM("PUR-NUM"),AVG("PUR-NUM")
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2011-09-03'
        GROUP BY "PUR-CODE"
        HAVING SUM("PUR-NUM")<=20
```

Retrieval results

PUR-CODE	SUM(PUR-NUM)	AVG(PUR-NUM)
P001	16	5
P003	17	5

Sums

Averages

Retrieves only product codes where 20 or fewer items were purchased.

1.13.5 Example 5: Aggregate data from the sales history table and customer table

From the sales history table (SALESLIST) and customer table (USERSLIST), obtain the sum by customer of the quantities purchased (PUR-NUM) on or after September 4, 2011 for product code P002.

Table to search

■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00358	P001	1	2011-09-04
U00555	P002	5	2011-09-06
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00358	P002	6	2011-09-07
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07
U00212	P001	6	2011-09-05
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04

■ USERSLIST

USERID	NAME	SEX
U00555	Mike Johnson	M
U00358	Nancy White	F
U00212	Maria Gomez	F
U00687	Taro Tanaka	M
U00869	Bob Clinton	M

Specification example

```
SELECT "NAME",SUM("PUR-NUM")
    FROM "SALESLIST","USERSLIST"
        WHERE "PUR-DATE">=DATE'2011-09-04'
        AND "PUR-CODE"='P002'
        AND "SALESLIST"."USERID"="USERSLIST"."USERID"
        GROUP BY "NAME"
```

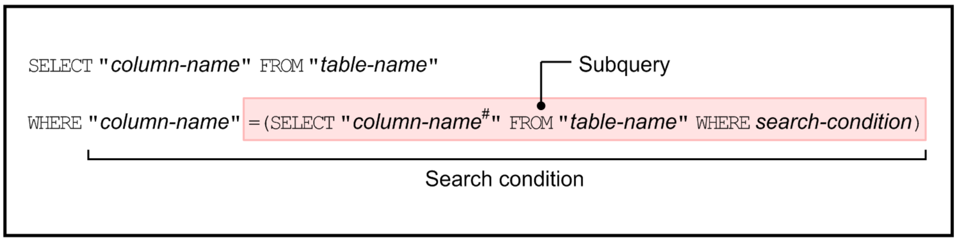
Retrieval results

NAME	SUM(PUR-NUM)
Maria Gomez	12
Nancy White	9
Mike Johnson	5
Taro Tanaka	8

1.14 Retrieving by specifying a SELECT statement in the search condition (subquery)

Subqueries can be used to retrieve data from a table by specifying a `SELECT` statement in the search condition. This provides a way to make retrieval using a `SELECT` statement more powerful and flexible, by using a search condition based on the retrieval results obtained in another `SELECT` statement. The `SELECT` statement that is specified in the search condition is called a *subquery*. The following figure shows an example of specifying a subquery:

Figure 1-2: Example of specifying a subquery



```
SELECT "column-name" FROM "table-name"
WHERE "column-name" =(SELECT "column-name#" FROM "table-name" WHERE search-condition)
```

#: You can specify a set function in addition to a column name.

Description

You can retrieve data using a search condition based on the results of a `SELECT` statement specified in a subquery.

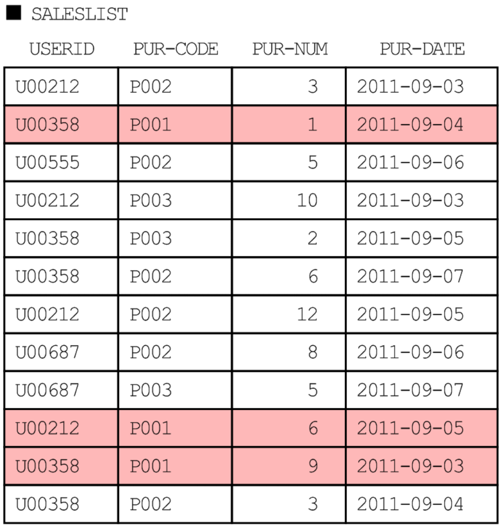
Note

For details about the syntax of subqueries, see 7.3.1 Specification format and rules for subqueries.

1.14.1 Example: Find the customer who purchased the greatest quantity of a product

From the sales history table (`SALESLIST`), find the customer ID (`USERID`) and quantity purchased (`PUR-NUM`) for the customer who purchased the greatest quantity of product code `P001`.

Table to search



■ SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2011-09-03
U00358	P001	1	2011-09-04
U00555	P002	5	2011-09-06
U00212	P003	10	2011-09-03
U00358	P003	2	2011-09-05
U00358	P002	6	2011-09-07
U00212	P002	12	2011-09-05
U00687	P002	8	2011-09-06
U00687	P003	5	2011-09-07
U00212	P001	6	2011-09-05
U00358	P001	9	2011-09-03
U00358	P002	3	2011-09-04

Specification example

```
SELECT "USERID","PUR-NUM"
    FROM "SALESLIST"
        WHERE "PUR-NUM"=(SELECT MAX("PUR-NUM") FROM "SALESLIST"
                             WHERE "PUR-CODE"='P001')
```

Retrieval results

USERID	PUR-NUM
U00358	9

Tip

The subquery specified in the underlined portion searches the sales history table (`SALESLIST`) to find the greatest quantity purchased (9) for product code `P001`.

Next, it finds the customer ID (`USERID`) and quantity purchased (`PUR-NUM`) where `PUR-NUM` equals the greatest quantity purchased (9) identified in the subquery.

1.15 Common errors in SQL statements and how to handle them

This section describes some common errors that occur when executing SQL statements, and how to handle them.

Only the most typical ways of dealing with the most common mistakes are described here; there might be cases where you cannot use the solutions described here. For such cases, follow the action recommended for the message that is output.

1.15.1 If message KFAA30104-E is displayed

Check if there is an error such as the following:

- **Character string not enclosed in single quotation marks (')**

  If a value specified in a conditional expression is a `CHAR` type or `VARCHAR` type character string, you must enclose the character string in single quotation marks (`'`).

  Example:

  ```
  SELECT "USERID" FROM "USERSLIST" WHERE "NAME"=Taro Tanaka

  KFAA30104-E There is an unnecessary part "Tanaka" in the SQL statement.
  ```

  In this example, the underlined portion of the statement is incorrect. The correct specification is `'Taro Tanaka'`.

- **Column name or table name not properly enclosed in double quotation marks (")**

  Example:

  ```
  SELECT "USERID,"PUR-CODE","PUR-DATE" FROM "SALESLIST"
      WHERE "PUR-DATE">=DATE'2011-09-06'

  KFAA30104-E There is an unnecessary part ",(0x2c)" in the SQL statement.
  ```

  In this example, the underlined portion of the statement is incorrect. There is no double quotation mark (`"`) specified after `USERID`.

- **WHERE not correctly spelled**

  Example:

  ```
  SELECT "USERID","PUR-CODE","PUR-DATE" FROM "SALESLIST"
      WHRER "PUR-DATE">=DATE'2011-09-06'

  KFAA30104-E There is an unnecessary part ""PUR-DATE">=DATE'2011-09-06'"
  in the SQL statement.
  ```

  In this example, the underlined portion of the statement is incorrect. The correct specification is `WHERE`.

1.15.2 If message KFAA30105-E is displayed

Check if there is an error such as the following:

Example:

```
SELECT "USERID","PUR-CODE","PUR-DATE" FROM "SALESLIST"
    WHERE "PUR-DATE"=>DATE'2011-09-06'
```

```
KFAA30105-E Token ">"(non-reserved word), which is after token "=", is invalid.
```

The underlined portion of the statement is incorrect. The correct specification is >=.

The syntax error occurred at the location of the character string `">"`, which is displayed after `Token` in message `KFAA30105-E`.

1.15.3 If message KFAA30119-E is displayed

Check whether the column specified immediately after the `SELECT` is different from the column specified in the `GROUP BY` clause.

Example 1:

```
SELECT "USERID",COUNT(*) FROM "SALESLIST"
    GROUP BY "PUR-CODE"

KFAA30119-E In a query using a GROUP BY clause or a set function specification,
the column "USERID" specified in a select expression, a HAVING clause
or an ORDER BY clause must be specified as an argument of the GROUP BY clause
or the set function. (query number = 1, 1)
```

In this example, the underlined portions of the statement are incorrect. Make the column names specified in the underlined portions the same.

Example 2:

```
SELECT "USERID","PUR-CODE",COUNT(*) FROM "SALESLIST"
    GROUP BY "USERID"

KFAA30119-E In a query using a GROUP BY clause or a set function specification,
the column "PUR-CODE" specified in a select expression, a HAVING clause
or an ORDER BY clause must be specified as an argument of the GROUP BY clause
or the set function. (query number = 1, 1)
```

In this example, the underlined portions of the statement are incorrect. Make the number of columns and the column names specified in the underlined portions the same.

1.15.4 If message KFAA30202-E is displayed

Check if there is an error in a specified column name.

Example:

```
SELECT "USRID","PUR-CODE","PUR-DATE" FROM "SALESLIST"
    WHERE "PUR-DATE">=DATE'2011-09-06'

KFAA30202-E Column "USRID" is not found in any table. (query number = 1)
```

In this example, the underlined portion of the statement is incorrect. The correct column name is `"USERID"`.

1.15.5 If message KFAA30203-E is displayed

In a retrieval that spans two tables, if both tables contain a column with the same name, the column name must be specified in the format `"`*table-name*`"."`*column-name*`"` in order to identify which table the column is in.

Example:

```
SELECT "USERID","NAME","PUR-CODE","PUR-DATE"
    FROM "SALESLIST","USERSLIST" WHERE "PUR-DATE">=DATE'2011-09-06'
    AND "SALESLIST"."USERID"="USERSLIST"."USERID"

KFAA30203-E Column "USERID" cannot be determined in the SQL statement.(query number =
 1)
```

In this example, the underlined portion of the statement is incorrect. Specify it using the format `"`*table-name*`"."USERID"` (for example, `"SALESLIST"."USERID"`).

1.15.6 If message KFAA30204-E is displayed

Check if there is an error in a specified table name.

Example:

```
SELECT "USERID","PUR-CODE","PUR-DATE" FROM "SALELIST"
    WHERE "PUR-DATE">=DATE'2011-09-06'

KFAA30204-E The table or index "ADBUSER01"."SALELIST" is not found in the system.
```

In this example, the underlined portion of the statement is incorrect. The correct table name is `"SALESLIST"`.

1.15.7 If message KFAA30401-E is displayed

Check if there is an error in the specification of a search condition.

Example 1:

```
SELECT "USERID","PUR-CODE","PUR-DATE" FROM "SALESLIST"
    WHERE "USERID">=DATE'2011-09-06'

KFAA30401-E The data types of both operands specified in predicate
"COMPARISON" are not compatible. (query number = 1)
```

In this example, the underlined portion of the statement is incorrect. It specifies an impossible condition in which you are attempting to compare the `USERID` column (customer ID) to the date range September 6, 2011 or later. Examples of correct specifications are as follows:

- `"USERID">='U00500'`
- `"PUR-DATE">=DATE'2011-09-06'`

Example 2:

```
SELECT "USERID","PUR-CODE","PUR-DATE" FROM "SALESLIST"
    WHERE "PUR-DATE">='2011-9-6'

KFAA30401-E The data types of both operands specified in predicate
"COMPARISON" are not compatible. (query number = 1)
```

In this example, the underlined portion of the statement is incorrect. The correct specification is `DATE'2011-09-06'`.

1.16 List of references by purpose

The table below lists the clauses, predicates, and functions that can be used in SELECT statements, as well as references to examples, organized according to the purpose of the retrieval.

Note that the descriptions given as examples assume that you are using the example tables used above, the sales history table (SALESLIST) and the customer table (USERSLIST).

Table 1-2: List of clauses, predicates, and functions usable in SELECT statements, with references to examples

No.	Purpose of retrieval	Clause, predicate, or function to use	Reference to example
1	You want to see all the data in a table. Examples: • See customer information for all customers. • See all sales history information for a product.	--	1.2 Retrieving all the rows from a table
2	You want to sort retrieved results in ascending or descending order. Examples: • Sort customer information by customer ID. • Sort a product's sales history information by date.	ORDER BY clause	1.3 Sorting retrieval results (ORDER BY clause)
3	You want to specify a maximum number of rows for the retrieval results. Examples: • See some but not all customer data. • See some but not all sales history information for a product.	LIMIT clause	1.4 Specifying the maximum number of rows of retrieval results (LIMIT clause)
4	You want to limit the search by specifying conditions. Examples: • Obtain yesterday's product sales history information. • Obtain the product purchase history for a specific customer.	WHERE clause	1.5 Retrieving data with search conditions specified
5	You want to search within a specified range. Example: • Obtain this week's product sales history information.	BETWEEN predicate	1.6 Retrieving data with a search range specified (BETWEEN predicate)
6	You want to search for data that matches any of multiple conditions. Example: • Find customers who purchased a product of product code P001 or P003.	IN predicate	1.7 Retrieving data that meets one of multiple conditions (IN predicate)
7	You want to search for data that contains a specific character string. Examples: • Obtain customer information for customers with the last name Johnson. • Obtain customer information for customers with a name whose initial letter is A.	LIKE predicate	1.8 Retrieving data that contains a specific character string (LIKE predicate)
8	You want to retrieve data from multiple tables (table join). Example:	WHERE clause	1.9 Retrieving data with multiple tables specified (table join),

No.	Purpose of retrieval	Clause, predicate, or function to use	Reference to example
	- Obtain customer information for customers who purchased a product yesterday.		1.10.1 Example: Retrieve customers who purchased products, and 1.13.5 Example 5: Aggregate data from the sales history table and customer table
9	You want to eliminate duplication in the retrieval results. Examples: - Find the names of customers who purchased products. - Obtain the product codes of items sold.	`SELECT DISTINCT`	1.10 Eliminating duplication in retrieval results (SELECT DISTINCT)
10	You want to determine the total number of data items in a table. Example: - Determine the total number of customers.	Set function `COUNT(*)`	1.11.1 Example 1: Determine the total number of customers
11	You want to determine the number of retrieved rows. Examples: - Determine the number of people who purchased a product. - Determine the number of product sales yesterday. - Determine the number of times a product was purchased by a specific customer.	Set function `COUNT(*)`	1.11.2 Example 2: Determine the number of people who purchased a product
12	You want to determine a maximum value from the retrieved data Example: - Determine the maximum quantity purchased of a product.	Set function `MAX`	1.12.1 Example 1: Determine the maximum, minimum, and average quantities purchased
13	You want to determine a minimum value from the retrieved data. Example: - Determine the minimum quantity purchased of a product.	Set function `MIN`	
14	You want to determine an average value from the retrieved data. Example: - Determine the average quantity purchased of a product	Set function `AVG`	
15	You want to determine the sum of the retrieved data. Examples: - Determine the quantity purchased yesterday of a product. - Determine the number of products purchased for a particular customer.	Set function `SUM`	1.12.2 Example 2: Determine the sum of quantities purchased)
16	You want to aggregate data into groups. Examples: - For each customer, determine the number of times a product was purchased or the quantities purchased. - For each product code, determine the number of sales or the quantities sold.	`GROUP BY` clause `HAVING` clause	1.13 Aggregating retrieved data by group (GROUP BY clause, HAVING clause)
17	You want to retrieve data based on the results of another `SELECT` statement specified in a search condition.	Subquery	1.14 Retrieving by specifying a SELECT

No.	Purpose of retrieval	Clause, predicate, or function to use	Reference to example
	Example: • Find information on the customer who purchased the greatest quantity of a product.		statement in the search condition (subquery)

Legend: --: Not applicable.

2 List of SQL Statements

This chapter lists the SQL statements supported by HADB, and explains how to read the SQL syntax specification format.

2.1 List of SQL statements

The following table lists the SQL statements supported by HADB.

Table 2-1: List of SQL statements supported by HADB

No.	Classification	SQL statement supported by HADB	Description
1	Definition SQL	ALTER TABLE	Change a base table's definition information.
2		ALTER USER	Change an HADB user's information.
3		ALTER VIEW	Re-create a viewed table.
4		CREATE AUDIT	Define audit targets.
5		CREATE INDEX	Define an index on a column in a base table.
6		CREATE SCHEMA	Define a schema.
7		CREATE TABLE	Define a base table.
8		CREATE USER	Create an HADB user.
9		CREATE VIEW	Define a viewed table.
10		DROP AUDIT	Delete the audit target definition.
11		DROP INDEX	Delete an index.
12		DROP SCHEMA	Delete a schema.
13		DROP TABLE	Delete a base table.
14		DROP USER	Delete an HADB user.
15		DROP VIEW	Delete a viewed table.
16		GRANT	Grant privileges to an HADB user.
17		REVOKE	Revoke privileges of an HADB user.
18	Data manipulation SQL	DELETE	Delete rows.
19		INSERT	Insert rows into a table.
20		PURGE CHUNK	Delete all the rows in a chunk.
21		SELECT	Retrieve data from a table.
22		TRUNCATE TABLE	Delete all the rows in a base table.
23		UPDATE	Update values in a row.
24	Control SQL	COMMIT	Validate the database contents that were updated by a transaction, and terminate the transaction normally.
25		ROLLBACK	Invalidate the database contents that were updated by a transaction, and cancel the transaction.

Notes:

You can execute the above SQL statements from application programs or by using the adbsql command. However, control SQL statements (COMMIT and ROLLBACK) cannot be used in application programs.

- If you are using the JDBC driver, use the `commit` method or `rollback` method in the `Connection` interface. For details about these methods, see the *HADB Application Development Guide*.
- If you are using the ODBC driver, use the ODBC function `SQLEndTran`. For details about `SQLEndTran`, see the *HADB Application Development Guide*.
- If you are using CLI functions, use `a_rdb_SQLEndTran()`. For details about `a_rdb_SQLEndTran()`, see the *HADB Application Development Guide*.

Note

- The `SELECT` statement is also called the retrieval SQL statement.
- The `INSERT`, `UPDATE`, `DELETE`, `PURGE CHUNK`, and `TRUNCATE TABLE` statements are generically called update SQL statements.

2.2 How to read the SQL syntax specification format

This section describes the SQL statement syntax specification format using BNF notation. To explain how to read the SQL statement syntax specification format, the `LIKE` predicate is used as an example.

Description of the specification format of the LIKE predicate

```
LIKE-predicate ::= match-value [NOT] LIKE pattern-character-string [ESCAPE escape-cha
racter]  ...1

  match-value ::= value expression  ...2
  pattern-character-string ::= value expression  ...2
  escape-character ::= value expression  ...2
```

This notation means that the item on the left of the `::=` is described in the format shown on the right of it. Therefore, the meanings of the specifications indicated by 1 and 2 in the preceding format are as follows:

1. The `LIKE` predicate takes the form *match-value* `[NOT] LIKE` *pattern-character-string* `[ESCAPE` *escape-character*`]`
2. *match-value*, *pattern-character-string*, and *escape-character* take the form of value expressions.

In other words, a `LIKE` predicate is written in the following format:

```
value-expression [NOT] LIKE value-expression ESCAPE value-expression
```

Value expressions are described in 7.20 Value expression. Refer to that section for the specification format of a value expression.

3 Definition SQL

This chapter describes the functions, rules, and specification formats of definition SQL statements.

3.1 ALTER TABLE (alter table definition)

This section describes the specification format and rules for the ALTER TABLE statement.

3.1.1 Specification format and rules for the ALTER TABLE statement

You can use the ALTER TABLE statement to perform the following operations:

- Add a column to a base table
- Rename a column of a base table
- Change the maximum number of chunks in a multi-chunk table
- Change a regular multi-chunk table into an archivable multi-chunk table
- Change an archivable multi-chunk table into a regular multi-chunk table

Note that you cannot perform two or more operations at the same time by a single execution of the ALTER TABLE statement.

(1) Specification format and description

(a) Adding a column to a base table

Specification format

```
ALTER-TABLE-statement ::= ALTER TABLE table-name
                      ADD COLUMN column-definition

  column-definition ::= column-name data-type [NOT NULL] [BRANCH {YES | NO | AUTO}] [
compression-type-specification]
```

- *table-name*

  Specify the name of the base table to which to add a column. For rules on specifying a table name, see (2) Table name specification format in 6.1.5 Qualifying a name.

  Note that the following tables cannot be specified:

  - Viewed tables
  - Dictionary tables
  - System tables

- ADD COLUMN *column-definition*

  ```
  column-definition ::= column-name data-type [NOT NULL] [BRANCH {YES | NO | AUTO}]
  [compression-type-specification]
  ```

  Specify the column definition of the column to be added.

  The following conditions govern the specifications for adding a column:

  - Only one column can be added at a time. The column is added as the last column of the base table.
  - Null values are stored in the added column.
  - You cannot add a column to a base table under the following circumstances:

- The number of columns in the target base table has already reached the maximum (1,000).
- The target base table is a FIX table to which row storage segments have been allocated.
- The target base table was created by running the `CREATE TABLE` statement with `BRANCH ALL` specified, and row storage segments have been allocated to the target base table.

For details about the status in which row storage segments have been allocated, see *Notes on defining B-tree indexes (unfinished status of B-tree indexes)* in the *HADB Setup and Operation Guide*.

Note

You cannot specify a `DEFAULT` clause in an `ALTER TABLE` statement.

column-name:

Specify the name of the column to be added.

You cannot specify a column name that is already used in the table.

Do not specify a character string in the `EXP`*nnnn*`_NO_NAME` format as a column name. Such a column name might duplicate a derived column name that is automatically set by HADB. In this format, *nnnn* is an unsigned integer in the range from `0000` to `9999`.

data-type:

Specify the data type of the column to be added. The data types that can be specified are shown in the following table:

Table 3-1: Data types that can be specified (ALTER TABLE statement)

No.	Data type	Specification format
1	`INTEGER`	`INT` or `INTEGER`
2	`SMALLINT`	`SMALLINT`
3	`DECIMAL`	`DEC[(`*m*`[,`*n*`])]` or `DECIMAL[(`*m*`[,`*n*`])]` *m*: Precision (total number of digits) *n*: Scaling (number of decimal places) If *m* is omitted, `38` is assumed, and if *n* is omitted, `0` is assumed.
4	`DOUBLE PRECISION`	`DOUBLE` or `DOUBLE PRECISION`
5	`CHARACTER`	`CHAR(`*n*`)` or `CHARACTER(`*n*`)` *n*: Length of the character string (number of bytes) If `CHAR` or `CHARACTER` is specified without a length, the length of the character string is assumed to be `1`.
6	`VARCHAR`#1, #2	`VARCHAR(`*n*`)` *n*: Maximum length of the character string (number of bytes)
7	`DATE`	`DATE`
8	`TIME`	`TIME(`*p*`)` or `TIME` *p*: Fractional seconds precision (number of digits to the right of the decimal point) You can specify a value of `0`, `3`, `6`, `9`, or `12` for *p*. If `TIME` is specified, *p* is assumed to be `0`.
9	`TIMESTAMP`	`TIMESTAMP(`*p*`)` or `TIMESTAMP` *p*: Fractional seconds precision (number of digits to the right of the decimal point) You can specify a value of `0`, `3`, `6`, `9`, or `12` for *p*. If `TIMESTAMP` is specified, *p* is assumed to be `0`.

No.	Data type	Specification format
10	BINARY	BINARY(*n*) *n*: Length of the binary data (number of bytes) If BINARY is specified without a length, the length of the binary data is assumed to be 1.
11	VARBINARY#1	VARBINARY(*n*) *n*: Maximum length of the binary data (number of bytes)

#1: The VARCHAR and VARBINARY types cannot be specified for columns that are added to a FIX table.

#2: VARCHAR-type data whose data length exceeds 32,000 bytes cannot be specified.

For details about data types, see 6.2 Data types.

NOT NULL:

Specify this to define the NOT NULL constraint (the constraint that does not allow null values) on the column to be added.

Note the following:

- You cannot specify NOT NULL for a base table to which row storage segments have been allocated. For details about the status in which row storage segments have been allocated, see *Notes on defining B-tree indexes (unfinished status of B-tree indexes)* in the *HADB Setup and Operation Guide*.
- In FIX tables, the NOT NULL constraint is set for all columns. When a column is added to a FIX table, the NOT NULL constraint is considered to be specified even if it is omitted.

BRANCH {YES | NO | AUTO}:

Specifies how to store VARCHAR-type and VARBINARY-type column data.

For details about situations for which it is better to specify YES or NO for BRANCH, see *Branch specification for column data of variable-length data types (BRANCH) [Row store table]* in the *HADB Setup and Operation Guide*.

The types of storage methods that can be specified are the same as those that can be specified for a CREATE TABLE statement. In addition, if BRANCH is omitted, this statement operates in the same way as when BRANCH is omitted from the CREATE TABLE statement. For details about BRANCH in the CREATE TABLE statement, see (d) column-definition [Common] of (2) Explanation of specification format in 3.7.1 Specification format and rules for the CREATE TABLE statement.

This option cannot be specified for the following tables and columns:

- Tables for which the BRANCH ALL table option is specified
- Columns of data types other than VARCHAR and VARBINARY
- Column store tables to which a column is to be added

compression-type-specification:

```
compression-type-specification ::= COMPRESSION TYPE {AUTO|NONE|RUNLENGTH|DICTION
ARY|DELTA|DELTA_RUNLENGTH}
```

Specifies the compression type to be used to compress the data of the column to be added (column-data compression type).

The compression types that can be specified are the same as those that can be specified for the CREATE TABLE statement. In addition, if *compression-type-specification* is omitted, this statement operates in the same way as when *compression-type-specification* is omitted from the CREATE TABLE statement. For details about *compression-type-specification* in the CREATE TABLE statement, see (d) column-definition [Common] of (2) Explanation of specification format in 3.7.1 Specification format and rules for the CREATE TABLE statement.

Note that you cannot specify this option if the table to which a column is to be added is a row store table.

(b) Renaming a column in a base table

Specification format

```
ALTER-TABLE-statement ::= ALTER TABLE table-name
                          RENAME COLUMN [FROM] current-column-name TO new-column-name
```

- *table-name*

  Specify the name of the base table of which to rename a column. For rules on specifying a table name, see (2) Table name specification format in 6.1.5 Qualifying a name.

  Note that the following tables cannot be specified:

  - Viewed tables
  - Dictionary tables
  - System tables

- `RENAME COLUMN [FROM]` *current-column-name* `TO` *new-column-name*

  Specify the current column name and new column name. The current column name is changed to the new column name.

  The following rules apply:

  - An existing column name cannot be specified as the new column name.
  - The same name cannot be specified for both the current and new column names.
  - Do not specify a character string in the `EXP`*nnnn*`_NO_NAME` format as a new column name. Such a column name might duplicate a derived column name that is automatically set by HADB. In this format, *nnnn* is an unsigned integer in the range from `0000` to `9999`.

> **Important**
>
> If a column of a table is renamed, all viewed tables that are dependent on that table are invalidated. For details about how to check the viewed tables that will be invalidated, see *Checking dependent viewed tables* in the *HADB Setup and Operation Guide*.

(c) Changing the maximum number of chunks in a multi-chunk table

Specification format

```
ALTER-TABLE-statement ::= ALTER TABLE table-name
                          CHANGE OPTION CHUNK=maximum-number-of-chunks
```

- *table-name*

  Specify the name of the multi-chunk table for which you want to change the maximum number of chunks. For rules on specifying a table name, see (2) Table name specification format in 6.1.5 Qualifying a name.

- `CHANGE OPTION CHUNK=`*maximum-number-of-chunks*

  ~ <unsigned integer> ((`2` to `30,000`)) (unit: chunks)

  Specify a new maximum number of chunks, replacing the maximum number of chunks that was specified using the chunk specification in the `CREATE TABLE` statement.

  Note the following points:

  - This option can be specified only for multi-chunk tables.
  - You cannot specify a value that is less than the number of chunks that have already been created in the table.

(d) Changing a regular multi-chunk table to an archivable multi-chunk table

Specification format

```
ALTER-TABLE-statement::=LTER TABLE table-name
                      CHANGE OPTION CHUNK chunk-archive-specification

  chunk-archive-specification::=ARCHIVABLE
                                RANGECOLUMN=column-name
                               [RANGEINDEXNAME=index-identifier]
                               [IN DB-area-name]
                                ARCHIVEDIR=archive-directory-name
```

- *table-name*

  Specify the name of the regular multi-chunk table that you want to change to an archivable multi-chunk table. For rules on specifying a table name, see (2) Table name specification format in 6.1.5 Qualifying a name.

  Note that the following tables cannot be specified:

  - FIX tables
  - Single-chunk tables
  - Column store tables

- CHANGE OPTION CHUNK *chunk-archive specification*

  ```
  chunk-archive-specification::=ARCHIVABLE
                                RANGECOLUMN=column-name
                               [RANGEINDEXNAME=index-identifier]
                               [IN DB-area-name]
                                ARCHIVEDIR=archive-directory-name
  ```

  Specify this option if you change a regular multi-chunk table to an archivable multi-chunk table.

  - RANGECOLUMN=*column-name*

    Specify a column name. The column specified here becomes the archive range column.

    The following rules apply:

    - You cannot use columns of the following data types as archive range columns:
      - CHARACTER type (only if the defined length is 33 bytes or more)
      - VARCHAR type
      - BINARY type
      - VARBINARY type
    - When you specify a column to be used as an archive range column, make sure that the NOT NULL constraint is specified for that column.

  - RANGEINDEXNAME=*index-identifier*

    When the ALTER TABLE statement is run, the HADB server automatically defines a range index that uses an archive range column as the indexed column. Specify the index identifier to be assigned to this range index.

    > **Important**
    >
    > If a range index has already been defined for the archive range column, no range index is automatically defined when the ALTER TABLE statement is run. In this case, the already defined range index is used. Therefore, if you specify RANGEINDEXNAME when a range index has already been defined for the archive range column, the ALTER TABLE statement will result in an error.

If RANGEINDEXNAME is not specified, the HADB server determines the index identifier in the following format:

```
ARCHIVE_RANGE_INDEX_nnnnnnnn
```

nnnnnnnn is an eight-digit character string that represents the ID of the archivable multi-chunk table in hexadecimal notation.

If the automatically determined index identifier already exists in the same schema, the ALTER TABLE statement will result in an error. Therefore, if you use the CREATE INDEX statement to define an index, we recommend that you do not use a name whose format resembles the preceding format.

> **Note**
>
> The range index that is automatically defined here is subject to the same rules as a range index defined by the CREATE INDEX statement.

- IN *DB-area-name*

  Specify the name of the DB area in which to store the range indexes that are automatically defined by the HADB server.

  In the following case, omit specifying IN *DB-area-name*:

  - When a range index has already been defined for the archive range column

  In this case, because the HADB server does not automatically define a range index, you do not need to specify IN *DB-area-name*.

  Also, if all of the following conditions are met, a range index automatically defined by the HADB server is stored in the DB area specified for the adb_sql_default_dbarea_shared operand in the server definition:

  - When the IN *DB-area-name* specification is omitted
  - No range index is defined for the archive range column.

  Note that, when both of the preceding two conditions are met, if the adb_sql_default_dbarea_shared operand is not specified in the server definition, the ALTER TABLE statement will result in an error. Also, if the DB area specified for the adb_sql_default_dbarea_shared operand in the server definition does not exist or if a DB area other than the data DB area is specified, the ALTER TABLE statement will result in an error.

- ARCHIVEDIR=*archive-directory-name*

  Specify the absolute path name of the archive directory in which to store archive files.

  The following rules apply:

  - Specify the archive directory name in the character string literal format. For details about character string literals, see 6.3 Literals.
  - Specify an existent directory for the archive directory. Make sure that read, write, and execution permissions for the HADB administrator are assigned to the directory that you specify.

    Also, make sure that execution permission for the HADB administrator is assigned to all directories that are included in the path of the archive directory.

    (Example) If the archive directory is /HADB/archive:

    For the /HADB/archive directory, read, write, and execution permissions must be set.

    For the / directory and the /HADB directory, the execution permission is necessary.
  - The following directories cannot be used as the archive directory:
    - Server directory
    - Subdirectory of a server directory

- Directory that contains a server directory
- DB directory
- Subdirectory of a DB directory
- Directory that contains a DB directory
- Root directory

The following shows examples of directories that can be and cannot be used as the archive directory when the DB directory is `/HADB/db`:

Directory		Reason
Example of directory that can be used as the archive directory	`/HADB/archive`	None.
Example of directory that cannot be used as the archive directory	`/HADB/db`	This directory is the same as the DB directory.
	`/HADB/db/archive`	This directory is a subdirectory of the DB directory.
	`/HADB`	This directory contains the DB directory.

- Do not specify (as the archive directory) a directory in which installation data was stored when the HADB server was installed.
- The name of the archive directory must be 1 to 400 bytes long except the heading and trailing spaces.

Note

If you specify a directory name that begins and/or ends with spaces, these spaces are deleted (the resulting character string is used as the archive directory name).

- Make sure that each element of the archive directory name is no more than `NAME_MAX` bytes long. The `NAME_MAX` value differs depending on the environment.

If a symbolic link is specified as the archive directory name, the system checks whether the absolute path name that the symbolic link substitutes for obeys the rules that are described here.

About the multi-node function:

If the multi-node function is enabled, note the following points:

- Use the NFS or other means to share the archive directory by all nodes. Note that the archive directory must be shared by all nodes when the `ALTER TABLE` statement is run.
- On the master node, when the `ALTER TABLE` statement is run, a check to see whether the archive directory name obeys the specification rules that are described here is conducted. On the slave nodes, this check is not conducted. Therefore, after the `ALTER TABLE` statement, check the archive directory name on each slave node.

About the location table that is defined when a regular multi-chunk table is changed to an archivable multi-chunk table

If a regular multi-chunk table is changed to an archivable multi-chunk table by running the `ALTER TABLE` statement, the HADB server automatically defines the location table and the index of the location table. The HADB server uses the location table and index. Therefore, no user can directly manipulate, redefine, or delete the location table or index. For details about the location table, see *Searching an archivable multi-chunk table* in the *HADB Setup and Operation Guide*.

The location table and its index are stored in the same DB area as the archivable multi-chunk table.

The names of the location table and its index are determined according to the rules that are described in the following table.

Table 3-2: Naming rules for the location table and location table index

Item	Naming rule	Information managed by the index	Columns in the index
Location table	"HADB"."LOCATION_TABLE_*nnnnnnnn*"	--	--
Location table index	"HADB"."LOCATION_INDEX_*nnnnnnnn*_CHUNK_ID"	Manages the chunk ID of the chunk that corresponds to the archive file.	CHUNK_ID
	"HADB"."LOCATION_INDEX_*nnnnnnnn*_RANGE_01"	Manages the range (upper and lower limits) of values in the archive range column of data stored in the archive file.	• RANGE_MAX • RANGE_MIN
	"HADB"."LOCATION_INDEX_*nnnnnnnn*_RANGE_02"	Manages the lower limit of values in the archive range column of data stored in the archive file.	RANGE_MIN

Legend:

--: Not applicable.

Note:

nnnnnnnn is an eight-digit character string that represents the ID of the archivable multi-chunk table in hexadecimal notation.

The schema name of the location table and location table index is HADB.

Note the following points when changing a regular multi-chunk table to an archivable multi-chunk table:

- If a regular multi-chunk table is changed to an archivable multi-chunk table, the HADB server automatically defines the location table, the location table index, and the range index of the archive range column. However, if a range index has been defined for the column that is specified as the archive range column, a new range index is not defined. In this case, the already defined range index is used. For details about how to check whether a range index has already been defined for a column, see *Investigating whether range indexes are defined in the column specified as the archive range column* in *Searching a dictionary table* in the *HADB Setup and Operation Guide*.
- If a regular multi-chunk table is changed to an archivable multi-chunk table, all viewed tables that are dependent on the table to be changed are invalidated. For details about how to check the viewed tables that will be invalidated, see *Checking dependent viewed tables* in the *HADB Setup and Operation Guide*.

For details about how to change a regular multi-chunk table to an archivable multi-chunk table, see *Changing a regular multi-chunk table to an archivable multi-chunk table* in the *HADB Setup and Operation Guide*.

(e) Changing an archivable multi-chunk table to a regular multi-chunk table

Specification format

```
ALTER-TABLE-statement ::= ALTER TABLE table-name
                          CHANGE OPTION CHUNK UNARCHIVABLE
```

- *table-name*

  Specify the name of the archivable multi-chunk table that you want to change into a regular multi-chunk table. For rules on specifying a table name, see (2) Table name specification format in 6.1.5 Qualifying a name.

- CHANGE OPTION CHUNK UNARCHIVABLE

  Specify this option if you change an archivable multi-chunk table to a regular multi-chunk table.

Note the following points when changing an archivable multi-chunk table to a regular multi-chunk table:

- You cannot change an archivable multi-chunk table to a regular multi-chunk table if there are archived chunks. In this case, unarchive the chunks, and then change an archivable multi-chunk table to a regular multi-chunk table if there are archived chunks.
- When an archivable multi-chunk table is changed to a regular multi-chunk table, the location table and the index that has been defined for the location table are deleted. However, the range index that has automatically been defined for the archive range column is not deleted. If this range index is unnecessary, delete it by using the `DROP INDEX` statement after changing an archivable multi-chunk table to a regular multi-chunk table.
- If an archivable multi-chunk table is changed to a regular multi-chunk table, all viewed tables that are dependent on the table to be changed are invalidated. For details about how to check the viewed tables that will be invalidated, see *Checking dependent viewed tables* in the *HADB Setup and Operation Guide*.

For details about how to change an archivable multi-chunk table to a regular multi-chunk table, see *Changing an archivable multi-chunk table to a regular multi-chunk table* in the *HADB Setup and Operation Guide*.

(2) Privileges required at execution

To execute the `ALTER TABLE` statement, the `CONNECT` privilege and schema definition privilege are required.

(3) Rules

1. You can only alter the definition of a table in the schema of the current user (the HADB user whose authorization identifier is currently connected to the HADB server). You cannot alter the definition of a table in a schema owned by another HADB user.
2. You cannot add a column unless the sum of the sizes of all columns in the base table (the row length) satisfies the following formula:

   ● **Formula (if the target base table is a row store table)**

   ```
   ROWSZ-(row-length) ≤ page-size - 56 (unit: bytes)
   ```

   ● **Formula (if the target base table is a column store table)**

   ```
   ROWSZ-(row-length) ≤ page-size - 80 (unit: bytes)
   ```

   For details about the formula for calculating *ROWSZ* (row length), see *Determining the number of pages for storing each type of row* in the *HADB Setup and Operation Guide*.
3. You cannot change the definition of a table that has been rendered non-updatable due to an interrupted command.

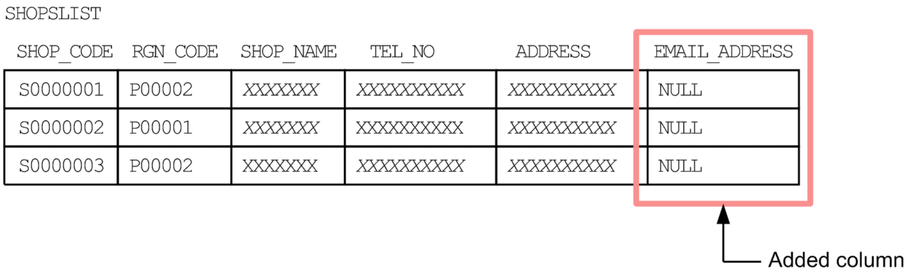
(4) Examples

Example 1: Adding a column to a row store table

Add a column for the email address of each shop (`EMAIL_ADDRESS`) to the shops table (`SHOPSLIST`), which is a row store table.

- Column name: `EMAIL_ADDRESS`
- Data type: `VARCHAR(100)`
- Branch the data in the column

```
ALTER TABLE "SHOPSLIST"
    ADD COLUMN "EMAIL_ADDRESS" VARCHAR(100) BRANCH YES
```



SHOPSLIST

SHOP_CODE	RGN_CODE	SHOP_NAME	TEL_NO	ADDRESS	EMAIL_ADDRESS
S0000001	P00002	XXXXXXX	XXXXXXXXXX	XXXXXXXXXX	NULL
S0000002	P00001	XXXXXXX	XXXXXXXXXX	XXXXXXXXXX	NULL
S0000003	P00002	XXXXXXX	XXXXXXXXXX	XXXXXXXXXX	NULL

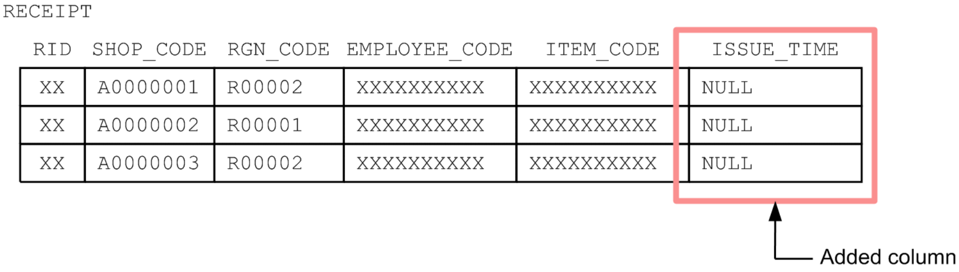
Added column

Example 2: Adding a column to a column store table

Add a column for the times that receipts were issued (ISSUE_TIME) to the receipt table (RECEIPT), which is a column store table.

- Column name: ISSUE_TIME
- Data type: TIME
- Compress the data in the column by using the delta run-length encoding algorithm (DELTA_RUNLENGTH).

```
ALTER TABLE "RECEIPT"
    ADD COLUMN  "ISSUE_TIME" TIME COMPRESSION TYPE DELTA_RUNLENGTH
```



RECEIPT

RID	SHOP_CODE	RGN_CODE	EMPLOYEE_CODE	ITEM_CODE	ISSUE_TIME
XX	A0000001	R00002	XXXXXXXXXX	XXXXXXXXXX	NULL
XX	A0000002	R00001	XXXXXXXXXX	XXXXXXXXXX	NULL
XX	A0000003	R00002	XXXXXXXXXX	XXXXXXXXXX	NULL

Added column

Example 3: Renaming a column

In the shops table (SHOPSLIST), rename the EMAIL_ADDRESS column to EMAIL.

```
ALTER TABLE "SHOPSLIST"
    RENAME COLUMN FROM EMAIL_ADDRESS TO EMAIL
```

Example 4: Changing the maximum number of chunks

Change the maximum number of chunks in the shops table (SHOPSLIST) to 300.

```
ALTER TABLE "SHOPSLIST"
    CHANGE OPTION CHUNK=300
```

Example 5: Changing a regular multi-chunk table to an archivable multi-chunk table

Change the format of the shops table (SHOPSLIST), which is a row store table, from that of a regular multi-chunk table to that of an archivable multi-chunk table. The specifications related to the archive range column and other items are as follows:

- The RECORD_DAY column is used as the archive range column.
- The /mnt/nfs/archivedir directory is used as the archive directory.
- The DB area that stores the range indexes that are automatically defined by the HADB server is DBAREA02.

```
ALTER TABLE "SHOPSLIST"
    CHANGE OPTION CHUNK ARCHIVABLE
        RANGECOLUMN="RECORD_DAY"
        IN "DBAREA02"
        ARCHIVEDIR='/mnt/nfs/archivedir'
```

Example 6: Changing an archivable multi-chunk table to a regular multi-chunk table

Change the format of the shops table (SHOPSLIST) from an archivable multi-chunk table to a regular multi-chunk table.

```
ALTER TABLE "SHOPSLIST"
    CHANGE OPTION CHUNK UNARCHIVABLE
```

3.2 ALTER USER (alter an HADB user's information)

This section describes the specification format and rules for the `ALTER USER` statement.

3.2.1 Specification format and rules for the ALTER USER statement

Change the following information for an HADB user:

- Password

(1) Specification format

```
ALTER-USER-statement ::= ALTER USER authorization-identifier IDENTIFIED BY new-password
```

(2) Explanation of specification format

● *authorization-identifier*

Specify the authorization identifier of the HADB user whose user information is to be changed.

For rules about specifying authorization identifiers, see 6.1.4 Specifying names.

● `IDENTIFIED BY` *new-password*

Specify the new password.

The rules for specifying the password are as follows:

- The password can include single-byte uppercase and lowercase letters, numbers, backslashes (\), as well as the following characters:
  `@ ` ! " # $ % & ' ( ) * : + ; [ ] { } , = < > | - . ^ ~ / ? _`
- Specify the password in the form of a character string literal. Therefore, you must enclose the password in single quotation marks. The following are examples:

  Example 1: Set the new password to `Password01`

  `IDENTIFIED BY 'Password01'`

  Example 2: Set the new password to `Pass'01`

  `IDENTIFIED BY 'Pass''01'`

  If the password itself includes a single quotation mark (`'`), specify two single quotation marks (`''`) to represent the single quotation mark (`'`), as shown in the example above.

  For rules on specifying a character string literal, see Table 6-10: Description formats and assumed data types of literals.
- The password cannot be empty. That is, the following is not permitted:

  `IDENTIFIED BY ''`
- The password cannot exceed 255 characters (255 bytes).

Note

- If you are using the JDBC driver, we recommend that you not use the following character in the password:

&

- If you are using the ODBC driver, we recommend that you not use the following characters in the password:
  [ ] { } ( ) , ; ? * = ! @

(3) Privileges required at execution

To execute the ALTER USER statement, the CONNECT privilege is required.

(4) Rules

1. An HADB user with the DBA privilege can change the user information of every HADB user. However, the user information of HADB users with the audit privilege cannot be changed. The user information of an HADB user with the audit privilege can be changed by that HADB user only.
2. An HADB user without the DBA privilege can change the user information of only the current user (the HADB user whose authorization identifier is currently connected to the HADB server).

(5) Examples

Example

Change the password of HADB user ADBUSER01 to #HelloHADB_02.

```
ALTER USER "ADBUSER01" IDENTIFIED BY '#HelloHADB_02'
```

3.3 ALTER VIEW (re-create a viewed table)

This section describes the specification format and rules for the `ALTER VIEW` statement.

3.3.1 Specification format and rules for the ALTER VIEW statement

Re-create a viewed table.

Run the `ALTER VIEW` statement to re-create a viewed table in the following cases:

- When you revalidate a viewed table

  After you have removed the cause that invalidated a viewed table, the viewed table is revalidated when you run the `ALTER VIEW` statement to re-create the viewed table.

- When you become unsure of the reason why a viewed table has been invalidated

  If you run the `ALTER VIEW` statement for a viewed table for which the cause of invalidation has not been removed, the `ALTER VIEW` statement results in an error. In this case, an error message is output. From this error message, you can identify the reason why the viewed table has been invalidated.

> **Important**
>
> The `ALTER VIEW` statement cannot change the definition of a viewed table. To change the definition of a viewed table, use the `DROP VIEW` statement to delete the viewed table, and then use the `CREATE VIEW` statement to redefine the viewed table.

(1) Specification format

```
ALTER-VIEW-statement ::= ALTER VIEW table-name RECREATE
```

(2) Explanation of specification format

● *table-name*

Specifies the name of the viewed table to be re-created. For rules on specifying a table name, see (2) Table name specification format in 6.1.5 Qualifying a name.

The following tables cannot be specified:

- Base tables
- Dictionary tables
- System tables

● `RECREATE`

Specify this to re-create a viewed table.

(3) Privileges required at execution

To run the `ALTER VIEW` statement, the `CONNECT` privilege and the schema definition privilege are required.

(4) Rules

1. If the authentication identifier connected to the HADB server is different from the schema name of a viewed table, the ALTER VIEW statement results in an error.
2. Even if viewed tables that depend on the viewed table to be re-created have been defined, the viewed table is re-created when the ALTER VIEW statement is run. In this case, the viewed tables that depend on the re-created viewed table are invalidated.
3. Re-creating a viewed table by using the ALTER VIEW statement does not affect the access privileges for the viewed tables that depend on the re-created viewed table.
4. The viewed table specified in the ALTER VIEW statement is always re-created regardless of whether the viewed table is valid or invalid.
5. If the ALTER VIEW statement is used to re-create a viewed table, the number of columns or column names of the viewed table might be changed. For example, the following cases apply.

   Example of defining viewed table V1:

   ```
   CREATE VIEW "V1" AS SELECT * FROM "T1" WHERE "C1">100
   ```

   - Case where the number of columns of a viewed table increases

     1. The CREATE VIEW statement is used to define viewed table V1.
     2. The ALTER TABLE statement is used to add a column (for example, column C5) to underlying table T1.
     3. The ALTER VIEW statement is used to re-create viewed table V1.

     In this case, because column C5 is added to viewed table V1, the number of columns in the viewed table increases.

   - Case where a column name of a viewed table changes

     In step 2 of the preceding procedure, assume that, for example, you use the ALTER TABLE statement to change the column name of column C2. In this case, if you then use the ALTER VIEW statement to re-create viewed table V1, the column name of column C2 in viewed table V1 changes.

6. When the ALTER VIEW statement is used to re-create a viewed table, the access privilege settings for the underlying table might have been changed# since the viewed table was defined. In such a case, the dependent privileges of the access privilege for the re-created viewed table might be revoked.

   #

   Either of the following changes applies.

   - An access privilege with the grant option has been changed to an access privilege without the grant option.
   - An access privilege with the grant option has been removed so that no access privilege is granted.

   The following shows an example in which a dependent privilege of the access privilege for a re-created viewed table is revoked.

   Example:

   1.

      HADB user A, who has the SELECT privilege with the grant option for table X.T1, defines viewed table A.V1 by using table X.T1 as the underlying table.

   2.

      HADB user A grants the SELECT privilege for viewed table A.V1 to another HADB user. The SELECT privilege that was granted to another HADB user becomes a dependent privilege.

   3.

      HADB user A has the SELECT privilege with the grant option for table X.T1 revoked. At this time, viewed table A.V1 is invalidated because table X.T1 is an underlying table of the viewed table.

4.

   To revalidate viewed table A.V1, the SELECT privilege without the grant option for table X.T1 is granted to HADB user A. The SELECT privilege with the grant option for table X.T1 that HADB user A had when defining viewed table A.V1 in step 1 has been changed to the SELECT privilege without the grant option.

5.

   The ALTER VIEW statement is run to re-create viewed table A.V1.

   Because the SELECT privilege was changed to a SELECT privilege without the grant option in step 4, the SELECT privilege that was granted to another HADB user and became a dependent privilege in step 2 is revoked.

(5) Examples

Example

Because a viewed table (VSHOPSLIST) for the shops table was invalidated, the ALTER VIEW statement is run to revalidate VSHOPSLIST.

```
ALTER VIEW "VSHOPSLIST" RECREATE
```

3.4 CREATE AUDIT (define audit targets)

This section describes the specification format and rules for the `CREATE AUDIT` statement.

Note that information defined by using the `CREATE AUDIT` statement is called an *audit target definition*.

3.4.1 Specification format and rules for the CREATE AUDIT statement

The CREATE AUDIT statement defines audit targets.

> **Important**
>
> You can execute the `CREATE AUDIT` statement when the audit trail facility is enabled. To check whether the audit trail facility is enabled, execute the `adbaudittrail -d` command.

(1) Specification format

```
CREATE-AUDIT-statement::=CREATE AUDIT AUDITTYPE EVENT
                              FOR ANY OPERATION
```

(2) Explanation of specification format

- `AUDITTYPE EVENT`

  Specify this if you want to output an audit trail of the final event results.

- `FOR ANY OPERATION`

  Specify this if the audit-target event is in the following table.

Table 3-3: Audit-target events

Event type	Audit-target event
Session management event	Execution of `CONNECT` (connection to an HADB server)
	Execution of `DISCONNECT` (disconnection from an HADB server)
Privilege management event	Executions of the following SQL statements: • `GRANT` statement • `REVOKE` statement • `CREATE USER` statement • `DROP USER` statement • `ALTER USER` statement
Definition SQL event	Executions of the following definition SQL statements: • `ALTER TABLE` statement • `ALTER VIEW` statement • `CREATE AUDIT` statement • `CREATE INDEX` statement • `CREATE SCHEMA` statement • `CREATE TABLE` statement • `CREATE VIEW` statement

Event type	Audit-target event
	• DROP AUDIT statement • DROP INDEX statement • DROP SCHEMA statement • DROP TABLE statement • DROP VIEW statement
Data manipulation SQL event	Executions of the following data manipulation SQL statements: • SELECT statement • INSERT statement • UPDATE statement • DELETE statement • TRUNCATE TABLE statement • PURGE CHUNK statement
Command operation event	Executions of the following commands: • adbimport command • adbexport command • adbidxrebuild command • adbgetcst command • adbdbstatus command • adbmergechunk command • adbchgchunkcomment command • adbchgchunkstatus command • adbarchivechunk command • adbunarchivechunk command • adbreorgsystemdata command • adbsyndict command

(3) Privileges required at execution

To execute the CREATE AUDIT statement, the CONNECT privilege and the audit admin privilege are required.

(4) Rules

1. You cannot define multiple instances of the same audit target.
2. An HADB server checks the audit target definition during the determination processing for outputting an audit trail. Therefore, depending on the audit trail output time, an audit trail about operations that were performed before the audit targets are defined might be output although those operations are not to be audited.

(5) Examples

Example

The events listed in Table 3-3: Audit-target events are defined as audit targets.

```
CREATE AUDIT AUDITTYPE EVENT
             FOR ANY OPERATION
```

3.5 CREATE INDEX (define an index)

This section describes the specification format and rules for the CREATE INDEX statement.

3.5.1 Specification format and rules for the CREATE INDEX statement

The CREATE INDEX statement defines an index (a B-tree index, text index, or range index) on a column in a base table. For details about B-tree indexes, text indexes, and range indexes, see *B-tree indexes*, *Text indexes*, and *Range indexes* in the *HADB Setup and Operation Guide*.

A B-tree index can be defined on multiple columns. A B-tree index defined on only one column is called a *single-column index*, and a B-tree index defined on multiple columns is called a *multiple-column index*.

Important

If you define an index for a base table to which row storage segments have been allocated, the index is placed in unfinished status (status in which no index data is created).

For example, no row storage segments have been allocated at the following times. If you define an index for a base table in this status, the index is created normally.

- Immediately after a base table is defined
- Immediately after the TRUNCATE TABLE statement is run

While a B-tree index is in unfinished status, you cannot perform searches that use the unfinished B-tree index, nor can you execute INSERT, UPDATE, or DELETE statements on the table.

While a text index is in unfinished status, you cannot perform searches that use the unfinished text index, nor can you execute INSERT, UPDATE, or DELETE on the table.

While a range index is in unfinished status, you cannot perform searches that use the unfinished range index, nor can you execute INSERT or UPDATE on the table.

For details about how to release indexes from unfinished status, see the following sections (whichever is applicable) in the *HADB Setup and Operation Guide*: *Steps to take when unfinished status is applied to a B-tree index*, *Steps to take when unfinished status is applied to a text index*, or *Steps to take when unfinished status is applied to a range index*.

For details about the status in which row storage segments have been allocated, see *Notes on defining B-tree indexes (unfinished status of B-tree indexes)* in the *HADB Setup and Operation Guide*.

(1) Specification format

```
CREATE-INDEX-statement ::=
     CREATE [UNIQUE] INDEX index-name
          ON table-name (column-name [{ASC|DESC}][,column-name [{ASC|DESC}]]...)
        [IN DB-area-name]
        [PCTFREE=percentage-of-unused-area]
          EMPTY
        [INDEXTYPE {BTREE|TEXT [WORDCONTEXT]|RANGE}]
        [CORRECTIONRULE]
```

```
         [DELIMITER {DEFAULT|ALL}]
         [EXCLUDE NULL VALUES]
```

Note

- PCTFREE, EMPTY, INDEXTYPE, CORRECTIONRULE, DELIMITER, and EXCLUDE NULL VALUES are generically called *index options*.
- Index options can be specified in any order.

The following table shows the different options that can be specified depending on which type of index is defined.

Table 3-4: Options for defining an index

No.	CREATE INDEX option	When defining a B-tree index	When defining a text index	When defining a range index
1	UNIQUE	Y	N	N
2	*index-name*	Y	Y	Y
3	ON *table-name*	Y	Y	Y
4	*column-name*	Y	Y	Y
5	{ASC|DESC}	Y	N	N
6	IN *DB-area-name*	Y	Y	Y
7	PCTFREE	Y	Y	N
8	EMPTY	Y	Y	Y
9	INDEXTYPE	Y	Y	Y
10	CORRECTIONRULE	N	Y	N
11	DELIMITER	N	Y	N
12	EXCLUDE NULL VALUES	Y	N	N

Legend:

Y: An option that can be specified, or one that must be specified.

N: An option that cannot be specified.

Note

It is not possible to define a primary key using the CREATE INDEX statement. To define a primary key, specify a uniqueness constraint definition using the CREATE TABLE statement.

(2) Explanation of specification format

In the option descriptions, options marked [B-tree index] can be specified during definition of a B-tree index. Options marked [Text index] can be specified during definition of a text index. Options marked [Range index] can be specified during definition of a range index. Options marked [Common] are common to B-tree indexes, text indexes, and range indexes.

- UNIQUE [B-tree index]

  Specify this if you want the B-tree index to be a unique index. A unique index is a B-tree index that does not allow duplicate key values (the values of the columns on which the B-tree index is being defined). However, if the key values can include null values, duplicate null values do not result in duplicate keys.

  For a multiple-column index, a key value is considered different if its value in any one of the columns is different.

  If UNIQUE is specified, you cannot update or add data that would result in a duplicate key value.

  Note that you cannot define a unique index for a base table created by using the CREATE TABLE statement with chunk specification.

- *index-name* [Common]

  Specifies the name of the index to be defined. For rules on specifying an index name, see (3) Index name specification format in 6.1.5 Qualifying a name.

  Note that you cannot specify the index name of an index that has already been defined.

- ON *table-name* [Common]

  Specifies the name of the base table for which the index is to be defined. For rules on specifying a table name, see (2) Table name specification format in 6.1.5 Qualifying a name.

  Note that a viewed table cannot be specified in *table-name*.

- (*column-name* [{ASC|DESC}] [,*column-name* [{ASC|DESC}]]...) [Common]

  - **For a B-tree index**

    Specifies the names of the columns on which the B-tree index is being defined, and the ordering of the B-tree index's key values.

    column-name:

    Specifies the names of the columns on which the B-tree index is being defined. A maximum of 16 column names can be specified. If multiple column names are specified, each column name must be unique.

    If multiple column names are specified, the resulting B-tree index is a multiple-column index.

    ASC:

    Specifies that the B-tree index is to be organized in ascending order of the key values.

    DESC:

    Specifies that the B-tree index is to be organized in descending order of the key values.

    For a single-column index, DESC is ignored. The index's key values are always arranged in ascending order (ASC is assumed).

    If neither ASC nor DESC is specified, the system assumes that ASC is specified.

  - **For a text index or range index**

    Specifies the name of the column on which the text index or range index is being defined.

    In the case of a text index or range index, only one column name can be specified. In addition, ASC and DESC cannot be specified.

    Therefore, the specification format in the case of a text index or range index is as follows:

    (*column-name*)

- IN *DB-area-name* [Common]

  Specifies the name of the DB area in which the index is to be defined.

  If the IN *DB-area-name* specification is omitted, the index is stored in the DB area specified for the adb_sql_default_dbarea_shared operand in the server definition.

  Note that if the IN *DB-area-name* specification is omitted when either of the following conditions is met, the CREATE INDEX statement will result in an error:

    - Specification of the adb_sql_default_dbarea_shared operand is omitted in the server definition.
    - A non-existent DB area or a DB area other than a data DB area is specified for the adb_sql_default_dbarea_shared operand in the server definition.

- PCTFREE=*percentage-of-unused-area* [B-tree index] [Text index]

  ~ <unsigned integer> ((0 to 99)) <<30>> (unit: %)

  Specifies the percentage of unused area to maintain in the index page of a B-tree index or text index. Specify a percentage from 0 to 99. If omitted, 30 (%) is assumed.

  When data is imported and an index is created or when the index is rebuilt, the B-tree index data or text index data will be stored leaving the percentage of unused area specified here.

  For details about the percentage of unused area in an index page, see *Allocating an unused area inside a B-tree index page (PCTFREE)* or *Allocating an unused area inside a text index page (PCTFREE)* in the *HADB Setup and Operation Guide*.

  Note that PCTFREE cannot be specified more than once.

- EMPTY [Common]

  EMPTY must be specified. If EMPTY is omitted, the CREATE INDEX statement cannot be executed.

  EMPTY cannot be specified more than once.

- INDEXTYPE {BTREE|TEXT [WORDCONTEXT]|RANGE} [Common]

  Specifies the type of index to be defined.

  BTREE:

  Specify this if you want to define a B-tree index.

  TEXT [WORDCONTEXT]:

  Specify this if you want to define a text index. To define a text index for a word-context search, specify TEXT WORDCONTEXT.

  RANGE:

  Specify this if you want to define a range index.

  If specification of INDEXTYPE is omitted, the system assumes that BTREE (B-tree index) is specified.

  INDEXTYPE can only be specified once.

- CORRECTIONRULE [Text index]

  Specify this option when you define a text index that supports correction search. For details about the correction search in a text index, see *Correction search* in the *HADB Setup and Operation Guide*.

  Note that correction search cannot be used if the character encoding used on the HADB server is Shift-JIS (if the value of the ADBLANG environment variable is SJIS). In such a case, you cannot specify CORRECTIONRULE.

  Also note that you cannot specify the CORRECTIONRULE option more than once.

  > **Note**
  >
  > This option specification is referred to as the *notation-correction-search text-index specification*.

- DELIMITER {DEFAULT|ALL} [Text index]

  Specifies the group of characters that can be used as word delimiters during a word-context search.

  DEFAULT:

  Handles the following characters as delimiters during a word-context search:

    - Half-width space (0x20)

- Tab (`0x09`)
- Line break (`0x0A`)
- Return (`0x0D`)
- Period (`0x2E`)
- Question mark (`0x3F`)
- Exclamation mark (`0x21`)

`ALL`:

Handles the following characters as delimiters during a word-context search:

- Half-width space (`0x20`)
- Tab (`0x09`)
- Line break (`0x0A`)
- Return (`0x0D`)
- Single-byte symbols including periods, question marks, and exclamation marks (`0x21` to `0x2F`, `0x3A` to `0x40`, `0x5B` to `0x60`, and `0x7B` to `0x7E`)

To specify this option, `TEXT WORDCONTEXT` must be specified for `INDEXTYPE`.

If specification of `DELIMITER` is omitted when `TEXT WORDCONTEXT` is specified for `INDEXTYPE`, the system assumes that `DEFAULT` is specified.

Note

The specification of this option is called *text-index delimiter specification*.

● `EXCLUDE NULL VALUES` [B-tree index]

If this option is specified and a B-tree index is created, no B-tree index key values that are composed of null values alone will be created. Consider specifying this option if you want to index columns in which most of the values are null.

Specifying this option can reduce the time it takes to create a B-tree index, because no B-tree index key values composed of null values alone will be created. Among other benefits, this can reduce the time it takes to import data and reduce the amount data required for the B-tree index.

Note that you cannot specify this option for a B-tree index that is defined on columns on which the `NOT NULL` constraint is defined.

Also note that you cannot specify the `EXCLUDE NULL VALUES` option more than once.

Note

This option is referred to as the *null-value exclusion specification*.

(3) Privileges required at execution

To execute the `CREATE INDEX` statement, the `CONNECT` privilege and schema definition privilege are required.

(4) Rules

(a) Common rules for indexes

1. An index can only be defined for a base table owned by the current user (the HADB user whose authorization identifier is currently connected to the HADB server). You cannot define an index for a base table owned by another HADB user.
2. Indexes cannot be defined on viewed tables.
3. A maximum combined total of 32 B-tree, text, and range indexes can be created for one table.
4. A maximum combined total of 8,192 B-tree, text, and range indexes can be defined in the system (excluding indexes defined for the base tables of dictionary tables and system tables).
5. A maximum of 400 indexes can be stored in one DB area.
6. The same column can have B-tree indexes (single-column indexes), text indexes, and range indexes defined on it.
7. To define an index for a multi-chunk table, see *Points to consider in storing a multi-chunk table in the data DB area* in the *HADB Setup and Operation Guide*.
8. You cannot define an index for a table that has become non-updatable due to interruption of a command.

(b) Rules for B-tree indexes

1. When a single-column index is defined, it must satisfy the formula below. You cannot define a single-column index that does not satisfy this formula.

   ```
   size-of-column-that-comprises-single-column-index ≤ MIN{(a ÷ 3) - 128, 4036 } (unit: bytes)
   ```

   a: Page size of the DB area where the B-tree index is to be stored

   The size of a column that comprises a single-column index can be calculated by using the information in the following table.

   Table 3-5: Size of a column that comprises a single-column index

No.	Data type of the column		Size of the column (unit: bytes)
1	INTEGER		8
2	SMALLINT		4
3	DECIMAL(*m*,*n*)	If 1 ≤ *m* ≤ 4	2
		If 5 ≤ *m* ≤ 8	4
		If 9 ≤ *m* ≤ 16	8
		If 17 ≤ *m* ≤ 38	16
4	DOUBLE PRECISION		8
5	CHAR(*n*)		*n*
6	VARCHAR(*n*)		*n*
7	DATE		4
8	TIME(*p*)		3 + ↑*p* ÷ 2↑
9	TIMESTAMP(*p*)		7 + ↑*p* ÷ 2↑
10	BINARY(*n*)		*n*

No.	Data type of the column	Size of the column (unit: bytes)
11	`VARBINARY(`*n*`)`	*n*

Legend:

m, *n*: Positive integers

p: `0`, `3`, `6`, `9`, or `12`

2. To define a multiple-column index, the following conditional expression must be satisfied. You cannot define a multiple-column index that does not satisfy this formula.

```
total-size-of-columns-that-comprise-multiple-column-index ≤ MIN{(a ÷ 3) - 128, 403
6 } (unit: bytes)
```

a: Page size of the DB area where the B-tree index is to be stored

To obtain the total size of the columns that comprise a multiple-column index, see the following table.

Table 3-6: Size of columns that comprise a multiple-column index

No.	Data type of a column		Size of the columns that comprise a multiple-column index (unit: bytes)#		
			If the total defined size of all columns does not exceed 255 bytes	If the total defined size of all columns exceeds 255 bytes	
				If only fixed size columns are included	If variable size columns are also included
1	`INTEGER`		9	9	10
2	`SMALLINT`		5	5	6
3	`DECIMAL(`*m*`,`*n*`)`	If 1 ≤ *m* ≤ 4	3	3	4
		If 5 ≤ *m* ≤ 8	5	5	6
		If 9 ≤ *m* ≤ 16	9	9	10
		If 17 ≤ *m* ≤ 38	17	17	18
4	`DOUBLE PRECISION`		9	9	10
5	`CHARACTER(`*n*`)`		*n* + 1	*n* + 1	*n* + 2
6	`VARCHAR(`*n*`)`		*n* + 1	--	*n* + 2
7	`DATE`		5	5	6
8	`TIME(`*p*`)`		4 + ↑*p* ÷ 2↑	4 + ↑*p* ÷ 2↑	5 + ↑*p* ÷ 2↑
9	`TIMESTAMP(`*p*`)`		8 + ↑*p* ÷ 2↑	8 + ↑*p* ÷ 2↑	9 + ↑*p* ÷ 2↑
10	`BINARY(`*n*`)`		*n* + 1	*n* + 1	*n* + 2
11	`VARBINARY(`*n*`)`		*n* + 1	--	*n* + 2

Legend:

m, *n*: Positive integers

p: `0`, `3`, `6`, `9`, or `12`

--: Not applicable

#

If the result calculated based on the formulas under *If the total defined size of all columns does not exceed 255 bytes* yields a total that exceeds 255 bytes, re-calculate the sizes of the columns using the formulas under *If the total defined size of all columns exceeds 255 bytes*.

3. You cannot define more than one of the following kinds of B-tree indexes:
   - B-tree indexes that have the same column structure, and where the same ascending or descending order is specified for all columns.
   - B-tree indexes that have the same column structure, but where the opposite ascending or descending order is specified for all columns.
4. A column on which a single-column index is defined can be specified when defining a multiple-column index.
5. When a multiple-column index is defined, the order in which the columns are specified determines the order of precedence for creating key values.

(c) Rules for text indexes

1. Text indexes can be defined on columns of the following data types:
   - `CHARACTER` types
   - `VARCHAR` types
2. You cannot define multiple text indexes with the same indexed columns.
3. You cannot define a text index for column store tables.

(d) Rules for range indexes

1. Range indexes cannot be defined on columns of the following data types:
   - `CHARACTER` types whose length exceeds 32 bytes
   - `VARCHAR` types
   - `BINARY` types
   - `VARBINARY` types
2. You cannot define multiple range indexes with the same indexed columns.

(5) Examples

Example 1: Define a B-tree index

Define a B-tree index for the shops table (`SHOPSLIST`) as follows:

- Define a single-column index (`SHOP_CODE_IDX`) on the shop code column (`SHOP_CODE`).
- Make the B-tree index a unique index.
- Store the B-tree index in the DB area `DBAREA01`.
- Because rows are added frequently to the shops table (`SHOPSLIST`), let the percentage of unused area in an index page be 50 percent.

```
CREATE UNIQUE INDEX "SHOP_CODE_IDX"
          ON "SHOPSLIST" ("SHOP_CODE")
        IN "DBAREA01"
        PCTFREE = 50
        EMPTY
```

Example 2: Define a B-tree index

Define a B-tree index for the shops table (SHOPSLIST) as follows:

- Define a multiple-column index (SHOP_RGN_IDX) with the shop code column (SHOP_CODE) and the region code column (RGN_CODE) as the indexed columns.
- Sort the key values of the index in ascending order (ASC) for the shop code, and in descending order (DESC) for the region code.
- Store the B-tree index in the DB area DBAREA01.

```
CREATE INDEX "SHOP_RGN_IDX"
          ON "SHOPSLIST" ("SHOP_CODE" ASC,"RGN_CODE" DESC)
        IN "DBAREA01"
        EMPTY
```

Example 3: Define a text index

Define a text index for the employee table (EMPLOYEE) as follows:

- Define a text index (ADDRESS_IDX) on the address column (ADDRESS).

```
CREATE INDEX "ADDRESS_IDX"
          ON "EMPLOYEE" ("ADDRESS")
        IN "DBAREA01"
        EMPTY
        INDEXTYPE TEXT
```

If you want the text index to support correction search, define the text index as follows. In this example, the underlined option is added.

```
CREATE INDEX "ADDRESS_IDX"
      ON "EMPLOYEE" ("ADDRESS")
    IN "DBAREA01"
    EMPTY
    INDEXTYPE TEXT
    CORRECTIONRULE
```

If you want to define the text index for a word-context search, define the text index as follows. In this example, the underlined options are added.

```
CREATE INDEX "ADDRESS_IDX"
      ON "EMPLOYEE"("ADDRESS")
    IN "DBAREA01"
    EMPTY
    INDEXTYPE TEXT WORDCONTEXT
    DELIMITER DEFAULT
```

Example 4: Define a range index

Define a range index for the shops table (SHOPSLIST) as follows:

- Define a range index (SHOP_CODE_RIDX) on the shop code column (SHOP_CODE)
- Store the range index in the DB area DBAREA01.

```
CREATE INDEX "SHOP_CODE_RIDX"
          ON "SHOPSLIST" ("SHOP_CODE")
        IN "DBAREA01"
        EMPTY
        INDEXTYPE RANGE
```

3.6 CREATE SCHEMA (define a schema)

This section describes the specification format and rules for the CREATE SCHEMA statement.

3.6.1 Specification format and rules for the CREATE SCHEMA statement

The CREATE SCHEMA statement defines a schema.

(1) Specification format

```
CREATE-SCHEMA-statement ::= CREATE SCHEMA [schema-name]
```

(2) Explanation of specification format

- *schema-name*

  Specifies the name of the schema to be defined. In *schema-name*, specify the authorization identifier of the current user (the HADB user whose authorization identifier is currently connected to the HADB server).

  If the schema name is omitted, the assumed value is the authorization identifier of the HADB user who executed the CREATE SCHEMA statement.

  For rules on specifying a schema name, see (1) Schema name specification format in 6.1.5 Qualifying a name.

  Note that you cannot specify ALL, HADB, MASTER, or PUBLIC for *schema-name*.

(3) Privileges required at execution

To execute the CREATE SCHEMA statement, the CONNECT privilege and schema definition privilege are required.

(4) Rules

1. Each HADB user can own only one schema.
2. You can only define a schema for the current user (the HADB user whose authorization identifier is currently connected to the HADB server). You cannot define a schema for another HADB user. For example, if the adbsql command is executed with ADBUSER01 specified as the authorization identifier, schema ADBUSER01 is the only schema that can be defined with CREATE SCHEMA.

(5) Examples

Example

Define a schema with the schema name ADBUSER01.

```
CREATE SCHEMA "ADBUSER01"
```

3.7 CREATE TABLE (define a table)

This section describes the specification format and rules for the `CREATE TABLE` statement.

3.7.1 Specification format and rules for the CREATE TABLE statement

The `CREATE TABLE` statement defines a base table.

(1) Specification format

```
CREATE-TABLE-statement ::=
     CREATE [FIX] TABLE table-name(table-element[,table-element]...)
       [IN DB-area-name]
       [PCTFREE=percentage-of-unused-area]#
       [BRANCH ALL]#
       [chunk-specification]#
       [STORAGE FORMAT {ROW|COLUMN}]#


  table-element ::= {column-definition|table-constraint}

    column-definition ::= column-name data-type [DEFAULT-clause] [NOT NULL] [BRANCH {
YES|NO|AUTO}]
                             [compression-type-specification]
      DEFAULT-clause ::= DEFAULT default-option
        default-option ::= {literal|CURRENT_DATE|CURRENT_TIME[(p)]
                          |CURRENT_TIMESTAMP[(p)]|CURRENT_USER|NULL}

      compression-type-specification ::= COMPRESSION TYPE {AUTO|NONE|RUNLENGTH|DICTIO
NARY|DELTA
                                          |DELTA_RUNLENGTH}


    table-constraint ::= {uniqueness-constraint-definition|referential-constraint-def
inition}
      uniqueness-constraint-definition ::= [CONSTRAINT constraint-name] PRIMARY KEY (
column-name [{ASC|DESC}]
                                [,column-name [{ASC|DESC}]]...)
                                [IN DB-area-name]
                                [PCTFREE=percentage-of-unused-area]

      referential-constraint-definition ::= [CONSTRAINT constraint-name] FOREIGN KEY
(column-name[,column-name]...)
                                 REFERENCES table-name DISABLE


  chunk-specification ::= CHUNK[=maximum-number-of-chunks]
                         [chunk-archive-specification]
    chunk-archive-specification ::= ARCHIVABLE
                                  RANGECOLUMN=column-name
                                 [RANGEINDEXNAME=index-identifier]
                                 [IN DB-area-name]
                                  ARCHIVEDIR=archive-directory-name
```

#

`PCTFREE`, `BRANCH ALL`, *chunk-specification*, and `STORAGE FORMAT` can be specified in any order.

Note

`PCTFREE`, `BRANCH ALL`, *chunk-specification*, and `STORAGE FORMAT` are referred to collectively as *table options*.

The following table lists the options for defining row store tables and the options for defining column store tables.

Table 3-7: Options for defining row store tables or column store tables

No.	CREATE TABLE option		For defining row store tables	For defining column store tables
1	`FIX`		Y	N
2	*table-name*		Y	Y
3	*column-definition*	*column-name*	Y	Y
4		*data-type*	Y	Y
5		*DEFAULT-clause*	Y	Y
6		`NOT NULL`	Y	Y
7		`BRANCH`	Y	N
8		*compression-type-specification*	N	Y
9	*table-constraint*		Y	Y
10	`IN` *DB-area-name*		Y	Y
11	`PCTFREE`		Y	N
12	`BRANCH ALL`		Y	N
13	*chunk-specification*	*maximum-number-of-chunks*	Y	Y
14		*chunk-archive-specification*	Y	N
15	`STORAGE FORMAT`		Y	Y

Legend:

Y: An option that can be specified, or one that must be specified.

N: An option that cannot be specified.

(2) Explanation of specification format

In the option descriptions, options marked [Row store table] can be specified to define a row store table. Options marked [Column store table] can be specified to define a column store table. Options marked [Common] can be specified to define both a row store table and a column store table.

(a) FIX [Row store table]

Defines a base table in which every row has a fixed length (a *FIX table*).

The following rules apply:

- If `FIX` is specified, the following data type cannot be specified for any row in this table:
    - `VARCHAR`

  - VARBINARY

- If FIX is specified, NOT NULL constraint is set for all columns of the base table.
- Only FIX tables allow you to perform reference, update, and insert operations by row (ROW specification).
- For archivable multi-chunk tables, FIX cannot be specified.

(b) table-name [Common]

Specifies the name of the base table to be defined. You cannot specify the table name of a table that has already been defined. For rules on specifying a table name, see (2) Table name specification format in 6.1.5 Qualifying a name.

(c) table-element [Common]

```
table-element ::= {column-definition | table-constraint}
```

A table element specifies either a column definition or a table constraint.

(d) column-definition [Common]

```
column-definition ::= column-name data-type [DEFAULT-clause] [NOT NULL] [BRANCH {YES
| NO | AUTO}]
                                  [compression-type-specification]
```

Specifies the definitions of the columns that make up the base table. At least one column definition must be specified.

● *column-name* [Common]

Specifies the names of the columns that comprise the table. Each column name must be unique.

Do not specify a character string in the EXP*nnnn*_NO_NAME format as a column name. Such a column name might duplicate a derived column name that is automatically set by HADB. In this format, *nnnn* is an unsigned integer in the range from 0000 to 9999.

● *data-type* [Common]

Specifies the data types of the columns. The following table lists the data types that can be specified.

Table 3-8: Data types that can be specified (CREATE TABLE statement)

No.	Data type	Specification format
1	INTEGER	INT or INTEGER
2	SMALLINT	SMALLINT
3	DECIMAL	DEC[(*m*[,*n*])] or DECIMAL[(*m*[,*n*])] *m*: Precision (total number of digits) *n*: Scaling (number of decimal places) If *m* is omitted, 38 is assumed, and if *n* is omitted, 0 is assumed.
4	DOUBLE PRECISION	DOUBLE or DOUBLE PRECISION
5	CHARACTER	CHAR(*n*) or CHARACTER(*n*) *n*: Length of character string (in bytes) If CHAR or CHARACTER is specified without a length, the length of the character string is assumed to be 1.
6	VARCHAR	VARCHAR(*n*) *n*: Maximum length of character string (in bytes)

No.	Data type	Specification format
7	DATE	DATE
8	TIME	TIME(*p*) or TIME *p*: Fractional seconds precision (number of digits to the right of the decimal point) You can specify a value of 0, 3, 6, 9, or 12 for *p*. If TIME is specified, *p* is assumed to be 0.
9	TIMESTAMP	TIMESTAMP(*p*) or TIMESTAMP *p*: Fractional seconds precision (number of digits to the right of the decimal point) You can specify a value of 0, 3, 6, 9, or 12 for *p*. If TIMESTAMP is specified, *p* is assumed to be 0.
10	BINARY	BINARY(*n*) *n*: Length of the binary data (number of bytes) (number of bytes) If BINARY is specified without a length, the length of the binary data is assumed to be 1.
11	VARBINARY	VARBINARY(*n*) *n*: Maximum length of the binary data (number of bytes)

For details about data types, see 6.2 Data types.

Important

A VARCHAR-type column whose defined length exceeds 32,000 bytes cannot be specified.

- *DEFAULT-clause* [Common]

```
DEFAULT-clause ::= DEFAULT default-option
  default-option ::= {literal|CURRENT_DATE|CURRENT_TIME[(p)]
                    |CURRENT_TIMESTAMP[(p)]|CURRENT_USER|NULL}
```

Specify a DEFAULT clause when you want to set a default value for a column.

For details about the specification format of the DEFAULT clause and the default values for columns, see 7.10 DEFAULT clause.

- NOT NULL [Common]

Specify this to define the NOT NULL constraint (the constraint to not allow null values) for the column.

You cannot specify NULL for *default-option* in the DEFAULT clause for columns on which the NOT NULL constraint is specified.

- BRANCH {YES|NO|AUTO} [Row store table]

Specifies how to store VARCHAR-type and VARBINARY-type column data.

For details about situations for which it is better to specify YES or NO for BRANCH, see *Branch specification for column data of variable-length data types (BRANCH) [Row store table]* in the *HADB Setup and Operation Guide*.

YES:

Branch the specified VARCHAR-type or VARBINARY-type column data.

NO:

Do not branch the specified VARCHAR-type or VARBINARY-type column data.

AUTO:

Do not branch if the defined length of the specified VARCHAR-type or VARBINARY-type column data is less than or equal to 255 bytes. If the defined length is 256 bytes or greater, branch when the base row does not fit on one page.

If the BRANCH specification is omitted, the system assumes that AUTO is specified.

This option cannot be specified for the following tables and columns:

- Tables for which the BRANCH ALL table option is specified
- Columns of data types other than VARCHAR and VARBINARY

● *compression-type-specification* [Column store table]

```
compression-type-specification ::= COMPRESSION TYPE {AUTO|NONE|RUNLENGTH|DICTIONAR
Y|DELTA|DELTA_RUNLENGTH}
```

Specifies the compression type to be used to compress the column data in a column store table (column-data compression type). This option can be specified for each column in the column store table.

If *compression-type-specification* is not specified, the system assumes that AUTO is specified.

AUTO:

If this type is specified, the HADB server automatically determines the data compression type of this column in the column store table.

NONE:

If this type is specified, the data in this column of the column store table is not compressed.

RUNLENGTH:

If this type is specified, the data in this column of the column store table is compressed by using the run-length encoding algorithm.

DICTIONARY:

If this type is specified, the data in this column of the column store table is compressed by using the dictionary encoding algorithm.

DELTA:

If this type is specified, the data in this column of the column store table is compressed by using the delta encoding algorithm.

DELTA_RUNLENGTH:

If this type is specified, the data in this column of the column store table is compressed by using the delta run-length encoding algorithm.

For details about each compression type, see *Column-data compression types for column store tables* in *Criteria for selecting row store tables and column store tables* in the *HADB Setup and Operation Guide*.

(e) table-constraint [Common]

```
table-constraint ::= {uniqueness-constraint-definition | referential-constraint-defin
ition}
```

For the table constraint, specify a uniqueness constraint definition or a referential constraint definition.

(f) uniqueness-constraint-definition [Common]

```
uniqueness-constraint-definition ::= [CONSTRAINT constraint-name] PRIMARY KEY (column
-name [{ASC|DESC}]
                          [,column-name [{ASC|DESC}]]...)
                          [IN DB-area-name]
                          [PCTFREE=percentage-of-unused-area]
```

Specify this if you want to define a primary key for the base table. Only one primary key can be defined for each table.

The uniqueness constraint and NOT NULL constraint are applied to the columns that make up the primary key. The *uniqueness constraint* disallows duplicate column values (or duplicate combinations of values from multiple columns). The *NOT NULL constraint* disallows the null value as a column value.

The primary key must be selected from among the columns or combinations of columns that can uniquely identify a row in the table (candidate keys).

> **Note**
>
> The columns or combinations of columns that can uniquely identify a row in the table are referred to as *candidate keys*.

- CONSTRAINT *constraint-name* [Common]

  Specifies a name for the uniqueness constraint definition specified here. For rules on specifying a constraint name, see (2) Rules for characters that can be used in names in 6.1.4 Specifying names.

  Note the following rules:

  - If the same constraint name (including referential constraint names) already exists in the same schema, executing the CREATE TABLE statement will result in an error.
  - If this specification is omitted, a name of the following form is generated as the constraint name:

    PRIMARY_*nnnnnnnn*

    nnnnnnnn: The table ID of the table for which the primary key is to be defined, converted to a character string of eight hexadecimal digits

    If a constraint with the same name as the generated name already exists in the same schema, executing the CREATE TABLE statement will result in an error. It is therefore recommended that you avoid the above format when specifying constraint names (including referential constraint names) or index identifiers.

- *column-name* [Common]

  Specify the names of the columns that are to make up the primary key. Note the following rules:

  - A maximum of 16 column names can be specified.
  - If multiple column names are specified, each must be unique.

  When the CREATE TABLE statement is executed, it automatically defines a B-tree index as a unique index consisting of the specified columns. This B-tree index is subject to the following rules:

  - If only one column name is specified, a single-column index is defined.
  - If multiple column names are specified, a multiple-column index is defined.
  - The index identifier will be the same as the constraint name.
  - If the index identifier, which is the same as the constraint name, already exists in the same schema, executing the CREATE TABLE statement will result in an error.

  > **Note**
  >
  > The B-tree index that is automatically defined here is subject to the same rules as a B-tree index defined by the CREATE INDEX statement.

- {ASC|DESC} [Common]

  Specifies the sort order of the key values of the B-tree index corresponding to the primary key.

ASC:

Specify this if you want the key values of the B-tree index corresponding to the primary key to be sorted in ascending order.

DESC:

Specify this if you want the key values of the B-tree index corresponding to the primary key to be sorted in descending order.

If you specify DESC for a single-column index, it will be ignored. The key values of a single-column index are always sorted in ascending order (it is assumed that ASC is specified).

If neither ASC nor DESC is specified, the system assumes that ASC is specified.

● IN *DB-area-name* [Common]

Specify the name of the DB area in which to store the B-tree index corresponding to the primary key.

If the IN *DB-area-name* specification is omitted, the B-tree index for the primary key is stored in the DB area specified for the adb_sql_default_dbarea_shared operand in the server definition.

Note that if the IN *DB-area-name* specification is omitted when either of the following conditions is met, the CREATE TABLE statement will result in an error:

- Specification of the adb_sql_default_dbarea_shared operand is omitted in the server definition.
- A non-existent DB area or a DB area other than a data DB area is specified for the adb_sql_default_dbarea_shared operand in the server definition.

● PCTFREE=*percentage-of-unused-area* [Common]

~ <unsigned integer> ((0 to 99)) <<30>> (unit: %)

Specifies the percentage of unused area to maintain in the index pages of the B-tree index corresponding to the primary key. Specify a percentage from 0 to 99. If omitted, 30% is assumed.

When data is imported and the B-tree index for the primary key is created, the B-tree index data will be stored leaving the percentage of unused area specified here. The B-Tree index data will also be stored in this way when the B-tree index for the primary key is rebuilt.

For details about the percentage of unused area in an index page, see *Allocating an unused area inside a B-tree index page (PCTFREE)* or *Allocating an unused area inside a text index page (PCTFREE)* in the *HADB Setup and Operation Guide*.

(g) referential-constraint-definition [Common]

```
referential-constraint-definition ::= [CONSTRAINT constraint-name] FOREIGN KEY (colum
n-name[,column-name]...)
                              REFERENCES table-name DISABLE
```

Specify this if you want to define a referential constraint (foreign key) for the base table. The foreign key can be defined as a column (or combination of multiple columns) that references the primary key of another table.

For more information about the benefits of defining a foreign key, see *Specifying a foreign key (FOREIGN KEY)* in the *HADB Setup and Operation Guide*.

Note

The columns that make up the foreign key and the columns that make up the primary key must be the same in all of the following respects:

- The number of columns

- The data type of each column
- The data length of each column

Note the following rules:

- A maximum of 255 foreign keys can be defined for one table.
- A maximum of 255 foreign keys can be defined that reference one primary key.
- You cannot define multiple referential constraints that reference the same primary key from the same foreign key. In this context, *the same foreign key* means a foreign key that satisfies the following condition:
  - The columns that make up the foreign key are the same

  A foreign key composed of multiple columns is considered the same foreign key even when the order of columns differs from the order in the definition.

● `CONSTRAINT` *constraint-name* [Common]

Specifies a name for the referential constraint definition specified here. For rules on specifying a constraint name, see (2) Rules for characters that can be used in names in 6.1.4 Specifying names.

Note the following rules:

- If the same constraint name (including uniqueness constraint names) exists in the same schema, executing the `CREATE TABLE` statement will result in an error.
- If this specification is omitted, a name of the following form is generated as the constraint name:

  `FOREIGN_`*nnnnnnnn*`_`*YYYYMMDDhhmmss*`th`

  nnnnnnnn: The table ID of the table for which the foreign key is to be defined, converted to a character string of eight hexadecimal digits

  YYYYMMDDhhmmss`th`: The time stamp when the foreign key was defined (output to the hundredth of a second)

  If a constraint with the same name as the generated name exists in the same schema, executing the `CREATE TABLE` statement will result in an error. It is therefore recommended that you avoid the above format when specifying constraint names (including uniqueness constraint names) or index identifiers.

● *column-name* [Common]

Specifies the names of the columns that are to make up the foreign key.

Note the following rules:

- A maximum of 16 column names can be specified.
- If multiple column names are specified, each must be unique.

● *table-name* [Common]

Specifies the name of the referenced table (the base table where the primary key is defined).

Note the following rules:

- The referenced table cannot be the table where the foreign key is defined.

● `DISABLE` [Common]

This option (referential constraint check suppression) specifies to suppress checking of the foreign key referential constraint.

This option must be specified. If it is not specified, the `CREATE TABLE` statement will result in an error.

(h) IN DB-area-name [Common]

Specifies the name of the DB area where the table is to be stored.

If the IN *DB-area-name* specification is omitted, the table is stored in the DB area specified for the adb_sql_default_dbarea_shared operand in the server definition.

Note that if the IN *DB-area-name* specification is omitted when either of the following conditions is met, the CREATE TABLE statement will result in an error:

- Specification of the adb_sql_default_dbarea_shared operand is omitted in the server definition.
- A non-existent DB area or a DB area other than a data DB area is specified for the adb_sql_default_dbarea_shared operand in the server definition.

(i) PCTFREE=percentage-of-unused-area [Row store table]

~ <unsigned integer> ((0 to 99)) <<30>> (unit: %)

Specifies the percentage of unused area to maintain in a data page (the pages that store the data for the table). Specify a percentage from 0 to 99. If omitted, 30 (%) is assumed.

When data is imported, the data will be stored leaving the percentage of unused area specified here.

Note that when you add a row with an INSERT statement, or update a row with an UPDATE statement, the percentage of unused area specified here does not apply (the added or updated data is stored in the unused area reserved by this keyword).

For details about the percentage of unused area in a data page, see *Allocating an unused area inside the data page (PCTFREE) [Row store table]* in the *HADB Setup and Operation Guide*.

Note that PCTFREE cannot be specified more than once.

(j) BRANCH ALL [Row store table]

Branch all the VARCHAR-type and VARBINARY-type column data defined in the table.

For information about cases where BRANCH ALL is appropriate, see *Branch specification for column data of variable-length data types (BRANCH) [Row store table]* in the *HADB Setup and Operation Guide*.

Note that you cannot specify BRANCH ALL when FIX is specified.

(k) chunk-specification [Common]

```
chunk-specification ::= CHUNK[=maximum-number-of-chunks]
                       [chunk-archive-specification]
  chunk-archive-specification ::= ARCHIVABLE
                                RANGECOLUMN=column-name
                               [RANGEINDEXNAME=index-identifier]
                               [IN DB-area-name]
                                ARCHIVEDIR=archive-directory-name
```

Use the preceding specification when you define a base table as a multi-chunk table.

For details about designing a multi-chunk table, see *Points to consider in defining a multi-chunk table* in the *HADB Setup and Operation Guide*.

The following table shows the relationship between the types of multi-chunk tables and the functions that can be used.

Table 3-9: Types of multi-chunk tables and the functions that can be used

Function name	Multi-chunk table type	
	Regular multi-chunk table	Archivable multi-chunk table
Background-import facility	Y	Y
Deleting all rows in a chunk by using the `PURGE CHUNK` statement	Y	Y
Chunk archiving function	N	Y

Legend:

Y: Can be used.

N: Cannot be used.

For details about the background-import facility and chunk archiving function, see the following sections in the *HADB Setup and Operation Guide*: *Background-import facility* and *Chunk archiving function (compressing data in a chunk)*.

- `CHUNK[=`*maximum-number-of-chunks*`]` [Common]

  ~ <unsigned integer> ((`2` to `30,000`)) <<`256`>> (unit: chunks)

  Specify the maximum number of chunks in a multi-chunk table. If you specify only the keyword `CHUNK` without specifying a value (*maximum-number-of-chunks*), the system assumes that `256` is set as *maximum-number-of-chunks*.

  If you define a regular multi-chunk table, specify only `CHUNK=`*maximum-number-of-chunks*.

  If you define an archivable multi-chunk table, specify `CHUNK=`*maximum-number-of-chunks* and the chunk-archive specification described later.

- *chunk-archive-specification* [Row store table]

```
chunk-archive-specification ::= ARCHIVABLE
                               RANGECOLUMN=column-name
                              [RANGEINDEXNAME=index-identifier]
                              [IN DB-area-name]
                               ARCHIVEDIR=archive-directory-name
```

  Use the preceding specification when you define a base table as an archivable multi-chunk table.

  - `RANGECOLUMN=`*column-name*

    Specify a column name. The column specified here becomes the archive range column.

    You cannot use columns that have the following data types as archive range columns:

    - `CHARACTER` type (only if the defined length is 33 bytes or more)
    - `VARCHAR` type
    - `BINARY` type
    - `VARBINARY` type

    Note that NOT NULL constraint is set for archive range columns.

  - `RANGEINDEXNAME=`*index-identifier*

    When the `CREATE TABLE` statement is run, the HADB server automatically defines a range index whose columns include an archive range column. Specify the index identifier to be assigned to this range index.

    If `RANGEINDEXNAME` is not specified, the HADB server determines the index identifier in the following format:

```
ARCHIVE_RANGE_INDEX_nnnnnnnn
```

nnnnnnnn is an eight-digit character string that represents the ID of the archivable multi-chunk table in hexadecimal notation.

If the automatically determined index identifier already exists in the same schema, the `CREATE TABLE` statement will result in an error. Therefore, if you use the `CREATE INDEX` statement to define an index, we recommend that you do not use a name whose format resembles the preceding format.

> **Note**
>
> The range index that is automatically defined here is subject to the same rules as a range index defined by the `CREATE INDEX` statement.

- `IN` *DB-area-name*

  Specify the name of the DB area in which to store the range indexes that are automatically defined.

  If the `IN` *DB-area-name* specification is omitted, automatically defined range indexes are stored in the DB area specified for the `adb_sql_default_dbarea_shared` operand in the server definition.

  Note that if the `IN` *DB-area-name* specification is omitted when either of the following conditions is met, the `CREATE TABLE` statement will result in an error:

  - Specification of the `adb_sql_default_dbarea_shared` operand is omitted in the server definition.
  - A non-existent DB area is specified for the `adb_sql_default_dbarea_shared` operand in the server definition.

- `ARCHIVEDIR=`*archive-directory-name*

  Specify the absolute path name of the archive directory in which to store archive files.

  The following rules apply:

  - Specify the archive directory name in the character string literal format. For details about character string literals, see 6.3 Literals.
  - Specify an existent directory for the archive directory. Make sure that read, write, and execution permissions for the HADB administrator are set on the directory that you specify.

    Also, make sure that execution permission for the HADB administrator is set on all directories that are included in the path of the archive directory.

    (Example) If the archive directory is `/HADB/archive`:

    For the `/HADB/archive` directory, read, write, and execution permissions must be set.

    For the `/` directory and the `/HADB` directory, the execution permission is necessary.

  - The following directories cannot be used as the archive directory:

    - Server directory
    - Subdirectory of a server directory
    - Directory that contains a server directory
    - DB directory
    - Subdirectory of a DB directory
    - Directory that contains a DB directory
    - Root directory

    The following shows examples of directories that can be and cannot be used as the archive directory when the DB directory is `/HADB/db`:

Directory		Reason
Example of directory that can be used as the archive directory	`/HADB/archive`	None.
Example of directory that cannot be used as the archive directory	`/HADB/db`	This directory is the same as the DB directory.
	`/HADB/db/archive`	This directory is a subdirectory of the DB directory.
	`/HADB`	This directory contains the DB directory.

- Do not specify (as the archive directory) a directory in which installation data was stored when the HADB server was installed.
- The name of the archive directory must be 1 to 400 bytes long except the heading and trailing spaces.

> **Note**
>
> If you specify a directory name that begins and/or ends with spaces, these spaces are deleted (the resulting character string is used as the archive directory name).

- Make sure that each element of the archive directory name is no more than `NAME_MAX` bytes long. The `NAME_MAX` value differs depending on the environment.

If a symbolic link is specified as the archive directory name, the system checks whether the absolute path name that the symbolic link substitutes for obeys the rules that are described here.

About the multi-node function:

If the multi-node function is enabled, note the following points:

- Use the NFS or other means to share the archive directory by all nodes. Note that it must have been shared by all nodes when the `CREATE TABLE` statement is run.
- On the master node, when the `CREATE TABLE` statement is run, a check to see whether the archive directory name obeys the specification rules that are described here is conducted. This check is not conducted on the slave nodes. Therefore, after the `CREATE TABLE` statement, check the archive directory name on each slave node.

About the location table that is defined when an archivable multi-chunk table is defined

If an archivable multi-chunk table is defined by running the `CREATE TABLE` statement, the HADB server automatically defines the location table and the index of the location table. The HADB server uses the location table and index. Therefore, no user can directly manipulate, redefine, or delete the location table or index. For details about the location table, see *Searching an archivable multi-chunk table* in the *HADB Setup and Operation Guide*.

The location table and its index are stored in the same DB area as the archivable multi-chunk table.

The names of the location table and its index are determined according to the rules that are described in the following table.

Table 3-10: Naming rules for the location table and location table index

Item	Naming rule	Information managed by the index	Columns in the index
Location table	`"HADB"."LOCATION_TABLE_`*nnnnnnnn*`"`	--	--
Location table index	`"HADB"."LOCATION_INDEX_`*nnnnnnnn*`_CHUNK_ID"`	Manages the chunk ID of the chunk that corresponds to the archive file.	`CHUNK_ID`

Item	Naming rule	Information managed by the index	Columns in the index
	`"HADB"."LOCATION_INDEX_`*nnnnnnnn*`_RANGE_01"`	Manages the range (upper and lower limits) of values in the archive range column of data stored in the archive file.	• `RANGE_MAX` • `RANGE_MIN`
	`"HADB"."LOCATION_INDEX_`*nnnnnnnn*`_RANGE_02"`	Manages the lower limit of values in the archive range column of data stored in the archive file.	`RANGE_MIN`

Legend:

--: Not applicable.

Note:

nnnnnnnn is an eight-digit character string that represents the ID of the archivable multi-chunk table in hexadecimal notation.

The schema name of the location table and location table index is HADB.

(l) STORAGE FORMAT {ROW|COLUMN} [Common]

Specifies the table-data storage format of a table to be defined.

`ROW`:

Specify this keyword when you define a table that has row store format as the table-data storage format. If `ROW` is specified, the table is defined as a row store table.

`COLUMN`:

Specify this keyword when you define a table that has column store format as the table-data storage format. If `COLUMN` is specified, the table is defined as a column store table.

If the `STORAGE FORMAT` specification is omitted, the system assumes that `ROW` is specified.

Note

- For details about the row store table, row store format, column store table, and column store format, see *Row store tables and column store tables* in the *HADB Setup and Operation Guide*.
- The specification of this option is called *table-storage-format specification*.

(3) Privileges required at execution

To execute the `CREATE TABLE` statement, the `CONNECT` privilege and schema definition privilege are required.

If you also want to specify a referential constraint (foreign key), the `REFERENCES` privilege on the referenced table is required.

(4) Rules

1. A base table can only be defined in the schema owned by the current user (the HADB user whose authorization identifier is currently connected to the HADB server). You cannot define a base table in a schema owned by another HADB user.

2. A maximum of 4,096 base tables can be defined in the system (excluding the base tables of dictionary tables and system tables).
3. A maximum of 200 base tables can be stored in one DB area.
4. A maximum of 1,000 columns can be defined in one table.
5. Columns must be defined such that the sum of the sizes of all columns in the base table (the row length) satisfies the following inequality:
   - If the base table is a row store table:

     ```
     ROWSZ (row-length) ≤ page-size - 56
     ```

   - If the base table is a column store table:

     ```
     ROWSZ (row-length) ≤ page-size - 80
     ```

   For details about the formula for calculating *ROWSZ* (row length), see *Determining the number of pages for storing each type of row* in the *HADB Setup and Operation Guide*.
6. If you use a chunk specification, you cannot define a primary key. Also, if you define a primary key, you cannot use a chunk specification.
7. You cannot define a primary key on a column on which a B-tree index cannot be defined.
8. You can define a primary key only when the following conditions are met. You cannot define a primary key unless the following conditions are met.
   - For a primary key consisting of one column

   ```
   Defined length of column in primary key#1 ≤ MIN{(a ÷ 3) - 128, 4,036} (unit: bytes)
   ```

   - For a primary key consisting of two or more columns

   ```
   Total defined length of columns in primary key#2 ≤ MIN{(a ÷ 3) - 128, 4,036} (unit:
    bytes)
   ```

   a: Page size of DB area where B-tree index corresponding to primary key is to be stored

   #1

   For details about the defined length of columns, see Table 3-5: Size of a column that comprises a single-column index.

   #2

   For details about the defined length of each column, see Table 3-6: Size of columns that comprise a multiple-column index. Then, obtain the total defined size of columns that comprise the primary key.
9. The tables defined after a transaction is started cannot be accessed from the transaction.

(5) Examples

Examples 1 to 6 are examples of defining a row store table. Example 7 is an example of defining a column store table.

Example 1: Define a base table that is not a FIX table

Define a shops table (SHOPSLIST). Let the shops table's column structure, percentage of unused area, and so on, be as follows:

- Shop code (SHOP_CODE): CHAR(8)
- Region code (RGN_CODE): CHAR(6)
- Shop name (SHOP_NAME): VARCHAR(20)

- Shop telephone number (TEL_NO): CHAR(10)
- Shop address (ADDRESS): VARCHAR(300)
- Define the NOT NULL constraint for every column.
- For storing, branch the data in the shop's address column (ADDRESS).
- Store the shops table in DB area DBAREA01.
- Let the percentage of unused area in a data page be 40%.
- Let the maximum number of chunks be 100.

```
CREATE TABLE "SHOPSLIST"
        ("SHOP_CODE" CHAR(8) NOT NULL,
         "RGN_CODE" CHAR(6) NOT NULL,
         "SHOP_NAME" VARCHAR(20) NOT NULL,
         "TEL_NO" CHAR(10) NOT NULL,
         "ADDRESS" VARCHAR(300) NOT NULL BRANCH YES)
      IN "DBAREA01"
      PCTFREE=40
      CHUNK=100
```

Example 2: Define a FIX table

Define a sales history table (SALESLIST). Let the sales history table's column structure, percentage of unused area, and so on, be as follows:

- Customer ID (USERID): CHAR(6)
- Product code (PUR-CODE): CHAR(4)
- Quantity purchased (PUR-NUM): SMALLINT
- Date of purchase (PUR-DATE): DATE
- Set a default column value for the date of purchase (PUR-DATE) column by specifying a DEFAULT clause.
- Store the sales history table in the DB area DBAREA01.
- Let the percentage of unused area in a data page be 20%.
- Specify FIX because the row length is fixed.
- Let the maximum number of chunks be 200.

```
CREATE FIX TABLE "SALESLIST"
        ("USERID" CHAR(6),
         "PUR-CODE" CHAR(4),
         "PUR-NUM" SMALLINT,
         "PUR-DATE" DATE DEFAULT CURRENT_DATE)
      IN "DBAREA01"
      PCTFREE=20
      CHUNK=200
```

Example 3: Define a base table with a primary key

Define a sales history table (SALESLIST). Let the sales history table's column structure, percentage of unused area, and so on, be as follows:

- Customer ID (USERID): CHAR(6)
- Product code (PUR-CODE): CHAR(4)
- Quantity purchased (PUR-NUM): SMALLINT
- Date of purchase (PUR-DATE): DATE

- Store the sales history table in the DB area DBAREA01.
- Let the percentage of unused area in a data page be 20%.
- Specify FIX because the row length is fixed.
- Define a primary key (let the customer ID column (USERID) be the column that comprises the primary key).
- Store the B-tree index corresponding to the primary key in the DB area DBAREA02.
- Set the percentage of unused area in the index page of the B-tree index corresponding to the primary key to 20%.

```
CREATE FIX TABLE "SALESLIST"
      ("USERID" CHAR(6),
       "PUR-CODE" CHAR(4),
       "PUR-NUM" SMALLINT,
       "PUR-DATE" DATE,
       CONSTRAINT "PK-USERID" PRIMARY KEY ("USERID" ASC)
            IN "DBAREA02" PCTFREE=20)
    IN "DBAREA01"
    PCTFREE=20
```

The underlined portion indicates the primary key definition (uniqueness constraint definition).

Example 4: Define a base table with a primary key

Define a shops table (SHOPSLIST). Let the shops table's column structure, percentage of unused area, and so on, be as follows:

- Shop code (SHOP_CODE): CHAR(8)
- Region code (RGN_CODE): CHAR(6)
- Shop name (SHOP_NAME): VARCHAR(20)
- Shop telephone number (TEL_NO): CHAR(10)
- Shop address (ADDRESS): VARCHAR(300)
- For storing, branch the data in the shop's address column (ADDRESS).
- Store the shops table in DB area DBAREA01.
- Let the percentage of unused area in a data page be 40%.
- Define a primary key (let the shop code column (SHOP_CODE) and the region code column (RGN_CODE) be the columns that comprise the primary key).
- Store the B-tree index corresponding to the primary key in the DB area DBAREA02.
- Set the percentage of unused area in the index page of the B-tree index corresponding to the primary key to 20%.

```
CREATE TABLE "SHOPSLIST"
      ("SHOP_CODE" CHAR(8),
       "RGN_CODE" CHAR(6),
       "SHOP_NAME" VARCHAR(20),
       "TEL_NO" CHAR(10),
       "ADDRESS" VARCHAR(300) BRANCH YES,
       CONSTRAINT "PK-CODE" PRIMARY KEY ("SHOP_CODE" ASC,"RGN_CODE" ASC)
            IN "DBAREA02" PCTFREE=20)
    IN "DBAREA01"
    PCTFREE=40
```

The underlined portion indicates the primary key definition (uniqueness constraint definition).

Example 5: Define a base table with a foreign key

Define a shops table (SHOPSLIST) and an employee table (EMPLOYEE). Define the primary key and foreign key as follows:

- Define the primary key on the shops table (SHOPSLIST). The primary key will consist of the SHOP_CODE column and the RGN_CODE column from the shops table (SHOPSLIST).
- Define the foreign key on the employee table (EMPLOYEE). The foreign key will consist of the SHOP_CODE column and the RGN_CODE column from the employee table (EMPLOYEE).

■ Shops table (SHOPSLIST)

```
CREATE TABLE "SHOPSLIST"
      ("SHOP_CODE" CHAR(8),
       "RGN_CODE" CHAR(6),
       "SHOP_NAME" VARCHAR(20),
       "TEL_NO"    CHAR(10),
       "ADDRESS"   VARCHAR(300) BRANCH YES,
       CONSTRAINT "PK-CODE" PRIMARY KEY ("SHOP_CODE" ASC,"RGN_CODE" ASC)
              IN "DBAREA02" PCTFREE=20)
    IN "DBAREA01"
    PCTFREE=40
```

The underlined portion indicates the primary key definition.

■ Employee table (EMPLOYEE)

```
CREATE TABLE "EMPLOYEE"
      ("EMPLOYEE_CODE"    CHAR(8),
       "FIRST_NAME"       VARCHAR(8),
       "FIRST_NAME_YOMI"  VARCHAR(16),
       "FAMILY_NAME"      VARCHAR(8),
       "FAMILY_NAME_YOMI" VARCHAR(16),
       "SHOP_CODE"        CHAR(8),
       "RGN_CODE"        CHAR(6),
       "EMPLOYEE_TYPE"    CHAR(1),
       "TEL_NO"           CHAR(10),
       "ADDRESS"          VARCHAR(300) BRANCH YES,
       CONSTRAINT "PK-EMPLOYEE_CODE" PRIMARY KEY ("EMPLOYEE_CODE" ASC)
             IN "DBAREA02" PCTFREE=20,
       CONSTRAINT "FK-SHOP_CODE" FOREIGN KEY ("SHOP_CODE","RGN_CODE")
             REFERENCES "SHOPSLIST" DISABLE)
    IN "DBAREA01"
    PCTFREE=40
```

The underlined portion indicates the foreign key (referential constraint) definition.

Example 6: Define an archivable multi-chunk table

Define a receipt table (RECEIPT) as an archivable multi-chunk table. Specify the chunk-related settings under the following conditions:

- Let the maximum number of chunks be 120.
- The RECORD_DAY column is used as the archive range column.
- The /mnt/nfs/archivedir directory is used as the archive directory.

```
CREATE TABLE "RECEIPT"
      ("RID" INTEGER,
       "SHOP_CODE"      CHAR(8),
       "RGN_CODE"       CHAR(6),
       "EMPLOYEE_CODE"  CHAR(8),
       "CUSTOMER_CODE"  CHAR(8),
       "RECORD_DAY"     DATE,
       "ITEM_CODE"      CHAR(8),
       "ITEM_PRICE"     INTEGER)
    IN "DBAREA01"
    PCTFREE=30
    CHUNK=120
      ARCHIVABLE RANGECOLUMN="RECORD_DAY" IN "DBAREA02"
                 ARCHIVEDIR='/mnt/nfs/archivedir'
```

In the preceding example, the underlined parts are the settings for defining an archivable multi-chunk table.

Example 7: Define a column store table

Define a receipt table (RECEIPT) as a column store table.

```
CREATE TABLE "RECEIPT"
      ("RID"            INTEGER,
       "SHOP_CODE"      CHAR(8),
       "RGN_CODE"       CHAR(6),
       "EMPLOYEE_CODE"  CHAR(8),
       "CUSTOMER_CODE"  CHAR(8),
       "RECORD_DAY"     DATE,
       "ITEM_CODE"      CHAR(8),
       "ITEM_PRICE"     INTEGER)
    IN "DBAREA01"
    CHUNK=120
    STORAGE FORMAT COLUMN
```

In the preceding example, the underlined portion indicates a specification specific to column store tables.

3.8 CREATE USER (create an HADB user)

This section describes the specification format and rules for the `CREATE USER` statement.

3.8.1 Specification format and rules for the CREATE USER statement

The `CREATE USER` statement creates an HADB user.

Because no privileges are granted to the HADB user that is created, the `GRANT` statement must be used to grant the required privileges to the HADB user.

(1) Specification format

```
CREATE-USER-statement ::= CREATE USER authorization-identifier IDENTIFIED BY password
```

(2) Explanation of specification format

- *authorization-identifier*

  Specifies the authorization identifier of the HADB user to be created.

  The rules for specifying an authorization identifier are as follows:

  - The authorization identifier can include single-byte uppercase and lowercase letters, numbers, and the backslash (`\`), hash mark (`#`), and at mark (`@`) characters.
  - If you want to use lowercase letters in the authorization identifier, enclose the authorization identifier in double quotation marks (`"`).

    Example: `CREATE USER "ADBuser01" ...`

    When not enclosed in double quotation marks, lowercase letters are treated as uppercase. For example, `ADBuser01` is treated as `ADBUSER01`.
  - Because an authorization identifier is specified as a name, we recommend that you enclose it in double quotation marks (`"`).
  - You cannot specify `ALL`, `HADB`, `MASTER`, or `PUBLIC` as an authorization identifier.
  - The authorization identifier cannot exceed 100 characters (100 bytes).

  For details about the rules for specifying an authorization identifier, see 6.1.4 Specifying names.

- `IDENTIFIED BY` *password*

  Specify a password for the HADB user that is to be created.

  The rules for specifying a password are as follows:

  - The password can include single-byte uppercase and lowercase letters, numbers, backslashes (`\`), as well as the following characters:

    `` @ ` ! " # $ % & ' ( ) * : + ; [ ] { } , = < > | - . ^ ~ / ? _ ``
  - Specify the password in the form of a character string literal. Therefore, you must enclose the password in single quotation marks. The following are examples:

    Example 1: Specify `Password01` as the password

    `IDENTIFIED BY 'Password01'`

    Example 2: Specify `Pass'01` as the password

```
IDENTIFIED BY 'Pass''01'
```

If the password itself includes a single quotation mark ('), specify two single quotation marks to represent a single quotation mark (''), as shown in the example above.

For rules on specifying a character string literal, see Table 6-10: Description formats and assumed data types of literals.

- The password cannot be empty. That is, the following is not permitted:

```
IDENTIFIED BY ''
```

- The password cannot exceed 255 characters (255 bytes).

Note

- If you are using the JDBC driver, we recommend that you not use the following character in the password:

  &

- If you are using the ODBC driver, we recommend that you not use the following characters in the password:

  [ ] { } ( ) , ; ? * = ! @

(3) Privileges required at execution

To execute the CREATE USER statement, the DBA privilege and the CONNECT privilege are required.

(4) Rules

A maximum of 30,000 HADB users can be created.

(5) Examples

Example

Create an HADB user with the following authorization identifier and password:

- Authorization identifier: ADBUSER01
- Password: #HelloHADB_01

```
CREATE USER "ADBUSER01" IDENTIFIED BY '#HelloHADB_01'
```

3.9 CREATE VIEW (define a viewed table)

This section describes the specification format and rules for the CREATE VIEW statement.

3.9.1 Specification format and rules for the CREATE VIEW statement

The CREATE VIEW statement defines a viewed table.

(1) Specification format

```
CREATE-VIEW-statement ::= CREATE VIEW table-name [(column-name-list)] AS query-expres
sion [LIMIT-clause]

  column-name-list ::= column-name[,column-name]...
```

(2) Explanation of specification format

- *table-name*

  Specifies the name of the viewed table to be defined. You cannot specify a name that is the same as a base table, or a viewed table that has already been defined. For rules on specifying a table name, see (2) Table name specification format in 6.1.5 Qualifying a name.

- *column-name-list*

  ```
  column-name-list ::= column-name[,column-name]...
  ```

  Specifies the columns that will make up the viewed table.

  column-name:

  Specifies the name of a column that will make up the viewed table. The column names must be unique within a single viewed table.

  Do not specify a character string in the EXP*nnnn*_NO_NAME format as a column name. Such a column name might duplicate a derived column name that is automatically set by HADB. In this format, *nnnn* is an unsigned integer in the range from 0000 to 9999.

  Note the following points concerning column names:

  - The number of column names specified in *column-name-list* must be the same as the number of columns in the table derived by the query expression.
  - A maximum of 1,000 columns can be specified in *column-name-list*.
  - If *column-name-list* is omitted, the names of the columns that make up the viewed table will be the same as the names of the columns derived by the query expression. For details about derived column names, see 6.9 Derived column names.
  - You must specify *column-name-list* in the following circumstances:
    - If the derived column names are not unique
    - If one or more columns have no corresponding derived column name

- AS *query-expression* [*LIMIT-clause*]

  Specifies a query expression that determines the contents that will make up the viewed table. For details about query expressions, see 7.1 Query expression.

All tables specified in the query expression become tables upon which the viewed tables will be based (*underlying tables*).

The following items cannot be specified in the query expression:

- [*table-specification*.]ROW
- Dynamic parameters

Note

If you execute the CREATE VIEW statement with * or *table-specification*.* in the selection list in the outermost query specification in the query expression, and then add a column to the underlying table, that column is not added to the viewed table.

LIMIT-clause:

Specifies the maximum number of rows that will be retrieved from the results of the query expression.

For details about the LIMIT clause, see 7.9 LIMIT clause.

(3) Privileges required at execution

To execute the CREATE VIEW statement, all of the following privileges are required:

- CONNECT privilege
- Schema definition privilege
- SELECT privilege for all underlying tables that are to be specified in the query expression

(4) Rules

1. A maximum of 30,000 viewed tables can be defined in the system.
2. The maximum length of a CREATE VIEW statement is 64,000 bytes.
3. The total number of table names, derived tables, and table function derived tables in table references specified in the CREATE VIEW statement cannot exceed 2,047.

   Note that if the following items are specified in a table reference, the total number of derived tables is checked for the SQL statement after those items are equivalently exchanged into internal derived tables:

   - Query name
   - Viewed tables

     If a viewed table is specified in the CREATE VIEW statement, the total number of derived tables is checked after the viewed table specified in the CREATE VIEW statement is equivalently exchanged into a derived table.

   For rules and examples of how to count the number of tables, derived tables, and table function derived tables specified in an SQL statement, see (4) Rules in 4.4.1 Specification format and rules for the SELECT statement.
4. The total number of query specifications and table value constructors that can be included in the CREATE VIEW statement cannot exceed 1,023.
5. You cannot define a viewed table with a schema name that is different than the HADB user whose authorization identifier is connected to the HADB server.
6. The columns that comprise the viewed table will have the same attributes as the columns in the table from which the viewed table is derived as the result of executing the query expression in the CREATE VIEW statement. Note that these attributes are the data type, data length, and whether a NOT NULL constraint exists.

7. Viewed tables include *read-only viewed tables* and *updatable viewed tables*. You cannot insert, update, or delete rows in a read-only viewed table.

8. Whether a viewed table will be a read-only viewed table or an updatable viewed table depends on what is specified in `AS` *query-expression*. It will be a read-only viewed table in the following circumstances:

   - If the outermost query specification includes a table join, a joined table, a derived table#, table function derived table, `SELECT DISTINCT`, a `GROUP BY` clause, a `HAVING` clause, a window function, or a set function
   - If the same column from the underlying table is specified multiple times in the selection expression in the outermost query specification
   - If something other than a column specification is specified in the selection expression in the outermost query specification
   - If the same table as the table specified in the `FROM` clause in the outermost query specification is specified in the `FROM` clause in a subquery
   - If a read-only viewed table is specified in the `FROM` clause in the outermost query specification
   - If you specify a set operation with the outermost query specification as an operand
   - If you specify a `LIMIT` clause
   - If a recursive query name is specified in the `FROM` clause in the outermost query specification
   - If a dictionary table or system table is specified for the `FROM` clause in the outermost query specification

   #

   If the viewed table no longer meets the conditions for being a read-only viewed table once the derived table is expanded, it will become an updatable viewed table. For the rules about derived table expansion, see 7.30.3 Rules for derived table expansion.

   You can check whether the viewed table you defined is a read-only viewed table or an updatable viewed table by searching the dictionary table. For details on how to check this, see *Checking whether a viewed table is updatable* in the *HADB Setup and Operation Guide*.

9. The access privilege for the viewed table that you define is determined based on the following rules:

   - When an HADB user defines a viewed table, the user's access privileges to the viewed table are determined by the user's access privileges to all the underlying tables. For example, if you want to have the `INSERT` privilege on the viewed table you are defining, you must have the `INSERT` privilege on all of the underlying tables.
   - This item explains the rule to determine the access privilege for a viewed table defined by specifying either of the following elements in the query expression of the `CREATE VIEW` statement:

     - Table value constructors
     - Table function derived table

     Any HADB users who define a viewed table for a derived table derived by a table value constructor or for a table function derived table are assumed to have an access privilege with the grant option. Therefore, if another underlying table of the viewed table is specified in the query expression of the `CREATE VIEW` statement, the access privilege for the defined viewed table is as described later.

     Examples:

     - If a table value constructor and an underlying table having access privileges with the grant option are specified in the query expression of the `CREATE VIEW` statement

       For the viewed table, the access privilege that overlaps between the access privilege for the derived table derived by the table value constructor and the access privilege for the underlying table of the viewed table is applied. In this case, therefore, the HADB user who defines a viewed table will have the access privilege with the grant option for the viewed table.

    - If a table value constructor and an underlying table that has only the SELECT privilege are specified in the query expression of the CREATE VIEW statement

      For the viewed table, the access privilege that overlaps between the access privilege for the derived table derived by the table value constructor and the access privilege for the underlying table of the viewed table is applied. In this case, therefore, the HADB user who defines a viewed table will have only the SELECT privilege for the viewed table.

10. An HADB user who wants to grant an access privilege for a viewed table to another HADB user must have an access privilege with the grant option for all underlying tables of that viewed table.
11. When a new access privilege for an underlying table is granted, the access privilege for the viewed tables that depend on the underlying table is also granted. (Consequently, propagation of access privileges will occur.) For example, assume that HADB user A has defined viewed table A.V1 by using table X.T1 as the underlying table, and viewed table A.V2 by using viewed table A.V1 as the underlying table. In this case, if the INSERT privilege for table X.T1 is granted to HADB user A, INSERT privilege for viewed tables A.V1 and A.V2 is also granted to HADB user A.
12. If the access privilege for an underlying table is revoked, the access privilege for the viewed tables that depend on the underlying table is also revoked.
13. If you define a viewed table by specifying the scalar function CONTAINS (with synonym-search specification) in the query expression of the CREATE VIEW statement, the following rules apply:
    - If you delete the synonym dictionary that was specified in the synonym-search specification, an error occurs when the viewed table is accessed.
    - If you update the synonym dictionary that was specified in the synonym-search specification, the updates are applied to the viewed table that you define.
14. Assume that you have defined a viewed table for which the query name in the WITH clause specified in the query expression in the CREATE VIEW statement is not referenced in the CREATE VIEW statement. In this case, when you search the viewed table, the following rules apply. Also, the following rules apply when you specify the viewed table in another CREATE VIEW statement.
    - The upper limit check on the number of constituent elements is not conducted for the number of constituent elements in the query expression body for a query name that is not referenced in the CREATE VIEW statement.
    - No lock is obtained for any tables specified in the query expression body for a query name that is not referenced in the CREATE VIEW statement.

    The following shows examples.

    (Example) When a viewed table is defined

    ```
    CREATE VIEW "V1"
        AS WITH "Q1" AS (SELECT "T1"."C1","T2"."C2" FROM "T1","T2")
           SELECT * FROM "T3"
    ```

    [Explanation]

    The rules shown earlier are not applied when a viewed table is defined. Therefore, the number of queries specified in the CREATE VIEW statement is 2 (the query corresponding to query name Q1 and its main query). Also, the number of specified tables is 3 (base tables T1, T2, and T3).

    (Example) When a viewed table is searched

    ```
    SELECT * FROM "V1"
    ```

[Explanation]

- The rules shown earlier are applied when a viewed table is searched. Therefore, the number of queries specified in the `SELECT` statement is 2 (the main query and derived query for the derived table equivalently exchanged from viewed table `V1`).
- The number of specified tables is 2 (the derived table equivalently exchanged from viewed table `V1` and base table `T3` in the derived query for the derived table). The upper limit check on the number of queries and tables that can be specified in one SQL statement is conducted based on these rules.
- For base tables `T1` and `T2`, no lock is obtained.

15. In the query expression of the `CREATE VIEW` statement, subqueries can be specified in a nested form. In this case, the subquery nesting depth must not exceed 31. Note that if the table specified in the `FROM` clause is a viewed table, the subquery nesting depth must not exceed 31 after HADB generates the internal derived table specified in the underlying query expression. For details, see (a) Common rules for subqueries in (4) Rules in 7.3.1 Specification format and rules for subqueries.

Example 1:

```
CREATE VIEW "V1"
  AS SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (
     SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (
     SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (
     SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (
     SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (
     SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (
     SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (
     SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (
     SELECT * FROM (SELECT * FROM "T1") AS DT32
     ) AS DT31 ) AS DT30 ) AS DT29 ) AS DT28 ) AS DT27 ) AS DT26 ) AS DT25 ) AS DT
24
     ) AS DT23 ) AS DT22 ) AS DT21 ) AS DT20 ) AS DT19 ) AS DT18 ) AS DT17 ) AS DT
16
     ) AS DT15 ) AS DT14 ) AS DT13 ) AS DT12 ) AS DT11 ) AS DT10 ) AS DT9 ) AS DT8
     ) AS DT7 ) AS DT6 ) AS DT5 ) AS DT4 ) AS DT3 ) AS DT2 ) AS DT1 ) AS DT0
```

In the preceding example, the subquery nesting depth of viewed table `V1` is 32. In this case, because the maximum nesting depth is exceeded, the `CREATE VIEW` statement will result in an error.

Note that in this example, `T1` is the base table.

Example 2:

```
CREATE VIEW "V2"           <== Viewed table V2
  AS SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (
     SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (
     SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (
     SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (
     SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (
     SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (
     SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (
     SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM "T1") AS DT30
     ) AS DT29 ) AS DT28 ) AS DT27 ) AS DT26 ) AS DT25 ) AS DT24 ) AS DT23 ) AS DT
22
     ) AS DT21 ) AS DT20 ) AS DT19 ) AS DT18 ) AS DT17 ) AS DT16 ) AS DT15 ) AS DT
14
     ) AS DT13 ) AS DT12 ) AS DT11 ) AS DT10 ) AS DT9 ) AS DT8 ) AS DT7 ) AS DT6
     ) AS DT5 ) AS DT4 ) AS DT3 ) AS DT2 ) AS DT1 ) AS DT0
CREATE VIEW "V3" AS SELECT * FROM "V2"       <== Viewed table V3
CREATE VIEW "V4" AS SELECT * FROM "V3"       <== Viewed table V4
```

- For viewed table `V2`, the subquery nesting depth is 30. Therefore, the `CREATE VIEW` statement can run.

- For viewed table V3, the subquery nesting depth becomes 31 when the internal derived table is generated. Therefore, the CREATE VIEW statement can run.
- For viewed table V4, the subquery nesting depth becomes 32 when the internal derived table is generated. In this case, because the maximum nesting depth is exceeded, the CREATE VIEW statement will result in an error.

Note that in this example, T1 is the base table.

16. A viewed table whose view level is 33 cannot be specified in the query expression for the CREATE VIEW statement.
17. When the version of the HADB server is upgraded, the viewed tables that depend on dictionary tables or system tables might be re-created automatically. For details about the conditions in which viewed tables are re-created, see *Re-creation of viewed tables in the event of a version upgrade* in *Notes on version upgrading* in the *HADB Setup and Operation Guide*.
18. The viewed tables defined after a transaction is started cannot be accessed from the transaction.

(5) Examples

Example 1

Define a viewed table of shops (VSHOPSLIST) from which you can retrieve all the columns in the shops table (SHOPSLIST) except the address column (ADDRESS). Let the ordering and column names of the columns that make up the viewed table of shops be as follows:

- Shop code (SHOP_CODE)
- Region code (RGN_CODE)
- Shop name (SHOP_NAME)
- Telephone number (TEL_NO)

```
CREATE VIEW "VSHOPSLIST" ("SHOP_CODE","RGN_CODE","SHOP_NAME","TEL_NO")
    AS SELECT "SHOP_CODE","RGN_CODE","SHOP_NAME","TEL_NO"
           FROM "SHOPSLIST"
```

The viewed table of shops (VSHOPSLIST) is an updatable viewed table.

Example 2

This example defines (as a viewed table) a sales table (VSALES) that obtains the maximum sales value (QMAXSALES) for each product name (PUR-NAME) from the sales history table (SALESLIST) and product table (PRODUCTSLIST). Let the structure of columns in the sales table (VSALES) be as follows:

- Product name (VPUR_NAME)
- Maximum sales value (VQMAXSALES)

■ Defining a viewed table

```
CREATE VIEW "VSALES" ("VPUR-NAME","VQMAXSALES")
    AS WITH "QT1"("QCODE","QMAXSALES") AS (SELECT "PUR-CODE",MAX("PRICE" * "QUANTI
TY")
                                             FROM "SALESLIST"
                                               GROUP BY "PUR-CODE")
       SELECT "PUR-NAME","QMAXSALES"
         FROM "QT1" INNER JOIN "PRODUCTSLIST" ON "QCODE"="PUR-CODE"
```

SALESLIST

USERID	PUR-CODE	PRICE	QUANTITY
U0001	P001	500	3
U0001	P002	100	10
U0002	P001	500	1
U0002	P002	100	5
U0003	P001	500	4
U0003	P002	100	1
U0003	P003	50	10

PRODUCTSLIST

PUR-CODE	PUR-NAME
P001	PRODUCT_A
P002	PRODUCT_B
P003	PRODUCT_C

■ Searching a viewed table

```
SELECT * FROM "VSALES"
```

Example of search result

VPUR_NAME	VQMAXSALES
PRODUCT_A	2000
PRODUCT_B	1000
PRODUCT_C	500

3.10 DROP AUDIT (delete the audit target definition)

This section describes the specification format and rules for the DROP AUDIT statement.

3.10.1 Specification format and rules for the DROP AUDIT statement

The DROP AUDIT statement deletes the audit target definition created by using the CREATE AUDIT statement.

> **Important**
>
> You can execute the DROP AUDIT statement when the audit trail facility is enabled. To check whether the audit trail facility is enabled, execute the adbaudittrail -d command.

(1) Specification format

```
DROP-AUDIT-statement::=DROP AUDIT AUDITTYPE EVENT
                             FOR ANY OPERATION
```

(2) Explanation of specification format

- AUDITTYPE EVENT

  Specify this if you want to delete the audit target definition created by specifying EVENT for AUDITTYPE in the CREATE AUDIT statement. Specify this when you stop outputting an audit trail about the final event results.

- FOR ANY OPERATION

  Specify this if you want to exclude the events listed in Table 3-3: Audit-target events from the audit targets.

  Specify this when you delete the audit target definition created by specifying FOR ANY OPERATION in the CREATE AUDIT statement.

(3) Privileges required at execution

To execute the DROP AUDIT statement, the CONNECT privilege and the audit admin privilege are required.

(4) Rules

1. This statement can delete the audit target definition created by using the CREATE AUDIT statement.
2. An HADB server checks the audit target definition during the determination processing for outputting an audit trail. Therefore, depending on the audit trail output time, an audit trail about operations that were performed before the audit target definition is deleted might not be output although those operations are to be audited.

(5) Examples

Example

Delete the audit target definition created by using the CREATE AUDIT statement.

```
DROP AUDIT AUDITTYPE EVENT
           FOR ANY OPERATION
```

3.11 DROP INDEX (delete an index)

This section describes the specification format and rules for the DROP INDEX statement.

3.11.1 Specification format and rules for the DROP INDEX statement

The DROP INDEX statement deletes an index (a B-tree index, text index or range index).

(1) Specification format

```
DROP-INDEX-statement ::= DROP INDEX index-name
```

(2) Explanation of specification format

- *index-name*

  Specifies the index name of the index to be deleted. For rules on specifying an index name, see (3) Index name specification format in 6.1.5 Qualifying a name.

(3) Privileges required at execution

To execute the DROP INDEX statement, the CONNECT privilege and schema definition privilege are required.

(4) Rules

1. You can only delete an index owned by the current user (the HADB user whose authorization identifier is currently connected to the HADB server). You cannot delete an index owned by another HADB user.
2. You can delete an index that is defined for a table even if that table has data stored in it. When you delete the index, the data stored in the table will not be deleted.
3. You cannot delete an index defined on the base tables of a dictionary table or system table.
4. When you delete an index, it also deletes the cost information for the index.
5. You cannot use the DROP INDEX statement to delete a B-tree index corresponding to a primary key. To delete the index, use the DROP TABLE statement to delete both the index and table.
6. You cannot use the DROP INDEX statement to delete the range index that is defined for the archive range column. To delete the index, use the DROP TABLE statement to delete both the index and table.

(5) Examples

Example 1

Delete the B-tree index (SHOP_CODE_IDX) defined for the shops table (SHOPSLIST).

```
DROP INDEX "SHOP_CODE_IDX"
```

Example 2

Delete the text index (ADDRESS_IDX) defined for the employee table (EMPLOYEE).

```
DROP INDEX "ADDRESS_IDX"
```

Example 3

Delete the range index (`SHOP_CODE_RIDX`) defined for the shops table (`SHOPSLIST`).

```
DROP INDEX "SHOP_CODE_RIDX"
```

3.12 DROP SCHEMA (delete a schema)

This section describes the specification format and rules for the DROP SCHEMA statement.

3.12.1 Specification format and rules for the DROP SCHEMA statement

The DROP SCHEMA statement deletes a schema.

Deleting a schema affects tables, indexes, and foreign keys as follows:

- The tables (base and viewed tables) and indexes that are defined in the schema are also deleted.
- If viewed tables defined in other schemata depend on the tables that will be deleted by the DROP SCHEMA statement, those dependent viewed tables are also deleted (or invalidated).
- If the tables that will be deleted by the DROP SCHEMA statement are referenced by foreign keys defined in other schemata, those foreign keys are also deleted.

(1) Specification format

```
DROP-SCHEMA-statement ::= DROP SCHEMA [schema-name] [drop-behavior]

  drop-behavior ::= {CASCADE | RESTRICT}
```

(2) Explanation of specification format

● *schema-name*

Specifies the name of the schema to be deleted. If the schema name is omitted, the authorization identifier of the HADB user who executed the DROP SCHEMA statement is assumed.

For rules on specifying a schema name, see (1) Schema name specification format in 6.1.5 Qualifying a name.

Note that you cannot specify ALL, HADB, MASTER, or PUBLIC for *schema-name*.

• *drop-behavior*

```
drop-behavior ::= {CASCADE | RESTRICT}
```

Specifies whether to drop the schema if tables or indexes are defined in the schema to be deleted. The following table describes the specifications for drop-behavior.

Specification of drop-behavior	Description	Handling of viewed tables in other schemata	Handling of foreign keys in other schemata
If drop-behavior is omitted	Even if tables or indexes are defined in the schema to be deleted, the schema is deleted. In this case, any tables or indexes defined in the schema are also deleted.	If viewed tables defined in other schemata depend on the tables that will be deleted by the DROP SCHEMA statement, those dependent viewed tables are invalidated.	If the tables that will be deleted by the DROP SCHEMA statement are referenced by foreign keys defined in other schemata, those foreign keys are also deleted.
If CASCADE is specified		If viewed tables defined in other schemata depend on the tables that will be deleted by the DROP SCHEMA statement, those dependent viewed tables are also deleted.	

Specification of drop-behavior	Description	Handling of viewed tables in other schemata	Handling of foreign keys in other schemata
If RESTRICT is specified	If tables or indexes are defined in the schema to be deleted, the DROP SCHEMA statement results in an error.	The viewed tables in other schemata are not affected because the DROP SCHEMA statement results in an error.	The foreign keys in other schemata are not affected because the DROP SCHEMA statement results in an error.

(3) Privileges required at execution

To execute the DROP SCHEMA statement, the CONNECT privilege and schema definition privilege are required.

(4) Rules

1. You can only delete a schema owned by the current user (the HADB user whose authorization identifier is currently connected to the HADB server). You cannot delete a schema owned by another HADB user. For example, if the adbsql command is executed with ADBUSER01 specified as the authorization identifier, schema ADBUSER01 is the only schema that can be deleted with DROP SCHEMA statement.
2. When you delete a schema, the following cost information is also deleted:
   - The cost information for the tables defined in the schema
   - The cost information for any indexes defined in the schema
3. If you delete a schema in which tables are defined, all HADB users who have the access privileges for those tables will have the access privileges revoked. Revoking the access privileges might affect viewed tables and referential constraints. For details, see (4) Rules in 3.17.2 Revoking access privileges.

(5) Examples

Example 1

Delete the schema with schema name ADBUSER01.

```
DROP SCHEMA "ADBUSER01" CASCADE
```

Example 2

Delete the schema with schema name ADBUSER01. However, if a table or index has been defined for the schema, make the DROP SCHEMA statement result in an error.

```
DROP SCHEMA "ADBUSER01" RESTRICT
```

3.13 DROP TABLE (delete a table)

This section describes the specification format and rules for the DROP TABLE statement.

3.13.1 Specification format and rules for the DROP TABLE statement

The DROP TABLE statement deletes a base table.

Deleting a base table affects indexes, table constraints, and viewed tables as follows:

- The indexes defined in the base table are also deleted.
- The viewed tables that depend on the deleted base table are also deleted (or invalidated).
- The table constraints# defined in the base table are also deleted.

#

Primary keys, and any foreign keys defined in the table to be deleted, are deleted. The foreign keys that reference the deletion-target table are also deleted (even if the foreign keys are defined in other schemata).

(1) Specification format

```
DROP-TABLE-statement ::= DROP TABLE table-name [drop-behavior]


 drop-behavior ::= {CASCADE | RESTRICT}
```

(2) Explanation of specification format

- *table-name*

  Specifies the name of the base table to be deleted. For rules on specifying a table name, see (2) Table name specification format in 6.1.5 Qualifying a name.

  Note that you cannot specify the name of a viewed table.

- *drop-behavior*

  ```
  drop-behavior ::= {CASCADE | RESTRICT}
  ```

  Specifies whether to delete the base table if any of the following conditions are met:

  - There is an index defined for the base table to be deleted.
  - Viewed tables that depend on the base table to be deleted are defined.
  - A table constraint has been defined for the base table to be deleted.

  The following table describes the specifications for drop-behavior.

Specification of drop-behavior	Description	Handling of viewed tables that depend on the base table to be deleted
If drop-behavior is omitted	The base table is also deleted if any of the following conditions are met: • There is an index defined for the base table to be deleted.	The viewed tables that depend on the deleted base table are invalidated. Not only the viewed table in the relevant schema, but also the dependent viewed tables in other schemata, are invalidated.

Specification of drop-behavior	Description	Handling of viewed tables that depend on the base table to be deleted
If CASCADE is specified	• Viewed tables that depend on the base table to be deleted are defined. • A table constraint has been defined for the base table to be deleted. In this case, the following items are also deleted: • Indexes and table constraints defined in the base table • Foreign keys that reference the deletion-target table (including those in other schemata)	The viewed tables that depend on the deleted base table are deleted. Not only the viewed table in the relevant schema, but also the dependent viewed tables in other schemata, are deleted.
If RESTRICT is specified	If any of the following conditions are met, the DROP TABLE statement results in an error. • There is an index defined for the base table to be deleted. • Viewed tables that depend on the base table to be deleted are defined. • A table constraint has been defined for the base table to be deleted.	The dependent viewed tables are not affected because the DROP TABLE statement results in an error.

(3) Privileges required at execution

To execute the DROP TABLE statement, the CONNECT privilege and schema definition privilege are required.

(4) Rules

1. The DROP TABLE statement can be used to drop only base tables owned by the current user (the HADB user whose authorization identifier is currently connected to the HADB server). You cannot delete a base table owned by another HADB user.
2. Base tables with data stored in them can be deleted.
3. You cannot delete the base tables of a dictionary table or system table.
4. When you delete a table, the following cost information is also deleted:
    - The cost information for the table
    - The cost information for any indexes defined for the table
5. If you delete a table, all HADB users who have the access privileges for that table will have the access privileges revoked. Revoking the access privileges might affect viewed tables and referential constraints. For details, see (4) Rules in 3.17.2 Revoking access privileges.
6. When an archivable multi-chunk table is deleted, the following table and indexes are also deleted:
    - Range index automatically defined for the archive range column
    - Location table
    - Location table index
7. If an archivable multi-chunk table is deleted, data stored in chunks (either archived or not) is deleted.

(5) Examples

Example

Delete the shops table (SHOPSLIST).

```
DROP TABLE "SHOPSLIST" CASCADE
```

3.14 DROP USER (delete an HADB user)

This section describes the specification format and rules for the DROP USER statement.

3.14.1 Specification format and rules for the DROP USER statement

The DROP USER statement deletes an HADB user.

(1) Specification format

```
DROP-USER-statement ::= DROP USER authorization-identifier [drop-behavior]

  drop-behavior ::= {CASCADE | RESTRICT}
```

(2) Explanation of specification format

- *authorization-identifier*

  Specify the authorization identifier of the HADB user to be deleted.

  Note the following rules for specifying an authorization identifier:

  - If you want to use lowercase letters, enclose the authorization identifier in double quotation marks ("). When not enclosed in double quotation marks, lowercase letters will be treated as uppercase.

    Example: DROP USER adbuser01 ...

    In this case, the authorization identifier is treated as ADBUSER01.

  - Because an authorization identifier is specified as a name, we recommend that you enclose it in double quotation marks (").

  For details about the rules for specifying an authorization identifier, see 6.1.4 Specifying names.

- *drop-behavior*

  ```
  drop-behavior ::= {CASCADE | RESTRICT}
  ```

  Specifies whether to delete the HADB user if either of the following conditions is met:

  - The HADB user to be deleted owns a schema.
  - The HADB user to be deleted has granted access privileges to other HADB users.

  The following table describes the possible specifications for drop-behavior.

Specification of drop-behavior	Description	Handling of viewed tables in other schemata	Handling of foreign keys in other schemata
If drop-behavior is omitted	The HADB user is deleted even if either of the following conditions is met: • The HADB user to be deleted owns a schema. • The HADB user to be deleted has granted access privileges to other HADB users. When the DROP USER statement is run, the schemata owned by the	If viewed tables defined in other schemata depend on the tables that will be deleted by the DROP USER statement, those dependent viewed tables are invalidated.	If the tables that will be deleted by the DROP USER statement are referenced by foreign keys defined in other schemata, those foreign keys are also deleted.
If CASCADE is specified		If viewed tables defined in other schemata depend on the tables that will be deleted by the DROP USER statement, those dependent viewed tables are also deleted.	

Specification of drop-behavior	Description	Handling of viewed tables in other schemata	Handling of foreign keys in other schemata
	HADB user to be deleted are also deleted. Also, all access privileges that have been granted to other HADB users are revoked. In addition, all dependent privileges for the revoked access privileges are revoked.		
If RESTRICT is specified	The DROP USER statement results in an error if either of the following conditions is met: • The HADB user to be deleted owns a schema. • The HADB user to be deleted has granted access privileges to other HADB users.	The viewed tables in other schemata are not affected because the DROP USER statement results in an error.	The foreign keys in other schemata are not affected because the DROP USER statement results in an error.

(3) Privileges required at execution

To execute the DROP USER statement, the DBA privilege and the CONNECT privilege are required.

(4) Rules

1. It is possible to delete HADB users other than yourself.
2. The HADB user whose authorization identifier is currently connected to the HADB server cannot be deleted.
3. If the deleted HADB user had granted another HADB user the DBA privilege, the CONNECT privilege, or the schema definition privilege, those privileges are not revoked.
4. If the HADB user to be deleted has granted access privileges to other HADB users, all the granted access privileges are revoked. The dependent privileges for the revoked access privileges are also revoked. Therefore, revoking access privileges might affect viewed tables and referential constraints. For details, see (4) Rules in 3.17.2 Revoking access privileges.
5. The HADB users having the audit privilege cannot be deleted. To delete HADB users who have the audit privilege, ask an HADB user who has the audit admin privilege to revoke the audit privilege of the HADB users, and then delete them.

(5) Examples

Example 1

Delete HADB user ADBUSER01.

```
DROP USER "ADBUSER01" CASCADE
```

Example 2

Delete HADB user ADBUSER01. However, if ADBUSER01 owns a schema, make the DROP USER statement result in an error.

```
DROP USER "ADBUSER01" RESTRICT
```

3.15 DROP VIEW (delete a viewed table)

This section describes the specification format and rules for the DROP VIEW statement.

3.15.1 Specification format and rules for the DROP VIEW statement

The DROP VIEW statement deletes a viewed table.

(1) Specification format

```
DROP-VIEW-statement ::= DROP VIEW table-name [drop-behavior]

  drop-behavior ::= {CASCADE | RESTRICT}
```

(2) Explanation of specification format

- *table-name*

  Specifies the name of the viewed table to be deleted. For rules on specifying a table name, see (2) Table name specification format in 6.1.5 Qualifying a name.

  You cannot specify the table names of the following tables:

  - Base tables
  - Dictionary tables
  - System tables

- *drop-behavior*

  ```
  drop-behavior ::= {CASCADE | RESTRICT}
  ```

  Specifies whether to delete the viewed tables if the following condition is met:

  - Viewed tables that depend on the viewed table to be deleted exist.

  The following table describes the possible specifications for drop-behavior.

Specification of drop-behavior	Description	Handling of viewed tables that depend on the viewed table to be deleted
If drop-behavior is omitted	The viewed tables are also deleted if the following condition is met: • Viewed tables that depend on the viewed table to be deleted exist.	The viewed tables that depend on the viewed table to be deleted are invalidated. The viewed table in the relevant schema and the viewed tables in other schemata are invalidated.
If CASCADE is specified		The viewed tables that depend on the viewed table to be deleted are deleted. The viewed table in the relevant schema and the viewed tables in other schemata are deleted.
If RESTRICT is specified	The DROP VIEW statement results in an error if the following condition is met: • Viewed tables that depend on the viewed table to be deleted exist.	The dependent viewed tables are not affected because the DROP VIEW statement results in an error.

(3) Privileges required at execution

To execute the DROP VIEW statement, the CONNECT privilege and schema definition privilege are required.

(4) Rules

1. You cannot delete a viewed table that has a schema name different from the authorization identifier connected to the HADB server.
2. If you delete a viewed table, all HADB users who have the access privileges for that viewed table will have the access privileges revoked.

(5) Example

Example

Delete the viewed table of shops (`VSHOPSLIST`).

```
DROP VIEW "VSHOPSLIST" CASCADE
```

3.16 GRANT (grant privileges)

This section describes the specification format and rules for the `GRANT` statement.

3.16.1 Granting user privileges, schema operation privileges, and audit privileges

Grant the following privileges to an HADB user.

- User privileges
  - `DBA` privilege
  - `CONNECT` privilege
- Schema operation privileges
  - Schema definition privilege
- Audit privileges
  - Audit admin privilege
  - Audit viewer privilege

(1) Specification format

```
GRANT-statement::=GRANT privilege[,privilege]... TO authorization-identifier[,authori
zation-identifier]...

  privilege::={user-privilege|schema-operation-privilege|audit-privilege}
    user-privilege::={DBA|CONNECT}
    schema-operation-privilege::=SCHEMA
    audit-privilege::={AUDIT ADMIN|AUDIT VIEWER}
```

(2) Explanation of specification format

- *privilege*`[,`*privilege*`]...`

```
privilege::={user-privilege|schema-operation-privilege|audit-privilege}
```

Specify the privilege to be granted to an HADB user. You cannot specify the same privilege more than once.

```
user-privilege ::= {DBA | CONNECT}
```

Specify this to grant user privileges to an HADB user.

- `DBA`
  Specify this to grant the `DBA` privilege to the HADB user.
- `CONNECT`
  Specify this to grant the `CONNECT` privilege to the HADB user.

```
schema-operation-privilege ::= SCHEMA
```

Specify this to grant schema operation privileges to an HADB user.

- `SCHEMA`

Specify this to grant the schema definition privilege to the HADB user.

```
audit-privilege::={AUDIT ADMIN|AUDIT VIEWER}
```

Specify this to grant an audit privilege (audit admin privilege or audit viewer privilege) to an HADB user.

- `AUDIT ADMIN`

  Specify this to grant the audit admin privilege to an HADB user.

- `AUDIT VIEWER`

  Specify this to grant the audit viewer privilege to an HADB user.

- `TO` *authorization-identifier* `[,`*authorization-identifier*`]...`

  Specifies the authorization identifiers of the HADB users who are to be granted privileges. A maximum of 128 authorization identifiers can be specified.

  Note the following rules for specifying an authorization identifier:

  - If you want to use lowercase letters, enclose the authorization identifier in double quotation marks ("). When not enclosed in double quotation marks, lowercase letters will be treated as uppercase.

    Example: `GRANT DBA TO adbuser01`

    In this case, the authorization identifier is treated as `ADBUSER01`.

  - Because an authorization identifier is specified as a name, we recommend that you enclose it in double quotation marks (").

  For details about the rules for specifying an authorization identifier, see 6.1.4 Specifying names.

(3) Privileges required at execution

To execute a `GRANT` statement that grants user privileges, schema operation privileges, or audit privileges, the `DBA` privilege and the `CONNECT` privilege are required.

(4) Rules

1. An HADB user with the `DBA` privilege can grant the following privileges to other HADB users:

   - User privileges
   - Schema operation privileges
   - Audit privileges

   However, the audit admin privilege cannot be granted to HADB users who have the `DBA` privilege.

   Note that an HADB user can also grant a user privilege, a schema operation privilege, and the audit viewer privilege to himself or herself (the HADB user whose authorization identifier is connected to the HADB server).

   > **Important**
   >
   > An HADB user cannot have both the `DBA` privilege and the audit admin privilege. Therefore, it is impossible to grant the audit admin privilege to HADB users who have the `DBA` privilege. Similarly, it is also impossible to grant the `DBA` privilege to HADB users who have the audit admin privilege.

2. If an error occurs in the execution of the `GRANT` statement when multiple authorization identifiers are specified, the operation is cancelled for all of the targeted HADB users.

(5) Examples

Example 1

Grant the DBA privilege, CONNECT privilege, and schema definition privilege to HADB user ADBUSER01.

```
GRANT DBA,CONNECT,SCHEMA TO "ADBUSER01"
```

Example 2

Grant the CONNECT privilege and schema definition privilege to HADB users ADBUSER02 and ADBUSER03.

```
GRANT CONNECT,SCHEMA TO "ADBUSER02","ADBUSER03"
```

Example 3

Grant the CONNECT privilege and the audit admin privilege to HADB user ADBAUDITADMIN.

```
GRANT CONNECT,AUDIT ADMIN TO "ADBAUDITADMIN"
```

Example 4

Grant the CONNECT privilege and the audit viewer privilege to HADB user ADBAUDITOR.

```
GRANT CONNECT,AUDIT VIEWER TO "ADBAUDITOR"
```

3.16.2 Granting access privileges

Grant access privileges to an HADB user.

(1) Specification format

```
GRANT-statement ::= GRANT access-privilege ON object-name TO privilege-grantee [WITH
GRANT OPTION]

  access-privilege ::= {ALL [PRIVILEGES]|operation[,operation]...}
    operation ::= {SELECT|INSERT|UPDATE|DELETE|TRUNCATE|REFERENCES
                |IMPORT TABLE|REBUILD INDEX|GET COSTINFO|EXPORT TABLE
                |MERGE CHUNK|CHANGE CHUNK COMMENT|CHANGE CHUNK STATUS
                |ARCHIVE CHUNK|UNARCHIVE CHUNK}

  object-name ::= {[TABLE]table-name|ALL TABLES}
  privilege-grantee ::= {authorization-identifier[,authorization-identifier]...|PUBLI
C}
```

(2) Explanation of specification format

- *access-privilege*

```
access-privilege ::= {ALL [PRIVILEGES] | operation[,operation]...}
```

Specify the type of access privilege to be granted.

ALL [PRIVILEGES]:

Specify this to grant all access privileges.

Note that the access privileges that are granted if this clause is specified are all the access privileges that are supported at the time when the GRANT statement is run. If other access privileges are additionally supported as a result of version upgrade after the GRANT statement is run, those access privileges will not be granted automatically.

Important

If you run the GRANT statement with ALL PRIVILEGES specified when you have only some access privileges with the grant option, you cannot grant all types of access privileges. In this case, only the access privileges with the grant option are granted to the privilege grantee. For example, when you have the grant option for the INSERT privilege only, if you run the GRANT statement with ALL PRIVILEGES specified, only the INSERT privilege is granted to the privilege grantee.

Note

If you specify ALL PRIVILEGES when you have no access privilege with the grant option for the target object, the GRANT statement results in an error.

operation[,*operation*]...:

```
operation ::= {SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES
             | IMPORT TABLE | REBUILD INDEX | GET COSTINFO | EXPORT TABLE
             | MERGE CHUNK | CHANGE CHUNK COMMENT | CHANGE CHUNK STATUS
             | ARCHIVE CHUNK | UNARCHIVE CHUNK}
```

Specify the type of access privilege to be granted. You cannot specify the same operation more than once.

- SELECT
  Specify this to grant the SELECT privilege to an HADB user.
- INSERT
  Specify this to grant the INSERT privilege to an HADB user.
- UPDATE
  Specify this to grant the UPDATE privilege to an HADB user.
- DELETE
  Specify this to grant the DELETE privilege to an HADB user.
- TRUNCATE
  Specify this to grant the TRUNCATE privilege to an HADB user.
- REFERENCES
  Specify this to grant the REFERENCES privilege to an HADB user.
- IMPORT TABLE
  Specify this to grant the IMPORT TABLE privilege to an HADB user.
- REBUILD INDEX
  Specify this to grant the REBUILD INDEX privilege to an HADB user.
- GET COSTINFO
  Specify this to grant the GET COSTINFO privilege to an HADB user.
- EXPORT TABLE
  Specify this to grant the EXPORT TABLE privilege to an HADB user.
- MERGE CHUNK

Specify this to grant the MERGE CHUNK privilege to an HADB user.

- CHANGE CHUNKCOMMENT

  Specify this to grant the CHANGE CHUNK COMMENT privilege to an HADB user.

- CHANGE CHUNKSTATUS

  Specify this to grant the CHANGE CHUNK STATUS privilege to an HADB user.

- ARCHIVE CHUNK

  Specify this to grant the ARCHIVE CHUNK privilege to an HADB user.

- UNARCHIVE CHUNK

  Specify this to grant the UNARCHIVE CHUNK privilege to an HADB user.

● ON *object-name*

```
object-name ::= {[TABLE] table-name | ALL TABLES}
```

Specifies the object to which access privileges are to be granted.

Here, *object* refers to a schema object.

[TABLE] *table-name*:

Grant access privileges to the table specified here. For rules on specifying a table name, see (2) Table name specification format in 6.1.5 Qualifying a name.

Note that you cannot specify the table name of a viewed table that has been invalidated.

ALL TABLES:

Grant access privileges to all the base tables in the schema of the executing user. In this context, *executing user* means the HADB user executing the GRANT statement.

If the HADB user executing the GRANT statement has not defined a schema, or no base tables are defined in the schema, the GRANT statement terminates normally without granting access privileges.

Note

Specifying ALL TABLES grants access privileges to all the base tables owned by the executing user at the time of the GRANT statement. This does not include new base tables defined after the GRANT statement is executed (access privileges will not be granted for these newly-defined base tables).

● TO *privilege-grantee*

```
privilege-grantee ::= {authorization-identifier[,authorization-identifier]...|PUBL
IC}
```

Specifies the HADB users to grant access privileges to.

authorization-identifier[,*authorization-identifier*]:

Specifies the authorization identifiers of the HADB users who are to be granted access privileges. A maximum of 128 authorization identifiers can be specified.

Note the following rules for specifying an authorization identifier:

- If you want to use lowercase letters, enclose the authorization identifier in double quotation marks ("). When not enclosed in double quotation marks, lowercase letters will be treated as uppercase.
- Because an authorization identifier is specified as a name, we recommend that you enclose it in double quotation marks (").

For details about the rules for specifying an authorization identifier, see 6.1.4 Specifying names.

PUBLIC:

Specify this if you want to authorize access privileges to the specified object for all HADB users. In this context, *all HADB users* includes HADB users created after execution of the GRANT statement in which PUBLIC is specified.

Example:

```
GRANT SELECT,IMPORT TABLE ON "T1" TO PUBLIC
```

Executing the GRANT statement above authorizes the following access privileges for all HADB users:

- The SELECT privilege on table T1
- The IMPORT TABLE privilege on table T1

For a user who runs the GRANT statement to specify PUBLIC, he or she must own the object for which the access privileges are to be authorized.

Note

- The PUBLIC keyword can be thought of as an implicit, system-generated user who represents all HADB users.
- In this context, *authorizing access privileges* means authorizing access or operations that use access privileges.

● WITH GRANT OPTION

Specify this option when you grant an access privilege with the grant option to the privilege grantee.

Note that a user who runs the GRANT statement must have the relevant access privilege with the grant option.

Example:

```
GRANT SELECT ON "X"."T1" TO "ADBUSER01" WITH GRANT OPTION
```

When you run the preceding GRANT statement, the SELECT privilege for table X.T1 is granted to ADBUSER01 with the grant option. An HADB user who runs the GRANT statement must have the SELECT privilege with the grant option for table X.T1.

(3) Privileges required at execution

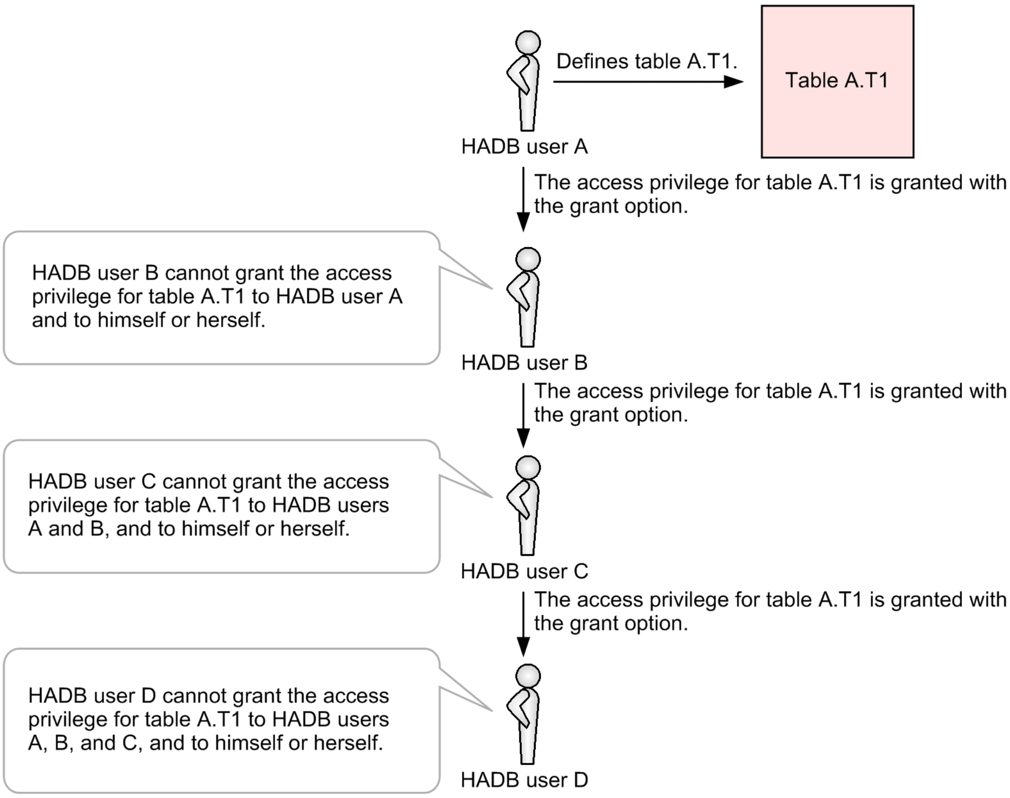
A user who runs the GRANT statement that grants an access privilege must have the following privileges:

- The CONNECT privilege
- The schema definition privilege or the access privilege with the grant option

(4) Rules

1. For you to grant an access privilege to other HADB users, you must have the grant option for that access privilege.
2. You cannot grant yourself access privileges to objects that you own.
3. Even if you have the grant option for an access privilege, you cannot grant the access privilege to the following HADB users:
    - HADB user who granted you an access privilege with the grant option
    - Any HADB users in the chain of granting the access privilege with the grant option up to the preceding HADB user
    - Yourself (you cannot grant yourself an access privilege that has been granted to you)

Example:


Defines table A.T1.
Table A.T1
HADB user A
The access privilege for table A.T1 is granted with the grant option.
HADB user B cannot grant the access privilege for table A.T1 to HADB user A and to himself or herself.
HADB user B
The access privilege for table A.T1 is granted with the grant option.
HADB user C cannot grant the access privilege for table A.T1 to HADB users A and B, and to himself or herself.
HADB user C
The access privilege for table A.T1 is granted with the grant option.
HADB user D cannot grant the access privilege for table A.T1 to HADB users A, B, and C, and to himself or herself.
HADB user D

4. For you to grant another HADB user an access privilege for a viewed table, you must have an access privilege for all underlying tables of the viewed table with the grant option.

5. If a new access privilege for an underlying table is granted, the access privilege for the viewed tables that depend on the underlying table is also granted. (Consequently, propagation of access privileges occurs.) For example, assume that HADB user A has defined viewed table `A.V1` by using table `X.T1` as the underlying table, and viewed table `A.V2` by using viewed table `A.V1` as the underlying table. In this case, if the `INSERT` privilege for table `X.T1` is granted to HADB user A, the `INSERT` privileges for viewed tables `A.V1` and `A.V2` are also granted to HADB user A.

Note

- Propagation of access privileges can occur for only viewed tables that are defined by an HADB user who is granted the access privilege.
- If viewed tables are invalidated, no propagation of an access privilege occurs for the viewed tables.

Important

Be careful when a viewed table has multiple underlying tables. In this case, if a new access privilege for an underlying table is granted, the access privilege for the viewed table can be changed only when the access privilege meets the conditions applied to the viewed table.

Example:

Assume that HADB user A has the `SELECT` privileges for tables `X.T1` and `X.T2`, and has defined viewed table `A.V1` by using tables `X.T1` and `X.T2` as the underlying tables. In this case, even if the `UPDATE` privilege for table `X.T1` is granted, the `UPDATE` privilege for viewed table `A.V1` is not granted. Unless the `UPDATE` privileges for both tables `X.T1` and `X.T2` are granted, the `UPDATE` privilege for viewed table `A.V1` is not granted. As shown earlier, even if the access privilege for only one underlying table is granted, the access privilege for the viewed table is not changed unless the conditions for access privileges applied to the viewed table are met.

6. To revoke only the grant option from an access privilege granted to another HADB user with the grant option, run the `REVOKE` statement with `GRANT OPTION FOR` specified. From an HADB user who is granted an access privilege with the grant option, you cannot revoke the grant option by running the `GRANT` statement without specifying `WITH GRANT OPTION` to regrant the same access privilege.
7. If you specify more than one authorization identifier for a privilege grantee and an error occurs in the execution of the `GRANT` statement, the granting of privileges to all of the specified HADB users is invalidated.
8. If you change the access privileges of an HADB user who is currently connected to the HADB server, the changed access privileges take effect at the following time:
   - The next time the HADB user executes a transaction

(5) Examples

Example 1

Grant the `SELECT` and `INSERT` privileges on table `T1` to HADB user `ADBUSER01`.

```
GRANT SELECT,INSERT ON "T1" TO "ADBUSER01"
```

Example 2

Grant all access privileges on table `T1` to HADB users `ADBUSER02` and `ADBUSER03`.

```
GRANT ALL PRIVILEGES ON "T1" TO "ADBUSER02","ADBUSER03"
```

Example 3

In this example, the `SELECT` privilege for table `X.T1` is granted with the grant option to HADB user `ADBUSER04`.

```
GRANT SELECT ON "X"."T1" TO "ADBUSER04" WITH GRANT OPTION
```

3.17 REVOKE (revoke privileges)

This section describes the specification format and rules for the REVOKE statement.

3.17.1 Revoking user privileges, schema operation privileges, and audit privileges

Revoke the following privileges that were granted to an HADB user.

- User privileges
  - DBA privilege
  - CONNECT privilege
- Schema operation privileges
  - Schema definition privilege
- Audit privileges
  - Audit admin privilege
  - Audit viewer privilege

(1) Specification format

```
REVOKE-statement::=REVOKE privilege[,privilege]...
                  FROM authorization-identifier[,authorization-identifier]... [drop-behavior]

  privilege::={user-privilege|schema-operation-privilege|audit-privilege}
    user-privilege::={DBA|CONNECT}
    schema-operation-privilege::=SCHEMA
    audit-privilege::={AUDIT ADMIN|AUDIT VIEWER}

  drop-behavior::={CASCADE|RESTRICT}
```

(2) Explanation of specification format

- *privilege*[,*privilege*]...

```
privilege::={user-privilege|schema-operation-privilege|audit-privilege}
```

Specifies the privilege to be revoked. You cannot specify the same privilege more than once.

user-privilege ::= {DBA | CONNECT}

Specify this to revoke user privileges.

- DBA

  Specify this to revoke the DBA privilege.

- CONNECT

  Specify this to revoke the CONNECT privilege.

schema-operation-privilege ::= SCHEMA

Specify this to revoke schema operation privileges.

- `SCHEMA`

  Specify this to revoke the schema definition privilege.

audit-privilege `::= {AUDIT ADMIN|AUDIT VIEWER}`

Specify this to revoke an audit privilege (audit admin privilege or audit viewer privilege).

- `AUDIT ADMIN`

  Specify this to revoke the audit admin privilege.

- `AUDIT VIEWER`

  Specify this to revoke the audit viewer privilege.

● `FROM` *authorization-identifier* `[,` *authorization-identifier* `]...`

Specifies the authorization identifiers of the HADB user(s) whose privileges are to be revoked. A maximum of 128 authorization identifiers can be specified.

Note the following rules for specifying an authorization identifier:

- If you want to use lowercase letters, enclose the authorization identifier in double quotation marks ("). When not enclosed in double quotation marks, lowercase letters will be treated as uppercase.

  Example: `REVOKE DBA FROM adbuser01`

  In this case, the authorization identifier is treated as `ADBUSER01`.

- Because an authorization identifier is specified as a name, we recommend that you enclose it in double quotation marks (").

For details about the rules for specifying an authorization identifier, see 6.1.4 Specifying names.

- *drop-behavior*

```
drop-behavior ::= {CASCADE | RESTRICT}
```

This specification only applies when revoking the schema definition privilege.

Specify whether to revoke the schema definition privilege if the HADB user who wants to revoke the schema definition privilege owns the schema. The following table describes the possible specifications for drop-behavior.

Specification of drop-behavior	Description	Handling of viewed tables in other schemata	Handling of foreign keys in other schemata
If drop-behavior is omitted	The schema definition privilege is revoked even if the targeted HADB user owns a schema. At this time, the schema owned by the targeted HADB user is also deleted.	If viewed tables defined in other schemata depend on the tables that will be deleted by the `REVOKE` statement, those dependent viewed tables are invalidated.	If the tables that will be deleted by the `REVOKE` statement are referenced by foreign keys defined in other schemata, those foreign keys are also deleted.
If `CASCADE` is specified		If viewed tables defined in other schemata depend on the tables that will be deleted by the `REVOKE` statement, those dependent viewed tables are also deleted.	
If `RESTRICT` is specified	If the targeted HADB user owns the schema, the `REVOKE` statement results in an error.	The viewed tables in other schemata are not affected because the `REVOKE` statement results in an error.	The foreign keys in other schemata are not affected because the `REVOKE` statement results in an error.

(3) Privileges required at execution

- To execute a `REVOKE` statement that revokes a user privilege or schema operation privilege:

  The `DBA` privilege and the `CONNECT` privilege are required.

- To execute a `REVOKE` statement that revokes an audit privilege:

The audit admin privilege and the CONNECT privilege are required.

(4) Rules

1. You cannot revoke the CONNECT privilege of the HADB user whose authorization identifier is currently connected to the HADB server.
2. You cannot revoke the DBA and CONNECT privileges that have been granted to yourself. You can revoke the schema definition privilege that has been granted to yourself.
3. If an error occurs in the execution of the REVOKE statement when multiple authorization identifiers are specified, the operation is cancelled for all of the targeted HADB users.
4. You cannot revoke the CONNECT privilege and the schema definition privilege of an HADB user who has an audit privilege.
5. An HADB user having the audit admin privilege can revoke the following privileges:
    - Other HADB users' audit admin privilege or audit viewer privilege
    - The HADB user's own audit admin privilege or audit viewer privilege
6. Audit privileges can be revoked if the audit trail facility is enabled.

    However, as an HADB user, you can revoke the audit admin privilege even when the audit trail facility is disabled if all of the following conditions are met:
    - There are no HADB users who have the audit viewer privilege.
    - You are the only HADB user who has the audit admin privilege.
7. If the audit trail facility is enabled and there is only one HADB user having both the audit admin privilege and the CONNECT privilege, the HADB user's audit admin privilege cannot be revoked.

(5) Examples

Example 1

Revoke the DBA privilege, CONNECT privilege, and schema definition privilege of HADB user ADBUSER01.

```
REVOKE DBA,CONNECT,SCHEMA FROM "ADBUSER01" CASCADE
```

Example 2

Revoke the CONNECT privilege and the schema definition privilege of HADB users ADBUSER02 and ADBUSER03. However, if HADB user ADBUSER02 or ADBUSER03 owns a schema, make the REVOKE statement result in an error.

```
REVOKE CONNECT,SCHEMA FROM "ADBUSER02","ADBUSER03" RESTRICT
```

For example, assume that ADBUSER02 owns a schema and ADBUSER03 does not. In this case, if the preceding REVOKE statement is executed, the processing of the REVOKE statement for both ADBUSER02 and ADBUSER03 results in an error.

Example 3

Revoke the audit admin privilege of HADB user ADBAUDITADMIN.

```
REVOKE AUDIT ADMIN FROM "ADBAUDITADMIN"
```

Example 4

Revoke the audit viewer privilege of HADB user ADBAUDITOR.

```
REVOKE AUDIT VIEWER FROM "ADBAUDITOR"
```

3.17.2 Revoking access privileges

Revoke access privileges that were granted to an HADB user.

(1) Specification format

```
REVOKE-statement ::= REVOKE [GRANT OPTION FOR] access-privilege ON object-name
                 FROM privilege-grantee [drop-behavior]

  access-privilege ::= {ALL [PRIVILEGES]|operation[,operation]...}
    operation ::= {SELECT|INSERT|UPDATE|DELETE|TRUNCATE|REFERENCES
              |IMPORT TABLE|REBUILD INDEX|GET COSTINFO|EXPORT TABLE
              |MERGE CHUNK|CHANGE CHUNK COMMENT|CHANGE CHUNK STATUS
              |ARCHIVE CHUNK|UNARCHIVE CHUNK}

  object-name ::= {[TABLE] table-name|ALL TABLES}

  privilege-grantee ::= {authorization-identifier[,authorization-identifier]...|PUBLI
C}

  drop-behavior ::= {CASCADE|RESTRICT}
```

(2) Explanation of specification format

- GRANT OPTION FOR

  Specify this keyword to revoke only the grant option of an access privilege. If the REVOKE statement is run with this keyword specified, the access privilege itself is not revoked. Only the grant option of the access privilege is revoked.

  Example:

  ```
  REVOKE GRANT OPTION FOR SELECT ON "X"."T1" FROM "ADBUSER01"
  ```

  If the preceding REVOKE statement is run, HADB user ADBUSER01 has only the grant option of the SELECT privilege for table X.T1 revoked. HADB user ADBUSER01 does not have the SELECT privilege for table X.T1 revoked.

- *access-privilege*

  ```
  access-privilege ::= {ALL [PRIVILEGES] | operation[,operation]...}
  ```

  Specifies the type of access privilege to be revoked.

  ALL [PRIVILEGES]:

  Specify this to revoke all access privileges.

  > **Important**
  >
  > The privileges that you can revoke by running the REVOKE statement with ALL PRIVILEGES specified are only the privileges that you granted. The access privileges granted by other HADB users are not revoked.
  >
  > Example:
  >
  > Assume that HADB user ADBUSER01 has the following access privileges for table X.T1:
  >
  > - SELECT and UPDATE privileges granted by HADB user ADBUSER02

- INSERT and DELETE privileges granted by HADB user ADBUSER03

If HADB user ADBUSER02 runs the following REVOKE statement, only the SELECT and UPDATE privileges are revoked:

```
REVOKE ALL PRIVILEGES ON "X"."T1" FROM "ADBUSER01"
```

Note

If you run the REVOKE statement with ALL PRIVILEGES specified when you have no access privilege with the grant option for the target object, the statement will result in an error.

operation [,*operation*]...:

```
operation ::= {SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES
             | IMPORT TABLE | REBUILD INDEX | GET COSTINFO | EXPORT TABLE
             | MERGE CHUNK | CHANGE CHUNK COMMENT | CHANGE CHUNK STATUS
             | ARCHIVE CHUNK | UNARCHIVE CHUNK}
```

Specifies the type of access privilege to be revoked. You cannot specify the same operation more than once.

- SELECT

  Specify this to revoke the SELECT privilege.

- INSERT

  Specify this to revoke the INSERT privilege.

- UPDATE

  Specify this to revoke the UPDATE privilege.

- DELETE

  Specify this to revoke the DELETE privilege.

- TRUNCATE

  Specify this to revoke the TRUNCATE privilege.

- REFERENCES

  Specify this to revoke the REFERENCES privilege.

- IMPORT TABLE

  Specify this to revoke the IMPORT TABLE privilege.

- REBUILD INDEX

  Specify this to revoke the REBUILD INDEX privilege.

- GET COSTINFO

  Specify this to revoke the GET COSTINFO privilege.

- EXPORT TABLE

  Specify this to revoke the EXPORT TABLE privilege.

- MERGE CHUNK

  Specify this to revoke the MERGE CHUNK privilege.

- CHANGE CHUNK COMMENT

  Specify this to revoke the CHANGE CHUNK COMMENT privilege.

    - CHANGE CHUNK STATUS
      Specify this to revoke the CHANGE CHUNK STATUS privilege.
    - ARCHIVE CHUNK
      Specify this to revoke the ARCHIVE CHUNK privilege.
    - UNARCHIVE CHUNK
      Specify this to revoke the UNARCHIVE CHUNK privilege.

- ON *object-name*

```
object-name ::= {[TABLE] table-name | ALL TABLES}
```

Specifies the object to which access privileges are to be revoked.

Here, *object* refers to a schema object.

[TABLE] *table-name*:

Revoke access privileges to the table specified here. For rules on specifying a table name, see (2) Table name specification format in 6.1.5 Qualifying a name.

Note that you cannot specify the table name of a viewed table that has been invalidated.

ALL TABLES:

Revoke access privileges to all the base tables in the schema of the executing user. In this context, *executing user* means the HADB user executing the REVOKE statement.

If the HADB user executing the REVOKE statement has not defined a schema, or no base tables are defined in the schema, the REVOKE statement terminates normally without revoking any access privileges.

- FROM *privilege-grantee*

```
privilege-grantee ::= {authorization-identifier[,authorization-identifier]...| PUB
LIC}
```

Specifies the HADB users whose access privileges are to be revoked.

authorization-identifier[,*authorization-identifier*]...:

Specifies the authorization identifiers of the HADB user(s) whose access privileges are to be revoked. A maximum of 128 authorization identifiers can be specified.

Note the following rules for specifying an authorization identifier:

- If you want to use lowercase letters, enclose the authorization identifier in double quotation marks ("). When not enclosed in double quotation marks, lowercase letters will be treated as uppercase.
- Because an authorization identifier is specified as a name, we recommend that you enclose it in double quotation marks (").

For details about the rules for specifying an authorization identifier, see 6.1.4 Specifying names.

PUBLIC:

Specify this to revoke access privileges that were authorized by the GRANT statement with PUBLIC specified.

Example:

```
REVOKE SELECT,IMPORT TABLE ON "T1" FROM PUBLIC
```

Executing the above REVOKE statement revokes the following access privileges that were authorized by a GRANT statement with PUBLIC specified:

- The SELECT privilege on table T1
- The IMPORT TABLE privilege on table T1

Note

The PUBLIC keyword can be thought of as an implicit, system-generated user who represents all HADB users.

- *drop-behavior*

```
drop-behavior ::= {CASCADE | RESTRICT}
```

This specification takes effect only in either of the following cases:

- When the SELECT or REFERENCES privilege is to be revoked
- When an access privilege for which dependent privileges exist is to be revoked

If specification of the drop behavior is omitted, the system assumes that CASCADE is specified.

CASCADE:

Specify this keyword if you want to revoke an access privilege even when any of the following conditions are met:

- There is a viewed table that was defined using the SELECT privilege to be revoked.
  In this case, the viewed table is invalidated. The viewed tables that depend on the viewed table to be invalidated are also invalidated.
- There is a referential constraint that was defined using the REFERENCES privilege to be revoked.
  In this case, the referential constraint is deleted.
- There are dependent privileges for the access privilege to be revoked (including when only the grant option is to be revoked).
  In this case, the dependent privileges are revoked. If there are viewed tables or referential constraints that use the dependent privileges, the viewed tables are invalidated and the referential constraints are deleted.

RESTRICT:

Specify this keyword if you want the REVOKE statement to result in an error in any of the following cases:

- There is a viewed table that was defined using the SELECT privilege to be revoked.
- There is a referential constraint that was defined using the REFERENCES privilege to be revoked.
- There are dependent privileges for the access privilege to be revoked (including when only the grant option is to be revoked).

(3) Privileges required at execution

A user who runs the REVOKE statement that revokes an access privilege must have the following privileges:

- CONNECT privilege
- The schema definition privilege or the access privilege with the grant option

(4) Rules

1. You can revoke only access privileges that you granted.
2. If you attempt to revoke an access privilege by running the REVOKE statement when you do not have the access privilege with the grant option, the statement will result in an error.
3. You cannot revoke your own access privileges to objects that you own.

4. If an error occurs during execution of the REVOKE statement that you ran by specifying multiple authorization identifiers as the privilege grantee, the revoking of privileges for all HADB users will become invalid.

5. If an access privilege for an underlying table is revoked, the access privilege for the viewed tables that depend on the underlying table is also revoked. (Consequently, propagation of access privileges occurs.)

   For example, assume that HADB user A has defined viewed table A.V1 by using table X.T1 as the underlying table, and viewed table A.V2 by using viewed table A.V1 as the underlying table. In this case, if the INSERT privilege for table X.T1 is revoked, the INSERT privileges for viewed tables A.V1 and A.V2 are also revoked.

   For invalidated viewed tables, however, the revoking of access privileges for those viewed tables is not propagated.

6. If the SELECT privilege for an underlying table is revoked, all viewed tables that depend on the underlying table are invalidated.

   Example:

   Assume that HADB user A has the SELECT privilege for table X.T1, and has defined viewed table A.V1 by using table X.T1 as the underlying table. Also assume that the user has defined viewed table A.V2 by using viewed table A.V1 as the underlying table, and viewed table A.V3 by using viewed table A.V2 as the underlying table.

   If the SELECT privilege for table X.T1 is revoked, viewed tables A.V1, A.V2, and A.V3, which depend on table X.T1, are invalidated.

7. If the REFERENCES privilege for a table is revoked, the referential constraints defined by using the REFERENCES privilege are deleted. For example, if the REFERENCES privilege for table X.T1 owned by HADB user A is revoked, the referential constraint that HADB user A defined by using table X.T1 as the referenced table is deleted.

8. The following describes the rules for revoking access privileges when an access privilege for the same table is granted by multiple HADB users or permitted with the PUBLIC specification. Note that the following description is an example for the SELECT privilege.

   Example:

   Assume that the following SQL statements are run:

```
GRANT SELECT ON "ADBUSER01"."T1" TO "ADBUSER03"      ...1  <= Run by HADB user AD
BUSER01
GRANT SELECT ON "ADBUSER01"."T1" TO "ADBUSER03"      ...2  <= Run by HADB user AD
BUSER02
GRANT SELECT ON "ADBUSER01"."T1" TO PUBLIC           ...3  <= Run by HADB user AD
BUSER01
REVOKE SELECT ON "ADBUSER01"."T1" FROM "ADBUSER03"   ...4  <= Run by HADB user AD
BUSER01
REVOKE SELECT ON "ADBUSER01"."T1" FROM "ADBUSER03"   ...5  <= Run by HADB user AD
BUSER02
REVOKE SELECT ON "ADBUSER01"."T1" FROM PUBLIC        ...6  <= Run by HADB user AD
BUSER01
```

   [Explanation]

   - In steps 1 to 3, the GRANT statements are run to grant (or permit) HADB user ADBUSER03 the SELECT privilege for table ADBUSER01.T1 (table T1, hereafter).
   - When the REVOKE statement in step 4 is run, only the SELECT privilege granted in step 1 is revoked. The SELECT privilege granted in step 2 and the SELECT privilege permitted in step 3 are not revoked.
   - Then, when the REVOKE statement is run in step 5, only the SELECT privilege granted in step 2 is revoked. The SELECT privilege permitted in step 3 is not revoked.
   - Then, when the REVOKE statement is run in step 6, the SELECT privilege permitted in step 3 is revoked.

     At this time, all SELECT privileges for table T1 are revoked. Therefore, if HADB user ADBUSER03 has defined a viewed table by using table T1 as the underlying table, the viewed table is invalidated at this time.

Note that the rules for revoking the SELECT privilege described in the preceding example also apply to the revoking of the REFERENCES privilege. Therefore, the referential constraints are deleted when all REFERENCES privileges for table T1 are revoked.

9. If you change the access privileges of an HADB user who is currently connected to the HADB server, the changed access privileges take effect at the following time:
    - The next time the HADB user executes a transaction

(5) Examples

Example 1

Revoke the SELECT and INSERT privileges on table T1 of HADB user ADBUSER01.

```
REVOKE SELECT,INSERT ON "T1" FROM "ADBUSER01"
```

If ADBUSER01 has defined a viewed table by using table T1 as the underlying table, the viewed table is invalidated when the preceding REVOKE statement is run. The viewed tables that depend on the viewed table to be invalidated are also invalidated.

Example 2

Revoke all the access privileges of HADB users ADBUSER02 and ADBUSER03 to table T1.

```
REVOKE ALL PRIVILEGES ON "T1" FROM "ADBUSER02","ADBUSER03" RESTRICT
```

Because RESTRICT is specified, while ADBUSER02 or ADBUSER03 is performing any of the following operations, the REVOKE statement results in an error:

- Defined a viewed table by using table T1 as the underlying table
- Defined a referential constraint whose referenced table is T1

Example 3

In the following example, HADB user ADBUSER01 has only the grant option of the SELECT privilege for table X.T1 revoked.

```
REVOKE GRANT OPTION FOR SELECT ON "X"."T1" FROM "ADBUSER01"
```

When the preceding REVOKE statement is run, HADB user ADBUSER01 does not have the SELECT privilege for table X.T1 revoked. Therefore, even if ADBUSER01 has defined a viewed table by using table X.T1 as the underlying table, the viewed table is not invalidated. However, in cases such as the following, the viewed tables defined by HADB users other than ADBUSER01 are invalidated.

- If ADBUSER01 has granted the SELECT privilege for table X.T1 to another HADB user (ADBUSER02, for example), ADBUSER02 has the SELECT privilege for table X.T1 revoked. Therefore, if ADBUSER02 has defined viewed table ADBUSER02.V1 by using table X.T1 as the underlying table, viewed table ADBUSER02.V1 is invalidated.
- Assume that ADBUSER01 has defined viewed table ADBUSER01.V1 by using table X.T1 as the underlying table. If the SELECT privilege for viewed table ADBUSER01.V1 has been granted to another HADB user (ADBUSER02, for example), ADBUSER02 has the SELECT privilege for viewed table ADBUSER01.V1 revoked. Therefore, if ADBUSER02 has defined viewed table ADBUSER02.V2 by using viewed table ADBUSER01.V1 as the underlying table, viewed table ADBUSER02.V2 is invalidated.

3.18 Definition SQL runtime considerations

1. When a definition SQL statement is executed, if it finishes normally, a `COMMIT` is automatically executed before the transaction terminates.
2. Definition SQL is not subject to rollback.
3. A definition SQL statement produces an error if both of the following conditions are met:
    - Within the connection in which the definition SQL statement is executed, there is a cursor performing retrieval from one of the following tables:
      • A dictionary table that is referenced or updated in the definition SQL statement
      • A table that is changed or deleted in the definition SQL statement
    - The above-mentioned cursor is open
4. A definition SQL statement produces an error if both of the following conditions are met:
    - The definition SQL statement is executed using the JDBC driver.
    - Within the same connection, there is a `Statement` object or a `PreparedStatement` object performing retrieval from one of the following tables:
      • A dictionary table that is referenced or updated in the definition SQL statement
      • A table that is changed or deleted in the definition SQL statement
5. If a definition SQL statement is executed using the JDBC driver, it produces an error if there is a `ResultSet` object within the same connection.

4 Data Manipulation SQL

This chapter describes the functions, rules, and specification formats of data manipulation SQL statements.

4.1 DELETE (delete rows)

This section describes the specification format and rules for the DELETE statement.

4.1.1 Specification format and rules for the DELETE statement

The DELETE statement deletes rows that satisfy the specified search conditions.

(1) Specification format

```
DELETE-statement ::= DELETE FROM table-name [[AS] correlation-name] [WHERE search-con
ditions]
```

(2) Explanation of specification format

- *table-name*

  Specifies the name of the table whose rows you want to delete (the *deletion target table*). For rules on specifying a table name, see (2) Table name specification format in 6.1.5 Qualifying a name.

  Note that you cannot specify a read-only viewed table.

- [AS] *correlation-name*

  Specifies the correlation name of the deletion target table. For details about correlation names, see (4) Table specification format in 6.1.5 Qualifying a name. For details about the effective scope of correlation names, see 6.8 Scope variables.

- WHERE *search-conditions*

  Specifies the conditions that identify the rows to be deleted in *search-conditions*. For details about search conditions, see 7.18 Search conditions.

  If the WHERE clause is omitted, all the rows in the specified table are deleted.

  The following rules apply:

  - You can specify dynamic parameters in the search conditions.

  If you specify an updatable viewed table in *table-name*, note the following points:

  - When you delete rows from the updatable viewed table, the rows of the underlying table are deleted.
  - The rows that are deleted from the underlying table are the rows that satisfy both the search conditions specified when the viewed table was defined and the search conditions specified here.
  - If the WHERE clause is omitted, the rows that are deleted from the underlying table are rows that satisfy the search conditions specified when the viewed table was defined.

(3) Privileges required at execution

To execute the DELETE statement, all of the following privileges are required:

- The CONNECT privilege
- DELETE privilege for a table whose rows are to be deleted
- SELECT privilege for a table specified in a query expression body

Example

```
DELETE FROM "T1"
    WHERE "T1"."C1" IN (SELECT "C1" FROM "T2" WHERE "C3"<=100)
```

The DELETE privilege for Table T1 and the SELECT privilege for Table T2 are required to execute the above DELETE statement.

(4) Rules

1. If there are no rows that meet the deletion conditions, SQLCODE is set to 100.
2. The total number of tables, derived tables, and table function derived tables specified in the DELETE statement cannot exceed 2,048. For rules on how to count the number of tables, derived tables, and table function derived tables specified in an SQL statement, see (4) Rules in 4.4.1 Specification format and rules for the SELECT statement.
3. If the set operations specified in the DELETE statement are all UNION, a maximum of 1,023 set operations can be specified. However, if the specified set operations include EXCEPT or INTERSECT, no more than 63 set operations can be specified.
4. A maximum of 63 outer joins (FULL OUTER JOIN) can be specified in the DELETE statement.
5. This statement cannot be used to delete rows from a dictionary table or system table.
6. The table containing the rows to be deleted cannot be specified in the FROM clause of a subquery within the search conditions.
7. The same operation or design that can be used when the DELETE statement is run for row store tables cannot be used when the statement is run for column store tables. For details, see *Criteria for selecting row store tables and column store tables*, *Checking whether a single-chunk table needs to be reorganized*, and *Checking whether a multi-chunk table needs to be reorganized* in the *HADB Setup and Operation Guide*.
8. This statement cannot delete archived rows. The DELETE statement that is run to delete archived rows will result in an error. To delete archived rows, first, unarchive the chunk that stores the rows to be deleted. Then, run the DELETE statement to delete the rows.
9. The DELETE statement can delete unarchived rows. Note, however, that the DELETE statement you run must meet all of the following conditions:
   - The archive range column is specified in a search condition.
   - In the search condition in which the archive range column is specified, only a comparison predicate, the IN predicate, or the BETWEEN predicate is specified.
   - OR, NOT, and other logical operators are not used in the search condition in which the archive range column is specified.
   - Archived rows are not specified as the deletion-target rows.

   Unless all of the preceding conditions are met, the DELETE statement will result in an error.

> **Important**
>
> In the search condition in which the archive range column is specified, the predicates that can be specified are limited. Even if logical operators such as OR and NOT are not specified in the search condition, the DELETE statement might result in an error. For details, see *Using the datetime information of the archive range column to narrow the search range* in the *HADB Application Development Guide*.

The following shows typical examples in which the `DELETE` statement can be run and cannot be run. Note that in the following examples, table `ARCHIVE-T1` is an archivable multi-chunk table, and column `RECORD-DAY` is the archive range column.

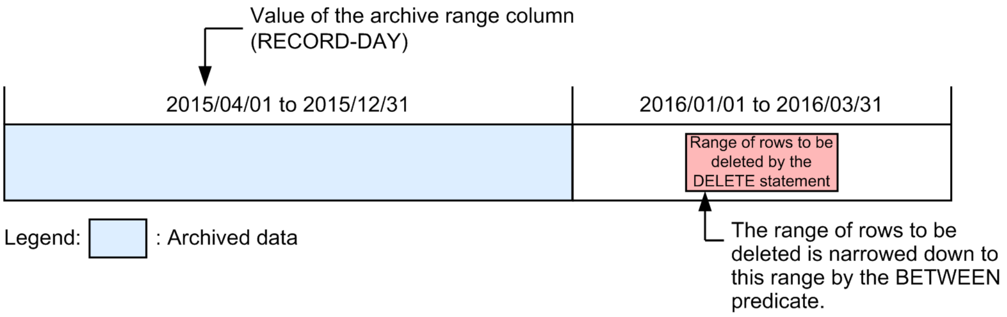
Example in which the DELETE statement can be run

Example:

```
DELETE FROM "ARCHIVE-T1"
    WHERE "RECORD-DAY" BETWEEN DATE'2016/02/01' AND DATE'2016/02/29'
      AND "CODE"='P001'
```

In the preceding example, the `DELETE` statement can be run because all of the following conditions are met:

- The archive range column (`RECORD-DAY`) is specified in a search condition.
- Only the `BETWEEN` predicate is specified in the search condition in which the archive range column is specified.
- `OR`, `NOT`, and other logical operators are not used in the search condition in which the archive range column is specified.
- Archived rows are not specified as the deletion-target rows.


Value of the archive range column (RECORD-DAY)
2015/04/01 to 2015/12/31
2016/01/01 to 2016/03/31
Range of rows to be deleted by the DELETE statement
Legend:
: Archived data
The range of rows to be deleted is narrowed down to this range by the BETWEEN predicate.

Important

For comparison with the archive range column specified in a search condition, we recommend that you specify a literal.

Example of recommended specification:

```
"RECORD-DAY" BETWEEN DATE'2016/01/01' AND DATE'2016/01/10'
"RECORD-DAY" >= DATE'2016/02/10'
```

We recommend that you specify only a literal.

Example of specification that is not recommended:

```
"RECORD-DAY" BETWEEN ? AND ?
"RECORD-DAY" >= CURRENT_DATE
```

Note

The HADB server determines whether the deletion-target data has been archived from the search condition in which the archive range column is specified. If you specify a literal as the comparison with the archive range column, you can reduce the time required for determination. If you do not specify a literal, determination might require a very long time.

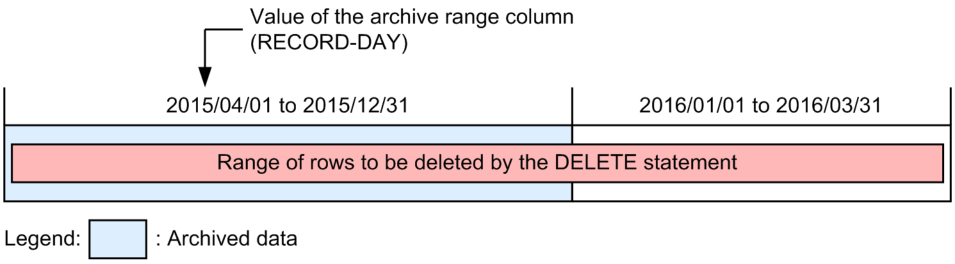
Examples in which the DELETE statement cannot be run

• No archive range column is specified in search conditions

Example 1:

```
DELETE FROM "ARCHIVE-T1" ______
```

In this example, because the archive range column (RECORD-DAY) is not specified in the search condition, the DELETE statement results in an error.


Value of the archive range column
(RECORD-DAY)
2015/04/01 to 2015/12/31
2016/01/01 to 2016/03/31
Range of rows to be deleted by the DELETE statement
Legend:
: Archived data

Example 2:

```
DELETE FROM "ARCHIVE-T1"
    WHERE "CODE"='P001'
```

In this example, because the archive range column (RECORD-DAY) is not specified in the search condition, the DELETE statement results in an error. An error occurs even when an attempt is made to delete unarchived rows.

- **Logical operations such as OR and NOT are specified** in the search condition in which the archive range column is specified

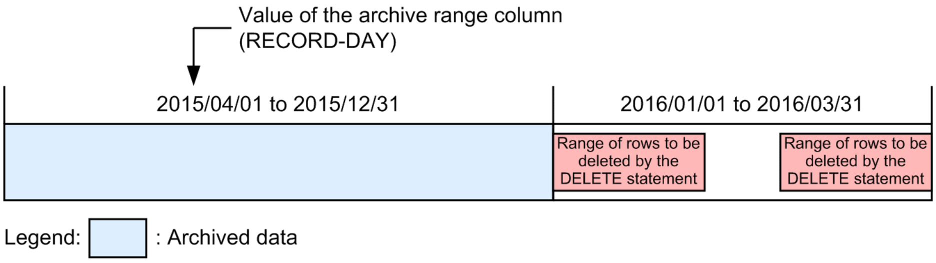
Example:

```
DELETE FROM "ARCHIVE-T1"
    WHERE "RECORD-DAY" BETWEEN DATE'2016-01-01' AND DATE'2016-01-31'
       OR "RECORD-DAY" BETWEEN DATE'2016-03-01' AND DATE'2016-03-31'
```

In this example, because the OR operator is specified in the search condition in which the archive range column is specified, the DELETE statement results in an error. The preceding statement will also result in an error when an attempt is made to delete unarchived rows.

In this case, you can delete the rows by running the DELETE statement twice as follows:

```
DELETE FROM "ARCHIVE-T1"
    WHERE "RECORD-DAY" BETWEEN DATE'2016-01-01' AND DATE'2016-01-31'
DELETE FROM "ARCHIVE-T1"
    WHERE "RECORD-DAY" BETWEEN DATE'2016-03-01' AND DATE'2016-03-31'
```

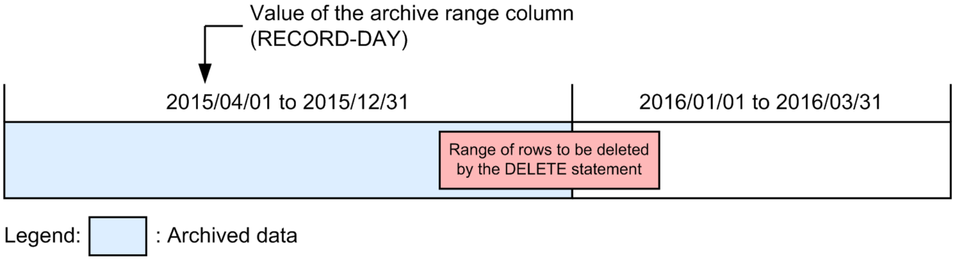

Value of the archive range column
(RECORD-DAY)
2015/04/01 to 2015/12/31
2016/01/01 to 2016/03/31
Range of rows to be deleted by the DELETE statement
Range of rows to be deleted by the DELETE statement
Legend:
: Archived data

- **Archived rows are specified as the deletion-target rows**

Example:

```
DELETE FROM "ARCHIVE-T1"
    WHERE "RECORD-DAY" BETWEEN DATE'2015/11/01' AND DATE'2016/01/31'
```

In this example, the DELETE statement results in an error because an attempt is made to delete archived rows.


Value of the archive range column
(RECORD-DAY)
2015/04/01 to 2015/12/31
2016/01/01 to 2016/03/31
Range of rows to be deleted by the DELETE statement
Legend:
: Archived data

- **The archive range column is specified together with other items**

Example:

```
DELETE FROM "ARCHIVE-T1"
    WHERE "RECORD-DAY" - 10 DAY > DATE'2016/02/01'
```

In this example, the `DELETE` statement results in an error because a datetime operation using the archive range column is specified.

- **A datetime operation is used in the comparison with the archive range column**

Example:

```
DELETE FROM "ARCHIVE-T1"
    WHERE "RECORD-DAY" >= CURRENT_DATE - 1 MONTH
```

In this example, the `DELETE` statement results in an error because a datetime operation is used in the comparison with the archive range column.

10. If an archivable multi-chunk table is specified in the `DELETE` statement, accesses to the location table and system table (`STATUS_CHUNKS`) occur. At this time, locked resources are secured for the system table (`STATUS_CHUNKS`). For details about locks, see *Locking* in the *HADB Setup and Operation Guide*.

(5) Examples

Example 1

Delete rows where the customer ID (`USERID`) is `U00212` from the customer table (`USERSLIST`).

```
DELETE FROM "USERSLIST"
    WHERE "USERID"='U00212'
```

Example 2

Delete rows where the date of purchase (`PUR-DATE`) is between September 4, 2011 and September 5, 2011 from the sales history table (`SALESLIST`).

```
DELETE FROM "SALESLIST"
    WHERE "PUR-DATE" BETWEEN DATE'2011-09-04' AND DATE'2011-09-05'
```

4.2 INSERT (insert rows)

This section describes the specification format and rules for the `INSERT` statement.

4.2.1 Specification format and rules for the INSERT statement

The `INSERT` statement inserts rows into a table. You can insert a single row by specifying a value, or insert one or more rows by using a query expression body.

(1) Specification format

■ To set insertion values on a column-by-column basis

```
INSERT-statement ::=
     INSERT INTO table-name [[AS] correlation-name]
                {[(column-name[, column-name]...)]
        {query-expression-body | VALUES(insertion-value[, insertion-value]...)}
                  | DEFAULT VALUES
                 }

insertion-value ::=  {value-expression | NULL | DEFAULT}
```

■ To insert by row

```
INSERT-statement ::=
     INSERT INTO table-name [[AS] correlation-name] (ROW)
         VALUES(row-insertion-value)

row-insertion-value ::=  dynamic-parameter
```

(2) Explanation of specification format

- *table-name*

  Specifies the name of the table into which rows are to be inserted (the *insertion target table*). For rules on specifying a table name, see (2) Table name specification format in 6.1.5 Qualifying a name.

  You cannot specify the same table that is specified in the query expression body.

  In addition, you cannot specify a read-only viewed table.

  If you specify an updatable viewed table in *table-name*, note the following points:

  - When you insert rows into an updatable viewed table, the rows are inserted into the underlying table. At this time, rows can be inserted regardless of the search conditions specified when the viewed table was defined.
  - When rows are inserted into an updatable viewed table, default values are stored in the columns of the underlying table that do not correspond to the columns of the updatable viewed table. For details about the default values for columns, see 7.10 DEFAULT clause.

    Note that if no default value for a column is specified in a `DEFAULT` clause, the null value is stored as the default value for the column.

Important

If the NOT NULL constraint (null values are not allowed) is defined on columns of the underlying table that do not correspond to the columns of the updatable viewed table, rows in which null values are stored in those columns cannot be inserted.

- [AS] *correlation-name*

  Specifies the correlation name of the insertion target table. For details about correlation names, see (4) Table specification format in 6.1.5 Qualifying a name.

- (*column-name* [, *column-name*] . . .)

  Specifies the names of the columns into which data is to be inserted.

  Columns whose names are not specified will be filled with the default values specified in the DEFAULT clauses in the CREATE TABLE statement. However, in the following cases, the null value is stored as the default value for the column:

  - When no default value for the column has been specified in a DEFAULT clause in the CREATE TABLE statement

  Note that if no column names are specified, it assumes that all the columns were specified, in the same order in which the columns were specified when the table was defined with the CREATE TABLE statement.

- *query-expression-body*

  Specifies a query expression body to be used to retrieve the data to be inserted. For details about the query expression body, see (b) query-expression-body in (2) Explanation of specification format in 7.1.1 Specification format and rules for query expressions.

  The following rules apply:

  - The table specified in the query expression body cannot be the same as the table that is the target of the insertion.

- VALUES (*insertion-value* [, *insertion-value*] . . .)

  ```
  insertion-value ::= {value-expression | NULL | DEFAULT}
  ```

  Specifies insertion values corresponding to the columns specified in the *column-name* specifications. Specify one of the following for *insertion-value*:

  value-expression:

  Specify the insertion value in the form of a value expression. For details about value expressions, see 7.20 Value expression.

  Note the following rules:

  - *value-expression* cannot include a column specification.
  - *insertion-value* cannot include a table that is the same as the insertion target table.

  NULL:

  Specify this to set the insertion value to the NULL value.

  DEFAULT:

  Specify this to set the insertion value to the default value for the column specified in the DEFAULT clause of the CREATE TABLE statement. If no default value for the column was specified in a DEFAULT clause, the null value is assumed as the default value for the column.

- DEFAULT VALUES

  Specify this if you want to insert the default column values in all of the columns in the insertion target table.

  Specifying DEFAULT VALUES is equivalent to specifying the following:

```
VALUES(DEFAULT,DEFAULT,...)
```

where the number of DEFAULT specifications is equal to the number of columns in the insertion target table.

If you specify DEFAULT VALUES for a table without a DEFAULT clause specification, the null value will be assumed as the default values for the columns, which means all of the columns will be assigned null values.

- ROW

  Specified to insert data by row. When you specify ROW, the entire row is inserted as a single item of data.

  The rules for specifying ROW are as follows:

  - It can be specified only for a FIX table.
  - You cannot specify a query expression body.

- VALUES(*row-insertion-value*)

  ```
  row-insertion-value ::= dynamic-parameter
  ```

  Specifies the data to be inserted into an entire row.

  The assumed data type of the dynamic parameter is CHAR type. The data length is the row length of the table into which data is being inserted. Align the boundaries so that there are no gaps in the structure. For details about how to calculate the row length, see the *ROWSZ* calculation formula in *Determining the number of pages for storing each type of row* in the *HADB Setup and Operation Guide*.

  Note that only one dynamic parameter can be specified.

(3) Privileges required at execution

To execute the INSERT statement, all of the following privileges are required:

- The CONNECT privilege
- INSERT privilege for a table to which rows are to be inserted
- SELECT privilege for a table specified in a query expression body

Example

```
INSERT INTO "T1"
    ("C1","C2","C3")
    SELECT "C1","C2","C3" FROM "T2" WHERE "C3"<=100
```

The INSERT privilege for Table T1 and the SELECT privilege for Table T2 are required to execute the above INSERT statement.

(4) Rules

1. The total number of tables, derived tables, and table function derived tables specified in the INSERT statement cannot exceed 2,048. For rules on how to count the number of tables, derived tables, and table function derived tables specified in an SQL statement, see (4) Rules in 4.4.1 Specification format and rules for the SELECT statement.
2. If the set operations specified in the INSERT statement are all UNION, a maximum of 1,023 set operations can be specified. However, if the specified set operations include EXCEPT or INTERSECT, no more than 63 set operations can be specified.
3. A maximum of 63 outer joins (FULL OUTER JOIN) can be specified in the INSERT statement.

4. When insertion values are set on a column-by-column basis, the number of insertion values must be the same as the number of column names. Note also that the data types of the insertion values must be the same as the data types of the columns into which the data is being inserted, or else they must be converted into assignable data types. For details about converting data into assignable data types, see 6.2.2 Data types that can be converted, assigned, and compared.

   Example:

   ```
   INSERT INTO "T1" ("C1","C2","C3")
       VALUES('U00358',5,DATE'2011-09-08')
   ```

   In this case, the following rules must be observed:

   - Because three columns (C1, C2, and C3) are specified, three insertion values must also be specified.
   - The data types of the insertion values must be the same as the data types of columns C1, C2, and C3, or they must be converted into assignable data types. For example, if column C3 is type DATE, its insertion value data must also be made type DATE.

5. If you specify a dynamic parameter as an insertion value, its assumed data type and data size will be the data type and data size of the column into which it is being inserted.
6. If you insert DECIMAL or DOUBLE PRECISION type data into a column with any of the data types listed below, the fractional (decimal) part will be truncated:

   - INTEGER
   - SMALLINT

   Furthermore, if you insert DECIMAL type data into a DECIMAL type column, any digits beyond the scaling specified for the column will be truncated. If you insert DOUBLE PRECISION type data into a DECIMAL type column, any digits beyond the scaling specified for the column will be rounded off (to the nearest even number).

7. You cannot insert character string data or binary data that is longer than the row length specified when the table was defined.
8. You cannot insert numeric data outside the numeric range of the data type defined for a column.
9. If the data being inserted into a CHAR type column is shorter than the column size, the data is left-aligned in the column and trailing spaces are added.
10. If the data being inserted into a BINARY type column is shorter than the column size, the data is left-aligned in the column and the rest of the field is set to X'00'.
11. This statement cannot be used to insert rows into a dictionary table or system table.
12. The same operation or design that can be used when the INSERT statement is run for row store tables cannot be used when the statement is run for column store tables. For details, see *Criteria for selecting row store tables and column store tables*, *Checking whether a single-chunk table needs to be reorganized*, and *Checking whether a multi-chunk table needs to be reorganized* in the *HADB Setup and Operation Guide*.

(5) Examples

Example 1: Insert rows by specifying VALUES

Insert the following data (row) into the sales history table (SALESLIST):

- Customer ID (USERID): U00358
- Product code (PUR-CODE): P003
- Quantity purchased (PUR-NUM): 5
- Date of purchase (PUR-DATE): 2011-09-08

```
INSERT INTO "SALESLIST"
      ("USERID","PUR-CODE","PUR-NUM","PUR-DATE")
      VALUES('U00358','P003',5,DATE'2011-09-08')
```

Example 2: Insert rows by specifying VALUES (specifying subqueries in the insertion value)

Insert the following data (row) into the sales history table (SALESLIST):

- Product code (PUR-CODE): P003
- Product name (PUR-NAME): the product name corresponding to product code P003 in the product table (PRODUCTLIST)
- Product color (PUR-COL): the product color corresponding to product code P003 in the product table (PRODUCTLIST)

```
INSERT INTO "SALESLIST"("PUR-CODE","PUR-NAME","PUR-COL")
    VALUES('P003',
          (SELECT "PUR-NAME" FROM "PRODUCTLIST" WHERE "PUR-CODE"='P003'),
          (SELECT "PUR-COL" FROM "PRODUCTLIST" WHERE "PUR-CODE"='P003'))
```

Example 3: Insert rows by specifying a query expression body

Insert data from the north district sales history table (SALESLIST_N) into the sales history table (SALESLIST).

- Assume that the sales history table (SALESLIST) and the north district sales history table (SALESLIST_N) have the same column structure.
- Insert data where the date of purchase (PUR-DATE_N) in the north district sales history table (SALESLIST_N) is on or after September 6, 2011.

```
INSERT INTO "SALESLIST"
      ("USERID","PUR-CODE","PUR-NUM","PUR-DATE")
      SELECT "USERID_N","PUR-CODE_N",
             "PUR-NUM_N","PUR-DATE_N"
          FROM "SALESLIST_N"
          WHERE "PUR-DATE_N">=DATE'2011-09-06'
```

Example 4: Insert rows using the ROW specification

Add new sales information to the sales history table (SALESLIST) (insert using the ROW specification). The columns that comprise the sales history table are customer ID (USERID), product code (PUR-CODE), quantity purchased (PUR-NUM), date of purchase (PUR-DATE).

```
INSERT INTO "SALESLIST"(ROW)
    VALUES(?)
```

4.3 PURGE CHUNK (delete all rows in a chunk)

This section describes the specification format and rules for the PURGE CHUNK statement.

4.3.1 Specification format and rules for the PURGE CHUNK statement

The PURGE CHUNK statement deletes all of the rows in a chunk.

The PURGE CHUNK statement can be run for only multi-chunk tables.

(1) Specification format

```
PURGE CHUNK-statement ::= PURGE CHUNK table-name [[AS] correlation-name]
                              WHERE search-condition
```

(2) Explanation of specification format

- *table-name*

  Specifies the name of the multi-chunk table to be processed (the chunk deletion target table). For rules on specifying a table name, see (2) Table name specification format in 6.1.5 Qualifying a name.

  Note that you cannot specify a viewed table.

- [AS] *correlation-name*

  Specifies the correlation name of the chunk deletion target table. For details about correlation names, see (4) Table specification format in 6.1.5 Qualifying a name.

- WHERE *search-condition*

  Specifies the ID of the chunk to be processed.

  Specify search conditions in which CHUNKID is specified. For details about search conditions, see 7.18 Search conditions.

  You must specify either a comparison predicate, IN predicate, or quantified predicate in *search-condition*.

  comparison predicate:

  For details about comparison predicates, see 7.19.7 Comparison predicate.

  The following rules apply to the PURGE CHUNK statement specifically:

  - The only comparison operator that can be specified is =.
  - CHUNKID must be specified for either comparison operand 1 or comparison operand 2.
  - The comparison operand to be compared to CHUNKID must be either an integer literal, dynamic parameter, or scalar subquery. For details about scalar subqueries, see 7.3.1 Specification format and rules for subqueries.
  - If you specify a dynamic parameter for a comparison operand, the assumed data type of the dynamic parameter is INTEGER type.

  The following are examples of the specification format:

```
WHERE CHUNKID=integer-literal
WHERE CHUNKID=?
WHERE CHUNKID=(scalar-subquery)
```

Important

When using a comparison predicate to specify the chunk ID, only one chunk ID can be specified. Therefore, to delete rows from multiple chunks using a comparison predicate, you must execute multiple PURGE CHUNK statements.

However, when using an IN predicate or quantified predicate, you can delete rows from multiple chunks that match the conditions in a single PURGE CHUNK statement.

IN predicate:

For details about IN predicates, see 7.19.3 IN predicate.

The following rules apply to the PURGE CHUNK statement specifically:

- The IN predicate must use a table subquery.
- CHUNKID must be the first value expression in the IN predicate.
- The IN predicate must not use NOT.

The following is an example of the specification format:

```
WHERE CHUNKID IN (table-subquery)
```

quantified predicate:

For details about quantified predicates, see 7.19.8 Quantified predicate.

The following rules apply to the PURGE CHUNK statement specifically:

- The only comparison operator that can be specified is =.
- Only ANY or SOME can be specified. ALL cannot be specified.
- You must specify CHUNKID as the value expression inside the quantified predicate.

The following are examples of the specification format:

```
WHERE CHUNKID=ANY(table-subquery)
WHERE CHUNKID=SOME(table-subquery)
```

(3) Privileges required at execution

To execute the PURGE CHUNK statement, both of the following privileges are required:

- The CONNECT privilege
- The TRUNCATE privilege on the chunk deletion target table

If a subquery is specified, the SELECT privilege is required on all of the tables specified in the FROM clause.

(4) Rules

1. Make sure the result of the search condition will have a data type of INTEGER or SMALLINT.
2. Logical operations (AND, OR, NOT) cannot be specified on the predicate.

   Examples that produce an error:

   ```
   WHERE CHUNKID=1 OR CHUNKID=5
   WHERE NOT(CHUNKID=1)
   ```

3. CHUNKID cannot be specified in a subquery.
4. If you specify a subquery, the selection expression to be compared to CHUNKID must have a data type of INTEGER or SMALLINT.

   Example:

   ```
   PURGE CHUNK "SALESLIST"
           WHERE CHUNKID=(
                                  SELECT "CHUNK_ID"
                                    FROM "MASTER"."STATUS_CHUNKS"
                                      WHERE "TABLE_SCHEMA" = 'ADBUSER01'
                                        AND "TABLE_NAME" = 'SALESLIST'
                                        AND "CHUNK_COMMENT" = '2015/01/24 additional data')
   ```

   The underlined portion is the selection expression to be compared to CHUNKID.
5. You cannot specify a column from the chunk deletion target table in the search condition.
6. You cannot specify the chunk deletion target table in the FROM clause of a subquery specified in the search condition.
7. If the chunk ID of the current chunk is specified, the PURGE CHUNK statement will result in an error.
8. If a non-existent chunk ID is specified, the specified chunk ID will be ignored and processing will continue.
9. The total number of tables, derived tables, and table function derived tables specified in the PURGE CHUNK statement cannot exceed 2,048. For rules and examples about how to count the number of tables, derived tables, and table function derived tables specified in an SQL statement, see (4) Rules in 4.4.1 Specification format and rules for the SELECT statement.
10. If all of the set operations specified in the PURGE CHUNK statement are UNION, a maximum of 1,023 set operations can be specified. However, if the specified set operations include EXCEPT or INTERSECT, no more than 63 set operations can be specified.
11. A maximum of 63 outer joins (FULL OUTER JOIN) can be specified in the PURGE CHUNK statement.
12. During the execution of the PURGE CHUNK statement, the DB area is locked in exclusive mode. Therefore, you cannot execute the PURGE CHUNK statement while performing operations on another table or index stored in the DB area that holds the chunk deletion target table, or the index of the chunk deletion target table.
13. If the PURGE CHUNK statement terminates successfully, a COMMIT statement is automatically executed before the transaction terminates. Therefore, there is no need to execute a COMMIT statement after the execution of the PURGE CHUNK statement.
14. If the PURGE CHUNK statement terminates successfully, the chunk to be processed (the chunk specified in the chunk ID) is deleted.
15. If both of the following conditions are met, the PURGE CHUNK statement will result in an error.
    - There is a cursor performing retrieval of the chunk deletion target table in the connection where the PURGE CHUNK statement is executing
    - The cursor is open
16. If the PURGE CHUNK statement is executed using the JDBC driver, and there is a Statement object or a PreparedStatement object performing retrieval from the chunk deletion target table within the same connection, the PURGE CHUNK statement will result in an error.

(5) Examples

Example 1

In the sales history table (SALESLIST), delete all the rows in the chunk whose ID is 1.

```
PURGE CHUNK "SALESLIST" WHERE CHUNKID=1
```

The above example uses an integer literal to specify the ID of the chunk to be deleted.

Example 2

In the sales history table (`SALESLIST`), delete all the rows in the chunk whose ID is specified in the dynamic parameter.

```
PURGE CHUNK "SALESLIST" WHERE CHUNKID=?
```

The above example uses a dynamic parameter to specify the ID of the chunk to be deleted.

Example 3

In the sales history table (`SALESLIST`), delete all the rows in the chunk meeting the following condition:

- The chunk's comment is set to `2015/01/24 additional data`

```
PURGE CHUNK "SALESLIST"
        WHERE CHUNKID=(
                       SELECT "CHUNK_ID"
                         FROM "MASTER"."STATUS_CHUNKS"
                           WHERE "TABLE_SCHEMA" = 'ADBUSER01'
                             AND "TABLE_NAME" = 'SALESLIST'
                             AND "CHUNK_COMMENT" = '2015/01/24 additional data')
```

The above example uses a scalar subquery to specify the ID of the chunk to be deleted.

Important

The above `PURGE CHUNK` statement is executed under the assumption that there is only one chunk whose comment is set to `2015/01/24 additional data`. If there are multiple such chunks, use an `IN` predicate or a quantified predicate.

Example 4

In the sales history table (`SALESLIST`), delete all the rows in the chunks meeting the following condition:

- The chunk's comment is set to `2015`*XXXX* `additional data`

where *XXXX* denotes a month and day.

```
PURGE CHUNK "SALESLIST"
        WHERE CHUNKID IN (
                          SELECT "CHUNK_ID"
                            FROM "MASTER"."STATUS_CHUNKS"
                              WHERE "TABLE_SCHEMA" = 'ADBUSER01'
                                AND "TABLE_NAME" = 'SALESLIST'
                                AND "CHUNK_COMMENT" LIKE '2015% additional data')
```

The above example uses an `IN` predicate to specify the IDs of the chunks to be deleted.

Example 5

In the sales history table (`SALESLIST`), delete all the rows in chunks that are in wait status.

```
PURGE CHUNK "SALESLIST"
        WHERE CHUNKID=ANY(
                          SELECT "CHUNK_ID"
                            FROM "MASTER"."STATUS_CHUNKS"
                              WHERE "TABLE_SCHEMA" = 'ADBUSER01'
                                AND "TABLE_NAME" = 'SALESLIST'
                                AND "CHUNK_STATUS" = 'Wait')
```

The above example uses a quantified predicate to specify the IDs of the chunks to be deleted.

4.4 SELECT (retrieve rows)

This section describes the specification format and rules for the `SELECT` statement.

4.4.1 Specification format and rules for the SELECT statement

The `SELECT` statement retrieves data from a table.

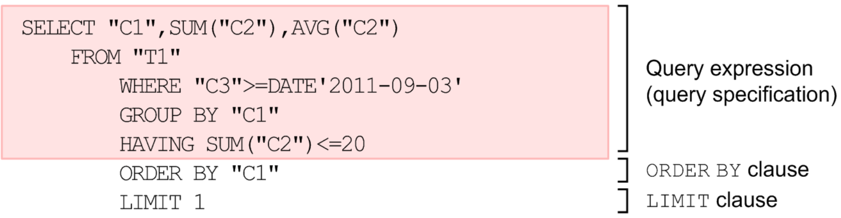
(1) Specification format

```
SELECT-statement ::= query-expression [ORDER-BY-clause] [LIMIT-clause]

  ORDER-BY-clause ::=  ORDER BY sort-specification-list
```

A `SELECT` statement consists of a query expression followed by clauses (an `ORDER BY` clause or a `LIMIT` clause). The configuration of an example `SELECT` statement is illustrated in the following figure.

Figure 4-1: Configuration of an example SELECT statement



```
SELECT "C1",SUM("C2"),AVG("C2")
    FROM "T1"
        WHERE "C3">=DATE'2011-09-03'
        GROUP BY "C1"
        HAVING SUM("C2")<=20
        ORDER BY "C1"
        LIMIT 1
```

For details about how to retrieve rows using a `SELECT` statement, see the following sections in the *HADB Application Development Guide*:

- Using the JDBC driver: *How to retrieve data* in *Retrieving data (executing the SELECT statement)*
- Using CLI functions: *Referencing data* in *How to use the CLI functions*

(2) Explanation of specification format

- *query-expression*

  Specifies a query expression. For details about query expressions, see 7.1 Query expression.

  Specify a query specification, or specify a query expression to find the union of tables derived by query specifications.

- *ORDER-BY-clause*

  ```
  ORDER-BY-clause ::= ORDER BY sort-specification-list
  ```

  Specify if you want to sort the results of the query expression in ascending or descending order. If the `ORDER BY` clause is omitted, the results of the query expression are not sorted in ascending or descending order.

  The sort specification list specifies the sort keys and the sorting order of the results. For details about the sort specification list, see 7.24 Sort specification list.

  Note the following points:

  - When the sort key is character string data, results are sorted in sort code order or bytecode order according to the sort order for character string data specified in the server definition, client definition, or connection attributes.

- When character string data is sorted in sort code order, it is sorted using the ISO/IEC 14651:2011 standard sort codes `<S0000>` to `<S2FFFF>` and subcodes `<T0000>` to `<TFFFF>` assigned to each character. Characters not assigned a sort code are sorted relative to each other in bytecode order.
- When character string data is sorted in sort code order, it is sorted as Unicode (UTF-8) bit patterns, with illegal characters treated as one-byte characters and returned at the end of the result.
- If an `ORDER BY` clause is specified, a work table might be created. If the size of the work table DB area where the work table will be created has not been estimated correctly, it might result in performance degradation. For details about estimating the size of the work table DB area, see the *HADB Setup and Operation Guide*. For details about work tables, see *Considerations when executing an SQL statement that creates work tables* in the *HADB Application Development Guide*.

● *LIMIT-clause*

Specifies the maximum number of rows to be retrieved from the results of the query expression.

For details about the `LIMIT` clause, see 7.9 LIMIT clause.

(3) Privileges required at execution

To execute the `SELECT` statement, both of the following privileges are required:

- The `CONNECT` privilege
- The `SELECT` privilege on all of the tables specified in the query specification of the `SELECT` statement

(4) Rules

1. The total number of tables, derived tables, and table function derived tables specified in all table references in a `SELECT` statement cannot exceed 2,048. However, if the SQL statement includes the following items, the total number check is performed for the SQL statement after those items are equivalently exchanged into internal derived tables:
    - Viewed tables

      If a viewed table is specified in a `CREATE VIEW` statement, the total number check is performed after the viewed table specified in the `CREATE VIEW` statement is equivalently exchanged into a derived table.
    - Query name
    - Archivable multi-chunk table

    Note

    For details about equivalent exchange of archivable multi-chunk tables, see *Equivalent exchange of SQL statements that search archivable multi-chunk tables* in the *HADB Application Development Guide*.

    The following shows an example of counting the number of tables, derived tables, and table function derived tables specified in an SQL statement.

    Example

```
WITH "Q1" AS (SELECT * FROM "T6","T7")
SELECT * FROM
  "T1",                                                 ...[a]
  "T2" LEFT OUTER JOIN "T3" ON "T2"."C1"="T3"."C1",     ...[b]
  (SELECT * FROM "T4","T5") W1,                         ...[c]
  "Q1",                                                 ...[d]
  TABLE(ADB_CSVREAD(MULTISET['/tmp/data.gz'],'COMPRESSION_FORMAT=GZIP;'))
  AS W2 ("C1" INTEGER)                                  ...[e]
  "V1",                                                 ...[f]
  "V2",                                                    ...[g]
  "T001"                                                   ...[h]
```

[Explanation]

a. A table (`T1`) is specified. Here, therefore, the number of specified tables is 1.

b. A joined table consisting of tables `T2` and `T3` is specified. Here, therefore, the number of specified tables is 2 (the total number of tables specified for the joined table).

c. A derived table is specified, and the derived query for this derived table includes two tables (`T4` and `T5`). Here, therefore, the number of specified tables is 3 in total.

d. A query name is specified. The query name is equivalently exchanged into a derived table, and the derived query for this derived table includes two tables (`T6` and `T7`). Here, therefore, the number of specified tables is 3 in total.

e. A table function derived table is specified. Here, therefore, the number of specified tables is 1.

f. A viewed table (`V1`) is specified. The viewed table is equivalently exchanged into a derived table, and the derived query for this derived table includes two tables (`T8` and `T9`). Here, therefore, the number of specified tables is 3 in total.

g. A viewed table (`V2`) is specified. The viewed table is equivalently exchanged into a derived table. The derived query for this derived table includes a viewed table (`V1`), which is equivalently exchanged into a derived table. The derived query for this derived table includes two tables (`T8` and `T9`). Here, therefore, the number of specified tables is 4 in total.

h. An archivable multi-chunk table is specified. The archivable multi-chunk table is equivalently exchanged into a derived table. The derived query for this derived table includes four tables. Here, therefore, the number of specified tables is 5 in total. `T001` is the archivable multi-chunk table that is equivalently exchanged into a derived table.

In the case of the preceding example, the total number of tables, derived tables, and table function derived tables specified in the SQL statement is 22.

Note that `V1` and `V2` are viewed tables that are defined in the following `CREATE VIEW` statements:

```
CREATE VIEW "V1" AS SELECT * FROM "T8","T9"
CREATE VIEW "V2" AS SELECT * FROM "V1"
```

2. If the set operations specified in the `SELECT` statement are all `UNION`, a maximum of 1,023 set operations can be specified. However, if the specified set operations include `EXCEPT` or `INTERSECT`, no more than 63 set operations can be specified.

3. A maximum of 63 outer joins (`FULL OUTER JOIN`) can be specified in the `SELECT` statement.

4. The names of the query expression result columns and derived columns are called retrieval item column names . When a query expression result column or derived column has no name (the length of the column name is 0), its retrieval item column name is set as follows:

   `EXP`*nnnn*`_NO_NAME`

   Legend: *nnnn*: Unsigned integer in the range from `0001` to `1000`

   Example:

```
SELECT "C1",MAX("C2"),MIN("C2")
    FROM "T1" GROUP BY "C1"
```

   When the preceding `SELECT` statement is executed, the retrieval item column names will be `C1`, `EXP0001_NO_NAME`, and `EXP0002_NO_NAME`.

5. Note that when you search an archivable multi-chunk table, you must consider the specification of search conditions in the SELECT statement. For details, see *Considerations when searching an archivable multi-chunk table* in the *HADB Application Development Guide*. Make sure that you read the preceding section when you specify a SELECT statement that searches an archivable multi-chunk table.

(5) Examples

Example 1

From the sales history table (SALESLIST), retrieve the customer ID (USERID), product code (PUR-CODE), and date of purchase (PUR-DATE) for customers who purchased product code P002 on or after September 6, 2013.

```
SELECT "USERID","PUR-CODE","PUR-DATE"
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2013-09-06'
        AND "PUR-CODE"='P002'
```

Example 2

From the employee table (EMPLIST), determine the average age (AGE) of the employees in each section (SCODE).

```
SELECT "SCODE",AVG("AGE")
    FROM "EMPLIST"
        GROUP BY "SCODE"
```

Several basic examples of the SELECT statement are shown in 1. SELECT Statement Examples. See also this chapter.

For an example of the SELECT statement in which the ORDER BY clause is specified, see (1) Examples of specifying a sort specification list in an ORDER BY clause in 7.24.4 Examples.

For an example of the SELECT statement in which the LIMIT clause is specified, see (4) Examples in 7.9.1 Specification format and rules for LIMIT clauses.

4.5 TRUNCATE TABLE (delete all rows in a base table)

This section describes the specification format and rules for the `TRUNCATE TABLE` statement.

4.5.1 Specification format and rules for the TRUNCATE TABLE statement

The `TRUNCATE TABLE` statement deletes all the rows in a base table.

(1) Specification format

```
TRUNCATE TABLE-statement ::= TRUNCATE TABLE table-name
```

(2) Explanation of specification format

- *table-name*

  Specifies the name of the base table whose rows are to be deleted (the *row deletion target table*). For the rules on specifying a table name, see (2) Table name specification format in 6.1.5 Qualifying a name.

  Note that the following tables cannot be specified:

  - Viewed tables
  - Dictionary tables
  - System tables

> **Note**
>
> If an archivable multi-chunk table is specified, the data stored in chunks that are archived and not archived is deleted.

(3) Privileges required at execution

To execute the `TRUNCATE TABLE` statement, both of the following privileges are required:

- The `CONNECT` privilege
- The `TRUNCATE` privilege on the table

(4) Rules

1. During the execution of the `TRUNCATE TABLE` statement, the DB area is locked in exclusive mode. Therefore, you cannot execute the `TRUNCATE TABLE` statement while performing operations on another table or index stored in the DB area that contains the table to be processed, or the index of the table to be processed.
2. If the `TRUNCATE TABLE` statement terminates successfully, a `COMMIT` statement is automatically executed before the transaction terminates. Therefore, there is no need to execute a `COMMIT` statement after the execution of the `TRUNCATE TABLE` statement.
3. If both of the following conditions are met, the `TRUNCATE TABLE` statement will result in an error.
   - There is a cursor performing retrieval from the table targeted by the `TRUNCATE TABLE` statement in the connection where the `TRUNCATE TABLE` statement is executing

    - The cursor is open

4. If both of the following conditions are met, the `TRUNCATE TABLE` statement will result in an error.
    - The `TRUNCATE TABLE` statement is executed using the JDBC driver
    - There is a `Statement` object or a `PreparedStatement` object performing retrieval from the table targeted by the `TRUNCATE TABLE` statement within the same connection

(5) Examples

Example

Delete all rows in the sales history table (`SALESLIST`).

```
TRUNCATE TABLE "SALESLIST"
```

4.6 UPDATE (update rows)

This section describes the specification format and rules for the UPDATE statement.

4.6.1 Specification format and rules for the UPDATE statement

The UPDATE statement updates values in a row.

(1) Specification format

■ **To update rows by specifying the names of the columns to update:**

```
UPDATE-statement ::=
     UPDATE table-name [[AS] correlation-name]
       SET update-target-column-name=update-value[, update-target-column-name=upda
te-value]...
          [WHERE search-condition]

 update-value ::=  {value-expression | NULL | DEFAULT}
```

■ To update an entire row by specifying ROW:

```
UPDATE-statement ::=
     UPDATE table-name [[AS] correlation-name]
       SET ROW=row-update-value
          [WHERE search-condition]

row-update-value ::=  dynamic-parameter
```

(2) Explanation of specification format

- *table-name*

  Specifies the name of the table to be updated (the update target table). For rules on specifying a table name, see (2) Table name specification format in 6.1.5 Qualifying a name.

  Note that you cannot specify a read-only viewed table.

- [AS] *correlation-name*

  Specifies the correlation name of the update target table. For details about correlation names, see (4) Table specification format in 6.1.5 Qualifying a name. For details about the effective scope of correlation names, see 6.8 Scope variables.

- *update-target-column-name*=*update-value* [, *update-target-column-name*=*update-value*]...

  ```
  update-value ::= {value-expression | NULL | DEFAULT}
  ```

  Specifies the columns to be updated and their update values (the values after the update).

  update-target-column-name can be specified in the form of a column specification. For details about column specifications, see (5) Column specification format in 6.1.5 Qualifying a name.

  Specify one of the following for *update-value*.

  value-expression:

  Specify the post-update value in the form of a value expression. For details about value expressions, see 7.20 Value expression.

NULL:

Specify this to set the post-update value to the null value.

DEFAULT:

Specify this to set the post-update value to the default value for the column specified in the DEFAULT clause of the CREATE TABLE statement. For details about the default values of columns, see 7.10 DEFAULT clause.

If no default value is specified in a DEFAULT clause for a column, the null value is assumed as the default value for the column.

- WHERE *search-condition*

Specifies the conditions for selecting the rows to update. If the WHERE clause is omitted, all the rows in the specified table are updated.

For details about search conditions, see 7.18 Search conditions.

The following rules apply:

- You can specify dynamic parameters in the search conditions.

If you specify an updatable viewed table in *table-name*, note the following points:

- When you update rows in an updatable viewed table, it updates the rows in the underlying table.
- The rows of the underlying table that will be updated are those that satisfy both the search conditions specified here and the search conditions specified when the viewed table was defined.
- If the WHERE clause is omitted, the rows of the underlying table that will be updated are those that satisfy the search conditions specified when the viewed table was defined.

- ROW=*row-update-value*

```
row-update-value ::= dynamic-parameter
```

Specified to insert data by row. ROW can be specified only for FIX tables. When you specify ROW, the entire row is updated as one item of data.

The assumed data type of the dynamic parameter is the CHAR type. The data length is the row length of the table being updated. Align the boundaries so that there are no gaps in the structure. For details about how to calculate the row length, see the *ROWSZ* calculation formula in *Determining the number of pages for storing each type of row* in the *HADB Setup and Operation Guide*.

Note that only one dynamic parameter can be specified.

(3) Privileges required at execution

To execute the UPDATE statement, all of the following privileges are required:

- The CONNECT privilege
- UPDATE privilege for a table whose rows are to be updated
- SELECT privilege for a table specified in a query expression body

Example

```
UPDATE "T1"
    SET "C1"='P001'
       WHERE "T1"."C2" IN (SELECT "C2" FROM "T2" WHERE "C3"<=100)
```

The UPDATE privilege for Table T1 and the SELECT privilege for Table T2 are required to execute the above UPDATE statement.

(4) Rules

1. The total number of tables, derived tables, and table function derived tables specified in the UPDATE statement cannot exceed 2,048. For rules and examples of how to count the number of tables, derived tables, and table function derived tables specified in an SQL statement, see (4) Rules in 4.4.1 Specification format and rules for the SELECT statement.
2. If the set operations specified in the UPDATE statement are all UNION, a maximum of 1,023 set operations can be specified. However, if the specified set operations include EXCEPT or INTERSECT, no more than 63 set operations can be specified.
3. A maximum of 63 outer joins (FULL OUTER JOIN) can be specified in the UPDATE statement.
4. You cannot specify the update target table in the FROM clause of a subquery in the search conditions or update values.
5. For the data types of the update values, use the data types of the columns to be updated or data types that can be converted and assigned to the columns' data types. For details about data types that can be converted or assigned, see 6.2.2 Data types that can be converted, assigned, and compared.
6. If you specify a dynamic parameter as a row update value, the assumed data type and data length will be the data type and data length of the column to be updated.
7. If you update DECIMAL or DOUBLE PRECISION type data in a column defined as any of the data types listed below, the fractional (decimal) part will be truncated:
   - INTEGER
   - SMALLINT

   Furthermore, if you use DECIMAL type data to update a DECIMAL type column, any digits beyond the scaling specified for the column will be truncated.

   If you use DOUBLE PRECISION type data to update a DECIMAL type column, any digits beyond the scaling specified for the column will be rounded off (to the nearest even number).
8. When updating a CHAR type, VARCHAR type, BINARY type or VARBINARY type column, if the data length of the update value is greater than the defined size of the column, the table cannot be updated.
9. When updating a CHAR type column, if the data length of the update value is shorter than the defined size of the column, the data is stored left-aligned and trailing spaces are added.
10. When updating a BINARY type column, if the data length of the update value is shorter than the defined size of the column, the data is stored left-aligned in the column and the rest of the field is set to X'00'.
11. When updating an INTEGER, SMALLINT, or DECIMAL type column, if the update value is outside the numeric range of the data type, the table cannot be updated.
12. A maximum of 1,000 update target column names can be specified in the SET clause.
13. If there are no rows to be updated, SQLCODE is set to 100.
14. Each column name must be unique among the columns to be updated.
15. When the ROW specification is used, you cannot specify more than one SET clause.
16. This statement cannot be used to update rows of a dictionary table or system table.
17. The same operation or design that can be used when the UPDATE statement is run for row store tables cannot be used when the statement is run for column store tables. For details, see *Criteria for selecting row store tables and column store tables*, *Checking whether a single-chunk table needs to be reorganized*, and *Checking whether a multi-chunk table needs to be reorganized* in the *HADB Setup and Operation Guide*.

18. Archived rows cannot be updated. The UPDATE statement that is run to update archived rows will result in an error. To update archived rows, first, unarchive the chunk that stores the rows to be updated. Then, run the UPDATE statement to update the rows.

19. The UPDATE statement can update unarchived rows. Note, however, that the UPDATE statement you run must meet all of the following conditions:

    - The archive range column is specified in a search condition.
    - In the search condition in which the archive range column is specified, only a comparison predicate, the IN predicate, or the BETWEEN predicate is specified.
    - OR, NOT, and other logical operators are not used in the search condition in which the archive range column is specified.
    - Archived rows are not specified as the update-target rows.

    Unless all of the preceding conditions are met, the UPDATE statement will result in an error.

    > **Important**
    >
    > In the search condition in which the archive range column is specified, the predicates that can be specified are limited. Even if logical operators such as OR and NOT are not specified in the search condition, the UPDATE statement might result in an error. For details, see *Using the datetime information of the archive range column to narrow the search range* in the *HADB Application Development Guide*.

    The following shows typical examples in which the UPDATE statement can be run and cannot be run. Note that in the following examples, table ARCHIVE-T1 is an archivable multi-chunk table, and column RECORD-DAY is the archive range column.

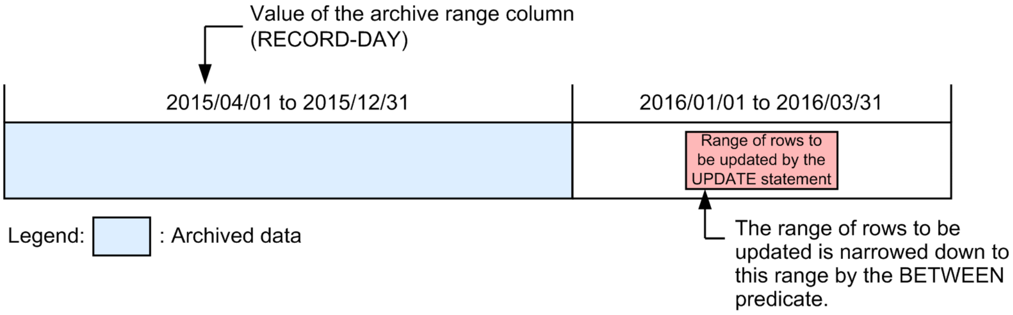
    Example in which the UPDATE statement can be run

    Example:

    ```
    UPDATE "ARCHIVE-T1" SET "NUMBER"=100
        WHERE "RECORD-DAY" BETWEEN DATE'2016/02/01' AND DATE'2016/02/29'
          AND "CODE"='P001'
    ```

    In the preceding example, the UPDATE statement can be run because all of the following conditions are met:

    - The archive range column (RECORD-DAY) is specified in a search condition.
    - Only the BETWEEN predicate is specified in the search condition in which the archive range column is specified.
    - OR, NOT, and other logical operators are not used in the search condition in which the archive range column is specified.
    - Archived rows are not specified as the update-target rows.


Value of the archive range column
(RECORD-DAY)
2015/04/01 to 2015/12/31
2016/01/01 to 2016/03/31
Range of rows to be updated by the UPDATE statement
Legend:
: Archived data
The range of rows to be updated is narrowed down to this range by the BETWEEN predicate.

Important

For the comparison with the archive range column specified in a search condition, we recommend that you specify a literal.

Example of recommended specification:

```
"RECORD-DAY" BETWEEN DATE'2016/01/01' AND DATE'2016/01/10'
"RECORD-DAY" >= DATE'2016/02/10'
```

We recommend that you specify only a literal.

Example of specification that is not recommended:

```
"RECORD-DAY" BETWEEN ? AND ?
"RECORD-DAY" >= CURRENT_DATE
```

Note

The HADB server determines whether the update-target data has been archived from the search condition in which the archive range column is specified. If you specify a literal as the comparison with the archive range column, you can reduce the time required for determination. If you do not specify a literal, determination might require a very long time.

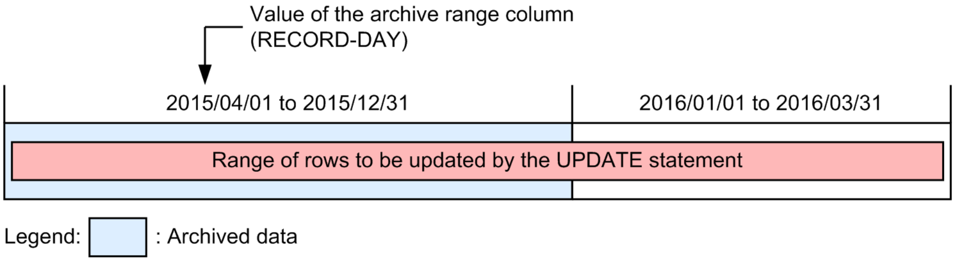
Examples in which the UPDATE statement cannot be run

• No archive range column is specified in search conditions

Example 1:

```
UPDATE "ARCHIVE-T1" SET "NUMBER"=100 _____
```

In this example, because the archive range column (RECORD-DAY) is not specified in the search condition, the UPDATE statement results in an error.


Value of the archive range column (RECORD-DAY)
2015/04/01 to 2015/12/31
2016/01/01 to 2016/03/31
Range of rows to be updated by the UPDATE statement
Legend:
: Archived data

Example 2:

```
UPDATE "ARCHIVE-T1" SET "NUMBER"=100
    WHERE "CODE"='P001'
```

In this example, because the archive range column (RECORD-DAY) is not specified in the search condition, the UPDATE statement results in an error. An error occurs even when an attempt is made to update unarchived rows.

• Logical operations such as OR and NOT are specified in the search condition in which the archive range column is specified

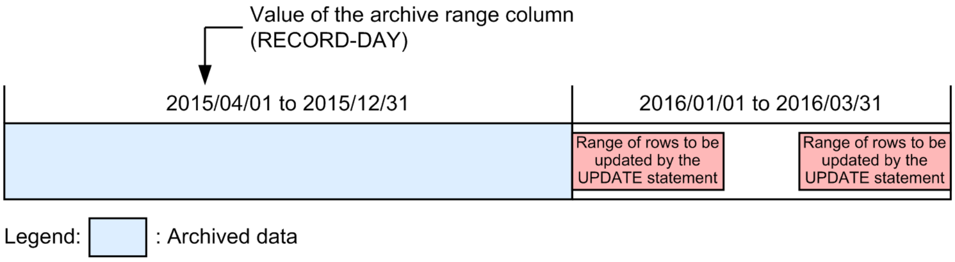
Example:

```
UPDATE "ARCHIVE-T1" SET "NUMBER"=100
    WHERE "RECORD-DAY" BETWEEN DATE'2016-01-01' AND DATE'2016-01-31'
       OR "RECORD-DAY" BETWEEN DATE'2016-03-01' AND DATE'2016-03-31'
```

In this example, because the `OR` operator is specified in the search condition in which the archive range column is specified, the `UPDATE` statement results in an error. The preceding statement will also result in an error when an attempt is made to update unarchived rows.

In this case, you can update the rows by running the `UPDATE` statement twice as follows:

```
UPDATE "ARCHIVE-T1" SET "NUMBER"=100
    WHERE "RECORD-DAY" BETWEEN DATE'2016-01-01' AND DATE'2016-01-31'
UPDATE "ARCHIVE-T1" SET "NUMBER"=100
    WHERE "RECORD-DAY" BETWEEN DATE'2016-03-01' AND DATE'2016-03-31'
```

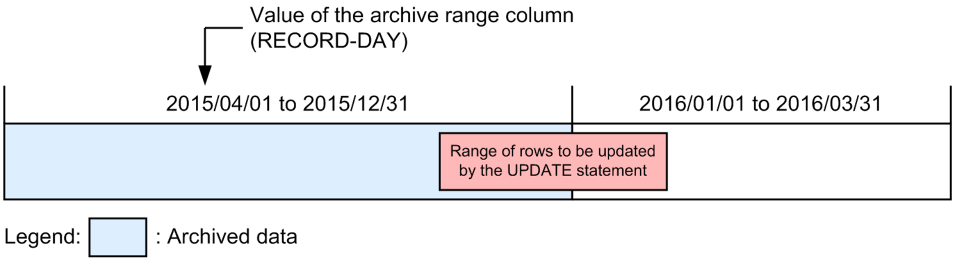

Value of the archive range column
(RECORD-DAY)
2015/04/01 to 2015/12/31
2016/01/01 to 2016/03/31
Range of rows to be updated by the UPDATE statement
Range of rows to be updated by the UPDATE statement
Legend:
: Archived data

- **Archived rows are specified as the update-target rows**

Example:

```
UPDATE "ARCHIVE-T1" SET "NUMBER"=100
    WHERE "RECORD-DAY" BETWEEN DATE'2015/11/01' AND DATE'2016/01/31'
```

In this example, the `UPDATE` statement results in an error because an attempt is made to update archived rows.


Value of the archive range column
(RECORD-DAY)
2015/04/01 to 2015/12/31
2016/01/01 to 2016/03/31
Range of rows to be updated by the UPDATE statement
Legend:
: Archived data

- **The archive range column is specified together with other items**

Example:

```
UPDATE "ARCHIVE-T1" SET "NUMBER"=100
    WHERE "RECORD-DAY" - 10 DAY > DATE'2016/02/01'
```

In this example, the `UPDATE` statement results in an error because a datetime operation using the archive range column is specified.

- **A datetime operation is used in the comparison with the archive range column**

Example:

```
UPDATE "ARCHIVE-T1" SET "NUMBER"=100
    WHERE "RECORD-DAY" >= CURRENT_DATE - 1 MONTH
```

In this example, the `UPDATE` statement results in an error because a datetime operation is used in the comparison with the archive range column.

20. If an archivable multi-chunk table is specified in the `UPDATE` statement, accesses to the location table and system table (`STATUS_CHUNKS`) occur. At this time, locked resources are secured for the system table (`STATUS_CHUNKS`). For details about locks, see *Locking* in the *HADB Setup and Operation Guide*.

(5) Examples

Example 1: Update rows by specifying the name of the column to be updated

In the sales history table (SALESLIST), update the quantity purchased (PUR-NUM) to 6 in rows that satisfy the following conditions:

- Customer ID (USERID): U00358
- Product code (PUR-CODE): P003
- Date of purchase (PUR-DATE): 2011-09-08

```
UPDATE "SALESLIST"
      SET "PUR-NUM"=6
      WHERE "USERID"='U00358'
        AND "PUR-CODE"='P003'
        AND "PUR-DATE"=DATE'2011-09-08'
```

Example 2: Update rows by specifying the name of the column to be updated (specifying a subquery for the update values)

Update the product color (PUR-COL) of the product whose product code (PUR-CODE) column's value is P003 in the sales history table (SALESLIST) so that it is the same color as the product whose product code (PUR-CODE) column's value is P003 in the product table (PRODUCTLIST).

```
UPDATE "SALESLIST"
    SET "PUR-COL" = (SELECT "PUR-COL" FROM "PRODUCTLIST" WHERE "PUR-CODE"='P003')
    WHERE "PUR-CODE"='P003'
```

Example 3: Update rows by ROW specification

Update the sales information in the sales history table (SALESLIST) (update the entire row using the ROW specification). The sales history table comprises the columns customer ID (USERID), product code (PUR-CODE), quantity purchased (PUR-NUM), and date of purchase (PUR-DATE).

```
UPDATE "SALESLIST"
    SET ROW=?
    WHERE "USERID"=?
```

5 Control SQL

This chapter describes the functions, rules, and specification formats of control SQL statements.

5.1 COMMIT (terminate a transaction normally)

This section describes the specification format for the COMMIT statement.

5.1.1 Specification format for the COMMIT statement

The COMMIT statement validates the database contents that were updated by a transaction, and terminates the transaction normally.

(1) Specification format

You can specify a COMMIT statement when you use the adbsql command to execute an SQL statement. You cannot specify a COMMIT statement in an application program.

The specification format of a COMMIT statement in the adbsql command is as follows:

```
COMMIT-statement ::= COMMIT
```

(2) Privileges required at execution

To execute the COMMIT statement, the CONNECT privilege is required.

5.2 ROLLBACK (cancel a transaction)

This section describes the specification format for the ROLLBACK statement.

5.2.1 Specification format for the ROLLBACK statement

The ROLLBACK statement invalidates the database contents that were updated by a transaction, and cancels the transaction.

(1) Specification format

You can specify a ROLLBACK statement when you use the adbsql command to execute an SQL statement. You cannot specify a ROLLBACK statement in an application program.

The specification format of a ROLLBACK statement in the adbsql command is as follows:

```
ROLLBACK-statement ::= ROLLBACK
```

(2) Privileges required at execution

To execute the ROLLBACK statement, the CONNECT privilege is required.

6 SQL Basics

This chapter describes the basic elements of SQL.

6.1 SQL writing conventions

This section presents SQL writing conventions.

6.1.1 Rules for writing SQL statements

(1) Specifying the order of options

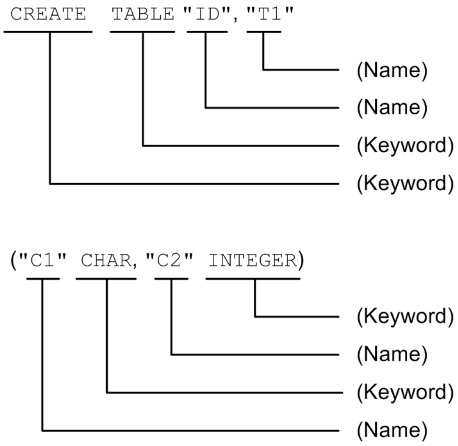
Specify options in the order in which they are described in the specification format of each SQL statement.

(2) Specifying keywords

Terms that have to be specified in order to use a built-in SQL capability, such as the names of SQL statements (`SELECT`, `UPDATE`, and so on), are called *keywords*. Because most keywords are registered as system-reserved words, they can be specified only at prescribed positions within SQL statements.

However, keywords that are not registered as reserved words can be used as names. Examples of keywords and names are given in the following figure.

Figure 6-1: Examples of keywords and names



The following are specified as names:

- Index identifiers
- Correlation names
- Query names
- Authorization identifiers
- Schema identifiers
- Table identifiers
- Column names
- Constraint names
- DB area names

For details about reserved words, see 6.10 Reserved words.

(3) Specifying numeric values

In SQL statements, specify numeric values that are not numeric literals using the conventions and restrictions of unsigned integers. The following are numeric values that are not numeric literals:

- Percentage of unused area (percentage of unused area in table and index definitions)
- Length, maximum length (length and maximum length of character string data and binary data)
- Precision (number of digits for decimal data)
- Scaling (number of digits to the right of the decimal point for decimal data)
- Fractional seconds precision (number of digits in the fractional seconds of time data and time stamp data)

(4) Maximum size of an SQL statement

The maximum size of an SQL statement is 16,000,000 bytes.

For a view definition (`CREATE VIEW` statement), the maximum size is 64,000 bytes.

6.1.2 Rules for separators

(1) About separators

When writing SQL statements, you need to put separators between two keywords, or between keywords and names, and so on. Separators include the following:

- Spaces
  The following white space characters are treated as spaces:
  - Space
  - CR (carriage return): Return
  - NL (new line): Line break
  - Tab
- Comments

(2) Where separators must be inserted

Separators must be inserted in the following places:

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

Examples of where separators must be inserted are shown in the following figure.

Figure 6-2: Examples of where separators must be inserted

```
SELECT   ▲      "USERID","PUR_CODE"
Keyword           Name

FROM     ▲      "SALESLIST"
Keyword           Name

WHERE    ▲      "USERID"='U00358'
Keyword           Name
```

Legend:
▲ : Separator

(3) Where separators cannot be inserted

Separators cannot be inserted in the following places:

- Inside a keyword
- Inside a name that is not enclosed in double quotation marks (`"`)
- After the opening double quotation mark (`"`) that encloses a name
- Before the closing double quotation mark (`"`) that encloses a name
- Inside a numeric literal (except after the sign specified at the beginning)
- Between the `X` and the following `'` in the hexadecimal-format binary literal representation `X'`...`'`
- Between the `B` and the following `'` in the binary-format binary literal representation `B'`...`'`
- Inside an operator (inside a comparison operator consisting of two characters)

Examples of where separators cannot be inserted are shown in the following figure.

Figure 6-3: Examples of where separators cannot be inserted

```
S▲ELECT   678▲9     "USERLIST▲""▲USERLIST"    <▲=
Keyword   Numeric literal   Name              Operator
```

Legend:
▲ : Separator

(4) Where separators can be inserted

Separators can be inserted in the following places:

- In places not prohibited under (3) Where separators cannot be inserted above, as well as before and after the following special characters:
  `, . -+ * '" ( )< > = ^ ! ?` tab NL CR space

Examples of where separators can be inserted are shown in the following figure.

Figure 6-4: Examples of where separators can be inserted

```
SCORE▲.SNAME    (Insert a space before the . special character.)
SCORE =▲1       (Insert a space after the = special character.)
```

Legend:

▲ : Separator

(5) Comments

You can add a comment at any location in an SQL statement where a separator can be inserted. Comments are illustrated in the example below.

Example

```
SELECT "C1","C2" FROM "T1"    /* comment1 */
    ORDER BY "C1" ASC         /* comment2 */
```

The underlined portions are comments. Everything between the `/*` and the `*/` is considered part of the comment.

When writing comments, the following guidelines must be observed:

- A comment cannot be placed inside an identifier or character string literal.
- Comments cannot be nested.

  Example

  ```
  SELECT * FROM  "T1" /* /* comment1 */ comment2 */
  ```

  Specifying nested comments as shown above results in a syntax error.
- If the `/*` and `*/` are enclosed in double quotation marks (`"`) or single quotation marks (`'`), they are not treated as defining a comment.

  Example

  ```
  SELECT * FROM "T1" WHERE "C1"='/* comment */'
  ```

  The underlined portion above is treated as a character string literal rather than a comment.
- Note that a character string that begins with `/*>>` and ends with `<<*/` is not treated as a comment.

  Example

  ```
  SELECT * FROM "T1" /*>> WITHOUT INDEX <<*/
  ```

  In the preceding example, the underlined portion is treated as an index specification rather than a comment. For details about index specifications, see 7.14 Index specification.

  However, a comment can appear within the index specification enclosed in `/*>>` and `<<*/`.

  Example

  ```
  SELECT * FROM "T1" /*>> WITHOUT INDEX /* WITH INDEX(INDEXNAME)*/ <<*/
  ```

  The underlined portion above is treated as a comment. When processed, it is equivalent to the following SQL statement:

  ```
  SELECT * FROM "T1" /*>> WITHOUT INDEX <<*/
  ```
- Note that a character string that begins with `/*>>` and ends with `<<*/` cannot be specified within a comment. For example, it is impossible to include an index specification in a comment.

  Example

```
SELECT * FROM "T1" /* /*>> WITHOUT INDEX <<*/ */
```

In the preceding example, an index specification (the underlined portion) appears within a comment. Therefore, this SQL statement results in a syntax error.

6.1.3 Characters permitted in SQL statements

The following table lists the characters that are permitted in SQL statements.

Table 6-1: Characters that are permitted in SQL statements

No.	Type	Characters permitted in SQL statements
1	Character string literals	All characters except for the character encoding X'00'
2	Other than above	• The following characters: Uppercase alphabetic characters (A to Z, #, @, \) Lowercase alphabetic characters (a to z) Numeric characters (0 to 9) space underscore character (_) • The following special characters: 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 (/) Question mark (?) Percent sign (%) Vertical bar (|) Left square bracket ([) Right square bracket (]) Tab NL CR

(1) Character encodings permitted in SQL statements

The character encodings that are permitted in SQL statements depend on the character encoding being used by HADB. The following table shows the relationship between the character encoding used by HADB and the character encodings permitted in SQL statements.

Table 6-2: Relationship between the character encoding used by HADB and the character encodings permitted in SQL statements

Character encoding used by HADB	Character encodings permitted in SQL statements
Unicode (UTF-8)	JIS X 0221
Shift-JIS	JIS X 0201 and JIS X 0208

(2) Character handling

In character string data, each character takes up a certain number of bytes. This number is determined according to the relationship between the character encoding range and the required number of bytes, shown in the table below. If the number of bytes to the end of the character string data is less than the required number of bytes, the data is assumed to begin with a one-byte character consisting of the first byte, and the next byte is assumed to be the starting point for the next character.

Table 6-3: Relationship between the character encoding range and the number of bytes

Character encoding used by HADB	Range of first byte	Range of second and subsequent bytes	Required number of bytes
Unicode (UTF-8)	`0x00` to `0x7F`	--	1
	`0xC0` to `0xDF`	N	2
	`0xE0` to `0xEF`	N	3
	`0xF0` to `0xF7`	N	4
	`0xF8` to `0xFB`	N	5
	`0xFC` to `0xFD`	N	6
	Other than above	--	1
Shift-JIS	`0x00` to `0x7F`	--	1
	`0x81` to `0x9F`	`0x40` to `0x7E` or `0x80` to `0xFC`	2
		Other than above	1
	`0xA1` to `0xDF`	--	1
	`0xE0` to `0xFC`	`0x40` to `0x7E` or `0x80` to `0xFC`	2
		Other than above	1
	Other than above	--	1

Legend:

N: No range specified.

--: Not applicable.

6.1.4 Specifying names

(1) About names

The following are specified as names:

- Index identifiers
- Correlation names
- Query names
- Authorization identifiers
- Schema identifiers
- Table identifiers
- Column names
- Constraint names
- DB area names

A name can be specified either enclosed in double quotation marks (") or not. When specifying a name, we recommend enclosing it in double quotation marks ("). If a name containing alphabetic characters is enclosed in double quotation marks ("), it becomes case sensitive.

> **Note**
>
> You cannot specify a name that is the same as a reserved word, unless you enclose the name in double quotation marks ("). As the functional scope of SQL is extended, reserved words might be added, so we recommend that you enclose all names in double quotation marks to avoid potential conflict with reserved words that might be added in the future.

Note that a name is specified as an *identifier*. Identifiers include *normal identifiers*, which are not enclosed in double quotation marks ("), and *delimited identifiers*, which are enclosed in double quotation marks ("). The following are examples of specifying a normal identifier and a delimited identifier.

- Specifying a table identifier using the normal identifier format
  Example: `table01`
- Specifying a table identifier using the delimited identifier format
  Example: `"table01"`

(2) Rules for characters that can be used in names

- The first character of a name must be an uppercase alphabetic character, lowercase alphabetic character, half-width katakana character, or full-width character. Note, however, that a name can begin with a (half-width) left or right parenthesis in the following sections of the first query specification (except the query specification in the `WITH` clause of the `SELECT` statement):
  - Name of the *selection-expression* `AS` *column-name* column
  - Name of a column in the `ORDER BY` clause (except the `ORDER BY` clause specified in a subquery)

  Note that if you specify a column name that includes a half-width parenthesis, you must enclose it in double quotation marks ("). That is, you must specify it as a delimited identifier.

- The following table lists restrictions on characters and lengths that are permitted for names.

Table 6-4: Restrictions on characters and lengths that are permitted for names

No.	Type of name	Maximum length (bytes)	Characters[#1]						Full-width characters[#1, #4]
			Uppercase and numeric characters	Lowercase characters[#2]	Katakana characters	Underscore (_)	Space[#3]	Hyphen (-)	
1	Index identifier	100	Y	Y	Y	Y	D	D	Y
2	Correlation name	100	Y	Y	Y	Y	D	D	Y
3	Query name	100	Y	Y	Y	Y	D	D	Y
4	Authorization identifier[#5]	100	Y	Y	N	N	N	N	N
5	Schema identifier	100	Y	Y	N	N	N	N	N
6	Table identifier	100	Y	Y	Y	Y	D	D	Y
7	Column name	100	Y	Y	Y	Y	D	D	Y
8	Constraint name	100	Y	Y	Y	Y	D	D	Y
9	DB area name	30	Y[#6]	Y	N	Y	N	D	N

Legend:

Y: Can be used.

D: Can be used when specifying a name using the delimited identifier format. This character cannot be used when specifying a name using the normal identifier format.

N: Cannot be used.

#1

Names can use a mixture of half-width characters and full-width characters.

#2

If the name is specified using the normal identifier format, single-byte lowercase letters are treated as single-byte uppercase letters.

If the name is specified using the delimited identifier format, single-byte lowercase and uppercase letters are distinguished.

#3

If the name is specified using the delimited identifier format, the final character of the name cannot be a single-byte space.

#4

Double-byte spaces are not permitted.

#5

`ALL`, `HADB`, `MASTER`, and `PUBLIC` cannot be specified as authorization identifiers.

#6

Not including `#`, `@`, and `\`.

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

Change the SQL statement by enclosing the name that is in conflict with a reserved word in double quotation marks ("), thus specifying it in the delimited identifier format.

Note that if a name containing alphabetic characters is enclosed in double quotation marks ("), it becomes case sensitive.

6.1.5 Qualifying a name

You can qualify a name so that you can explicitly specify a schema name or make a name unique, among other uses. You qualify a name by connecting one name (such as a schema name) to another name (such as a table identifier) using a dot (.).

(1) Schema name specification format

Format

```
schema-name ::= schema-identifier
```

Specify a schema name as a schema identifier.

If the owner of the table or index specified in the SQL statement is the HADB user who is connected to the HADB server, this user specifies his or her own authorization identifier.

If the schema name is omitted in the SQL statement, the authorization identifier specified when connecting to the HADB server is assumed as the schema name.

To search a dictionary table or system table, specify `MASTER` as the schema name.

(2) Table name specification format

Format

```
table-name ::= [schema-name.]table-identifier
```

Specify a table name as a table identifier, optionally qualified by a schema name. Specify either a base table name or a viewed table name as the table identifier.

If the schema name is omitted in the SQL statement, the authorization identifier specified when connecting to the HADB server is assumed as the schema name.

(3) Index name specification format

Format

```
index-name ::= [schema-name.]index-identifier
```

Specify an index name as an index identifier, optionally qualified by a schema name.

If the schema name is omitted in the SQL statement, the authorization identifier specified when connecting to the HADB server is assumed as the schema name.

(4) Table specification format

If you specify two or more tables in a single SQL statement, in order to identify which table the specified column or asterisk (*) corresponds to, you must qualify it using a table name, query name, or correlation name to uniquely identify the table.

For details about query names, see (a) WITH-clause in (2) Explanation of specification format in 7.1.1 Specification format and rules for query expressions.

A *correlation name* is an alias for a table. It is used in the following circumstances:

- When you want to join a table to itself
- When you want to reference the columns of a table from an outer query after specifying the same table in a subquery

By specifying a correlation name, a single table can be treated as if it were two different tables.

Format

```
table-specification ::= {table-name | query-name | correlation-name}
```

The following is an example of qualification using a table specification.

Example:

In order to reference columns with the same name (DNO) in multiple tables (EMP, DEPT), qualify them with their table names.

```
SELECT "ENO","ENAME","EMP"."DNO","DNAME"
    FROM "EMP","DEPT"
    WHERE "EMP"."DNO"="DEPT"."DNO"
```

The underlined portions are examples of qualification using a table specification.

(5) Column specification format

A column name that is qualified with a table specification is called a *column specification*.

Format

```
column-specification ::= [table-specification.]column-name
```

A column name cannot be qualified with a table specification if it is not within the scope of the specified table name or query name. For details about the scope of table names and query names, see 6.8 Scope variables.

The column name must exist in the position where it is specified in the table, or derived table, whose scope it falls under. For rules about the column names of derived tables, see 6.9 Derived column names.

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.

In the following case, a column name must be qualified with a table 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).

6.2 Data types

This section describes the data types supported by HADB.

6.2.1 List of data types

The following table lists the data types supported by HADB.

Table 6-5: List of data types supported by HADB

No.	Class	Data type	Data type code#1		Length of data storage (units: bytes)	Data format
			Decimal	Hex		
1	Numeric data	`INTEGER`	241	F1	8	Integer (8-byte)
2		`SMALLINT`	245	F5	4	Integer (4-byte)
3		`DECIMAL(`*m*`,`*n*`)`	229	E5	• If $1 \leq m \leq 4$: 2 • If $5 \leq m \leq 8$: 4 • If $9 \leq m \leq 16$: 8 • If $17 \leq m \leq 38$: 16	Fixed-point number
4		`DOUBLE PRECISION`	225	E1	8	Double-precision floating-point number
5	Character string data	`CHARACTER(`*n*`)`	197	C5	*n*	Fixed-length character string
6		`VARCHAR(`*n*`)`	193	C1	$n + 2$	Variable-length character string
7	Datetime data	`DATE`	113	71	4	Data type for dates, with fields for the year, month, and day
8		`TIME(`*p*`)`	121	79	$3 + \uparrow p \div 2 \uparrow$	Data type for time, with fields for the hour, minute, and seconds
9		`TIMESTAMP(`*p*`)`	125	7D	$7 + \uparrow p \div 2 \uparrow$	Data type for time stamps, with fields for the year, month, day, hour, minute, and seconds
10	Binary data	`BINARY(`*n*`)`	149	95	*n*	Fixed-length binary data
11		`VARBINARY(`*n*`)`	145	91	$n + 2$	Variable-length binary data
12	Row data	`ROW`	69	45	Row length#2	Data type used for row interface

Legend:

Class: Classification

Hex: Hexadecimal

#1

The code that represents the data type of the retrieval results column.

When using a CLI function, the data type code is stored in the structure `a_rdb_SQLDataType_t`.

#2

The row length is the sum of the data storage size of each column.

(1) Numeric data

■ INTEGER

- This data type handles integer values in the range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
- The following shows the format to use when specifying this data type:
  INT or INTEGER
- The data is in 8-byte binary format.
- Literals are written in the form 100, 200, and so on. For details about literals, see 6.3 Literals.

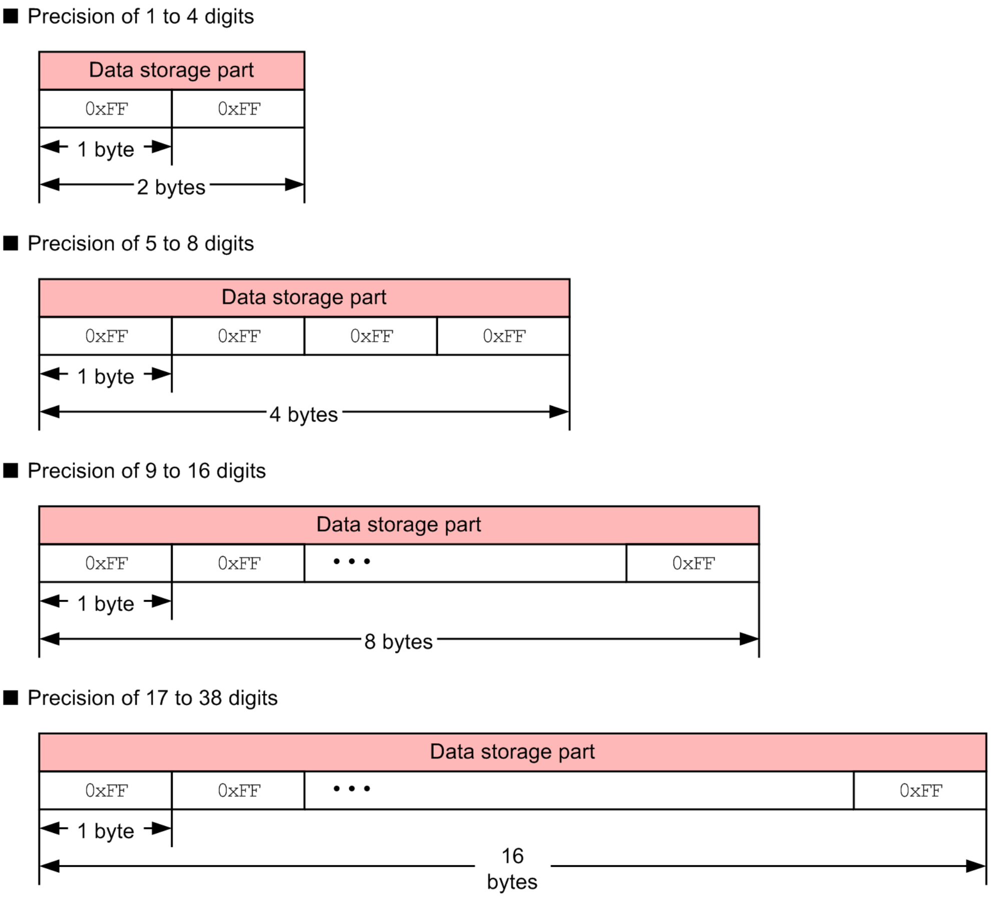
■ SMALLINT

- This data type handles integer values in the range -2,147,483,648 to 2,147,483,647.
- The following shows the format to use when specifying this data type:
  SMALLINT
- The data is in 4-byte binary format.
- Using the SMALLINT can reduce the size of the database compared with using the INTEGER.

■ DECIMAL

- This data type handles fixed-point numbers.
- The following shows the format to use when specifying this data type:
  {DEC | DECIMAL}[(*m*[,*n*])]
- The precision (overall number of digits) is specified in *m*, and the scaling (number of digits in the fractional part) is specified in *n*.
- *m* and *n* are positive integers such that $1 \leq m \leq 38$, $0 \leq n \leq 38$, $n \leq m$.
- If *m* is omitted, 38 is assumed, and if *n* is omitted, 0 is assumed.
- The data is stored internally in binary format. The binary value that is stored depends on the scaling.
- Negative values are represented in two's complement format.
- The data is stored as integer data of 2 to 16 bytes, depending on the precision, as illustrated in the following figure:

Figure 6-5: Data format of DECIMAL



- Literals are written in the form `123.4`, `12.345`, and so on. For details about literals, see 6.3 Literals.

■ `DOUBLE PRECISION`

- This data type handles double-precision floating-point numbers. The ranges of values covered include approximately -1.7 × 10^{308} to -2.3 × 10^{-308}, 0, and approximately 2.3 × 10^{-308} to 1.7 × 10^{308}.

  The exact range of values depends on the hardware representation.

- The following shows the format to use when specifying this data type:

  `DOUBLE` or `DOUBLE PRECISION`

- The data is an 8-byte floating-point number.
- In the case of literals such as `1.0e2` or `-3.4E-1`, the mantissa is represented by an integer or decimal literal, and the exponent is stored as an integer of no more than 3 digits. For details about literals, see 6.3 Literals.
- This data type cannot handle NaN (not a number) and infinite values.
- `-0` is converted to `+0`.
- Subnormal numbers are converted to `+0`.
- When floating-point data is rounded, it is rounded to the nearest even number.

(2) Character string data

■ `CHARACTER`

- This data type handles fixed-length character strings.
- The following shows the format to use when specifying this data type:

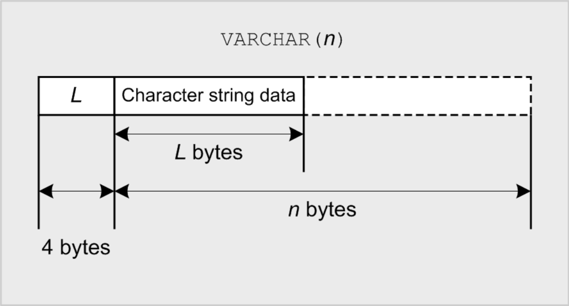
CHAR, CHAR(*n*), CHARACTER, or CHARACTER(*n*)

- The length of the character string (number of bytes) is specified in *n*, where *n* is an integer in the range 1 to 32,000. If *n* is omitted, 1 is assumed.
- Literals are written in the form 'char'. For details about literals, see 6.3 Literals.
- Both half-width and full-width characters can be handled.
- When you perform comparisons on character string data, the ordering of the character encoding determines the ordering of the data being compared.

■ VARCHAR

- This data type handles variable-length character strings.
- The following shows the format to use when specifying this data type:
  VARCHAR(*n*)
- In the preceding format, *n* specifies (in bytes) the maximum length of each character string. The value of *n* must be an integer in the range from 1 to 64,000. *n* cannot be omitted.
- The data format of the VARCHAR type is shown in the following figure.

Figure 6-6: Data format of VARCHAR type



The character string data length (*L*) is represented by four bytes.

- Both half-width and full-width characters can be handled. The length of the character string can be 0 bytes.
- When you perform comparisons on character string data, the ordering of the character encoding determines the ordering of the data being compared.
- You cannot specify VARCHAR-type data whose length exceeds 32,000 bytes in the following locations:
    - Data type specified in the column definition in an ALTER TABLE statement
    - Data type specified in the column definition in a CREATE TABLE statement
    - Data type specified in a table function column list
    - Post-conversion data type specified in the scalar function CAST
    - Post-conversion data type specified in the scalar function CONVERT

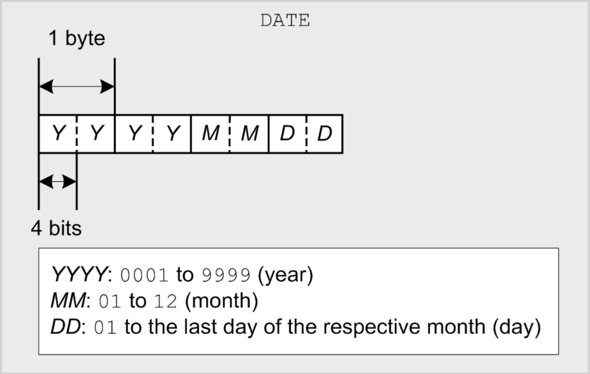
(3) Datetime data

■ DATE

- This is the data type for dates, with fields for the year, month, and day.
- The following shows the format to use when specifying this data type:
  DATE
- It can handle dates with a range of values from January 1, 0001 to December 31, 9999.

- The data length is 4 bytes. The data that is entered must be this length.
- The data format of the DATE type is shown in the following figure.

Figure 6-7: Data format of DATE type


DATE
1 byte
Y Y Y Y M M D D
4 bits
YYYY: 0001 to 9999 (year)
MM: 01 to 12 (month)
DD: 01 to the last day of the respective month (day)

- Literals are written in the form `DATE'2012-03-30'` or `DATE'2012/03/30'`. For details about literals, see 6.3 Literals.

■ TIME

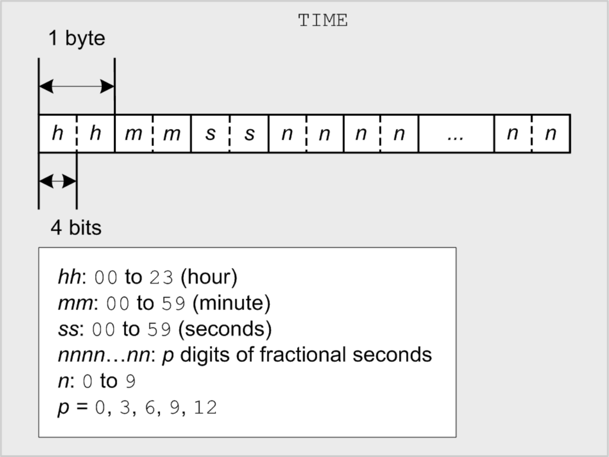
- This is the data type for time, with fields for the hour, minute, and seconds.
- The following shows the format to use when specifying this data type:

  TIME(*p*) or TIME

  p specifies the fractional seconds precision (the number of digits to the right of the decimal point). You can specify a value of 0, 3, 6, 9, or 12 for *p*.

  If TIME is specified, *p* is assumed to be 0.
- This format can handle times with a range of values from 0 hours, 0 minutes, and 0.000000000000 seconds to 23 hours, 59 minutes, and 59.999999999999 seconds.
- The data length is 3 + ↑*p* ÷ 2↑ bytes. The data that is entered must be this length.
- The data format of the TIME type is shown in the following figure.

Figure 6-8: Data format of TIME type



  One digit is represented in 4 bits. If the fractional seconds precision is an odd number, zeros are stored in the final 4 bits.
- Literals are written in the form `TIME'11:03:58.123456'`. For details about literals, see 6.3 Literals.

■ TIMESTAMP

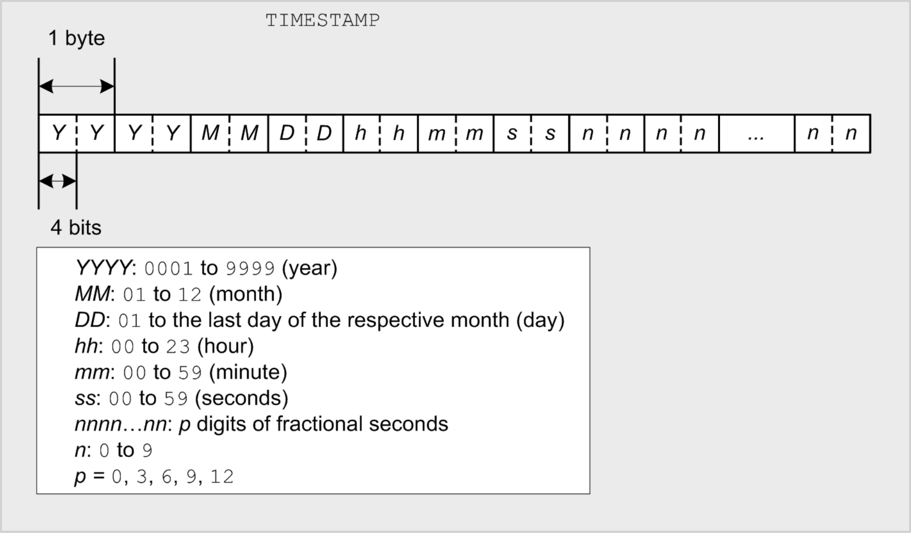
- This is the data type for time stamps, with fields for the year, month, day, hour, minute, and seconds.
- The following shows the format to use when specifying this data type:

  TIMESTAMP (*p*) or TIMESTAMP

  p specifies the fractional seconds precision (the number of digits to the right of the decimal point). You can specify a value of 0, 3, 6, 9, or 12 for *p*.

  If TIMESTAMP is specified, *p* is assumed to be 0.
- It can handle time stamps with a range of values from January 1, 0001 0:0:0.000000000000 to December 31, 9999 23:59:59.999999999999.
- The data length is 7 + ↑*p* ÷ 2↑ bytes. The data that is entered must be this length.
- The data format of the TIMESTAMP type is shown in the following figure.

Figure 6-9: Data format of TIMESTAMP type



YYYY: 0001 to 9999 (year)
MM: 01 to 12 (month)
DD: 01 to the last day of the respective month (day)
hh: 00 to 23 (hour)
mm: 00 to 59 (minute)
ss: 00 to 59 (seconds)
nnnn...nn: *p* digits of fractional seconds
n: 0 to 9
p = 0, 3, 6, 9, 12

One digit is represented in 4 bits. If the fractional seconds precision is an odd number, zeros are stored in the final 4 bits.

- Literals are written in the form TIMESTAMP'2012-03-30 11:03:58.123456' or TIMESTAMP'2012/03/30 11:03:58.123456'. For details about literals, see 6.3 Literals.

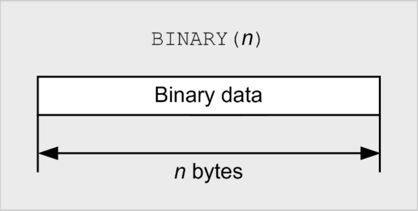
(4) Binary data

■ BINARY

- This is the data type for handling fixed-length binary data.
- The following shows the format to use when specifying this data type:

  BINARY (*n*) or BINARY
- The length of the binary data (number of bytes) is specified in *n*, where *n* is an integer in the range 1 to 32,000. If *n* is omitted, 1 is assumed.
- Literals are written in the form X'0A38ef92'. For details about literals, see 6.3 Literals.
- The data format of the BINARY type is shown in the following figure.

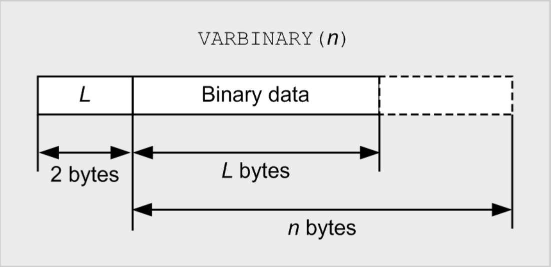
Figure 6-10: Data format of BINARY type



■ VARBINARY

- This is the data type for handling variable-length binary data.
- The following shows the format to use when specifying this data type:
  `VARBINARY(`*n*`)`
- The maximum length of the binary data (number of bytes) is specified in *n*, which must be an integer in the range 1 to 32,000, and cannot be omitted.
- Literals are written in the form `X'0A38ef92'`. For details about literals, see 6.3 Literals.
- The length of the binary data can be 0 bytes.
- The data format of the `VARBINARY` type is shown in the following figure.

Figure 6-11: Data format of VARBINARY type



The length of the binary data (*L*) is represented in two bytes.

6.2.2 Data types that can be converted, assigned, and compared

(1) Data types that can be compared

The following table shows the combinations of data types that can be compared.

Table 6-6: Combinations of data types that can be compared

Data type		**Data type of comparison target**					
		Numeric data	**Character string data**	**Datetime data**			**Binary data**
				DATE	**TIME**	**TIMESTAMP**	
Numeric data		Y	N	N	N	N	N
Character string data		N	Y	Y	Y	Y	N
Datetime data	`DATE`	N	Y	Y	N	Y	N
	`TIME`			N	Y	N	N

Data type		Data type of comparison target					
		Numeric data	Character string data	Datetime data			Binary data
				DATE	TIME	TIMESTAMP	
	`TIMESTAMP`			Y	N	Y	N
Binary data		N	N	N	N	N	Y

Legend:

Y: Can be compared.

N: Cannot be compared.

■ Comparing character string data

- If the lengths of the character string data being compared are different, spaces are added to the end of the shorter data string to make the lengths the same, and then the comparison is performed.
- Even when comparing `VARCHAR` types, the comparison is performed after the spaces are added.

■ Comparing numeric data

If the data types of the data being compared are different, the comparison is performed using the data type that has the larger range. The range sizes are ordered as follows:

`DOUBLE PRECISION` > `DECIMAL` > `INTEGER` > `SMALLINT`

■ Comparing datetime data to character string data

Datetime data can be compared to character string data only when the character string data is a literal written in the corresponding predefined input representation. For information about predefined input representations, see 6.3.3 Predefined character-string representations.

- Date data can be compared to character strings written in the predefined input representation for date data. The character string data in the predefined input representation for date data is converted to date data, and then the comparison is performed on the date data items.
- Date data can be compared to character strings written in the predefined input representation for time stamp data. A time of 0 hours, 0 minutes, and 0 seconds is set to the date data, and the date data is converted to time stamp data. The character string data in the predefined input representation for time stamp data is then converted to time stamp data. The comparison is then performed on the time stamp data items.
- Time data can be compared to character strings written in the predefined input representation for time data. The character string data in the predefined input representation for time data is converted to time data, and then the comparison is performed on the time data items.
- Time stamp data can be compared to character strings written in the predefined input representation for time stamp data. The character string data in the predefined input representation for time stamp data is converted to time stamp data, and then the comparison is performed on the time stamp data items.
- Time stamp data can be compared to character strings written in the predefined input representation for date data. A time of 0 hours, 0 minutes, and 0 seconds is set to the predefined input representation for date data, and the character string data is converted to time stamp data. The comparison is then performed on the time stamp data items.

However, if the datetime data is located in the selection expression of a subquery, it cannot be compared to the corresponding value expression.

■ **Comparing datetime data**

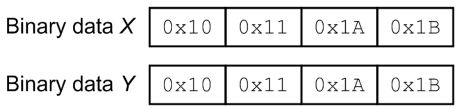
- When date data and time stamp data are compared, the date data is converted into time stamp data by setting the time to 0 hours, 0 minutes, and 0 seconds.

- When the number of digits in the fractional seconds are different, the lower-precision fractional seconds are padded with zeros until they align with the higher-precision data.

■ **Comparing binary data**

- When the data to be compared have the same length, they are considered equal when all the byte values match.

  Example:

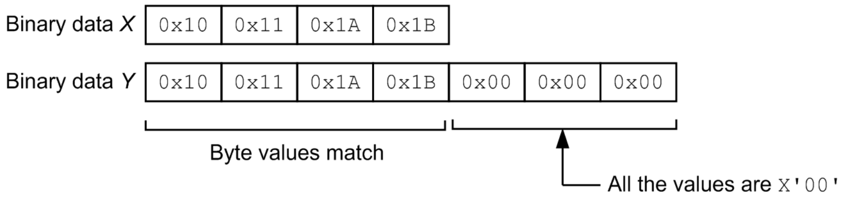


  In the above case, binary data *X* = binary data *Y*.

- When the data to be compared have different lengths, they are considered equal when the following two conditions are met:

  - All the byte values match when compared from the first byte through the end of the shorter data
  - The byte values in the longer portion are all `X'00'`

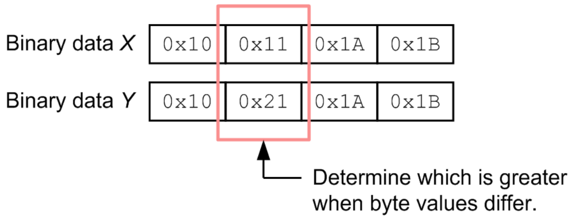
  Example:



  In the above case, binary data *X* = binary data *Y*.

- The data are compared in order starting from the first byte. When the byte values differ, the magnitudes of the first bytes that are different are compared, and this is used to determine which is greater.

  Example:

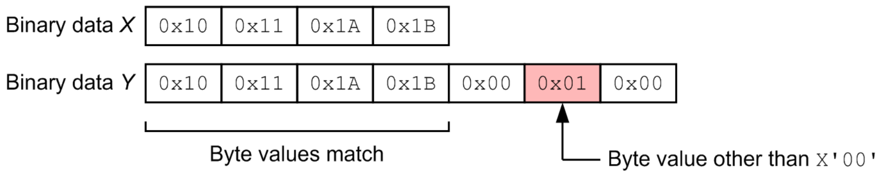


  In the above case, binary data *X* < binary data *Y*.

- When the data to be compared have different lengths, and the byte values match from the first byte through the end of the shorter data, which value is greater is determined as follows.

  Let *X* be the shorter data and *Y* be the longer data. If there are one or more byte values other than `X'00'` in the longer portion of *Y*, then *X* < *Y*.

  Example:



  In the above case, binary data *X* < binary data *Y*.

(2) Storage assignments between data types

The table below lists the combinations of data types that can be specified as an insertion value in an `INSERT` statement or an update value in an `UPDATE` statement. However, if you use a dynamic parameter to perform the storage assignment, align data types of the assignment source and assignment target. For details about dynamic parameters, see 6.6 Variables (dynamic parameters).

Table 6-7: Storage assignment relationships between combinations of data types

Data type of assignment source		Data type of assignment target						
		Numeric data	Character string data	Datetime data			Binary data	Row data
				DATE	TIME	TIMESTAMP		
Numeric data		Y	N	N	N	N	N	N
Character string data		N	Y	Y	Y	Y	N	Y#
Datetime data	`DATE`	N	N	Y	N	Y	N	N
	`TIME`			N	Y	N	N	
	`TIMESTAMP`			Y	N	Y	N	
Binary data		N	N	N	N	N	Y	N

Legend:

Y: Storage assignment can be performed.

N: Storage assignment cannot be performed.

#

A `CHAR` type can be assigned to row data (`ROW`).

■ Storage assignment of character string data

- If the data length of the assignment source is longer than the data length of the assignment target, the assignment cannot be performed.
- If the assignment target is `CHAR` type, and the data length of the assignment source is shorter than the data length of the assignment target, it is stored with spaces added at the end.

■ Storage assignment of numeric data

- If the assignment source exceeds the range of values that can be handled by the assignment target, the assignment cannot be performed.
- If the assignment target is `INTEGER` or `SMALLINT`, and the assignment source is `DECIMAL` or `DOUBLE PRECISION`, the fractional (decimal) part is truncated.
- If the assignment source and assignment target are both `DECIMAL`, any digits of the assignment source that are beyond the scaling of the assignment target are truncated. If the assignment source scaling is smaller than the assignment target scaling, it is stored with zeros added to the fractional part.
- If the assignment source is `DOUBLE PRECISION` and assignment target is `DECIMAL`, any digits of the assignment source that are beyond the scaling of the assignment target are rounded off (to the nearest even number). If the assignment source scaling is smaller than the assignment target scaling, it is stored with zeros added to the end of the fractional part.

■ Changing the storage assignment from character string data to datetime data

Changing the storage assignment from character string data to datetime data is possible only when the character string data is a literal written in the corresponding predefined input representation. For information about predefined input representations, see 6.3.3 Predefined character-string representations.

However, changing the storage assignment is not possible when the datetime data is a column that is to be inserted by an `INSERT` statement and the character string is specified as the selection expression of a query specification, even when the character string is a literal written in the corresponding predefined input representation.

- The storage assignment of character strings written in the predefined input representation for date data can be changed to date data. The character string data in the predefined input representation for date data is converted to date data, and then its storage assignment is changed to date data.
- The storage assignment of character strings written in the predefined input representation for time stamp data can be changed to date data. In this case, the character string data in the predefined input representation for time stamp data is converted to time stamp data, and then the storage assignment is changed for only the date portion of the time stamp data.
- The storage assignment of character strings written in the predefined input representation for time data can be changed to time data. The character string data in the predefined character-string representation for time data is converted to time data, and then its storage assignment is changed to time data.
- The storage assignment of character strings written in the predefined input representation for time stamp data can be changed to time stamp data. The character string data in the predefined input representation for time stamp data is converted to time stamp data, and then its storage assignment is changed to time stamp data.
- The storage assignment of character strings written in the predefined input representation for date data can be changed to time stamp data. In this case, a time of 0 hours, 0 minutes, and 0 seconds is set to the predefined character-string representation for date data, and the character string data is converted to time stamp data. Its storage assignment is then changed to time stamp data.

■ Storage assignment of datetime data

- When the storage assignment of date data is changed to time stamp data, a time of 0 hours, 0 minutes, and 0 seconds is set to the date data. Its storage assignment is then changed to time stamp data.
- When the source is time stamp data and the target is date data, storage assignment of only the date portion of the time stamp data is performed.
- If the number of digits in the fractional seconds of the assignment source is greater than the number of digits in the fractional seconds of the assignment target, the portion that cannot be assigned is truncated.
- If the number of digits in the fractional seconds of the assignment source is less than the number of digits in the fractional seconds of the assignment target, storage assignment is performed by padding the excess portion with 0.

■ Storage assignment of binary data

- If the data length of the source is greater than the data length of the target, the assignment cannot be performed.
- If the target type is `BINARY` and the data length of the source is less than the data length of the target, the data is padded with `X'00'` at the end before it is stored.

■ Storage assignment of row data

Match the data length of the assignment source to the assumed row data length of the assignment target (the row length of the table being updated or inserted into).

(3) Search assignments of data types

If you receive retrieval results, be sure to align the data types of the assignment source and assignment target.

If `ROW` (row data) is specified in the selection expression, the data type of the assignment target for receiving the search result can be a `CHAR` type variable.

(4) Storage assignment to a table function derived table (in the case of the ADB_CSVREAD function)

This subsection describes the rules for how field data in a CSV file is assigned to columns in a table function derived table. It also gives the rules for the description format of the field data.

Note that the table function derived table here means a table function derived table derived by means of the `ADB_CSVREAD` function.

> **Note**
>
> A *table function derived table* is a collection of data in table format derived by means of the `ADB_AUDITREAD` function or the `ADB_CSVREAD` function. For details about the `ADB_AUDITREAD` function, see 7.15.2 ADB_AUDITREAD function. For details about the `ADB_CSVREAD` function, see 7.15.3 ADB_CSVREAD function.

The description format of the field data must be compatible with the data type of the column in the table function derived table. The following table shows the relationship between the data type of the column in the table function derived table and the description format of the field data.

Table 6-8: Relationship between the data type of the column in the table function derived table and the description format of the field data

No.	Data type of the column in the table function derived table		Description format of the field data			
			Format	Examples	Notes	Examples of storage assignment of the null value
1	Numeric data	`INTEGER`	`[{+|-}]`*a...a* `+`, `-`: Sign *a...a*: Numeric value (*a* is `0` to `9`)	• `100` • `-123` • `000` • `0657`	• The sign and numeric value together cannot exceed 20 characters. • Regardless of the format and length restrictions, leading and trailing single-byte spaces and tabs are permitted around all characters.#1	• ...,`*`,... • ...,`"*"`,... • ...,,... • ...,`""`,... • ...,`""""`,... However, the examples using enclosing characters are invalid if the enclosing character specification option is set to `NONE`.
2		`SMALLINT`	`[{+|-}]`*a...a* `+`, `-`: Sign *a...a*: Numeric value (*a* is `0` to `9`)	• `100` • `-0123` • `0` • `+0657`	• The sign and numeric value together cannot exceed 11 characters. • Regardless of the format and length restrictions, leading and trailing single-byte spaces and tabs are	Same as No. 1.

No .	Data type of the column in the table function derived table		Description format of the field data			
			Format	Examples	Notes	Examples of storage assignment of the null value
					permitted around all characters.[#1]	
3		DECIMAL	[{+|-}] {*a*...*a*[.[*b*...*b*]]|.*b*...*b*} +, -: Sign *a*...*a*: Integer part (*a* is 0 to 9) *b*...*b*: Fractional part (*b* is 0 to 9)[#2]	• 100 • -123.00 • Δ.00 • 012. • -1.56 • +.560	• The integer and fractional parts together cannot exceed 38 characters (or 39 characters if the integer part (0) is omitted and the precision and scaling match the column where the value is to be stored). • Regardless of the format and length restrictions, leading and trailing single-byte spaces and tabs are permitted around all characters.[#1]	Same as No. 1.
4		DOUBLE PRECISION	[{+|-}] {*a*...*a*[.[*b*...*b*]]|.*b*...*b*}[{E|e}[[{+|-}]*c*...*c*]] +, -: Sign *a*...*a*: Integer part of the mantissa (*a* is 0 to 9) *b*...*b*: Fractional part of the mantissa (*b* is 0 to 9) *c*...*c*: Exponent (*c* is 0 to 9)[#3] E, e: Floating point numeric literal (literal identifying the exponent in a floating-point number)	• 100 • -Δ123 • 0.Δ • -1.5600 • .56 • -02.4e+9 • 000e • 2.4E+009	• The data cannot exceed 509 characters.[#4] • Regardless of the format and length restrictions, leading and trailing single-byte spaces and tabs are permitted around all characters.[#1]	Same as No. 1.
5	Character string data	CHARACTER	*a*...*a* *a*...*a*: Data consisting of one or more characters	• abcdef ΔΔ • ABCDEF • Δ	• The number of characters cannot exceed the defined length of the column where the value is to be stored. • The trailing single-byte spaces in the examples can be omitted.[#5]	• ...,,... • ...,"",... The character string data cannot include single-byte spaces and tabs. The examples using enclosing characters are invalid if the enclosing character specification option is set to NONE.

No.	Data type of the column in the table function derived table		Description format of the field data			
			Format	Examples	Notes	Examples of storage assignment of the null value
6		VARCHAR	*a*...*a* *a*...*a*: Data consisting of one or more characters	• abcdef ΔΔ • ABCDEF • ΔΔ	• The number of characters cannot exceed the defined length of the column where the value is to be stored.	• ...,,... Enclosing characters cannot be specified. The character string data cannot include single-byte spaces and tabs. **■ To specify data of length 0** • ...,"",... However, data of length 0 cannot be specified if the enclosing character specification option is set to NONE.
7	Datetime data	DATE	{*YYYY*-*MM*-*DD* |*YYYY*/*MM*/*DD*} *YYYY*: Year (0001 to 9999) *MM*: Month (01 to 12) *DD*: Day (01 to last day of month)	• 2013-06-10 • 2013/06/10	• Regardless of the format, leading and trailing single-byte spaces and tabs are permitted around all characters.[#1]	Same as No. 1.
8		TIME	*hh*:*mm*:*ss*[.[*nn*...*n*]] *hh*: Hour (00 to 23) *mm*: Minutes (00 to 59) *ss*: Seconds (00 to 59) *nn*...*n*: Fractional seconds (*n* is 0 to 9)	• 11:03:58 • 11:03:58. • 11:03:58 Δ.1234	• The fractional seconds (*nn*...*n*) cannot exceed 12 characters.[#6] • Regardless of the format and length restrictions, leading and trailing single-byte spaces and tabs are permitted around all characters.[#1]	Same as No. 1.
9		TIMESTAMP	{*YYYY*-*MM*-*DD* |*YYYY*/*MM*/*DD*} Δ*hh*:*mm*:*ss*[.[*nn*...*n*]] *YYYY*: Year (0001 to 9999) *MM*: Month (01 to 12) *DD*: Day (01 to last day of month) *hh*: Hour (00 to 23) *mm*: Minutes (00 to 59) *ss*: Seconds (00 to 59) *nn*...*n*: Fractional seconds (*n* is 0 to 9)	• 2013-06-10 Δ11:03:58 • 2013-06-10 Δ11:03:58 Δ.1234	• The fractional seconds (*nn*...*n*) cannot exceed 12 characters.[#6] • Regardless of the format and length restrictions, leading and trailing single-byte spaces and tabs are permitted around all characters.[#1]	Same as No. 1.

No .	Data type of the column in the table function derived table		Description format of the field data			
			Format	Examples	Notes	Examples of storage assignment of the null value
10	Binary data	BINARY	Hexadecimal string *a...a* *a*: 0 to 9, A to F, or a to f	• 12340000 • 90Δ AB • 90ab Δ CDEF	• The number of characters must be a multiple of 2, up to 2 times the defined length of the column where the value is to be stored.#7 • Trailing 00s are assumed and can be omitted.#8 • Regardless of the format and length restrictions, leading and trailing single-byte spaces and tabs are permitted around all characters.#1	Same as No. 1.
11			Binary string *a...a* *a*: 0 or 1	• 01010101 • 0101 Δ 0101	• The number of characters must be a multiple of 8, up to 8 times the defined length of the column where the value is to be stored.#7 • Trailing 00000000s are assumed and can be omitted.#8 • Regardless of the format and length restrictions, leading and trailing single-byte spaces and tabs are permitted around all characters.#1	Same as No. 1.
12		VARBINARY	Hexadecimal string *a...a* *a*: 0 to 9, A to F, or a to f	• 12340000 • 90Δ AB • 90ab Δ CDEF	• The number of characters must be a multiple of 2, up to 2 times the defined length of the column where the value is to be stored.#7 • Regardless of the format and length restrictions, leading and trailing single-byte spaces and tabs are permitted around all characters.#1	• ...,*,... • ...,"*",... • ...,,... However, the examples using enclosing characters are invalid if the enclosing character specification option is set to NONE. **■ To specify data of length 0**

No.	Data type of the column in the table function derived table		Description format of the field data			
			Format	Examples	Notes	Examples of storage assignment of the null value
						- …,"",… - …,"""",… However, data of length 0 cannot be specified if the enclosing character specification option is set to `NONE`.
13			Binary string *a...a* *a*: `0` or `1`	- `01010101` - `0101` Δ `0101`	- The number of characters must be a multiple of 8, up to 8 times the defined length of the column where the value is to be stored.[#7] - Regardless of the format and length restrictions, leading and trailing single-byte spaces and tabs are permitted around all characters.[#1]	Same as No. 12.

Legend:

Δ: One or more single-byte spaces or tabs

`,`: Delimiting character

`"`: Enclosing character

#1

Any leading or trailing single-byte space (`0x20`) or tab (`0x09`) characters are removed.

Example: Δ`1`Δ`23` ΔΔ `4` ΔΔΔ → `1234`

If the removal of white space leaves no data, the result is treated as a null value.

#2

Fractional digits beyond the scaling defined for the column where the value is to be stored are truncated.

#3

If the exponent is omitted, an exponent of +0 is assumed.

#4

Depending on the specified value, loss of precision might occur.

#5

If the input data is less than the defined length, the remaining portion is filled with single-byte spaces.

#6

If the number of digits in the fractional seconds (*nn*...*n*) is less than the fractional seconds precision of the data type in the table, the stored value is filled with zeros on the right.

If the number of digits in the fractional seconds (*nn*...*n*) exceeds the fractional seconds precision of the data type in the table, the input data is truncated.

#7

An error results if the number of characters in the hexadecimal string is not a multiple of 2.

An error results if the number of characters in the binary string is not a multiple of 8.

#8

When the input data is less than the defined length, the remaining portion is filled with `0x00`.

6.3 Literals

A *literal* is data whose value cannot be modified within the program.

6.3.1 Types of literals

A literal can be a numeric literal or a general literal. The following table lists types of literals.

Table 6-9: Types of literals

No.	Type of literal	Description	Type of literal		Description
1	Numeric literals	Literals that represent numeric values. The literals listed at the right are numeric literals.	Integer literal		A literal that represents an integer.
2			Decimal literal		A literal that represents a number with a decimal point.
3			Floating-point numeric literal		A literal that represents a number with a decimal point.
4	General literals	Literals that represent characters, dates, times, and binary data. The literals listed at the right are general literals.	Character string literal		A literal that represents characters.
5			Date literal		A literal that represents a date.
6			Time literal		A literal that represents time.
7			Time stamp literal		A literal that represents a date and time.
8			Binary literal	Hexadecimal-format binary literal	A binary literal represented in hexadecimal format.
9				Binary-format binary literal	A binary literal represented in binary format.

6.3.2 Description format of literals

The following table lists the description formats and assumed data types of literals.

Table 6-10: Description formats and assumed data types of literals

No.	Type of literal		Description format	Assumed data type
1	Numeric literal	Integer literal	- Description format [*sign*]*unsigned-integer* - Examples `45`, `6789`, `-123` - Explanation The *sign* portion is expressed as `+` or `-`. `+` can be omitted.	`INTEGER`
2		Decimal literal	- Description format [*sign*]*integer-part.fractional-part* - Examples	`DECIMAL(`*m*`[,`*n*`])` *m*,*n*: Number of specified digits

No.	Type of literal		Description format	Assumed data type
			12.3, -456., .789 • Explanation The *sign* portion is expressed as + or -. + can be omitted. The integer part and fractional part are represented as unsigned integers. Either the integer part or fractional part must be specified. A decimal point must be specified.	
3		Floating-point numeric literal	• Description format *mantissa*E*exponent*, or *mantissa*e*exponent* • Examples +1.0E+1, 1.0E2, -3.4e-01, .5E+67 • Explanation The mantissa is expressed in the form of an integer or decimal literal. The exponent is written in the form of an integer literal of 1 to 3 digits. The exponent represents powers of 10. The letter E or e is required.	DOUBLE PRECISION
4	General literal	Character string literal	• Description format '*character-string*' • Examples 'HITACHI', '88', '''95.7.30' • Explanation A character string is enclosed in single quotation marks ('). Half-width and/or full-width characters can be used. To use a single quotation mark within a character string, as in the example '95.7.30, specify two consecutive single quotation marks, as in the example above.	• When *n*>0 CHAR(*n*) • When *n*=0 VARCHAR(1) with an actual length of 0 (where *n* indicates the length of the character string)
5		Date literal	• Description format DATE'*YYYY-MM-DD*' or DATE'*YYYY/MM/DD*' • Examples DATE'2012-03-30' DATE'2012/03/30' • Explanation The year is expressed in four digits (*YYYY*), and the month (*MM*) and day (*DD*) in two digits. Pad the fields with zeros on the left as necessary. Specify values for *YYYY*, *MM*, and *DD* that are valid for the DATE type (for example, *MM* must be 01 to 12). No separators are permitted inside '*YYYY-MM-DD*' and '*YYYY/MM/DD*'.	DATE
6		Time literal	• Description format TIME'*hh:mm:ss.nn...n*' • Examples TIME'11:03:58' TIME'11:03:58.123' TIME'11:03:58.123456' TIME'11:03:58.123456789'	TIME[(*p*)] *p*: fractional seconds precision

No.	Type of literal		Description format	Assumed data type

			`TIME'11:03:58.123456789012'`
			`TIME'11:03:58.'`
			• Explanation
			The hour (*hh*), minutes (*mm*), and seconds (*ss*) are expressed in two digits. Pad the fields with zeros on the left as necessary.
			nn...*n* expresses the fractional seconds. *nn*...*n* represents 3, 6, 9, or 12 digits.
			To use fractional seconds, put a period between the seconds and the fractional seconds precision specification.
			If you omit the fractional seconds and specify only a period, the data is treated as having a fractional seconds precision of 0.
			An error results if the fractional seconds precision is more than 12.
			Specify values for *hh*, *mm*, *ss*, and *nn*...*n* that are valid for the `TIME` type (for example, *hh* must be `00` to `23`).
			No separators are permitted inside `'`*hh*`:`*mm*`:`*ss*`.`*nn*...*n*`'`.	

7		Time stamp literal	• Description format
			`TIMESTAMP'`*YYYY*-*MM*-*DD* *hh*`:`*mm*`:`*ss*`.`*nn*...*n*`'` or `TIMESTAMP'`*YYYY/MM/DD* *hh*`:`*mm*`:`*ss*`.`*nn*...*n*`'`
			• Examples
			`TIMESTAMP'2012-03-30 11:03:58'`
			`TIMESTAMP'2012/03/30 11:03:58'`
			`TIMESTAMP'2014-07-30 11:03:58.123'`
			`TIMESTAMP'2014/07/30 11:03:58.123456789012'`
			`TIMESTAMP'2014-07-30 11:03:58.'`
			• Explanation
			The format is *YYYY*-*MM*-*DD* (or *YYYY*/*MM*/*DD*) and *hh*`:`*mm*`:`*ss*, with a space between them.
			The year is expressed in four digits (*YYYY*), and the month (*MM*) and day (*DD*) in two digits. Pad the fields with zeros on the left as necessary.
			Similarly, pad the two-digit fields for hours (*hh*), minutes (*mm*), and seconds (*ss*) with zeros on the left as necessary.
			nn...*n* expresses the fractional seconds precision. *nn*...*n* represents 3, 6, 9, or 12 digits.
			To use fractional seconds, put a period between the seconds and the fractional seconds precision specification.
			If you omit the fractional seconds and specify only a period, the data is treated as having a fractional seconds precision of 0.
			An error results if the fractional seconds precision is more than 12.
			Specify values for *YYYY*, *MM*, *DD*, *hh*, *mm*, and *ss* that are valid for the `TIMESTAMP` type (for example, *hh* must be `00` to `23`).
				`TIMESTAMP[(`*p*`)]`
				p: fractional seconds precision

No.	Type of literal		Description format	Assumed data type
			No separators are permitted inside '*YYYY*-*MM*-*DD* *hh*:*mm*:*ss*.*nn*...*n*' and '*YYYY*/*MM*/*DD* *hh*:*mm*:*ss*.*nn*...*n*'.	

8		Hexadecimal-format binary literal	- Description format
			X'*hexadecimal-character-string*' or
			x'*hexadecimal-character-string*'
			- Examples
			`X'82A0'`
			`X'82a0'`
			`x'82A0'`
			- Explanation
			hexadecimal-character-string is expressed using the hexadecimal digits `0` to `9` and `A` to `F` (or `a` to `f`).
			The number of characters in *hexadecimal-character-string* must be a multiple of 2. Two hexadecimal characters make one byte.
			The number of characters in *hexadecimal-character-string* cannot exceed 64,000.
			No separators are permitted inside *hexadecimal-character-string*.
				- When $n > 0$
				BINARY($n \div 2$)
				- When $n = 0$
				`VARBINARY(1)`
				with an actual length of 0
				(where *n* indicates the length of the hexadecimal character string)

9		Binary-format binary literal	- Description format
			B'*binary-character-string*' or b'*binary-character-string*'
			- Examples
			`B'01010101'`
			`b'01010101'`
			- Explanation
			binary-character-string is expressed using the binary digits 0 and 1.
			The number of characters in *binary-character-string* must be a multiple of 8. Eight binary characters make one byte.
			The number of characters in *binary-character-string* cannot exceed 256,000.
			No separators are permitted inside *binary-character-string*.
				- When $n > 0$
				BINARY($n \div 8$)
				- When $n = 0$
				`VARBINARY(1)`
				with an actual length of 0
				(where *n* indicates the length of the binary character string)

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

Table 6-11: Restrictions on the use of numeric literals

No.	Numeric literal	Range	Maximum number of digits (including leading zeros)
1	Integer literal#1	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	19 digits
2	Decimal literal	$-(10^{38} - 1)$ to -10^{-38}, 0, and 10^{-38} to $(10^{38} - 1)$	38 digits
3	Floating-point numeric literal#2	Approximately -1.7×10^{308} to -2.3×10^{-308}, 0, and approximately 2.3×10^{-308} to 1.7×10^{308}	Mantissa part: 17 digits Exponent part: 3 digits

#1

If a literal that exceeds the range of values for an integer literal is written in the notation used to represent integer literals, it will be interpreted as a decimal literal, with an assumed decimal point to the right of the literal.

#2

The exact range of values depends on the hardware representation.

6.3.3 Predefined character-string representations

A character string literal in the format of the corresponding predefined character-string representation can be used to represent a date literal, time literal, or time stamp literal. This section describes the predefined character-string representations of dates, times, and time stamps.

(1) Predefined character-string representation of dates

The predefined character-string representations of dates include a predefined input representation and a predefined output representation.

(a) Predefined input representation

A character string literal that follows the format of the predefined input representation for dates can be used as a literal representing a date. The format of the predefined input representation of a date is as follows.

Format of the predefined input representation:

```
'YYYY-MM-DD' or 'YYYY/MM/DD'
```

- The year is expressed in four digits (*YYYY*), and the month (*MM*) and day (*DD*) in two digits. Pad the fields with zeros on the left, as necessary.
- Specify values for *YYYY*, *MM*, and *DD* that are valid for the `DATE` type (for example, *MM* must be `01` to `12`).

Example:

July 30, 2013 is expressed as follows.

- Character string literal (predefined input representation): `'2013-07-30'` or `'2013/07/30'`
- Date literal: `DATE'2013-07-30'` or `DATE'2013/07/30'`

(b) Predefined output representation

When date data is retrieved using the `adbsql` command (or similar commands), the results are output in a format that follows the predefined output representation.

Format of the predefined output representation:

```
'YYYY-MM-DD'
```

The year is expressed in four digits (*YYYY*), and the month (*MM*) and day (*DD*) in two digits. The fields are padded on the left with zeros, as necessary.

Example:

For the date data `X'20130730'`, the predefined output representation is as follows.

```
'2013-07-30'
```

(2) Predefined character-string representation of times

Predefined character-string representations of times include a predefined input representation and a predefined output representation.

(a) Predefined input representation

A character string literal that follows the format of the predefined input representation for times can be used as a literal representing a time. The format of the predefined input representation of a time is as follows.

Format of the predefined input representation:

```
'hh:mm:ss.nn...n'
```

- The hour (*hh*), minutes (*mm*), and seconds (*ss*) are expressed in two digits. Pad the fields with zeros on the left, as necessary.
- To use fractional seconds, add them in the `.`*nn...n* format. *nn...n* is represented by 3, 6, 9, or 12 digits. If the *nn...n* portion is not specified with 3, 6, 9, or 12 digits, the fractional seconds precision is assumed as described later. In that case, zero padding is applied to the missing digits on the right.

Number of digits in nn...n	Assumed fractional seconds precision
1, 2	3
4, 5	6
7, 8	9
10, 11	12

- An error results if *nn...n* contains 13 or more digits.
- A period is required between the seconds and the fractional seconds.
- There is no need to specify `.`*nn...n* unless you want to use fractional seconds.
- If you omit *nn...n* and specify only a period, the data is treated as having a fractional seconds precision of 0.
- Specify values for *hh*, *mm*, *ss*, and *nn...n* that are valid for the `TIME` type (for example, *hh* must be `00` to `23`).

Example:

The following representations express the time that is 3 minutes and 58.123456 seconds after the hour of 11 o'clock.

- Character string literal (predefined input representation): `'11:03:58.123456'`
- Time literal: `TIME'11:03:58.123456'`

(b) Predefined output representation

When time data is retrieved using the `adbsql` command (or a similar command), the results are output in a format that follows the predefined output representation.

Format of the predefined output representation:

```
'hh:mm:ss.nn...n'
```

- The hour (*hh*), minutes (*mm*), and seconds (*ss*) are expressed in two digits. The fields are padded on the left with zeros, as necessary.
- The fractional seconds are displayed in `.`*nn...n*. The number of digits in the fractional seconds depends on the specification of the fractional seconds precision in the time data.
- If the fractional seconds precision is 0, the `.`*nn...n* part is not displayed.

Example:

If the time data is `X'110358123'`, the predefined output representation is as follows.

```
'11:03:58.123'
```

(3) Predefined character-string representation of time stamps

The predefined character-string representation of time stamps include a predefined input representation and a predefined output representation.

(a) Predefined input representation

A character string literal that follows the format of the predefined input representation for time stamps can be used as a literal representing a time stamp. The format of the predefined input representation of a time stamp is as follows.

Format of the predefined input representation:

```
'YYYY-MM-DD hh:mm:ss.nn...n' or 'YYYY/MM/DD hh:mm:ss.nn...n'
```

- The year is expressed using four digits (*YYYY*), and the month (*MM*), day (*DD*), hour (*hh*), minutes (*mm*), and seconds (*ss*) using two digits. Pad the fields with zeros on the left, as necessary.
- A space is required between *YYYY*-*MM*-*DD* and *hh*:*mm*:*ss*.
- Specify `.`*nn*...*n* if you want to use fractional seconds. *nn*...*n* represents 3, 6, 9, or 12 digits. The table below shows the fractional seconds precision that is assumed in cases where the number of digits in *nn*...*n* is not 3, 6, 9, or 12. In that case, zero padding is applied to the missing digits on the right.

Number of digits in nn...n	Assumed fractional seconds precision
1, 2	3
4, 5	6
7, 8	9
10, 11	12

- An error results if *nn*...*n* contains 13 or more digits.
- A period is required between the seconds and the fractional seconds.
- There is no need to specify `.`*nn*...*n* unless you want to use fractional seconds.
- If you omit *nn*...*n* and specify only a period, the data is treated as having a fractional seconds precision of 0.
- Specify values for *YYYY*, *MM*, *DD*, *hh*, *mm*, *ss*, and *nn*...*n* that are valid for the `TIMESTAMP` type (for example, *hh* must be `00` to `23`).

Example

July 30, 2013 at 11:03:58.123456 is expressed as follows.

- Character string literal (predefined input representation): `'2013-07-30 11:03:58.123456'` or `'2013/07/30 11:03:58.123456'`
- Time stamp literal: `TIMESTAMP'2013-07-30 11:03:58.123456'` or `TIMESTAMP'2013/07/30 11:03:58.123456'`

(b) Predefined output representation

When time stamp data is retrieved using the `adbsql` command (or similar commands), the results are output in a format that follows the predefined output representation.

Format of the predefined output representation:

```
'YYYY-MM-DD hh:mm:ss.nn...n'
```

- The year is expressed using four digits (*YYYY*), and the month (*MM*), day (*DD*), hour (*hh*), minutes (*mm*), and seconds (*ss*) using two digits. The fields are padded with zeros on the left, as necessary.
- The fractional seconds are displayed in `.`*nn*...*n*. The number of digits in the fractional seconds depends on the specification of the fractional seconds precision in the time stamp data.
- If the fractional seconds precision is 0, the `.`*nn*...*n* part is not displayed.

Example:

For the time stamp data `X'20130730110358123'`, the predefined output representation is as follows.

```
'2013-07-30 11:03:58.123'
```

6.4 Datetime information acquisition functions

There are functions for acquiring datetime information:

- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP

This section describes each function in turn.

6.4.1 CURRENT_DATE

Returns the current date.

(1) Specification format

```
datetime-information-acquisition-function-CURRENT_DATE ::= CURRENT_DATE
```

(2) Rules

1. The data type of the execution result is the DATE type.
2. Specifying CURRENT_DATE multiple times in an SQL statement produces the same date value.
3. The value of CURRENT_DATE is acquired when the SQL statement is executed on the HADB server. For details about interfaces and execution methods for executing SQL statements, see the descriptions of the JDBC API, ODBC functions, and CLI functions in the *HADB Application Development Guide*.
4. CURRENT_DATE can be specified in places where a value specification can be specified.

(3) Examples

Example 1

Retrieve the customer ID (USERID) and product code (PUR-CODE) from the sales history table (SALESLIST) for customers who made purchases today.

```
SELECT "USERID","PUR-CODE"
    FROM "SALESLIST"
        WHERE "PUR-DATE"=CURRENT_DATE
```

Example 2

Insert the following data (row) into the sales history table (SALESLIST).

- Customer ID (USERID): U00358
- Product code (PUR-CODE): P003
- Quantity purchased (PUR-NUM): 5
- Date of purchase (PUR-DATE): Today's date

```
INSERT INTO "SALESLIST"
      ("USERID","PUR-CODE","PUR-NUM","PUR-DATE")
    VALUES('U00358','P003',5,CURRENT_DATE)
```

6.4.2 CURRENT_TIME

Returns the current time.

(1) Specification format

```
datetime-information-acquisition-function-CURRENT_TIME ::= CURRENT_TIME[(p)]
```

(2) Rules

1. The fractional seconds precision (the number of digits to the right of the decimal point) is specified in *p*. You can specify the value `0`, `3`, `6`, `9`, or `12` for *p*. For example, when *p* is `3`, there will be 3 digits in the fractional seconds of the execution result of `CURRENT_TIME`.
2. If `(`*p*`)` is omitted, *p* = 0 is assumed.
3. The data type of the execution result is the `TIME` type.
4. Specifying `CURRENT_TIME` multiple times in an SQL statement produces the same time value.
5. The value of `CURRENT_TIME` is acquired when the SQL statement is executed on the HADB server. For details about interfaces and execution methods for executing SQL statements, see the descriptions of the JDBC API, ODBC functions, and CLI functions in the *HADB Application Development Guide*.
6. The precision of the fractional seconds acquired by `CURRENT_TIME` depends on the capabilities of the hardware. For example, if you specify `CURRENT_TIME(12)`, depending on the hardware you are using you might not be able to acquire 12 digits of fractional seconds precision.

   Example:

   `10:35:55.123456000000`

   As shown above, if only 6 digits of fractional seconds precision can be acquired, the 7th and subsequent digits will be 0.
7. `CURRENT_TIME` can be specified in places where a value specification can be specified.

(3) Examples

Example

Add product sales information to the daily sales history table (`SALESLIST_DAY`). The columns in the sales history table are shown below. The current time is stored as the time goods are sold (`SALE_TIME`).

- Store code (`SCODE`)
- Goods code (`GCODE`)
- Sex of customer (`SEX`)
- Time of sale of goods (`SALE_TIME`)

```
INSERT INTO "SALESLIST_DAY"
      ("SCODE","GCODE","SEX","SALE_TIME")
    VALUES('S001','G03542','M',CURRENT_TIME)
```

6.4.3 CURRENT_TIMESTAMP

Returns the current time stamp (date and time).

(1) Specification format

```
datetime-information-acquisition-function-CURRENT_TIMESTAMP ::= CURRENT_TIMESTAMP[(p)
]
```

(2) Rules

1. The fractional seconds precision (the number of digits to the right of the decimal point) is specified in *p*. You can specify the value 0, 3, 6, 9, or 12 for *p*. For example, when *p* is 3, there will be 3 digits in the fractional seconds of the execution result of CURRENT_TIMESTAMP.
2. When (*p*) is omitted, *p* = 0 is assumed.
3. The data type of the execution result is the TIMESTAMP type.
4. Specifying CURRENT_TIMESTAMP multiple times in an SQL statement produces the same date and time values.
5. The value of CURRENT_TIMESTAMP is acquired when the SQL statement is executed on the HADB server. For details about interfaces and execution methods for executing SQL statements, see the descriptions of the JDBC API, ODBC functions, and CLI functions in the *HADB Application Development Guide*.
6. The precision of the fractional seconds acquired by CURRENT_TIMESTAMP depends on the capabilities of the hardware. For example, if you specify CURRENT_TIMESTAMP(12), depending on the hardware you are using you might not be able to acquire 12 digits of fractional seconds precision.

   Example:

   2014-09-25 10:35:55.123456000000

   As shown above, if only 6 digits of fractional seconds precision can be acquired, the 7th and subsequent digits will be 0.
7. CURRENT_TIMESTAMP can be specified in places where a value specification can be specified.

(3) Example

Example

Add new customer information to the customer table (USERSLIST). The column structure of the customer table is as follows.

- Customer ID (USERID)
- Name (NAME)
- Sex (SEX)
- Datetime (LAST_UPDATE_TIME) when the customer information was last updated

```
INSERT INTO "USERSLIST"
      ("USERID","NAME","SEX","LAST_UPDATE_TIME")
    VALUES
      ('U00887','Edward Connelly','M',CURRENT_TIMESTAMP)
```

6.5 User information acquisition function

This section describes the following user information acquisition function:

- CURRENT_USER

6.5.1 CURRENT_USER

Returns the authorization identifier of the currently executing HADB user.

(1) Specification format

```
user-information-acquisition-function-CURRENT_USER ::= CURRENT_USER
```

(2) Rules

1. The data type of the execution result is the VARCHAR type.
2. Specifying CURRENT_USER multiple times in an SQL statement produces the same value.
3. The value of CURRENT_USER is acquired when the SQL statement is executed on the HADB server. For details about interfaces and execution methods for executing SQL statements, see the descriptions of the JDBC API, ODBC functions, and CLI functions in the *HADB Application Development Guide*.
4. CURRENT_USER can be specified in places where a value specification can be specified.

(3) Example

Example

Retrieve a list of information (the contents of SQL_TABLES) about the tables owned by the current user (the HADB user whose authorization identifier is currently connected to the HADB server).

```
SELECT * FROM "MASTER"."SQL_TABLES"
    WHERE TABLE_SCHEMA=CURRENT_USER
```

6.6 Variables (dynamic parameters)

A dynamic parameter is a variable that passes a value to SQL. When passing a value to SQL, a `?` is specified in the place where the value is to be placed. This `?` is the dynamic parameter.

6.6.1 Rules for specifying dynamic parameters

1. The data type and data length that are assumed by a specified dynamic parameter differ depending on the location in which the parameter is specified. The following table describes the data type and data length that are assumed by dynamic parameters.

Table 6-12: Assumed data type and data length of dynamic parameters

No.	Where the dynamic parameter is specified	Assumed data type and data length
1	Specified alone in a predicate (other than `LIKE`, `NULL`, and `LIKE_REGEX` predicates)	Data type and data length of the result of the value expression it is compared against
2	• Specified alone as the insertion value specified in a `VALUES` clause in an `INSERT` statement • Specified alone as the update value specified in a `SET` clause in an `UPDATE` statement	Data type and data length of the column being assigned
3	Specified elsewhere	Refer to the description of each item.

2. A maximum of 1,000 dynamic parameters can be specified in an SQL statement.
3. When you pass a value to a dynamic parameter, pass a value of the assumed data type and data length.
4. Make sure that the total data length of dynamic parameters specified in an SQL statement does not exceed 32,000,000 bytes.

6.6.2 Where dynamic parameters can be specified

The following table lists the places where a dynamic parameter can be specified.

Table 6-13: Where dynamic parameters can be specified

No.	SQL statement	Where dynamic parameters can be specified
1	`SELECT`	Selection expression#1
2		Places where literals can be specified in the search conditions#2
3		`ORDER BY` clause#1
4		`LIMIT` clause
5	`INSERT`	Insertion value or row insertion value
6	`UPDATE`	Update value or row update value
7		Places where literals can be specified in the search conditions#2
8	`DELETE`	Places where literals can be specified in the search conditions#2
9	`PURGE CHUNK`	

#1

The dynamic parameter cannot be specified by itself.

#2

It cannot be specified in the following places:

- On both sides of a comparison predicate
- On the left side of a `BETWEEN` predicate
- In a view definition (`CREATE VIEW`)
- In a `WITH` clause

6.6.3 Notes

Note that operations involving data supplied to dynamic parameters specified in scalar operations are performed on every relevant row. In the portions of the scalar operations that do not include columns where dynamic parameters are specified (when the values are fixed), consider specifying literals in the SQL statement.

6.7 Null value

The null value is a special value indicating that either no value exists or no value has been set. The null value is set in any area that does not contain values or in which values have not been set. The following explains how the null value is handled.

Receiving a column value as a result of a retrieval

- If you are using the JDBC driver
  Determine whether the column value that was obtained is the null value using the `wasNull` method in the `ResultSet` interface.
- If you are using the ODBC driver
  When the value of a column of retrieval results is the null value, the `StrLen_or_IndPtr` argument of `SQLBindCol` or `SQLGetData` is set to `SQL_NULL_DATA`.
- If you are using CLI functions
  Use indicators to identify null values. For details, see *a_rdb_SQLInd_t (indicator)* in the *HADB Application Development Guide*.

Comparison

The predicate is undefined for rows in which the result of a value expression other than the following is a null value, or for rows in which the column value is a null value:

- A value expression on the left side of the `NULL` predicate
- A value expression specified in `ESCAPE` escape-character in the `LIKE` predicate

For details about how the scalar function `DECODE` handles comparisons to the null value, see 8.15.1 DECODE.

Sorting

The null value is sorted according to the specification of the null-value sort order in the sort specification list. For details about the specification of the null-value sort order, see 7.24.1 Specification format for the sort specification list.

Grouping

In the grouping condition columns, if a row contains null values, any SQL statement that performs grouping will treat the null values as being the same value.

Exclusion of duplicates

Multiple null values are treated as duplicates.

Set functions

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

Window functions

In window functions, when there are rows where the results of a value expression specified for the window specification are null values, the null values are treated as being the same value.

Indexing

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

6.8 Scope variables

This section describes the types of identifiers that can act as scope variables and the effective scope of scope variables.

6.8.1 About scope variables

An identifier that can serve as a qualifier for a column specification is called a *scope variable*. A scope variable has a name and an effective scope.

If a correlation name is specified, the table name or query name that it specifies loses its effective scope.

Among the scope variables that have an effective scope at the position of a table specification, the scope variable of the innermost (closest) query specification (table to be updated or deleted) that has the same name as the table specification will act as that table specification's scope variable.

The following table shows what types of identifiers can act as scope variables.

Table 6-14: Types of identifiers that can act as scope variables

No.	Scope variable		Scope?
1	Correlation name		Y
2	Table name or query name	Correlation name is not specified.	Y
3		Correlation name is specified.	N
4	Table whose contents are to be updated by an `UPDATE` statement or deleted by a `DELETE` statement	Correlation name is not specified.	Y
5		Correlation name is specified.	N

Legend:

Y: Can be a scope variable.

N: Cannot be a scope variable.

6.8.2 Scope variable names

The following table lists examples of scope variable names.

Table 6-15: Examples of names of scope variables

No.	SQL example	Name of scope variable
1	`WITH Q1 AS`	--
2	`(SELECT * FROM`	
3	`T0),`	`A.T0`
4	`Q2 AS`	--
5	`(SELECT * FROM`	
6	`Q1)`	`Q1`
7	`SELECT "T1"."C1","X"."C1","Y"."C1" FROM`	--

No.	SQL example	Name of scope variable
8	`"T1"`	`A.T1`
9	`,"A"."T1" "X"`	`X`
10	`,"A"."T2" "Y"`	`Y`

Legend:

--: Not applicable.

`A`: Schema name

`TO`, `T1`, `T2`: Table identifier

`X`, `Y`: Correlation name

`C1`: Column name

`Q1`, `Q2`: Query name

6.8.3 Effective scope of scope variables

The following table shows examples of the effective scope of scope variables.

Table 6-16: Examples of the effective scope of scope variables

No.	SQL example	Scope variable							
		A.T1	X	A.T2	A.T3	Y	A.T4	A.T5	Z
1	`SELECT "X"."C1","T2"."C2"`	N	Y	Y	N	Y	N	N	N
2	`FROM "A"."T1" "X",`	N	Y	Y	N	Y	N	N	N
3	`"A"."T2",`	N	Y	Y	N	Y	N	N	N
4	`(SELECT * FROM "T3"`	N	N	N	Y	N	N	N	N
5	`WHERE "T3"."C1"=100) "Y"`	N	N	N	Y	N	N	N	N
6	`WHERE "X"."C1"=100 AND`	N	Y	Y	N	Y	N	N	N
7	`"X"."C1"=ANY(`	N	Y	Y	N	Y	N	N	N
8	`SELECT "T4"."C1" FROM "T4",`	N	N	N	N	N	Y	N	Y
9	`(SELECT *`	N	N	N	N	N	N	Y	N
10	`FROM "T5"`	N	N	N	N	N	N	Y	N
11	`WHERE "T5"."C1"="X"."C1")"Z"`	N	Y	Y	N	Y	N	Y	N
12	`WHERE "T4"."C2"="A"."T2"."C2")`	N	Y	Y	N	Y	Y	N	Y

Legend:

Y: Has scope.

N: Does not have scope.

`A`: Schema name

`T1`~`T5`: Table identifier

`X,Y,Z`: Correlation name

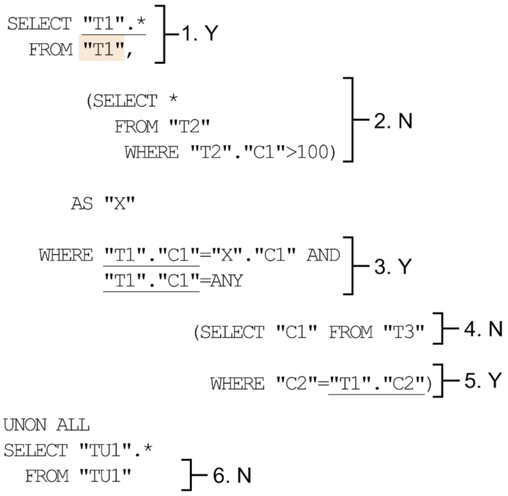
`C1`, `C2`: Column name

This section describes the effective scope of scope variables in `SELECT`, `UPDATE`, and `DELETE` statements.

(1) The effective scope of scope variables specified in the FROM clause of a SELECT statement

The effective scope of the scope variable encompasses the query specification that actually contains the scope variable identifier in its `FROM` clause, as well as any search conditions in its subqueries. However, the effective scope does not extend to derived tables specified in the `FROM` clause that actually contains the scope variable identifier. The following figure shows an example.

Figure 6-12: Example of the effective scope of scope variable T1 specified in a FROM clause (1 of 2)



```
SELECT "T1".*
  FROM "T1",

       (SELECT *
          FROM "T2"
           WHERE "T2"."C1">100)

      AS "X"

    WHERE "T1"."C1"="X"."C1" AND
          "T1"."C1"=ANY

                   (SELECT "C1" FROM "T3"

                     WHERE "C2"="T1"."C2")
UNION ALL
SELECT "TU1".*
  FROM "TU1"
```

Legend:

Y: Can reference scope variable `T1`.

N: Cannot reference scope variable `T1`.

Explanation

1. The scope variable `T1` can be referenced in the query specification that actually contains the scope variable in its `FROM` clause.
2. The scope variable `T1` cannot be referenced in the derived table specified in the same `FROM` clause as the `FROM` clause that actually contains the scope variable identifier.
3. The scope variable `T1` can be referenced in the query specification that actually contains the scope variable in its `FROM` clause.
4. The scope variable `T1` cannot be referenced outside of the search conditions of the subquery.
5. The scope variable `T1` can be referenced in the search conditions of the subquery contained in the query that immediately contains the scope variable in its `FROM` clause.
6. The scope variable `T1` cannot be referenced outside of the query.

Figure 6-13: Example of the effective scope of scope variable T1 specified in a FROM clause (2 of 2)

```
 SELECT "T1".*     ]-1. Y
   FROM ("T1"

         LEFT JOIN
         "T2"
          ON "T1"."C1"="T2"."C1"    ]-2. Y
        )
        LEFT JOIN
       ("T3"
        LEFT JOIN
        "T4"
         ON "T3"."C1"="T4"."C1")    ]-3. N
        )
        ON "T1"."C2"="T3"."C2"          ]
         AND EXISTS                     ]
           (SELECT * FROM "T5"          ]-4. Y
              WHERE "T5"."C3"="T1"."C3") ]
```

Legend:

Y: Can reference scope variable `T1`.

N: Cannot reference scope variable `T1`.

Explanation

These are examples of specifying joined tables.

1. The scope variable `T1` can be referenced in the query that immediately contains the scope variable in its `FROM` clause.
2. The scope variable `T1` can be referenced because `T1` is specified as a table reference in the specification of the joined table.
3. The scope variable `T1` cannot be referenced because `T1` is not specified as a table reference in the specification of the joined table.
4. The scope variable `T1` can be referenced because `T1` is specified as a table reference in the specification of the joined table.

Figure 6-14: Example of the effective scope of a scope variable (if a query name is specified)

```
WITH                                                                  ] Query name Q1
  "Q1"("C1","C2") AS (SELECT "C1,"C2" FROM "T1"                       ] can be referenced.
                      UNION ALL                                       ]
                      SELECT "Q1"."C1"+1,"T2"."C2" FROM "Q1","T2"     ]
                        WHERE "Q1"."C1"<5)                            ]
  "Q2"("C1","C2") AS (SELECT "T2"."C1,"T2"."C2" FROM "T2","Q1"        ]  ] Query name Q2
                        WHERE "T2"."C1"="Q1"."C1"),                   ]  ] can be referenced.
  "Q3"("C1","C2") AS (SELECT "Q1"."C1, "Q2"."C2" FROM "Q1","Q2"       ]  ]  ] Query name Q3
                        WHERE "Q1"."C1"="Q2"."C1")                    ]  ]  ] can be referenced.
SELECT * FROM "Q3"                                                    ]  ]  ]
```

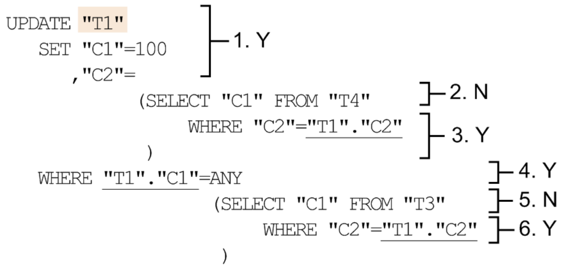
[Explanation]

A query name cannot be qualified with a schema name. If qualified with a schema name, the query name is treated as a table identifier rather than a query name. If there is a table identifier that has the same name as a query name, the table identifier is treated as a query name in the effective scope of the query name. However, outside the effective scope of the query name, the table identifier is treated as a table identifier. Therefore, when you specify a scope variable in the `FROM` clause as a table identifier, qualify the scope variable with a schema name.

(2) The effective scope of the table to be updated (scope variable) in an UPDATE statement

The effective scope of the scope variable encompasses the `SET` clause of the `UPDATE` statement, its search conditions, and the search conditions in any subqueries within those search conditions. The following figure shows an example.

Figure 6-15: Example of the effective scope of scope variable T1 specified in an UPDATE statement



```
UPDATE "T1"
   SET "C1"=100
      ,"C2"=
          (SELECT "C1" FROM "T4"
               WHERE "C2"="T1"."C2"
          )
   WHERE "T1"."C1"=ANY
                 (SELECT "C1" FROM "T3"
                      WHERE "C2"="T1"."C2"
                 )
```

Legend:

Y: Can reference scope variable `T1`.
N: Cannot reference scope variable `T1`.

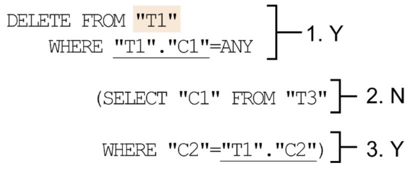
Explanation

1. The scope variable `T1` can be referenced in the `UPDATE` statement's `SET` clause and search conditions.
2. The scope variable `T1` cannot be referenced outside the search condition portion of the subquery.
3. The scope variable `T1` can be referenced in the search conditions of subqueries in the `SET` clause of the `UPDATE` statement.
4. The scope variable `T1` can be referenced in the `UPDATE` statement's `SET` clause and search conditions.
5. The scope variable `T1` cannot be referenced outside the search condition portion of the subquery.
6. The scope variable `T1` can be referenced in the search conditions of subqueries in the search conditions of the `UPDATE` statement.

(3) The effective scope of the table from which data is to be deleted (scope variable) in a DELETE statement

The effective scope of the scope variable encompasses the search conditions of the `DELETE` statement as well as the search conditions in any subqueries in those search conditions. The following figure shows an example.

Figure 6-16: Example of the effective scope of scope variable T1 specified in a DELETE statement



```
DELETE FROM "T1"
    WHERE "T1"."C1"=ANY

        (SELECT "C1" FROM "T3"

         WHERE "C2"="T1"."C2")
```

Legend:

Y: Can reference scope variable `T1`.
N: Cannot reference scope variable `T1`.

Explanation

1. The scope variable `T1` can be referenced in the search conditions of the `DELETE` statement.
2. The scope variable `T1` cannot be referenced outside of the search condition portion of the subquery.

3. The scope variable T1 can be referenced in the search conditions of subqueries in the search conditions of the DELETE statement.

(4) The effective scope of the table into which data is to be inserted into (non-scope variable) in an INSERT statement

The table that is the target of the INSERT statement does not have effective scope anywhere inside the insertion value (including subqueries) or the query expression body. This is illustrated in the following example.

Example

```
INSERT INTO "T1"     ...1
    VALUES(
            (SELECT "C1" FROM "T3"
                 WHERE "C2">="C3"     ...2
            )
           )
```

Explanation

The underlined insertion target table does not have effective scope anywhere inside the INSERT statement.

The insertion target table T1 cannot be referenced even in the search condition portion of the subquery.

6.9 Derived column names

The term *derived column* refers to a column in a table that was derived by the clauses in a query specification. Derived column names are determined according to the following rules.

6.9.1 Decision rules for derived column names in query specifications

In query specifications, derived column names are determined according to the following rules.

- In the case of a `FROM` clause

  The derived column names will be the column names of the table specified in the table reference.

  Example

  ```
  SELECT "C1","C2" FROM "T1"
  ```

  In the example above, the derived column names are the names of the columns of table `T1`.

- In the case of a `GROUP BY` clause

  The derived column names will be the names of the grouping columns.

  Example

  ```
  SELECT "C1","GC2",SUM("C3") FROM "T1"
      GROUP BY "C1",SUBSTR("C2",5,2) AS "GC2"
  ```

  In the example above, the derived column names are `"C1"` and `"GC2"`.

6.9.2 Decision rules for derived column names in query results

In query results, derived column names are determined according to the following rules.

(1) In the case of a query expression

The derived column names will be the column names derived from the results of the *query-primary* that is specified first.

(2) In the case of a query specification or subquery

■ If the `AS` clause is not specified in the i-th selection expression

- If the value expression specified in the i-th selection expression is a column specification

  The i-th derived column name will be that column name.

- If the value expression specified in the i-th selection expression is a subquery

  The i-th derived column name will be the column name derived from the result of the subquery.

- Other than above:

  No column name is set for the derived column.

■ If the `AS` clause is specified in the i-th selection expression

The i-th derived column name will be the column name in the `AS` clause specified in the i-th selection expression.

The following examples illustrate the decision rules for derived column names.

Example 1:

```
SELECT "C1","C2","C3" FROM "T1"
```

If you execute the `SELECT` statement above, the derived column names will be `"C1"`, `"C2"`, and `"C3"`. The column order of the derived columns is this same order.

Example 2:

```
SELECT "C1","C2" AS "X2","C3" FROM "T1"
```

If you execute the `SELECT` statement above, the derived column names will be `"C1"`, `"X2"`, and `"C3"`. The column order of the derived columns is this same order.

Example 3:

```
SELECT "C1",SUM("C2") AS "SUM-C2",AVG("C2") FROM "T1"
    WHERE "C3">=DATE'2011-09-03'
    GROUP BY "C1"
```

If you execute the `SELECT` statement above, the derived column names will be `"C1"`, `"SUM-C2"`, and `"no column name"`. The column order of the derived columns is this same order.

(3) In the case of a derived table

■ If a derived column list is specified

The i-th derived column name will be the i-th column name in the derived column list.

■ If no derived column list is specified

- If a table subquery is specified as a derived table

  The derived column names will be the column names of the table derived by the subquery.

  If names to be given to derived columns are not set, character strings in the `EXP`*nnnn*`_NO_NAME` format are used as the names of the derived columns. In this format, *nnnn* is an unsigned integer in the range from `0001` to `1000`. The integer *nnnn* is a sequence number that will be given in ascending order (`0001`, `0002`, ...) to each column for which a derived column name to be given is not set.

- If a table value constructor is specified as a derived table

  A character string in the `EXP`*nnnn*`_NO_NAME` format will become a derived column name. In this format, *nnnn* is an unsigned integer in the range from `0001` to `1000`. The integer *nnnn* is a sequence number that will be given in ascending order (`0001`, `0002`, ...) to each derived column.

(4) In the case of a table function derived table

■ If a table function column list is specified

The i-th derived column name will be the i-th column name specified in the table function column list.

■ If a table function column list is not specified

The derived column name will be the column name derived by means of the system-defined function specified in the table function derived table.

6.9.3 Effective scope of derived column names

The examples in the following tables illustrate the effective scope of derived column names.

Table 6-17: Example of the effective scope of derived column names (without GROUP BY clause)

Example SQL statement	Effective scope				
	C1	C2	C3	DC3	C4
SELECT "C1","C2"	Y	Y	N	Y	N
FROM "T1",	Y	Y	N	Y	N
(SELECT *	N	N	Y	N	N
FROM "T2"	N	N	Y	N	N
WHERE "C3"=100)	N	N	Y	N	N
"Y"("DC3")	N	N	N	Y	N
WHERE "C1"=100 AND	Y	Y	N	Y	N
"C2"=ANY	Y	Y	N	Y	N
(SELECT "C4"	N	N	N	N	Y
FROM "T3"	Y	Y	N	N	Y
WHERE "DC3"="C4")	Y	Y	N	Y	Y

Legend:

Y: Has scope.

N: Does not have scope.

T1,T2,T3: Table identifier

C1,C2: T1 column name

C3: T2 column name

C4: T3 column name

Y: Correlation name

DC3: Derived column name of derived table Y

Table 6-18: Example of the effective scope of derived column names (with GROUP BY clause)

Example SQL statement	Effective scope						
	C1	C2	GC2	C3	DC3	C4	C5
SELECT "C1","GC2"	Y	A	Y	N	A	N	N
FROM "T1",	Y	Y	N	N	Y	N	N
(SELECT *	N	N	N	Y	N	N	N
FROM "T2"	N	N	N	Y	N	N	N
WHERE "C3"=100)	N	N	N	Y	N	N	N
"Y"("DC3")	N	N	N	N	Y	N	N
WHERE "C1"=100 AND	Y	Y	N	N	Y	N	N
"C2"=ANY	Y	Y	N	N	Y	N	N
(SELECT "C4"	N	N	N	N	N	Y	N
FROM "T3"	Y	Y	N	N	Y	Y	N

Example SQL statement	Effective scope						
	C1	C2	GC2	C3	DC3	C4	C5
`        WHERE "DC3"="C4")`	Y	Y	N	N	Y	Y	N
`GROUP BY "C1","C2"+100 AS "GC2"`	Y	Y	N	N	Y	N	N
`HAVING "C1"=100 AND`	Y	A	Y	N	A	N	N
`  "GC2"=ANY`	Y	A	Y	N	A	N	N
`    (SELECT "C5"`	N	N	N	N	N	N	Y
`      FROM "T4"`	Y	N	N	N	N	N	Y
`      WHERE SUM("DC3")="C5")`	Y	A	N	N	A	N	Y

Legend:

Y: Has scope.

A: Has scope, but is not a grouping column, so can only be specified as the argument of a set function.

N: Does not have scope.

`T1,T2,T3,T4`: Table identifier

`C1`, `C2`: `T1` column name

`C3`: `T2` column name

`C4`: `T3` column name

`C5`: `T4` column name

`GC2`: the names of the grouping columns

Y: Correlation name

`DC3`: Derived column name of derived table Y

6.10 Reserved words

This section lists the reserved words used in HADB, and explains what to do when a name conflicts with a reserved word.

6.10.1 List of reserved words

Reserved words are keywords that are registered for use in SQL statements. Therefore, reserved words cannot be used as table or column names. If you want to specify a name with a character string that is the same as a reserved word, see 6.10.2 What to do when a name conflicts with a reserved word.

> **Note**
>
> HADB reserved words include the reserved words defined in SQL92 (ISO 9075-1992 Database Language SQL).

The following table lists the HADB reserved words.

Table 6-19: HADB reserved words

First letter	Reserved word
A	ABS, ABSOLUTE, ACCESS, ACTION, ADD, ADMIN, AFTER, AGGREGATE, AGGREGATES, ALIAS, ALL, ALLOCATE, **ALTER**, AND, ANDNOT, ANY, ARE, ARRAY, ARRAY_AGG, ARRAY_MAX_CARDINALITY, AS, ASC, ASENSITIVE, ASSERTION, **ASSIGN**, ASYMMETRIC, AT, ATOMIC, AUTHORIZATION, AVG
B	BEFORE, BEGIN, BEGIN_FRAME, BEGIN_PARTITION, BETWEEN, BIGINT, BINARY, BIT, BIT_AND_TEST, BIT_LENGTH, BLOB, BOOLEAN, BOTH, BREADTH, BY
C	CALL, CALLED, CARDINALITY, CASCADE, CASCADED, CASE, CAST, CATALOG, CEIL, CEILING, CHANGE, CHAR, CHAR_LENGTH, CHARACTER, CHARACTER_LENGTH, CHECK, CHUNK, CHUNKID, CLASS, CLASSIFIER, CLOB, **CLOSE**, CLUSTER, COALESCE, COLLATE, COLLATION, COLLECT, COLUMN, COLUMNS, COMMENT, **COMMIT**, COMPLETION, CONDITION, CONNECT, CONNECTION, CONSTRAINT, CONSTRAINTS, CONSTRUCTOR, CONTINUE, CONVERT, CORR, CORRESPONDING, COUNT, COUNT_FLOAT, COVAR_POP, COVAR_SAMP, **CREATE**, CROSS, CUBE, CUME_DIST, CURRENT, CURRENT_CATALOG, CURRENT_DATE, CURRENT_PATH, CURRENT_ROLE, CURRENT_ROW, CURRENT_SCHEMA, CURRENT_TIME, CURRENT_TIMESTAMP, CURRENT_USER, **CURRENT_USER_IS_DBA**, CURSOR, CYCLE
D	DATA, DATALINK, **DATE**, DAY, DAYS, DBA, DEALLOCATE, DEC, DECIMAL, **DECLARE**, DEFAULT, DEFERRABLE, DEFERRED, DEFINE, **DELETE**, DENSE_RANK, DEPTH, DEREF, DESC, DESCRIBE, DESCRIPTOR, DESTROY, DESTRUCTOR, DETERMINISTIC, DIAGNOSTICS, DICTIONARY, DIGITS, DISCONNECT, DISTINCT, DLNEWCOPY, DLPREVIOUSCOPY, DLURLCOMPLETE, DLURLCOMPLETEONLY, DLURLCOMPLETEWRITE, DLURLPATH, DLURLPATHONLY, DLURLPATHWRITE, DLURLSCHEME, DLURLSERVER, DLVALUE, DO, DOMAIN, DOUBLE, **DROP**, DYNAMIC
E	EACH, ELEMENT, ELSE, ELSEIF, END, END_FRAME, END_PARTITION, END-EXEC, EQUALS, ESCAPE, EVERY, EXCEPT, EXCEPTION, EXCLUSIVE, EXEC, **EXECUTE**, EXISTS, EXIT, EXP, EXTERNAL, EXTRACT
F	**FALSE**, **FETCH**, FILTER, FIRST, FIRST_VALUE, FIX, FLAT, FLOAT, FLOOR, FOR, FOREIGN, FOUND, FRAME_ROW, FREE, FROM, FULL, FUNCTION, FUSION
G	GENERAL, GET, GLOBAL, GO, GOTO, **GRANT**, GROUP, GROUPING, GROUPS
H	HANDLER, HASH, HAVING, HEX, HOLD, HOST, HOUR, HOURS

First letter	Reserved word
I	IDENTIFIED, IDENTITY, IF, IGNORE, IMMEDIATE, IMPORT, IN, INCREMENTAL, INDEX, INDICATOR, INITIAL, INITIALIZE, INITIALLY, INNER, INOUT, INPUT, INSENSITIVE, **INSERT**, INT, INTEGER, INTERSECT, INTERSECTION, INTERVAL, INTO, IS, ISOLATION, ITERATE
J	JAR, JOIN
K	KEY
L	LAG, LANGUAGE, LARGE, LAST, LAST_VALUE, LATERAL, LEAD, LEADING, LEAVE, LEFT, LENGTH, LESS, LEVEL, LIKE, LIKE_REGEX, LIMIT, LIST, LN, LOCAL, LOCALTIME, LOCALTIMESTAMP, LOCATOR, LOCK, LONG, LOOP, LOWER
M	MAP, MATCH, MATCH_NUMBER, MATCH_RECOGNIZE, MAX, MAXIMAL, MCHAR, MEASURES, MEMBER, MERGE, METHOD, MICROSECOND, MICROSECONDS, MILLISECOND, MILLISECONDS, MIN, MINUTE, MINUTES, MOD, MODE, MODIFIES, MODIFY, MODULE, MONTH, MONTHS, MULTISET, MVARCHAR
N	NAMES, NANOSECOND, NANOSECONDS, NATIONAL, NATURAL, NCHAR, NCLOB, NESTING, NEW, NEXT, NO, NONE, NORMALIZE, NOT, NOWAIT, NTH_VALUE, NTILE, **NULL**, NULLIF, NUMERIC, NVARCHAR
O	OBJECT, OCCURRENCES_REGEX, OCTET_LENGTH, OF, **OFF**, OFFSET, OLD, **ON**, ONE, ONLY, OPEN, OPERATION, OPTIMIZE, OPTION, OR, ORDER, ORDINALITY, OUT, OUTER, OUTPUT, OVER, OVERLAPS, OVERLAY
P	PAD, PAGE, PARAMETER, PARAMETERS, PARTIAL, PARTITION, PARTITIONED, PATH, PATTERN, PCTFREE, PER, PERCENT, PERCENT_RANK, PERCENTILE_CONT, PERCENTILE_DISC, PERIOD, PICOSECOND, PICOSECONDS, PORTION, POSITION, POSITION_REGEX, POSTFIX, POWER, PRECISION, PREFIX, PREORDER, **PREPARE**, PRESERVE, PRIMARY, PRIOR, PRIVATE, PRIVILEGES, PROCEDURE, PROGRAM, PROTECTED, PUBLIC, **PURGE**
R	RANGE, RANK, READ, READS, REAL, RECOVERY, RECURSIVE, REDO, REF, REFERENCES, REFERENCING, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, RELATIVE, RELEASE, REPEAT, RESIGNAL, RESTRICT, RESULT, RETURN, RETURNS, **REVOKE**, RIGHT, ROLE, **ROLLBACK**, ROLLUP, ROUTINE, ROW, ROW_NUMBER, ROWID, ROWS
S	SAVEPOINT, SCHEMA, SCOPE, SCROLL, SEARCH, SECOND, SECONDS, SECTION, SEEK, **SELECT**, SENSITIVE, SEQUENCE, SESSION, SESSION_USER, SET, SETS, SHARE, SIGNAL, SIMILAR, SIZE, SKIP, SMALLFLT, SMALLINT, SOME, SPACE, SPECIFIC, SPECIFICTYPE, SQL, SQLCODE, SQLCODE_OF_LAST_CONDITION, SQLCOUNT, SQLERRM_OF_LAST_CONDITION, SQLERROR, SQLEXCEPTION, SQLSTATE, SQLWARNING, SQRT, START, STATE, STATEMENT, STATIC, STDDEV_POP, STDDEV_SAMP, STRUCTURE, SUBMULTISET, SUBSE, SUBSTR, SUBSTRING, SUBSTRING_REGEX, SUM, SUPPRESS, SYMMETRIC, SYSTEM, SYSTEM_TIME, SYSTEM_USER
T	TABLE, TABLESAMPLE, TEMPORARY, TERMINATE, TEST, THAN, THEN, **TIME**, **TIMESTAMP**, TIMESTAMP_FORMAT, TIMEZONE_HOUR, TIMEZONE_MINUTE, TO, TRAILING, TRANSACTION, TRANSLATE, TRANSLATE_REGEX, TRANSLATION, TREAT, TRIGGER, TRIM, TRIM_ARRAY, **TRUE**, **TRUNCATE**, TYPE
U	UESCAPE, UNDER, UNDO, UNION, UNIQUE, **UNKNOWN**, UNNEST, UNTIL, **UPDATE**, UPPER, USAGE, USER, USING
V	VALUE, VALUE_OF, VALUES, VAR_POP, VAR_SAMP, VARBINARY, VARCHAR, VARCHAR_FORMAT, VARIABLE, VARYING, VERSIONING, VIEW
W	WAIT, WHEN, WHENEVER, WHERE, WHILE, WIDTH_BUCKET, WINDOW, **WITH**, WITHIN, WITHOUT, WORK, WRITE
X	XLIKE, XML, XMLAGG, XMLATTRIBUTES, XMLBINARY, XMLCAST, XMLCOMMENT, XMLCONCAT, XMLDOCUMENT, XMLELEMENT, XMLEXISTS, XMLFOREST, XMLITERATE, XMLNAMESPACES, XMLPARSE, XMLPI, XMLQUERY, XMLSERIALIZE, XMLTABLE, XMLTEXT, XMLVALIDATE
Y	YEAR, YEARS
Z	ZONE

Note

- Underlined reserved words are reserved words defined in SQL92.

- The shaded reserved words (highlighted reserved words) cannot be deleted by using the method described in (2) Unregistering a duplicated reserved word in 6.10.2 What to do when a name conflicts with a reserved word.

6.10.2 What to do when a name conflicts with a reserved word

Shown below are two ways to respond when a naming conflict with a reserved word arises.

- Enclose the name in double quotation marks.
- Unregister the reserved word.

Basically, changing the SQL statement to enclose the name in double quotation marks (`"`) is the recommended method. Unregistering the reserved word is only for cases in which changing the SQL statement would be too difficult; for example, when the amount of work would be too great.

(1) Enclosing the name in double quotation marks

Enclose the character string that is the same as a reserved word in double quotation marks (`"`). If you enclose the reserved word in double quotation marks (`"`), you can use it in SQL statements in the same way as any other string.

(2) Unregistering a duplicated reserved word

By unregistering a reserved word, you will be able to use it as a name without having to enclose it in double quotation marks (`"`). Note, however, that there are reserved words that cannot be unregistered. For the reserved words that cannot be unregistered, see 6.10.1 List of reserved words.

Important

If you delete a reserved word, you can no longer execute any SQL statements that include the deleted reserved word.

If you delete reserved words after defining a table, the reserved words remain valid in the definition. Therefore, doing so might cause an error to occur when the table is redefined.

To unregister a reserved word, specify it in the `adb_sql_prep_delrsvd_words` operand in the server definition. For details about the `adb_sql_prep_delrsvd_words` operand, see *Operands related to SQL statements (set format)* in *Detailed descriptions of the server definition operands* under *Designing the Server Definition* in the *HADB Setup and Operation Guide*.

If you want to prevent an application from actually unregistering any reserved words specified in the `adb_sql_prep_delrsvd_words` operand in the server definition, specify `N` for the client definition's `adb_sql_prep_delrsvd_use_srvdef` operand. For details about the `adb_sql_prep_delrsvd_use_srvdef` operand, see *Operands related to SQL* in *Contents of operands in the client definition* in *Designing Client Definitions* in the *HADB Application Development Guide*.

When you unregister a reserved word using the `adb_sql_prep_delrsvd_words` operand in the server definition, the effects extend even to the authorization identifiers used to connect to the HADB server. For example, if you specify `ABS` in the operand `adb_sql_prep_delrsvd_words`, you can then specify `ABS` to the `-u` option of the `adbsql` command to connect to the HADB server using the authorization identifier `ABS`.

7 Constituent Elements

This chapter describes the elements that make up SQL.

7.1 Query expression

This section describes the query expression.

7.1.1 Specification format and rules for query expressions

A query expression is a combination of a `WITH` clause and a query expression body.

The query expression body specifies either a query specification or a set operation that determines the union set, difference set, or intersection set of the tables derived from two query expression bodies. You can specify the set operator `UNION` to determine the union set, the set operator `EXCEPT` to determine the difference set, and the set operator `INTERSECT` to determine the intersection set of the tables.

When the `WITH` clause is used, the derived table produced by the derived query expression body can be given a query name, which can be specified in the query expression body itself.

Also, the query name specified in a `WITH` list element can be referenced from the query expression body in the `WITH` list element (recursive search can be performed). In this case, the query name specified in a `WITH` list element is called a *recursive query name*, and the query expression body specified in the `WITH` list element is called a *recursive query*.

(1) Specification format

```
query-expression::=[WITH-clause] query-expression-body


  WITH-clause::=WITH WITH-list-element[,WITH-list-element]...
    WITH-list-element::=query-name [(WITH-column-list)] AS (query-expression-body [LI
MIT-clause]) [maximum-number-of-recursions-specification]
      WITH-column-list::=column-name[,column-name]...
      maximum-number-of-recursions-specification::=/*>> MAX RECURSION maximum-number-
of-recursions <<*/

  query-expression-body::={query-term
                     |query-expression-body {UNION|EXCEPT} [{ALL|DISTINCT}][set-operat
ion-method-specification] query-term}
    query-term::={query-primary
                   |query-term INTERSECT [{ALL|DISTINCT}] query-primary}
      query-primary::={query-specification|(query-expression-body)}
    set-operation-method-specification::=/*>> SET OPERATION NOT BY HASH <<*/
```

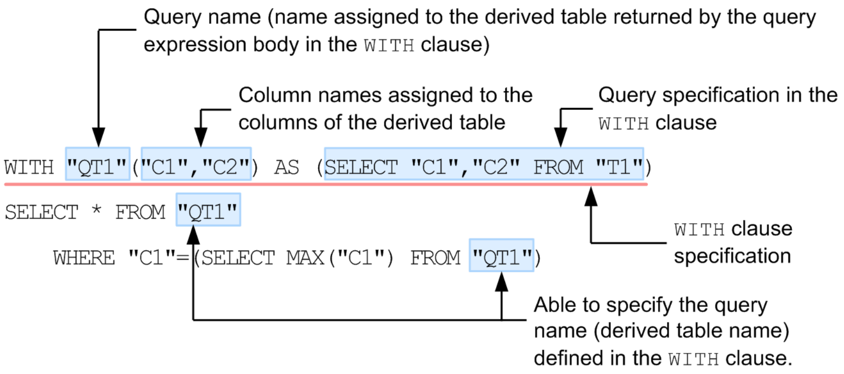
(2) Explanation of specification format

(a) WITH-clause

```
WITH-clause ::= WITH  WITH-list-element[,WITH-list-element]...
  WITH-list-element ::= query-name [(WITH-column-list)] AS (query-expression-body [LI
MIT-clause]) [maximum-number-of-recursions-specification]
    WITH-column-list ::= column-name[,column-name]...
    maximum-number-of-recursions-specification ::= /*>> MAX RECURSION maximum-number-
of-recursions <<*/
```

Specify the `WITH` clause if you want to define the result of the query specified by `AS (`*query-expression-body*`)` to be held in a temporary derived table. The following figure shows an example of specifying a `WITH` clause.

Figure 7-1: Example of specifying a WITH clause


Query name (name assigned to the derived table returned by the query expression body in the WITH clause)
Column names assigned to the columns of the derived table
Query specification in the WITH clause
WITH "QT1"("C1","C2") AS (SELECT "C1","C2" FROM "T1")
SELECT * FROM "QT1"
WHERE "C1"=(SELECT MAX("C1") FROM "QT1")
WITH clause specification
Able to specify the query name (derived table name) defined in the WITH clause.

query-name:

Specifies the name of the derived table. The name specified here is defined as the name of the query. You cannot specify the same query name as the one in the `WITH` clause.

WITH-column-list:

Specify a column name for each column in *query-name* (the derived table).

The number of column names specified in *WITH-column-list* must be the same as the number of columns derived by the outermost query in the query expression body in the corresponding `AS (`*query-expression-body*`)`.

If *WITH-column-list* is omitted, the names of the columns in *query-name* will be the names of the columns derived by the outermost query in the query expression body in the corresponding `AS (`*query-expression-body*`)`. For rules on derived column names, see 6.9 Derived column names.

Note the following points:

- The column names in *WITH-column-list* must be unique.
- If *WITH-column-list* is omitted, the column names that are derived by the query expression body must be unique.
- Do not specify a character string in the `EXP`*nnnn*`_NO_NAME` format as a column name in *WITH-column-list*. Such a column name might duplicate a derived column name that is automatically set by HADB. In this format, *nnnn* is an unsigned integer in the range from `0000` to `9999`.
- The number of columns derived by the outermost query in the query expression body specified in the corresponding `AS (`*query-expression-body*`)` cannot exceed 1,000.

`AS (`*query-expression-body* `[`*LIMIT-clause*`])`:

Specifies a query expression body.

The derived table is created from the query expression body specified here. The name of the derived table will be the name specified in *query-name*.

Note that you cannot specify a dynamic parameter inside the query expression body.

LIMIT-clause:

Specifies the maximum number of rows that will be retrieved from the results of the query expression body.

For details about the `LIMIT` clause, see 7.9 LIMIT clause.

Note that you cannot specify the `LIMIT` clause for a recursive query.

maximum-number-of-recursions-specification:

```
maximum-number-of-recursions-specification ::= /*>> MAX RECURSION maximum-number-o
f-recursions <<*/
```

Specifies the maximum number of recursions that can be performed when a recursive query is made. Use an unsigned integer literal to specify the maximum number of times recursion can be performed. The following rules apply:

- If the maximum-number-of-recursions specification is omitted, `100` is assumed as the maximum number of recursions.
- Specify an unsigned integer literal in the range from 0 to 32,767 as the maximum number of recursions.
- If the number of times recursion is performed exceeds the maximum number of recursions, the SQL statement will result in an error.
- If you specify `0` as the maximum number of recursions, recursion can be performed indefinitely. Therefore, if you specify `0`, execution of the SQL statement might be repeated indefinitely.
- If no recursive query is specified, the maximum-number-of-recursions specification is invalid.

(b) query-expression-body

```
query-expression-body::={query-term
                  |query-expression-body {UNION|EXCEPT} [{ALL|DISTINCT}][set-operatio
n-method-specification] query-term}
  query-term::={query-primary|query-term INTERSECT [{ALL|DISTINCT}] query-primary}
    query-primary::={query-specification|(query-expression-body)}
  set-operation-method-specification::=/*>> SET OPERATION NOT BY HASH <<*/
```

Specify one of the following in *query-expression-body*:

- *query-term*
- A set operation that determines the union or difference of the tables derived from a *query-expression-body* and a *query-term*.

Note that `UNION ALL`, `UNION DISTINCT`, `EXCEPT ALL`, `EXCEPT DISTINCT`, `INTERSECT ALL`, and `INTERSECT DISTINCT` are referred to as *set operators*.

query-term:

Specify one of the following in *query-term*:

- *query-primary*
- A set operation that determines the intersection of a *query-term* and a *query-primary*

`{UNION | EXCEPT}`:

Specify `UNION` to determine the union, or `EXCEPT` to determine the difference.

`{ALL|DISTINCT}`:

Specify whether to eliminate duplicate rows in the results of the set operation.

`ALL`: Do not eliminate duplicate rows in the results of the set operation.

`DISTINCT`: If there are duplicate rows in the results of the set operation, or in the operands of the set operation, consolidate the duplicate rows into a single row.

If neither `ALL` nor `DISTINCT` is specified, the system assumes that `DISTINCT` is specified.

query-primary:

In *query-primary*, specify a *query-specification* or `(`*query-expression-body*`)`.

`INTERSECT`:

Specify this to determine the intersection.

query-specification:

Specifies a query specification. For details about query specifications, see 7.2 Query specification.

(*query-expression-body*) :

Specifies a query expression body.

set-operation-method-specification:

If a set operation method specification is used, a processing method other than hash execution is used as the method for processing the set operation. For details about the method for processing the set operation, see *Methods for processing set operations* in the *HADB Application Development Guide*.

Note that, normally, there is no need to specify this. If the set operation method specification is omitted, HADB determines the method for processing the set operation.

The following shows the rules that apply when a set operation method specification is used:

- A set operation method specification that is specified for the `EXCEPT` set operation is ignored.
- A set operation method specification is applied to all set operations in a query expression body (all occurrences of `UNION DISTINCT` or `UNION ALL`). Whether a set operation method specification is applied can be checked in the access path information. For details about access path information, see *Set operation method specification* in the *HADB Application Development Guide*.

Example

```
SELECT "C1" FROM "T1"
UNION                                       ...[1]
SELECT "C1" FROM "T2"
UNION /*>> SET OPERATION NOT BY HASH <<*/   ...[2]
SELECT "C1" FROM "T3"
```

As shown in the preceding example, if a set operation method specification is written for the set operation on row [2], the set operation method specification is also applied to the set operation on row [1] (underlined portion).

(3) Rules

(a) Rules for the WITH clause

1. When there is one query name, the effective scope of the query name does not extend beyond the query expression body that follows the `WITH` clause. When there are two or more query names, the effective scope is different for each query name. For examples of the effective scope of a query name, see (1) The effective scope of scope variables specified in the FROM clause of a SELECT statement in 6.8.3 Effective scope of scope variables.
2. In the query expression body of a `WITH` list element, subqueries can be specified in a nested form. In this case, the subquery nesting depth must not exceed 31. Note that if the table specified in the `FROM` clause is a viewed table or query name, the subquery nesting depth after HADB generates the internal derived table to the viewed table or query name must not exceed 31. For details, see (a) Common rules for subqueries in (4) Rules in 7.3.1 Specification format and rules for subqueries.

   Example 1:

```
WITH "Q1" AS
   (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM
   (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM
   (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM
   (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM
   (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM
   (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM
   (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM "T1") AS DT32
   ) AS DT31 ) AS DT30 ) AS DT29 ) AS DT28 ) AS DT27 ) AS DT26 ) AS DT25 ) AS DT24
   ) AS DT23 ) AS DT22 ) AS DT21 ) AS DT20 ) AS DT19 ) AS DT18 ) AS DT17 ) AS DT16
   ) AS DT15 ) AS DT14 ) AS DT13 ) AS DT12 ) AS DT11 ) AS DT10 ) AS DT9 ) AS DT8
   ) AS DT7 ) AS DT6 ) AS DT5 ) AS DT4 ) AS DT3 ) AS DT2 ) AS DT1 ) AS DT0 )
SELECT * FROM "Q1"
```

In the preceding example, the subquery nesting depth of query name `Q1` is 32. In this case, because the maximum nesting depth is exceeded, the `SELECT` statement will result in an error.

Note that in this example, `T1` is the base table.

Example 2:

```
WITH "Q2" AS
  (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM
  (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM
  (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM
  (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM
  (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM
  (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM
  (SELECT * FROM (SELECT * FROM "T1") AS DT30
  ) AS DT29 ) AS DT28 ) AS DT27 ) AS DT26 ) AS DT25 ) AS DT24 ) AS DT23 ) AS DT22
  ) AS DT21 ) AS DT20 ) AS DT19 ) AS DT18 ) AS DT17 ) AS DT16 ) AS DT15 ) AS DT14
  ) AS DT13 ) AS DT12 ) AS DT11 ) AS DT10 ) AS DT9 ) AS DT8 ) AS DT7 ) AS DT6
  ) AS DT5 ) AS DT4 ) AS DT3 ) AS DT2 ) AS DT1 ) AS DT0 ),
  "Q3" AS (SELECT * FROM "Q2"),
  "Q4" AS (SELECT * FROM "Q3")
SELECT * FROM "Q4"
```

- For query name `Q2`, the subquery nesting depth is 30. In this case, the maximum nesting depth is not exceeded.
- For query name `Q3`, the subquery nesting depth becomes 31 when the internal derived table is generated. In this case, the maximum nesting depth is not exceeded.
- For query name `Q4`, the subquery nesting depth becomes 32 when the internal derived table is generated. In this case, because the maximum nesting depth is exceeded, the `SELECT` statement will result in an error.

Note that in this example, `T1` is the base table.

Example 3:

```
WITH "Q5" AS (SELECT "C1" FROM (SELECT "C1" FROM "T1") AS DT
              UNION ALL
              SELECT "C1"+1 FROM "Q5" WHERE "C1"+1 < 5),
     "Q6" AS (SELECT * FROM
              (SELECT * FROM
               (SELECT * FROM
                (SELECT * FROM
                 (SELECT * FROM
                  (SELECT * FROM "Q5") AS DT4
                 ) AS DT3
                ) AS DT2
               ) AS DT1
              ) AS DT0),
      "Q7" AS
      (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM
      (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM
      (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM
      (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM
      (SELECT * FROM (SELECT * FROM (SELECT * FROM (SELECT * FROM
      (SELECT * FROM "Q6") AS DT23
      ) AS DT22 ) AS DT21 ) AS DT20 ) AS DT19 ) AS DT18
      ) AS DT17 ) AS DT16 ) AS DT15 ) AS DT14 ) AS DT13
      ) AS DT12 ) AS DT11 ) AS DT10 ) AS DT9 ) AS DT8
      ) AS DT7 ) AS DT6 ) AS DT5 ) AS DT4 ) AS DT3
      ) AS DT2 ) AS DT1 ) AS DT0 )
SELECT * FROM "Q7"
```

- For query name `Q5`, the subquery nesting depth is 1. In this case, the maximum nesting depth is not exceeded.

- For query name Q6, the subquery nesting depth becomes seven when the internal derived table is generated. In this case, the maximum nesting depth is not exceeded.
- For query name Q7, the subquery nesting depth becomes 32 when the internal derived table is generated. In this case, because the maximum nesting depth is exceeded, the SELECT statement will result in an error.

Note that in this example, T1 is the base table.

(b) Rules for recursive queries

1. A recursive query must include the following items: one or more query specifications# that do not include a recursive query name that references the recursive query, and one or more query specifications# that include a recursive query name that references the recursive query. Query specifications# that do not include a recursive query name that references the recursive query are called *anchor members*, and query specifications# that include a recursive query name that references the recursive query are called *recursive members*.

   #: Subqueries do not apply.

2. To specify multiple recursive members, make sure that each of them is an operand of a set operation (UNION ALL).
3. Also, make sure that the last anchor member and first recursive member that are specified in a recursive query are operands of a set operation (UNION ALL). The following shows an example of specifying anchor and recursive members.

   Example:

```
WITH "Q1"("C1","C2")
    AS (SELECT "C1","C2" FROM "T1" WHERE "C1" > 0                            ...1
        UNION ALL
        SELECT "Q1"."C1"+1,"T1"."C2" FROM "Q1","T1" WHERE "Q1"."C1" < 5)    ...2
SELECT * FROM "Q1"
```

   [Explanation]

   1. The underlined entry is an anchor member.
   2. The underlined entry is a recursive member.

4. All anchor members must be specified before the first recursive member specified in the recursive query.
5. In the FROM clause of a recursive member in a recursive query, two or more recursive query names that reference the recursive query cannot be specified.
6. In a subquery of a recursive query, a recursive query name that references the recursive query cannot be specified.
7. The following items cannot be specified in recursive members. The following items also cannot be specified in subqueries in recursive members.

   - SELECT DISTINCT
   - GROUP BY clause
   - HAVING clause
   - LIMIT clause
   - Set functions
   - LEFT OUTER JOIN
   - RIGHT OUTER JOIN
   - FULL OUTER JOIN

   However, the preceding items can be specified in the following locations:

   - A viewed table specified in a recursive member

- A viewed table specified in a subquery in a recursive member
- A derived query derived from a query name

8. The following two items must have the same data type and data length:

   - Data type and data length of the columns#1 that make up the table that is derived from the result of a set operation for all recursive members that are specified in the recursive query
   - Data type and data length of the columns#2 that make up the table that is derived from the result of a set operation for all anchor members

   #1: If there is only one recursive member, the selection expression of the recursive member applies.

   #2: If there is only one anchor member, the selection expression of the anchor member applies.

9. An overview of a recursive query search performed under the following conditions is described later:

   - The query expression body that performs a set operation for all anchor members that are specified in the recursive query is Q_0.
   - The search result of Q_0 is X_0.
   - The query expression body that performs a set operation for all recursive members that are specified in the recursive query is Qi.
   - The search result of Qi is Xi.
   - The number of recursions is i.

   Overview of a recursive query search

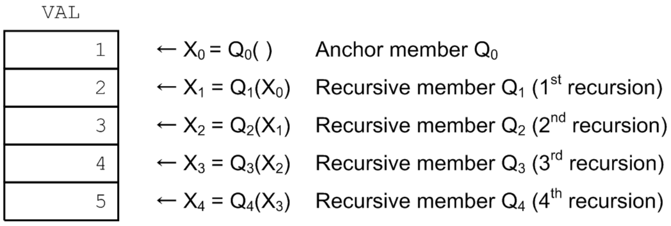
   1. A search is performed with Q_0 (the search result is X_0).
   2. The search result X_0 becomes the result of the recursive query.
   3. Based on the previous search result X_{i-1}, a search is performed with Qi (the search result is Xi).
   4. Either of the following operations is performed according to the search result Xi:

   - If the search result Xi is not a null row, the search result Xi becomes the result of the recursive query, and the processing returns to step 3.
   - If the search result Xi is a null row, the recursive query terminates.

   Example:

```
WITH "REC"("VAL") AS (
    SELECT * FROM (VALUES(1))                           <= Anchor member
    UNION ALL
    SELECT "VAL" + 1 FROM "REC" WHERE "VAL" + 1 <= 5    <= Recursive member
    )
SELECT "VAL" FROM "REC"
```

   Example of running the preceding SQL statement



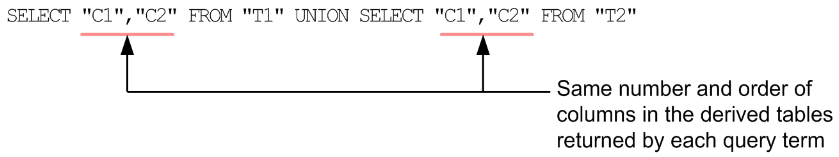
VAL		
1	← $X_0 = Q_0(\,)$	Anchor member Q_0
2	← $X_1 = Q_1(X_0)$	Recursive member Q_1 (1st recursion)
3	← $X_2 = Q_2(X_1)$	Recursive member Q_2 (2nd recursion)
4	← $X_3 = Q_3(X_2)$	Recursive member Q_3 (3rd recursion)
5	← $X_4 = Q_4(X_3)$	Recursive member Q_4 (4th recursion)

   When the preceding SQL statement is run, recursion occurs four times.

(c) Rules for set operations

1. In the evaluation order of set operations, parentheses take precedence over `INTERSECT`, which takes precedence over `UNION` and `EXCEPT`.
2. Derived tables returned by a *query-term* and *query-term*, a *query-term* and *query-primary*, or a *query-primary* and *query-primary* combination that are targeted by the set operation are treated as a set of rows, on which the set operation is performed.
3. The number and order of the columns must be identical in the tables targeted by the set operation (the derived tables returned by the query terms).

   Example:



```
SELECT "C1","C2" FROM "T1" UNION SELECT "C1","C2" FROM "T2"
```

Same number and order of columns in the derived tables returned by each query term

   Furthermore, the data types of the corresponding columns must be data types that can be compared. In the above example, column `C1` in table `T1` and column `C1` in table `T2` must have data types that can be compared. Similarly, column `C2` in table `T1` and column `C2` in table `T2` must have data types that can be compared.

   For details about data types that can be compared, see (1) Data types that can be compared in 6.2.2 Data types that can be converted, assigned, and compared.

   However, the following data cannot be compared with the set operation:

   - `DATE` type data cannot be compared to character string data (even to the predefined input representation of a date).
   - `TIME` type data cannot be compared to character string data (even to the predefined input representation of a time).
   - `TIMESTAMP` type data cannot be compared to character string data (even to the predefined input representation of a time stamp).

   For information about predefined input representations, see 6.3.3 Predefined character-string representations.
4. The column names of a table derived by a set operation are determined by the column names of the tables derived by the query terms specified in the set operation. For rules about the column names of tables derived from set operations, see (1) In the case of a query expression in 6.9.2 Decision rules for derived column names in query results.
5. The number and order of columns in the derived table produced by the set operation will be the same as the columns that make up the corresponding tables that were targeted by the set operation (the derived tables returned by query terms). Note that if even one of the corresponding columns does not have the `NOT NULL` constraint (null values are allowed), the set operation is executed without the `NOT NULL` constraint on all the columns of the derived table (null values are allowed).
6. The data types and data lengths of the columns in the derived table produced by the set operation are determined by the data types and data lengths of the columns that make up the corresponding tables that were targeted by the set operation (the derived tables returned by the query terms). For details, see 7.20.2 Data types of the results of value expressions.
7. *Q1* and *Q2* are set operation operands of the set operation. In this case, the number of occurrences of a given row in the results of *Q1 set-operation Q2* is as shown in the following table.

Table 7-1: Number of occurrences of a given row in the results of the set operation

Set operator	Number of occurrences of a given row R in the results of the set operation	
	When ALL is not specified	When ALL is specified
UNION	• 0 (when *m* = 0 and *n* = 0) • 1 (when *m* > 0 or *n* > 0)	*m* + *n*
EXCEPT	• 0 (when *m* = 0 or *n* > 0) • 1 (when *m* > 0 and *n* = 0)	MAX(*m* - *n*, 0)
INTERSECT	• 0 (when *m* = 0 or *n* = 0) • 1 (when *m* > 0 and *n* > 0)	MIN(*m*, *n*)

Notes:

In the table, *m* represents the number of occurrences of *R* in *Q1*, and *n* represents the number of occurrences of row *R* in *Q2*.

8. If the set operations specified in the SQL statement specified in the query expression body are all UNION, a maximum of 1,023 set operations can be specified. However, if the specified set operations include EXCEPT or INTERSECT, no more than 63 set operations can be specified.

   Note that when a viewed table is specified in an SQL statement, HADB uses an internal derived table based on the query expression specified in the CREATE VIEW statement. The rules for the maximum number of set operations apply to this internal derived table.

9. A maximum of 63 outer joins with FULL OUTER JOIN specified as the joined table mode can be specified in the SQL statement specified in the query expression body.

10. A set operation specified in a set operation that has DISTINCT specification might be treated as one having DISTINCT specification.

(4) Examples

Example 1 (WITH clause example)

This example obtains the maximum sales value (QMAXSALES) for each product name (PUR-NAME) from the sales history table (SALESLIST) and product table (PRODUCTSLIST).

```
WITH "QT1"("QCODE","QMAXSALES") AS
    (SELECT "PUR-CODE",MAX("PRICE"*"QUANTITY") FROM "SALESLIST"
         GROUP BY "PUR-CODE")
SELECT "PUR-NAME","QMAXSALES" FROM "QT1"
    INNER JOIN "PRODUCTSLIST" ON "QCODE"="PUR-CODE"
```

The underlined portion indicates the WITH clause.

Example 2 (union set operation example)

From branch A's sales history table (SALESLIST_A) and branch B's sales history table (SALESLIST_B), this example obtains the combined sales history of branch A and branch B.

```
SELECT "A"."USERID","A"."PUR-CODE","A"."PUR-NUM"
    FROM "SALESLIST_A" "A"
UNION ALL
SELECT "B"."USERID","B"."PUR-CODE","B"."PUR-NUM"
    FROM "SALESLIST_B" "B"
```

■ Branch A's sales history table (SALESLIST_A)

USERID	PUR-CODE	PUR-NUM
U00555	P001	1
U00555	P003	2
U00358	P001	3
U00614	P001	1

■ Branch B's sales history table (SALESLIST_B)

USERID	PUR-CODE	PUR-NUM
U00358	P002	6
U00212	P005	2
U00614	P001	1

■ Retrieval results

USERID	PUR-CODE	PUR-NUM
U00555	P001	1
U00555	P003	2
U00358	P001	3
U00614	P001	1
U00358	P002	6
U00212	P005	2
U00614	P001	1

Example 3 (union set operation example)

From branch A's sales history table (SALESLIST_A) and branch B's sales history table (SALESLIST_B), this example obtains the customer ID (USERID) of every customer who has made a purchase at either branch A or branch B.

```
SELECT "A"."USERID"
    FROM "SALESLIST_A" "A"
UNION DISTINCT
SELECT "B"."USERID"
    FROM "SALESLIST_B" "B"
```

■ Branch A's sales history table (SALESLIST_A)

USERID	PUR-CODE	PUR-NUM
U00555	P001	1
U00555	P003	2
U00358	P001	3
U00614	P001	1

■ Branch B's sales history table (SALESLIST_B)

USERID	PUR-CODE	PUR-NUM
U00358	P002	6
U00212	P005	2
U00614	P001	1

■ Retrieval results

USERID
U00212
U00358
U00555
U00614

Example 4 (difference set operation example)

From branch A's sales history table (SALESLIST_A) and branch B's sales history table (SALESLIST_B), this example obtains the customer ID (USERID) of every customer who has made a purchase at branch A but not at branch B.

```
SELECT "A"."USERID"
    FROM "SALESLIST_A" "A"
EXCEPT
SELECT "B"."USERID"
    FROM "SALESLIST_B" "B"
```

■ Branch A's sales history table (SALESLIST_A)

USERID	PUR-CODE	PUR-NUM
U00555	P001	1
U00555	P003	2
U00358	P001	3
U00614	P001	1

■ Branch B's sales history table (SALESLIST_B)

USERID	PUR-CODE	PUR-NUM
U00358	P002	6
U00212	P005	2
U00614	P001	1

■ Retrieval results

USERID
U00555

Example 5 (intersection set operation example)

From branch A's sales history table (SALESLIST_A) and branch B's sales history table (SALESLIST_B), this example obtains the customer ID (USERID) of every customer who has made a purchase at both branch A and branch B.

```
SELECT "A"."USERID"
    FROM "SALESLIST_A" "A"
INTERSECT
SELECT "B"."USERID"
    FROM "SALESLIST_B" "B"
```

■ Branch A's sales history table (SALESLIST_A)

USERID	PUR-CODE	PUR-NUM
U00555	P001	1
U00555	P003	2
U00358	P001	3
U00614	P001	1

■ Branch B's sales history table (SALESLIST_B)

USERID	PUR-CODE	PUR-NUM
U00358	P002	6
U00212	P005	2
U00614	P001	1

■ Retrieval results

USERID
U00358
U00614

Example 6 (recursive query example)

Part Parts_B consists of some other parts. This example obtains the parts that are out of stock.

```
WITH "V1"("ID","PARENT","NAME","QUANTITY") AS (
  SELECT "A"."ID","A"."PARENT","A"."NAME","A"."QUANTITY"
      FROM "BOMS" "A" WHERE "A"."ID"=2
  UNION ALL
  SELECT "A"."ID","A"."PARENT","A"."NAME","A"."QUANTITY"
      FROM "V1", "BOMS" "A" WHERE "A"."PARENT" = "V1"."ID"
  )
SELECT "NAME","QUANTITY" FROM "V1" WHERE "QUANTITY"=0
```

■ Search result

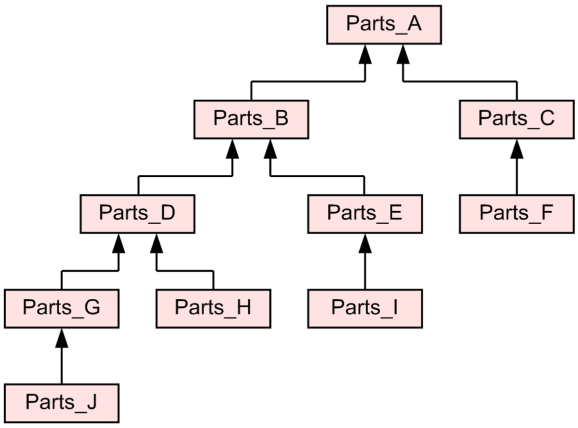
NAME	QUANTITY
Parts_D	0
Parts_E	0

The following shows the configuration of the bill of materials (BOMS) that contains the quantity in stock and other information, and the hierarchical structure of parts.

■ Bill of materials (BOMS)

ID	PARENT	NAME	QUANTITY
(part ID)	(parent ID)	(part name)	(quantity in stock)
1	NULL	Parts_A	1
2	1	Parts_B	150
3	1	Parts_C	250
4	2	Parts_D	0
5	2	Parts_E	0
6	3	Parts_F	0
7	4	Parts_G	11
8	4	Parts_H	15000
9	5	Parts_I	2000
10	7	Parts_J	100

■ Hierarchical structure of parts



(5) Notes

1. When a set operation is specified, a work table might be created. If the size of the work table DB area where the work table is to be created has not been estimated correctly, performance might be degraded. For details about estimating the size of the work table DB area, see the *HADB Setup and Operation Guide*. For details about work tables, see *Considerations when executing an SQL statement that creates work tables* in the *HADB Application Development Guide*.
2. If hash execution is used as the method for processing the set operation, a hash table area of an appropriate size is required. The size of the hash table area is specified in the `adb_sql_exe_hashtbl_area_size` operand in the server definition or client definition. For details about the method for processing the set operation, see *Methods for processing set operations* in the *HADB Application Development Guide*.
3. If hash execution is used as the method for processing the set operation, a derived table is created. HADB automatically assigns a correlation name in the following format to the derived table:

   ```
   ##DRVTBL_xxxxxxxxxx
   ```

   In the preceding format, *xxxxxxxxxx* is a 10-digit integer.
4. When the following predicates are evaluated by using a B-tree index, the set operation specified in a table subquery might be treated as one having `DISTINCT` specification:
   - `IN` predicate in which a table subquery is specified
   - Quantified predicate (`=ANY` or `=SOME` specification)

7.2 Query specification

This section describes the query specification.

7.2.1 Specification format and rules for query specifications

The query specification specifies the retrieval results to be output (the selection list) and the table retrieval criteria (the table expression).

(1) Specification format

```
query-specification::=SELECT [{ALL|DISTINCT}][SELECT-deduplication-method-specificati
on] selection-list table-expression

  SELECT-deduplication-method-specification::=/*>> SELECT DISTINCT NOT BY HASH <<*/
  selection-list::={*|selection-expression[,selection-expression]...}
    selection-expression::={value-expression [AS-clause]|NULL [AS-clause]|table-speci
fication.*|[table-specification.]ROW}
      AS-clause::=[AS] column-name
```

(2) Explanation of specification format

(a) {ALL|DISTINCT}

Specifies whether to exclude duplicate rows from the retrieval results.

ALL:

The retrieval results are output as-is, including duplicate rows.

DISTINCT:

If there are duplicate rows in the retrieval results, the retrieval results are output with all duplicates eliminated.

For details about differences in retrieval results when DISTINCT is specified, see 1.10.1 Example: Retrieve customers who purchased products.

Note the following points:

- If DISTINCT is specified, a work table might be created. If the size of the work table DB area where the work table is to be created has not been estimated correctly, it might result in performance degradation. For details about estimating the size of the work table DB area, see the *HADB Setup and Operation Guide*. For details about work tables, see *Considerations when executing an SQL statement that creates work tables* in the *HADB Application Development Guide*.
- If hash execution is used as the method for processing SELECT DISTINCT, a hash table area of an appropriate size is required. The size of the hash table area is specified in the adb_sql_exe_hashtbl_area_size operand in the server definition or client definition. For details about the method for processing SELECT DISTINCT, see *Method for processing SELECT DISTINCT* in the *HADB Application Development Guide*.

If neither ALL nor DISTINCT is specified, the system assumes that ALL is specified.

(b) SELECT deduplication method specification

If a SELECT deduplication method specification is used, a processing method other than hash execution is used as the method for processing SELECT DISTINCT. For details about the method for processing SELECT DISTINCT, see *Method for processing SELECT DISTINCT* in the *HADB Application Development Guide*.

Note that, normally, there is no need to specify this. If the SELECT deduplication method specification is omitted, HADB determines the method for processing SELECT DISTINCT.

(c) Selection list

```
selection-list ::= {* | selection-expression[,selection-expression]...}
```

The selection list specifies the retrieval results to be output.

*:

Specify this to output all columns of the table to be output in the retrieval results.

If * is specified, all columns from all tables specified in the FROM clause will be output in the order in which the tables were specified in the FROM clause. The order of the columns in each table will be the order specified when the table was defined.

selection-expression[, *selection-expression*]...:

Specifies the retrieval results to be output.

(d) Table expression

The table expression specifies the tables from which output is to be retrieved. You can also specify the conditions for retrieving from the tables (search conditions), and the conditions for selecting groups when performing grouping. For details about table expressions, see 7.4 Table expression.

(e) Selection expression

```
selection-expression ::= {value-expression [AS-clause]|NULL [AS-clause]|table-specifi
cation.*|[table-specification.]ROW}
  AS-clause ::= [AS] column-name
```

The selection expression specifies the retrieval results to be output.

You cannot specify an external reference column in a selection expression. For details about external reference columns, see (a) Common rules for subqueries in (4) Rules in 7.3.1 Specification format and rules for subqueries.

value-expression [*AS-clause*]:

Specify the retrieval results to be output in the form of a value expression.

Specify the AS clause if you want to change the column names in the retrieval results.

For details about column names and column ordering in the retrieval results, see 6.9 Derived column names.

Note that for the first query specification (except the query specification in the WITH clause of the SELECT statement), the column name in the AS clause can include a half-width (left or right) parenthesis. For the second and subsequent query specifications, the column name in the AS clause cannot include a half-width parenthesis.

Do not specify a character string in the EXP*nnnn*_NO_NAME format as a column name in the AS clause. Such a column name might duplicate a derived column name that is automatically set by HADB. In this format, *nnnn* is an unsigned integer in the range from 0000 to 9999.

■ Notes on specifying the GROUP BY clause, the HAVING clause, or a set function

When you specify the GROUP BY clause, the HAVING clause, or a set function, the column specification included in the value expression in a selection expression must meet any of the following conditions:

- **It specifies a grouping column name.**

  Example of a correct SQL statement:

  ```
  SELECT "C1" FROM "T1" GROUP BY "C1" HAVING "C1">100
  ```

  In the preceding example, the grouping column name specified in the GROUP BY clause is specified in the value expression in a selection expression.

- **It specifies the argument to a set function.**

  Example of a correct SQL statement:

  ```
  SELECT COUNT("C2") FROM "T1" HAVING MAX("C1")>100
  ```

  In the preceding example, the argument to a set function is a column specification.

  Example of an SQL statement that generates an error:

  ```
  SELECT COUNT("C1")+"C1" FROM "T1"
  ```

  In the preceding example, a column specification is used in a location other than the argument to a set function.

- **It specifies the same value expression as the value expression included in a grouping specification (value expression that includes a column specification).**

  Example of a correct SQL statement:

  ```
  SELECT "C1"+"C2" FROM "T1" GROUP BY "C1"+"C2"
  ```

  In the preceding example, the same value expression that is included in the grouping specification in the GROUP BY clause is specified as the value expression in a selection expression.

  Note that if a column specification having a table specification is specified in a selection expression and the name of the column is the same as an existing grouping column, the grouping column cannot be referenced.

  Example of an SQL statement that generates an error:

  ```
  SELECT "T1"."C2" FROM "T1" GROUP BY "C1"+1 AS "C2"
  ```

  In the preceding example, because a column specification having a table specification ("T1."C2") is specified in a selection expression, grouping column "C1"+1 AS "C2" cannot be referenced even though the column name is the same. Therefore, the preceding SQL statement will result in an error.

NULL [*AS-clause*]:

Specify this if you want null values to be output to the retrieval results.

To add column names to the retrieval results, specify the AS clause.

The following rules apply:

- NULL can be specified in the selection expression in the outermost query specification of the SELECT statement.
- NULL cannot be specified in the selection expression of a query specification subject to a set operation.

  Example of an SQL statement that generates an error:

  ```
  SELECT NULL FROM "T1" UNION SELECT "C1" FROM "T1"
  ```

- NULL cannot be specified in the selection expression of the query specification in the WITH clause.

For the retrieval results, the following rules apply:

- The data type of the result of NULL will be INTEGER.
- The NOT NULL constraint does not apply to the result of NULL (the null value is allowed).

table-specification.*:

If this is specified, the retrieval results will consist of all of the columns in the specified table. The order of columns in the retrieval results will be the same as the order of columns in the specified table.

If this is specified and you want to specify a GROUP BY clause, HAVING clause, or set function in the query specification, specify the column specification in the selection expression as follows:

- Grouping column

[*table-specification*.]ROW:

If the preceding specification is included, the entire row is retrieved to one area as a single entity. ROW means the entire row.

Regardless of the data types of the columns that make up the row, the ROW that is retrieved is stored in a CHAR type variable. Be sure to remove any leading or trailing spaces. The data length of the retrieved row is its row length (sum of the data lengths of the columns that make up the row). For details about how to calculate the row length, see the *ROWSZ* calculation formula in *Determining the number of pages for storing each type of row* in the *HADB Setup and Operation Guide*.

The rules for specifying ROW are as follows:

- It can be specified only for a FIX table.
- If ROW is specified, none of the following can be specified in the query specification:
  - Set function
  - Window function
  - GROUP BY clause
  - SELECT DISTINCT
  - Set operation
- If you specify LEFT OUTER JOIN as the joined table mode in a FROM clause, you cannot specify ROW for the table on the right.
- If you specify RIGHT OUTER JOIN as the joined table mode in a FROM clause, you cannot specify ROW for the table on the left.
- If you specify FULL OUTER JOIN as the joined table mode in a FROM clause, you cannot specify ROW for the table on either side.
- You cannot specify ROW in a query specification in a WITH clause.

(3) Privileges required at execution

To execute a query specification, the SELECT privilege is required on all tables specified in the query specification.

(4) Rules

1. A maximum of 1,000 columns are allowed in the retrieval results for a query specification.
2. If *table-specification*.* or *table-specification*.ROW is specified in the selection expression, the table specification must be equivalent to the scope variable that includes that selection expression in the scope variable's effective scope.
3. When you include an archivable multi-chunk table in a query specification, be careful about the specification of search conditions. For details, see *Considerations when searching an archivable multi-chunk table* in the *HADB Application Development Guide*. Make sure that you read the preceding section when you include an archivable multi-chunk table in a query specification.
4. When the following predicates are evaluated by using a B-tree index, the query specification included in a table subquery might be treated as one having SELECT DISTINCT specification:

- IN predicate in which a table subquery is specified
- Quantified predicate (=ANY or =SOME specification)

5. If DISTINCT is specified for a set operation, the query specification in the set operation might be treated as one having SELECT DISTINCT specification.

(5) Example

The following example illustrates a query specification.

Example

Using the data in the sales history table (SALESLIST), this example determines the sum and average of the quantities purchased on or after September 3, 2011 by product code (PUR-CODE). Furthermore, it retrieves only the product codes for which the sum of the quantities purchased is 20 or fewer.

```
SELECT "PUR-CODE",SUM("PUR-NUM"),AVG("PUR-NUM")          ...1
    FROM "SALESLIST"                                     ...2
        WHERE "PUR-DATE">=DATE'2011-09-03'               ...2
        GROUP BY "PUR-CODE"                              ...2
        HAVING SUM("PUR-NUM")<=20                        ...2
```

Explanation

In this example, the entire SELECT statement is a query specification.

1. The underlined portion indicates the selection list.
2. The underlined portion indicates the table expression.

7.3 Subqueries

This section describes subqueries.

7.3.1 Specification format and rules for subqueries

A *subquery* is an inner query specification. There are the following two types of subqueries:

- Scalar subquery

  This is a subquery for which the result of the query is at most a single row containing a single column (a single value).

- Table subquery

  This is a subquery for which the result of the query is zero or more rows containing one or more columns.

(1) Specification format

```
subquery::=([subquery-processing-method-specification] query-expression-body [LIMIT-c
lause])

  subquery-processing-method-specification::=/*>> SUBQUERY NOT BY HASH[subquery-proce
ssing-delegation-specification]<<*/
    subquery-processing-delegation-specification::=(DELEGATION)
```

(2) Explanation of specification format

subquery-processing-method-specification:

If a subquery processing method specification is used, a processing method other than hash execution is used as the method for processing the subquery. For details about the method for processing the subquery, see *Subquery processing methods* in the *HADB Application Development Guide*.

Note that, normally, there is no need to specify this. If the subquery processing method specification is omitted, HADB determines the method for processing the subquery.

Also note that a subquery processing method specification cannot be used for the following items:

- Table subquery for a derived table
- Table subquery for a multiset value constructor by query

subquery-processing-delegation-specification:

If a subquery processing delegation specification is used, SQL processing real threads that are used for other processing can be assigned to the search processing of a subquery that includes external reference columns.

Note that, normally, there is no need to specify this. If much search processing is required to obtain the results of a subquery that includes external reference columns, consider using a subquery processing delegation specification.

Example

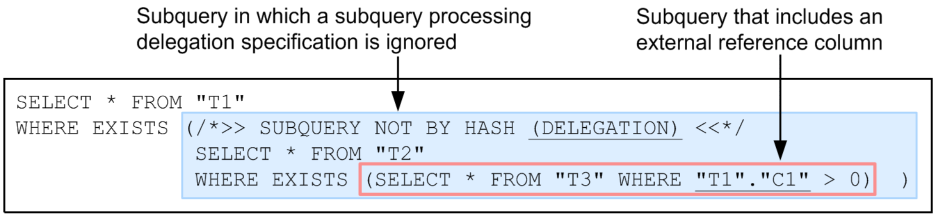
```
SELECT COUNT(*) FROM "T1" WHERE "T1"."C1" = ANY(
  SELECT "T2"."C1" FROM "T2" WHERE "T1"."C2" = "T2"."C2")
```

In the preceding SQL statement, table `T2` is searched each time a result of table `T1` is obtained. If table `T2` that satisfies the search condition `"T1"."C2" = "T2"."C2"` has many rows, the search performance might be improved by using a subquery processing delegation specification. However, if a subquery processing delegation specification is used, the processing that assigns other SQL processing real threads to the search processing of a subquery that includes external reference columns becomes an overhead. Therefore, the search performance might be lowered depending on the search conditions.

The following shows the rules that apply when a subquery processing delegation specification is used:

- A subquery processing delegation specification is ignored if it is used for a subquery that satisfies either of the following conditions:
  - Subquery that does not include external reference columns
  - Subquery that is specified in an SQL statement for which out-of-order execution is not used

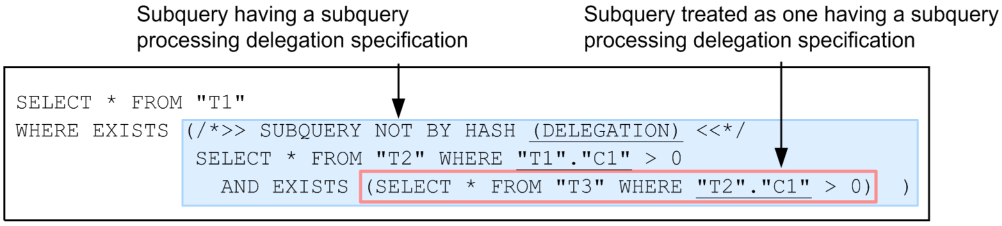
  Example of when a subquery processing delegation specification is ignored:

  
  Subquery in which a subquery processing delegation specification is ignored
Subquery that includes an external reference column

  ```
  SELECT * FROM "T1"
  WHERE EXISTS (/*>> SUBQUERY NOT BY HASH (DELEGATION) <<*/
                SELECT * FROM "T2"
                WHERE EXISTS (SELECT * FROM "T3" WHERE "T1"."C1" > 0)   )
  ```

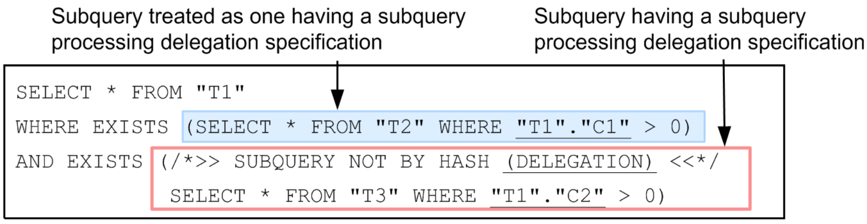
  In the preceding example, a subquery processing delegation specification is included in a subquery that does not include an external reference column. However, no subquery processing delegation specification is included in a subquery that includes an external reference column. In this case, because no subquery processing delegation specification is included in a subquery that includes an external reference column, the subquery processing delegation specification is ignored.

- If there are nested subqueries that include external reference columns and at least one of the nested subqueries has a subquery processing delegation specification, all the nested subqueries are treated as those having a subquery processing delegation specification. The following shows an example of an SQL statement that contains nested subqueries that include an external reference column. In this example, two subqueries that include an external reference column are specified, and only one of them has a subquery processing delegation specification. In this case, both subqueries are assumed to have a subquery processing delegation specification.

  
  Subquery having a subquery processing delegation specification
Subquery treated as one having a subquery processing delegation specification

  ```
  SELECT * FROM "T1"
  WHERE EXISTS (/*>> SUBQUERY NOT BY HASH (DELEGATION) <<*/
                SELECT * FROM "T2" WHERE "T1"."C1" > 0
                  AND EXISTS (SELECT * FROM "T3" WHERE "T2"."C1" > 0)   )
  ```

- If there are subqueries that reference the same table as external reference columns and at least one of them has a subquery processing delegation specification, the other subqueries are also assumed to have a subquery processing delegation specification. The following shows an example. In this example, the SQL statement contains two subqueries that reference the same table as an external reference column, and only one of them has a subquery processing delegation specification. In this case, both subqueries are assumed to have a subquery processing delegation specification.

  
  Subquery treated as one having a subquery processing delegation specification
Subquery having a subquery processing delegation specification

  ```
  SELECT * FROM "T1"
  WHERE EXISTS (SELECT * FROM "T2" WHERE "T1"."C1" > 0)
  AND EXISTS (/*>> SUBQUERY NOT BY HASH (DELEGATION) <<*/
               SELECT * FROM "T3" WHERE "T1"."C2" > 0)
  ```

query-expression-body:

For details about *query-expression-body*, see (2) Explanation of specification format in 7.1.1 Specification format and rules for query expressions.

LIMIT-clause:

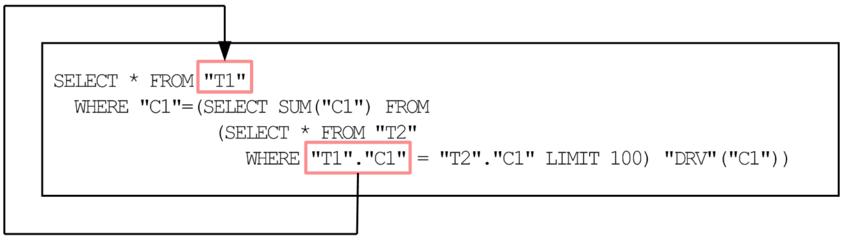
Specifies the maximum number of rows that will be retrieved from the results of the query expression body.

For details about the `LIMIT` clause, see 7.9 LIMIT clause.

The `LIMIT` clause can only be specified for a derived table returned by a table subquery, or in a scalar subquery. However, the following is a case of a derived table where a `LIMIT` clause is not permitted:

- A derived table that references a table that is outside the derived table in which the `LIMIT` clause is specified

  Example of an SQL statement that generates an error:



```
SELECT * FROM "T1"
  WHERE "C1"=(SELECT SUM("C1") FROM
               (SELECT * FROM "T2"
                  WHERE "T1"."C1" = "T2"."C1" LIMIT 100) "DRV"("C1"))
```

  In this example, `"T1"."C1"` references a table that is outside the derived table in which the `LIMIT` clause is specified (correlation name: `DRV`). The `LIMIT` clause is therefore not permitted here.

  For details about derived tables, see 7.11.1 Specification format for table references.

(3) Privileges required at execution

To execute a subquery, the `SELECT` privilege is required on all tables referenced in the subquery.

(4) Rules

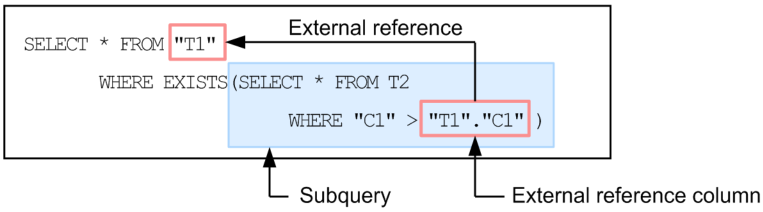
(a) Common rules for subqueries

1. The data type of the result of the subquery will the same as the data type of the result of the query expression body.
2. For details about the column names of tables derived in the results of subqueries, see 6.9 Derived column names.
3. The following cannot be specified in a selection expression in a subquery:
   - An external reference column
   - `[`*table-specification*`.]ROW`

   ■ External reference column

   A reference in the search conditions of a subquery to a table specified in the `FROM` clause of the outer query is known as an *external reference*. A corresponding referenced column is called an *external reference column*. An example of an external reference column is shown in the following figure.

   Figure 7-2: External reference column



```
SELECT * FROM "T1"
       WHERE EXISTS(SELECT * FROM T2
                      WHERE "C1" > "T1"."C1")
```

4. Subqueries can be nested to a maximum of 32 levels deep (31 in the case of a view definition or a `WITH` clause). In addition, the following rules apply:
   - If the table specified in the `FROM` clause is a viewed table or query name

After HADB generates the internal derived table to the viewed table or query name, the subquery nesting depth must not exceed 32 (31 in the case of a view definition or a `WITH` clause).

- If the table specified in the `FROM` clause is a recursive query name that references a recursive query that includes the table specified in the `FROM` clause

  After HADB generates the internal derived table that corresponds to the recursive query name once, the subquery nesting depth must not exceed 32 (31 in the case of a view definition or a `WITH` clause).

- If an archivable multi-chunk table is specified in the `FROM` clause

  The archivable multi-chunk table is equivalently exchanged into an internal derived table. If the subquery nesting depth becomes 33 or more after equivalent exchange into an internal derived table, the SQL statement results in an error. For details about equivalent exchange of archivable multi-chunk tables, see *Equivalent exchange of SQL statements that search archivable multi-chunk tables* in the *HADB Application Development Guide*.

- If the derived table derived by a table value constructor is specified in the `FROM` clause

  The subquery nesting depth increases by 1.

The following shows examples of counting the subquery nesting depth.

Example 1:

In the following example, the `SELECT` statement includes subqueries that are nested to a depth of eight.

```
SELECT * FROM "TT" WHERE EXISTS(
    SELECT * FROM "T0" WHERE EXISTS(
    SELECT * FROM "T1" WHERE EXISTS(     <= 1st nest
    SELECT * FROM "T2" WHERE EXISTS(     <= 2nd nest
    SELECT * FROM "T3" WHERE EXISTS(     <= 3rd nest
    SELECT * FROM "T4" WHERE EXISTS(     <= 4th nest
    SELECT * FROM "T5" WHERE EXISTS(     <= 5th nest
    SELECT * FROM "T6" WHERE EXISTS(     <= 6th nest
    SELECT * FROM "T7" WHERE EXISTS(     <= 7th nest
    SELECT * FROM "T8"                   <= 8th nest
    )))))))))
```

Example 2:

In the following example, the `CREATE VIEW` statement includes subqueries that are nested to a depth of seven.

```
CREATE VIEW "V1" AS SELECT * FROM "TT" WHERE EXISTS(
    SELECT * FROM "T0" WHERE EXISTS(
    SELECT * FROM "T1" WHERE EXISTS(     <= 1st nest
    SELECT * FROM "T2" WHERE EXISTS(     <= 2nd nest
    SELECT * FROM "T3" WHERE EXISTS(     <= 3rd nest
    SELECT * FROM "T4" WHERE EXISTS(     <= 4th nest
    SELECT * FROM "T5" WHERE EXISTS(     <= 5th nest
    SELECT * FROM "T6" WHERE EXISTS(     <= 6th nest
    SELECT * FROM "T7"                   <= 7th nest
    ))))))))
```

When the following `SELECT` statement is run, an internal derived table is generated. As a result, the subquery nesting depth becomes eight.

```
SELECT * FROM "V1"
```

Example 3:

When the following `SELECT` statement is run, an internal derived table is generated. As a result, the subquery nesting depth becomes eight.

```
WITH "Q1" AS (SELECT * FROM "TT" WHERE EXISTS(
    SELECT * FROM "T0" WHERE EXISTS(
    SELECT * FROM "T1" WHERE EXISTS(     <= 1st nest
    SELECT * FROM "T2" WHERE EXISTS(     <= 2nd nest
    SELECT * FROM "T3" WHERE EXISTS(     <= 3rd nest
    SELECT * FROM "T4" WHERE EXISTS(     <= 4th nest
    SELECT * FROM "T5" WHERE EXISTS(     <= 5th nest
    SELECT * FROM "T6" WHERE EXISTS(     <= 6th nest
    SELECT * FROM "T7"                   <= 7th nest
    )))))))))
SELECT * FROM "Q1"                  <= 8th nest produced because an internal der
ived table was generated
```

Example 4:

In the following SELECT statement, recursive query name Q1 is specified in the FROM clause. When the following SELECT statement is run, an internal derived table is generated. As a result, the subquery nesting depth becomes eight.

SQL statement to be run

```
WITH "Q1" AS (SELECT "C1" FROM "TT"
              UNION ALL
              SELECT "C1"+1 FROM "Q1" WHERE "C1"+1 < 5)
    SELECT * FROM "TT" WHERE EXISTS(
    SELECT * FROM "T0" WHERE EXISTS(
    SELECT * FROM "T1" WHERE EXISTS(
    SELECT * FROM "T2" WHERE EXISTS(
    SELECT * FROM "T3" WHERE EXISTS(
    SELECT * FROM "T4" WHERE EXISTS(
    SELECT * FROM "T5" WHERE EXISTS(
    SELECT * FROM "Q1")))))))
```

SQL statement after an internal derived table is generated

```
SELECT * FROM "TT" WHERE EXISTS(
SELECT * FROM "T0" WHERE EXISTS(
SELECT * FROM "T1" WHERE EXISTS(     <= 1st nest
SELECT * FROM "T2" WHERE EXISTS(     <= 2nd nest
SELECT * FROM "T3" WHERE EXISTS(     <= 3rd nest
SELECT * FROM "T4" WHERE EXISTS(     <= 4th nest
SELECT * FROM "T5" WHERE EXISTS(     <= 5th nest
SELECT * FROM                        <= 6th nest
              (SELECT "C1" FROM "TT"                       <= 7th nest
               UNION ALL
               SELECT "C1"+1 FROM
                                  (SELECT "C1" FROM "TT"   <= 8th nest
                                   UNION ALL
                                   SELECT "C1"+1 FROM "Q1" WHERE "C1"+1 < 5)"Q1"
               WHERE "C1"+1 < 5)"Q1")))))))
```

Example 5:

In the following example, the SELECT statement includes subqueries that are nested to a depth of seven. Also, because a table derived by a table value constructor is specified in the FROM clause, the subquery nesting depth is incremented by one. Therefore, in total, the nesting depth is assumed to be eight.

```
SELECT * FROM "TT" WHERE EXISTS(
    SELECT * FROM "T0" WHERE EXISTS(
    SELECT * FROM "T1" WHERE EXISTS(      <= 1st nest
    SELECT * FROM "T2" WHERE EXISTS(      <= 2nd nest
    SELECT * FROM "T3" WHERE EXISTS(      <= 3rd nest
    SELECT * FROM "T4" WHERE EXISTS(      <= 4th nest
    SELECT * FROM "T5" WHERE EXISTS(      <= 5th nest
    SELECT * FROM "T6" WHERE EXISTS(      <= 6th nest
    SELECT * FROM (VALUES (1,2,3)) "T7"   <= 7th and 8th nests
    ))))))))
```

5. Subqueries are not permitted in set functions.

6. Subqueries are not permitted in window functions.
7. Subqueries are not permitted in the grouping specification of a GROUP BY clause.
8. Subqueries are not permitted in the ON search condition of a joined table with FULL OUTER JOIN specified as the joined table mode.
9. You cannot specify a column that makes an external reference to a table reference in a joined table with FULL OUTER JOIN specified as the joined table mode.

   Example: The underlined portion indicates an incorrect external reference column.

```
SELECT * FROM ("T1" LEFT OUTER JOIN "T2" ON "T1"."C1"="T2"."C1")
                    FULL OUTER JOIN "T3" ON "T1"."C2"="T3"."C2"
    WHERE "T1"."C3">(SELECT MAX(C3) FROM "T4"
                        WHERE "C1"="T1"."C1"
                          AND "C2"="T3"."C2")
```

(b) Rules for scalar subqueries

1. A scalar subquery cannot return more than one column.
2. A scalar subquery cannot return more than one row. An SQL error results if there is more than one row in the results.
3. If the number of rows in the result of a scalar subquery is zero, the result is the null value.
4. The NOT NULL constraint does not apply to the result of a scalar subquery (the null value is allowed).

(c) Rules for table subqueries

1. The maximum number of columns permitted in the results of a table subquery are as follows.
   - If the table subquery specifies a derived table: 1,000
   - If the table subquery is specified in an IN predicate or on the right side of a quantified predicate: 1
   - If the table subquery is specified in an EXISTS predicate: 1,000

(5) Examples

Example 1

This example retrieves the names (NAME) and salary (SAL) of the employees who earn the highest salary.

```
SELECT "NAME","SAL"
    FROM "SALARYLIST"
       WHERE "SAL"=(SELECT MAX("SAL") FROM "SALARYLIST")
```

The underlined portion indicates the subquery.

Example 2

This example finds the sections (SCODE) in which the average salary is greater than the average salary for all employees.

```
SELECT "SCODE",AVG("SAL")
    FROM "SALARYLIST"
      GROUP BY "SCODE"
      HAVING AVG("SAL")>(SELECT AVG("SAL") FROM "SALARYLIST")
```

The underlined portion indicates the subquery.

Example 3

This example retrieves 100 rows from the sales history table (SALESLIST), and then calculate the total quantity purchased (PUR-NUM) for each product code (PUR-CODE) in those results.

```
SELECT "PUR-CODE",SUM("PUR-NUM")
    FROM (SELECT * FROM "SALESLIST" LIMIT 100) "SALESLIST"
        GROUP BY "PUR-CODE"
```

The underlined portion indicates the subquery.

(6) Notes

1. When a subquery is specified, a work table might be created. If the size of the work table DB area where the work table is to be created has not been estimated correctly, performance might be degraded. For details about estimating the size of the work table DB area, see the *HADB Setup and Operation Guide*. For details about work tables, see *Considerations when executing an SQL statement that creates work tables* in the *HADB Application Development Guide*.
2. If hash execution is used during subquery processing, a hash table area of an appropriate size is required. The size of the hash table area is specified in the `adb_sql_exe_hashtbl_area_size` operand in the server definition or client definition. If hash execution is used as the method for processing the subquery, a hash filter area to store hash filters is also required. The size of the hash filter area is specified by using the `adb_sql_exe_hashflt_area_size` operand in the server definition or client definition. For details about the method for processing the subquery, see *Subquery processing methods* in the *HADB Application Development Guide*.

7.4 Table expression

This section describes table expressions.

7.4.1 Specification format and rules for table expressions

The FROM clause, WHERE clause, GROUP BY clause, and HAVING clause are referred to collectively as *table expressions*. A table expression is specified within a query specification.

(1) Specification format

```
table-expression ::= FROM-clause
                    [WHERE-clause]
                    [GROUP-BY-clause]
                    [HAVING-clause]
```

(2) Explanation of specification format

FROM-clause:

The FROM clause specifies the tables from which results are to be retrieved. For details about the FROM clause, see 7.5 FROM clause.

WHERE-clause:

The WHERE clause specifies the search conditions. For details about the WHERE clause, see 7.6 WHERE clause.

GROUP-BY-clause:

Specify the GROUP BY clause when you want to aggregate the retrieval data into groups. For details about the GROUP BY clause, see 7.7 GROUP BY clause.

HAVING-clause:

The HAVING clause specifies criteria for the data aggregated of the groups created by the GROUP BY clause. For details about the HAVING clause, see 7.8 HAVING clause.

(3) Rules

1. Any column in the results of a table expression can be referenced as a column specification.
2. If there is no WHERE clause, GROUP BY clause, or HAVING clause, the result of the table expression will be determined using only the FROM clause. Otherwise, the results of each clause you specify are applied to the clause specified immediately after it. The result of the table expression will be the result of the last specified clause.

   For example, consider what happens when the SELECT statement shown below is executed.

   Example:

   Using the data in the sales history table (SALESLIST), this example determines the sum of the quantities purchased on or after September 3, 2011 by product code (PUR-CODE). Furthermore, it retrieves only the product codes for which the sum of the quantities purchased is 20 or fewer.

```
SELECT "PUR-CODE",SUM("PUR-NUM")
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2011-09-03'
        GROUP BY "PUR-CODE"
        HAVING SUM("PUR-NUM")<=20
```

Explanation

The underlined portion indicates the table expression. When the SELECT statement above is executed, the result of the table expression is determined by the following steps.

1. The result of the FROM clause is applied to the WHERE clause. In this step, data from the SALESLIST table where the PUR-DATE column is September 3, 2011 or later is extracted (data from both the PUR-CODE and PUR-NUM columns).

2. The results extracted in step 1 are grouped using the GROUP BY clause. In this case, the results are aggregated by PUR-CODE.

3. The results aggregated in step 2 are filtered using the HAVING clause. In this case, only data where the sum of the PUR-NUM column values does not exceed 20 are selected. This final set of data becomes the result of the table expression.

(4) Example

An example of a table expression is given below.

Example

From the sales history table (SALESLIST), this example retrieves the customer ID (USERID), product code (PUR-CODE), and date of purchase (PUR-DATE) for customers who purchased product code P002 on or after September 6, 2011.

```
SELECT "USERID","PUR-CODE","PUR-DATE"
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2011-09-06'
        AND "PUR-CODE"='P002'
```

The underlined portion indicates the table expression.

7.5 FROM clause

This section describes the FROM clause.

7.5.1 Specification format and rules for FROM clauses

The FROM clause specifies the tables from which be retrieved data.

(1) Specification format

```
FROM-clause ::= FROM table-reference[, table-reference]...
```

(2) Explanation of specification format

table-reference:

Specifies the tables from which to retrieved data in the form of a table reference. For details about table references, see 7.11 Table reference.

When you perform a query across multiple table references (a query containing multiple table names, query names, derived tables, or table function derived tables in the FROM clause), it is called a *table join*.

Also, when you perform a join by using a comma-separated list of multiple table references, it is called a *comma join*.

(3) Rules

1. A total of 64 table names, query names, derived tables, and table function derived tables can be specified in all table references in a FROM clause. The table specification counts are computed as follows.
   - When a table name is specified in a table reference: 1
   - When a derived table is specified in a table reference: 1
   - When a joined table is specified in a table reference: the total number of table names and derived tables specified in the joined table
   - When a query is specified in a WITH clause: 1
   - When a table function derived table is specified in a table reference: 1

   The following shows an example of computing the table specification counts.

   Example:

```
WITH "Q1" AS (SELECT * FROM "T6","T7")
SELECT * FROM "T1",                                          ...[a]
    "T2" LEFT OUTER JOIN "T3" ON "T2"."C1"="T3"."C1",        ...[b]
    (SELECT * FROM "T4","T5") "W1",                          ...[c]
    "Q1",                                                    ...[d]
    TABLE(ADB_CSVREAD(MULTISET['/dir/file.csv.gz'],
                     'COMPRESSION_FORMAT=GZIP;'))
    AS W2 (C1 INT)                                           ...[e]
```

   [Explanation]

   a. Table name T1 is specified. Here, therefore, the number of table names is 1.

   b. A joined table is specified, and two table names (T2 and T3) are specified in the joined table. Here, therefore, the number of table names is 2.

c. A derived table (W1) is specified. Here, therefore, the number of derived tables is 1.

d. Query name Q1 that is specified in the WITH clause is specified. Here, therefore, the number of query names is 1.

e. A table function derived table is specified. Here, therefore, the number of derived tables is 1.

As described earlier, in the preceding example, the number of table names specified in all table references is six in total.

2. The column descriptors used for the results of the FROM clause will be the same as the column descriptors from the tables specified in the FROM clause. In addition, the order of the columns in the result of the FROM clause will be the order of the columns in the tables specified in the FROM clause. For example, consider what happens when the SELECT statement shown below is executed.

   Example:

   ```
   SELECT * FROM "T1","T2"
   ```

   Assume that columns C1 and C2 are defined in table T1, and columns C3 and C4 are defined in table T2. In this case, the order of columns in the results of the FROM clause is as follows: C1 → C2 → C3 → C4

   > **Note**
   >
   > A column descriptor contains attribute information for a column. It consists of the column's name, data type, data length, column ID (numbered from the first column), and whether it contains null values.

(4) Example

The following example illustrates a FROM clause.

Example

From the sales history table (SALESLIST), this example retrieves the customer ID (USERID), product code (PUR-CODE), and date of purchase (PUR-DATE) for customers who purchased product code P002 on or after September 6, 2011.

```
SELECT "USERID","PUR-CODE","PUR-DATE"
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2011-09-06'
        AND "PUR-CODE"='P002'
```

The underlined portion indicates the FROM clause.

(5) Notes

If you specify multiple table references in the FROM clause in the situations listed below, a work table might be created. If the size of the work table DB area where the work table is created has not been estimated correctly, it might result in performance degradation. For details about estimating the size of the work table DB area, see the *HADB Setup and Operation Guide*.

- When multiple table references are specified in a single FROM clause
- When the same viewed table name is specified in more than one place in a single SQL statement
- When the query name specified in a WITH clause is specified in more than one place in a single SQL statement

For details about work tables, see *Considerations when executing an SQL statement that creates work tables* in the *HADB Application Development Guide*.

7.6 WHERE clause

This section describes the `WHERE` clause.

7.6.1 Specification format for WHERE clauses

The `WHERE` clause specifies search conditions.

(1) Specification format

```
WHERE-clause ::= WHERE search-condition
```

(2) Explanation of specification format

search-condition:

For details about search conditions, see 7.18 Search conditions.

(3) Example

The following example illustrates the `WHERE` clause.

Example

From the sales history table (`SALESLIST`), this example retrieves the customer ID (`USERID`), product code (`PUR-CODE`), and date of purchase (`PUR-DATE`) for customers who purchased product code `P002` on or after September 6, 2011.

```
SELECT "USERID","PUR-CODE","PUR-DATE"
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2011-09-06'
        AND "PUR-CODE"='P002'
```

The underlined portion indicates the `WHERE` clause.

7.7 GROUP BY clause

This section describes the `GROUP BY` clause.

7.7.1 Specification format and rules for GROUP BY clauses

Specify the `GROUP BY` clause when you want to aggregate the retrieval data into groups.

(1) Specification format

```
GROUP-BY-clause ::= GROUP BY [grouping-method-specification] grouping-specification[,
 grouping-specification]...

  grouping-method-specification ::= /*>> WITHOUT GLOBAL HASH GROUPING <<*/
  grouping-specification ::= value-expression [[AS] column-name]
```

(2) Explanation of specification format

- *grouping-method-specification*

```
grouping-method-specification ::= /*>> WITHOUT GLOBAL HASH GROUPING <<*/
```

When *grouping-method-specification* is specified, global hash grouping is not used as the processing method for the grouping.

Normally there is no need to specify this.

For details about grouping methods, see *Grouping Methods* in the *HADB Application Development Guide*.

Note that the character string enclosed in `/*>>` and `<<*/` is not a comment. An error results if you specify something other than a grouping method specification.

- *grouping-specification*

```
grouping-specification ::= value-expression [[AS] column-name]
```

Specifies a group by which the retrieval data is to be aggregated, in the form of a value expression. For details about value expressions, see 7.20 Value expression.

`GROUP BY` clauses are illustrated in the following examples. Example 1:

This example aggregates the retrieval data by product code (`PUR-CODE`)

```
GROUP BY "PUR-CODE"
```

Example 2: This example aggregates the retrieval data by month

```
GROUP BY EXTRACT(MONTH FROM "SALE-DAY") AS "GMONTH"
```

The `SALE-DAY` column stores the sale date of the product in `DATE` type format. The scalar function `EXTRACT` is used to extract the month part of the `SALE-DAY` column.

`[AS]` *column-name*:

The column name specified here becomes the grouping column name.

Example:

```
GROUP BY SUBSTR("C1",5,2) AS "GC1"
```

In the preceding example, `GC1` becomes the grouping column name.

> **Note**
>
> A column derived from the result of a `GROUP BY` clause is called a *grouping column*. The column name assigned to the grouping column is called a *grouping column name*.

Example 1:

```
GROUP BY "C1"
```

In the preceding example, the underlined item becomes a grouping column with a grouping column name of `C1`.

Example 2:

```
GROUP BY "T1"."C1"
```

In the preceding example, the underlined item becomes a grouping column with a grouping column name of `C1`.

Example 3:

```
GROUP BY "C1" AS "GC1"
```

In the preceding example, the underlined item becomes a grouping column with a grouping column name of `GC1`.

Example 4:

```
GROUP BY SUBSTR("C1",5,2) AS "GC1"
```

In the preceding example, the underlined item becomes a grouping column with a grouping column name of `GC1`.

Example 5:

```
GROUP BY SUBSTR("C1",5,2)
```

In the preceding example, the underlined item becomes a grouping column. No name is assigned to the grouping column.

Example 6:

```
GROUP BY "C1","C2"
```

In the preceding example, the underlined items become grouping columns. Two grouping columns are created. These grouping columns are named `C1` and `C2`.

Example 7:

```
GROUP BY 1 AS "GC1"
```

In the preceding example, the underlined item becomes a grouping column with a grouping column name of `GC1`.

Example 8:

```
GROUP BY 1
```

In the preceding example, the underlined item becomes a grouping column. No name is assigned to the grouping column.

(3) Rules

1. The maximum number of grouping columns is 64.
2. Set functions are not permitted in *value-expression*.
3. Subqueries are not permitted in *value-expression*.

4. Dynamic parameters are not permitted in *value-expression*.
5. The column name of a column specification included alone in another grouping column cannot be specified in AS *column-name*.

   Example that generates an error:

   ```
   GROUP BY "C1","C3" AS "C1"
   ```

   ```
   GROUP BY "C1" AS "C2","C3" AS "C1"
   ```

6. Do not specify a character string in the EXP*nnnn*_NO_NAME format as the column name in AS *column-name* in a grouping specification. Such a column name might duplicate a derived column name that is automatically set by HADB. In this format, *nnnn* is an unsigned integer in the range from 0000 to 9999.
7. Each column name specified in AS *column-name* must be unique.

   Example that generates an error:

   ```
   GROUP BY "C1"+1 AS "GC1","C2"+1 AS "GC1"
   ```

8. The column name specified in AS *column-name* cannot be referenced from a subquery in the selection expression or from a subquery in a HAVING clause.

   Example that generates an error:

   ```
   SELECT "GC1",MAX("C2") FROM "T1"
       GROUP BY SUBSTR("C1",5,2) AS "GC1"
       HAVING EXISTS(SELECT * FROM "T2" WHERE "T2"."C1"="GC1")
   ```

9. When a GROUP BY clause is specified, only the following items can be specified in the selection expression:

   1. Grouping column name
   2. Set function
   3. Value specification
   4. Scalar subquery
   5. Value expression specifying at least one of the preceding items
   6. Same value expression that is included in a grouping specification (value expression including a column specification)

   Example of a correct specification:

   ```
   SELECT "C1","C2",COUNT(*)  ← Selection expression contains grouping column names a
   nd set functions
       FROM "T1"
           GROUP BY "C1","C2"
   ```

   Example that generates an error:

   ```
   SELECT "C1","C2",COUNT(*)  ← Selection expression includes column C2, which is not
    a grouping column name
       FROM "T1"
           GROUP BY "C1"
   ```

10. The column specifications in the GROUP BY clause must meet the following conditions:

    - They must specify columns from tables specified in the FROM clause of the table expression in which the GROUP BY clause is specified
    - The column names must be uniquely identified

    For example, consider what happens when the SELECT statement shown below is executed.

    Example:

```
SELECT "SALESLIST"."USERID",SUM("PUR-NUM")
    FROM "SALESLIST","USERSLIST"
        WHERE "PUR-CODE"='P002'
        AND "SALESLIST"."USERID"="USERSLIST"."USERID"
        GROUP BY "SALESLIST"."USERID"
```

The sales history table (`SALESLIST`) and the customer table (`USERSLIST`) both have `USERID` columns with the same column name. In this case, if you want to specify the `USERID` column in the `GROUP BY` clause, you must do so in a way that uniquely specifies which `USERID` column is intended. Therefore, you cannot specify `GROUP BY "USERID"`. Instead, specify the column qualified with a table name, as in `GROUP BY "SALESLIST"."USERID"`.

11. The grouping column referenced by the column specified in the value expression in a selection expression or in the value expression in the `HAVING` clause is determined by the following priority. A smaller number indicates a higher priority level. (1 is the highest.)

1. If the column name is the same as a grouping column name

Example of a correct specification:

```
SELECT "C1"+"C2" FROM "T1" GROUP BY "C1"+"C2" AS "C1","C2"
```

In the preceding example, `C1` in the selection expression is the same as the name of grouping column `C1`. Therefore, grouping column `"C1"+"C2" AS "C1"` is referenced.

`C2` in the selection expression is the same as the name of grouping column `C2`. Therefore, grouping column `C2` is referenced.

2. If there is a grouping column that has a single column specification, or if there is a grouping column whose value expressions are specified in the same format

Example of a correct specification (1):

```
SELECT "C1"+"C2" FROM "T1" GROUP BY "C1"+"C2" AS "C3"
```

In the preceding example, `C1` and `C2` in the selection expression reference grouping column `"C1"+"C2" AS "C3"` because the value expressions of grouping column `"C1"+"C2" AS "C3"` have the same format.

Example of a correct specification (2):

```
SELECT "GC1","C1" FROM "T1" GROUP BY "C1" AS "GC1"
```

In the preceding example, `C1` in the selection expression references grouping column `"C1" AS "GC1"`, which has the same value expression format as `C1`.

Also, because `GC1` in the selection expression is the same as a grouping column name, the SQL statement in the preceding example meets condition 1 (the column name is the same as a grouping column name) shown earlier. Therefore, `GC1` also references the grouping column `"C1" AS "GC1"`.

3. Specification order of grouping columns (former item has higher priority)

Example of a correct specification:

```
SELECT "C1"+"C2" FROM "T1" GROUP BY "C1"+"C2" AS "C3","C1"+"C2"
```

In the preceding example, the value expressions of grouping columns `"C1"+"C2" AS "C3"` and `"C1"+"C2"` have the same format. In this case, because the former item has the higher priority, `C1` and `C2` in the selection expression reference grouping column `"C1"+"C2" AS "C3"`.

■ Specification example that generates an error

```
SELECT "C1"+"C2" FROM "T1" GROUP BY "C1"+"C2" AS "C1"
```

In the preceding example, `C1` in the selection expression references the grouping column that corresponds to the underlined grouping column name (`C1`). However, `C2` in the selection expression does not have a grouping column name with the same name. Therefore, the preceding SQL statement will result in an error.

12. We recommend that you do not specify the same name that was specified in the value expression of a grouping specification as the grouping column name in the same grouping specification. If the column name specified in the value expression of a grouping specification is the same as a grouping column name, an unintended grouping column might be referenced.

    Example:

    ```
    SELECT "C1"+1 FROM "T1" GROUP BY "C1"+1 AS "C1"
    ```

    In the preceding example, `C1` is specified in the value expression of a grouping specification, and `C1` is also specified as the grouping column name. In this case, `C1` in the selection expression references grouping column `"C1"+1 AS "C1"`.

13. A value expression that includes a column specification provided as a grouping column cannot be referenced from a subquery in a selection expression or a subquery in the `HAVING` clause.

    Example that generates an error:

    ```
    SELECT "T1"."C1"+"T1"."C2" FROM "T1"
        GROUP BY "T1"."C1"+"T1"."C2"
        HAVING (SELECT "T2"."C1" FROM "T2"
                   WHERE "T2"."C1" > "T1"."C1"+"T1"."C2") > 0
    ```

    `"T1"."C1"+"T1"."C2"` specified in a subquery in the `HAVING` clause cannot reference grouping column `"T1"."C1"+"T1"."C2"`. Therefore, the preceding SQL statement will result in an error.

14. Grouping columns are restricted to the following data types:

    - Character string data
    - Numeric data
    - Datetime data
    - Binary data

15. The results of the `WHERE` clause are grouped using the `GROUP BY` clause. For details about the order in which the results of the table expression are derived, see (3) Rules in 7.4.1 Specification format and rules for table expressions.

16. Let set *T* denote the results of the preceding `WHERE` clause (or the preceding `FROM` clause if no `WHERE` clause is specified).

    - When the `GROUP BY` clause is specified, set *T* will be divided into multiple groups (where each group is a set with identical values in the grouping column). Because duplicate rows are then eliminated from each group, the number of groups created will be the same as the number of rows in the results of the `GROUP BY` clause.

      Note that when there are `NULL` values in the grouping column, all the null values are treated alike and placed in a single group.

    - If the `GROUP BY` clause is omitted, but a `HAVING` clause or set function is specified in the query specification, it creates a single group consisting of set *T* in its entirety.

(4) Examples

Example 1

Using the data in the sales history table (`SALESLIST`), this example determines the number of purchases for each customer.

```
SELECT "USERID",COUNT(*) AS "COUNT"
    FROM "SALESLIST"
        GROUP BY "USERID"
```

The underlined portion indicates the `GROUP BY` clause.

Example of execution results

USERID	COUNT
U00212	4
U00358	5
U00555	1

Example 2

Using the data in the sales history table (SALESLIST), this example determines the sum and average of the quantities purchased on or after September 3, 2011 by product code (PUR-CODE).

```
SELECT "PUR-CODE",SUM("PUR-NUM") AS "SUM",AVG("PUR-NUM") AS "AVG"
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2011-09-03'
        GROUP BY "PUR-CODE"
```

The underlined portion indicates the GROUP BY clause.

Example of execution results

PUR-CODE	SUM	AVG
P001	16	5
P002	37	6
P003	17	5

Example 3

Using the data in the sales history table (SALESLIST) and the customer table (USERSLIST), this example determines, for each customer, the sum of quantities purchased (PUR-NUM) on or after September 4, 2011 for product code P002.

```
SELECT "NAME",SUM("PUR-NUM") AS "SUM"
    FROM "SALESLIST","USERSLIST"
        WHERE "PUR-DATE">=DATE'2011-09-04'
        AND "PUR-CODE"='P002'
        AND "SALESLIST"."USERID"="USERSLIST"."USERID"
        GROUP BY "NAME"
```

The underlined portion indicates the GROUP BY clause.

Example of execution results

NAME	SUM
Maria Gomez	12
Nancy White	9
Mike Johnson	5

Example 4

Using the data in the employee table (EMPLIST), this example organizes the employees' ages into 10-year groups and determines the number of employees in each group. Employees age 60 and over are grouped with the 60-year-old group.

```
SELECT "GAGE",COUNT(*) AS "COUNT"
    FROM "EMPLIST"
        GROUP BY CASE WHEN "AGE">=60 THEN 60
                      ELSE TRUNC("AGE",-1)
                 END AS "GAGE"
```

The underlined portion indicates the GROUP BY clause.

Example of execution results

GAGE	COUNT
20	212
30	375
40	186
50	113
60	35

Example 5

Using the data in the sales history table (`SALESLIST`), this example determines the sales amounts from 2013 on a monthly basis.

- The `SALE-DAY` column stores the sale date of the product in `DATE` type format.
- The `AMOUNT` column stores the price at which the customer purchased the product.

```
SELECT "GMONTH",SUM("AMOUNT") AS "SUM"
    FROM "SALESLIST"
        WHERE EXTRACT(YEAR FROM "SALE-DAY")=2013
        GROUP BY EXTRACT(MONTH FROM "SALE-DAY") AS "GMONTH"
```

The underlined portion indicates the `GROUP BY` clause.

Example of execution results

GMONTH	SUM
1	302245
2	258764
3	378847
4	402213
5	437022
⋮	⋮

(5) Notes

1. If the `GROUP BY` clause is specified, a work table might be created. If the size of the work table DB area where the work table is to be created has not been estimated correctly, performance might be degraded. For details about estimating the size of the work table DB area, see the *HADB Setup and Operation Guide*. For details about work tables, see *Considerations when executing an SQL statement that creates work tables* in the *HADB Application Development Guide*.
2. If global hash grouping is used as the grouping method, a hash table area of an appropriate size is required. The size of the hash table area is specified in the operand `adb_sql_exe_hashtbl_area_size` in the server definition or client definition.

   In addition, if local hash grouping is used as the grouping method, a hash group area of an appropriate size is required. The size of the hash group area is specified in the operand `adb_sql_exe_hashgrp_area_size` in the server definition or client definition.

   For details about grouping methods, see the *HADB Application Development Guide*.

7.8 HAVING clause

This section describes the HAVING clause.

7.8.1 Specification format and rules for HAVING clauses

The HAVING clause specifies the selection criteria for the data aggregation to be performed by the preceding GROUP BY clause.

If no GROUP BY clause was specified, the selection criteria is applied to the results of the preceding WHERE or FROM clause, which is treated as the group.

(1) Specification format

```
HAVING-clause ::= HAVING search-condition
```

(2) Explanation of specification format

search-condition:

For details about search conditions, see 7.18 Search conditions.

(3) Rules

1. Each column specification in *search-condition* must meet one of the following conditions:
   - It specifies a grouping column name.
     Example 1:
     ```
     SELECT COUNT("C2") FROM "T1" GROUP BY "C1" HAVING "C1">100
     ```
     For the underlined portions, the same column name (grouping column name) must be specified.
     Example 2:
     ```
     SELECT "GC1",COUNT(*) FROM "MEMBERS"
         GROUP BY CASE WHEN "AGE">=90 THEN 90 ELSE TRUNC("AGE",-1) END AS "GC1"
         HAVING "GC1">=20
     ```
     For the underlined portions, the same column name (grouping column name) must be specified.
   - It specifies the same value expression as the value expression included in a grouping specification (value expression that includes a column specification).
     Example:
     ```
     SELECT "C1"+"C2" FROM "T1" GROUP BY "C1"+"C2" HAVING "C1"+"C2" > 100
     ```
     The same value expression (value expression including a column specification) must be specified for the underlined items.
   - It specifies the argument to a set function.
     Example:
     ```
     SELECT COUNT("C2") FROM "T1" HAVING MAX("C1")>100
     ```
     The underlined portion indicates the argument to the set function.

- It specifies an external reference column.

  Example:

  ```
  SELECT * FROM "T1"
      WHERE EXISTS(SELECT * FROM "T2" HAVING MAX("C1")<"T1"."C1")
  ```

  The underlined portion indicates the external reference column.

2. Each column specification that is contained in any subqueries of *search-condition* and that references a table reference column specified in the preceding FROM clause must meet the following conditions:

   - It specifies the column specification included alone in the preceding GROUP BY clause (regardless of whether AS *column-name* is specified in *grouping-specification*).

     Example:

     ```
     SELECT "C1" FROM "T1"
         GROUP BY "C1"
         HAVING EXISTS(SELECT * FROM "T2" WHERE "C1"<"T1"."C1")
     ```

     For the underlined portion, the same column specification that is included alone in the grouping column in the preceding GROUP BY clause must be specified.

   - It specifies the argument to a set function.

     Example:

     ```
     SELECT COUNT("C1") FROM "T1"
         HAVING EXISTS(SELECT * FROM "T2" WHERE "C1"<MAX("T1"."C1"))
     ```

     The underlined portion indicates the set function in which the external reference column is specified as an argument.

3. The search conditions specified in the HAVING clause are applied to the results of the GROUP BY clause. For details about the order in which the results of the table expression are derived, see (3) Rules in 7.4.1 Specification format and rules for table expressions.

(4) Examples

Example 1

Using the data in the sales history table (SALESLIST), this example determines the sum and average of the quantities purchased on or after September 3, 2011 by product code (PUR-CODE).

Furthermore, retrieve only the product code groups for which the sum of the quantities purchased is 20 or fewer.

```
SELECT "PUR-CODE",SUM("PUR-NUM") AS "SUM",AVG("PUR-NUM") AS "AVG"
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2011-09-03'
        GROUP BY "PUR-CODE"
        HAVING SUM("PUR-NUM")<=20
```

The underlined portion indicates the HAVING clause.

Example of execution results

PUR-CODE	SUM	AVG
P001	16	5
P003	17	5

Example 2

This example selects the departments (SCODE) that have an average member age that is less than the average age of all employees, and obtains the average member age for each of those departments.

```
SELECT "SCODE",AVG("AGE") AS "AVG"
    FROM "EMPLIST"
       GROUP BY "SCODE"
       HAVING AVG("AGE")<(SELECT AVG("AGE") FROM "EMPLIST")
```

The underlined portion indicates the `HAVING` clause.

Example of execution results

SCODE	AVG
S001	28
S003	26

7.9 LIMIT clause

This section describes the LIMIT clause.

7.9.1 Specification format and rules for LIMIT clauses

The LIMIT clause specifies the maximum number of rows that will be retrieved from the results of a query expression or query expression body.

A LIMIT clause can be specified in the following locations:

- The outermost query specification or query expression body in a SELECT statement
- Derived table#

  Note that the LIMIT clause cannot be specified for a derived table in a recursive query.
- A scalar subquery

  Note that the LIMIT clause cannot be specified for a scalar subquery in a recursive query.
- WITH list element in a WITH clause

  Note that the LIMIT clause cannot be specified for the WITH list element that corresponds to a recursive query.
- A CREATE VIEW statement

#

Only a table derived by a table subquery applies. The derived tables in 7.9.1 Specification format and rules for LIMIT clauses refer to tables derived by table subqueries.

(1) Specification format

■ Specifying a LIMIT clause in the outermost query specification or query expression body in a SELECT statement

```
LIMIT-clause ::= LIMIT [offset,]row-count

  offset ::= value-specification
  row-count ::= value-specification
```

■ Specifying a LIMIT clause in a derived table, scalar subquery, WITH clause, or CREATE VIEW statement

```
LIMIT-clause ::= LIMIT row-count

  row-count ::= value-specification
```

(2) Explanation of specification format

offset:

Specifies the offset of the first row to return from the retrieval results of the query expression. For example, if you specify LIMIT 10,5 (*offset* is 10, *row-count* is 5), processing skips the first 10 rows of the retrieval results of the query expression and retrieves rows 11 to 15.

The following rules apply:

- The offset can only be specified in a `LIMIT` clause in the outermost query specification or query expression body in a `SELECT` statement. An offset is not allowed in a `LIMIT` clause in a derived table, scalar subquery, `WITH` clause, or `CREATE VIEW` statement.
- The offset is expressed in the form of a value specification. For details about value specifications, see 7.21 Value specification.
- An integer from `0` to `2147483647` (`INTEGER` type data) must be specified for *offset*.
- Specifying `0` for *offset* is equivalent to not having an offset. In this case, the number of rows specified in *row-count* is retrieved starting from the first row of the results of the query expression.
- If *offset* is a dynamic parameter, the assumed data type of the dynamic parameter will be `INTEGER` type.
- You cannot specify the null value for *offset*.

row-count:

Specifies the maximum number of rows that will be retrieved from the results of a query expression or query expression body.

The following rules apply:

- The maximum number of rows is specified in *row-count*, which is expressed in the form of a value specification. For details about value specifications, see 7.21 Value specification.
- An integer from 0 to 2,147,483,647 (`INTEGER` type data) must be specified for *row-count*.
- If *row-count* is `0`, the number of retrieval results will be `0`.
- If *row-count* is a dynamic parameter, the assumed data type of the dynamic parameter will be `INTEGER` type.
- You cannot specify the null value for *row-count*.

(3) Rules

(a) Rules for specifying a LIMIT clause in the outermost query specification or query expression body in a SELECT statement

1. When a `LIMIT` clause is specified, the number of rows in the results of the query expression will be the following:

   MAX{MIN(*number of rows in results of query expression when no LIMIT clause is specified* - *offset*, *row-count*), 0 }

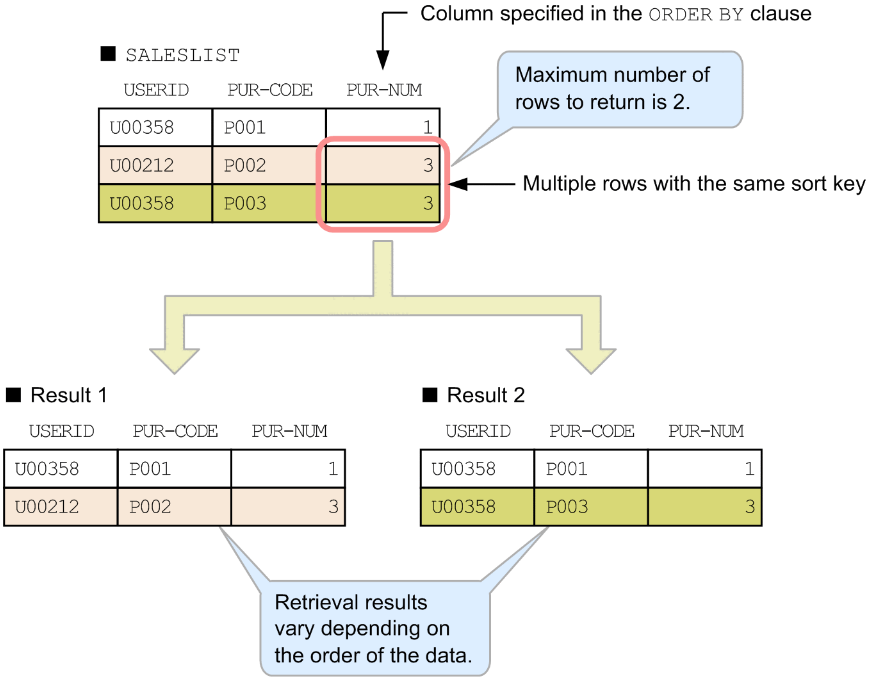
2. If the number of rows in the results of the query expression is greater than the sum of *offset* and *row-count*, the retrieval results will not be uniquely determined in the following cases:

   - When no `ORDER BY` clause is specified
   - When an `ORDER BY` clause is specified, but there is another row with the same sort key value as the last row of the results retrieved by the `LIMIT` clause (see Example 1)
   - When an `ORDER BY` clause is specified, but there is another row with the same sort key value as the last row that was skipped by specifying the offset (see Example 2)

Example 1

This example searches the sales history table (`SALESLIST`) by executing the following `SELECT` statement, setting *row-count* to `2`.

```
SELECT "USERID","PUR-CODE","PUR-NUM"
    FROM "SALESLIST"
       ORDER BY "PUR-NUM" ASC
       LIMIT 2
```



Column specified in the ORDER BY clause

■ SALESLIST

USERID	PUR-CODE	PUR-NUM
U00358	P001	1
U00212	P002	3
U00358	P003	3

Maximum number of rows to return is 2.

Multiple rows with the same sort key

■ Result 1

USERID	PUR-CODE	PUR-NUM
U00358	P001	1
U00212	P002	3

■ Result 2

USERID	PUR-CODE	PUR-NUM
U00358	P001	1
U00358	P003	3

Retrieval results vary depending on the order of the data.

Explanation

The ORDER BY clause arranges the results of the query expression in ascending order, using the value of the PUR-NUM column as the sort key.

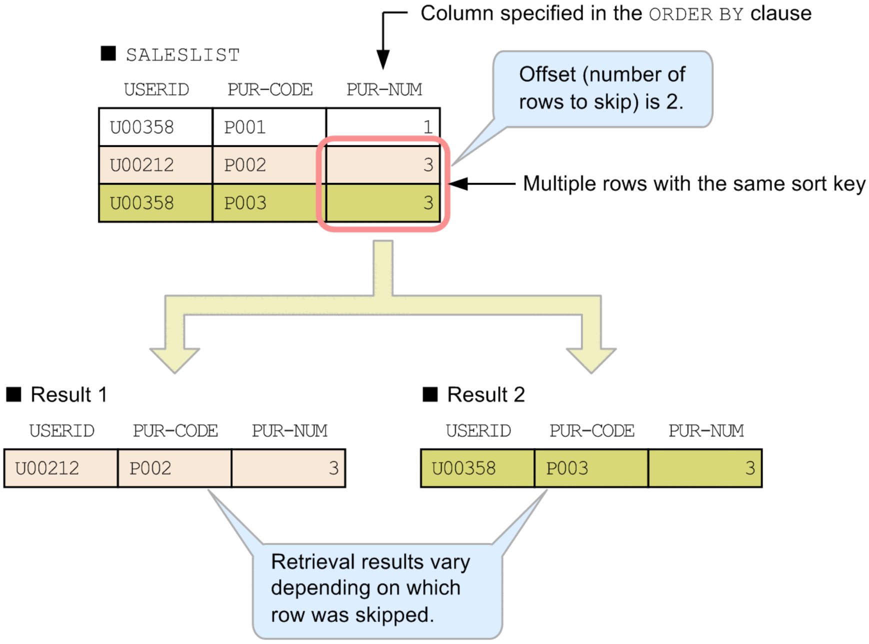
Because of *row-count*, the first two rows are retrieved as the retrieval results.

Because there are two rows with the same sort key (3) as the last row (row 2), the retrieval results are indeterminate.

Example 2

This example searches the sales history table (SALESLIST) by executing the following SELECT statement, setting *offset* to 2, and *row-count* to 1.

```
SELECT "USERID","PUR-CODE","PUR-NUM"
    FROM "SALESLIST"
       ORDER BY "PUR-NUM" ASC
       LIMIT 2,1
```



■ SALESLIST

USERID	PUR-CODE	PUR-NUM
U00358	P001	1
U00212	P002	3
U00358	P003	3

■ Result 1

USERID	PUR-CODE	PUR-NUM
U00212	P002	3

■ Result 2

USERID	PUR-CODE	PUR-NUM
U00358	P003	3

Column specified in the ORDER BY clause
Offset (number of rows to skip) is 2.
Multiple rows with the same sort key
Retrieval results vary depending on which row was skipped.

Explanation

The ORDER BY clause arranges the results of the query expression in ascending order, using the value of the PUR-NUM column as the sort key.

Because of *offset*, the first two rows are skipped.

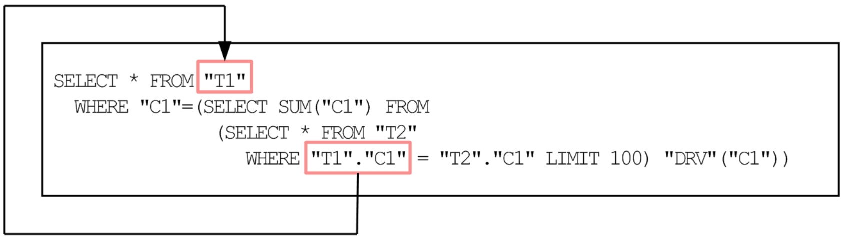
Because there are two rows with the same sort key (3) as the last skipped row (row 2), the retrieval results vary depending on which row is skipped.

(b) Rules for specifying a LIMIT clause in a derived table, scalar subquery, WITH clause, or CREATE VIEW statement

1. A LIMIT clause is not permitted in a derived table of the following type:

   - A derived table that references a table that is outside the derived table in which the LIMIT clause is specified

   Example of an SQL statement that generates an error:



```
SELECT * FROM "T1"
  WHERE "C1"=(SELECT SUM("C1") FROM
               (SELECT * FROM "T2"
                  WHERE "T1"."C1" = "T2"."C1" LIMIT 100) "DRV"("C1"))
```

   In this example, "T1"."C1" references a table that is outside the derived table in which the LIMIT clause is specified (correlation name: DRV). The LIMIT clause is therefore not permitted here.

   For details about derived tables, see 7.11.1 Specification format for table references.

2. When a LIMIT clause is specified, the number of rows in the results of the query expression body will be the following:

   MIN(*number of rows in results of query expression body when no LIMIT clause is specified*, *row-count*)

3. If the number of rows in the results of the query expression body is greater than *row-count*, the retrieval results will not be uniquely determined in the following cases:

   - When the LIMIT clause is specified in a derived table, scalar subquery, or WITH clause (because the ORDER BY clause is not permitted in these cases)

- When retrieving from a viewed table defined by a CREATE VIEW statement in which the LIMIT clause is specified (because the ORDER BY clause is not permitted in a CREATE VIEW statement)

Example:

```
CREATE VIEW "SALESLIST_VIEW" AS SELECT * FROM "SALESLIST" LIMIT 2
SELECT * FROM "SALESLIST_VIEW" ORDER BY "USERID"
```

When you execute the above SELECT statement, the retrieval results are not uniquely determined, as illustrated in the following figure:

■ SALESLIST

USERID	PUR-CODE	PUR-NUM
U00358	P001	1
U00212	P002	8
U00555	P003	5

■ Result 1

USERID	PUR-CODE	PUR-NUM
U00358	P001	1
U00212	P002	8

■ Result 2

USERID	PUR-CODE	PUR-NUM
U00555	P003	5
U00212	P002	8

Only two retrieval results are shown above, but other results might also appear.

4. When a LIMIT clause is specified in a query expression body that references a column in an outer query, the LIMIT clause does not apply to the total number of rows of results from that query expression body. Rather, the LIMIT clause applies to the number of rows of query expression body results for a single value of the outer query column.

Example:

```
SELECT (SELECT "PRODUCTLIST"."PUR-NAME" FROM "PRODUCTLIST"
          WHERE "SALESLIST"."PUR-CODE"="PRODUCTLIST"."PUR-CODE" LIMIT 1)
       ,"SALESLIST"."PUR-NUM"
    FROM "SALESLIST"
```

In the above example, the underlined portion "SALESLIST"."PUR-CODE" references a column in an outer query.

When you execute the above SELECT statement, the retrieval results will be as follows:

■ SALESLIST

PUR-CODE	PUR-NUM
P001	1
P002	8
P003	5

■ Retrieval results

PUR-NAME	PUR-NUM
File	1
Pen	8
Highlighter	5

(4) Examples

Example 1 (`LIMIT` clause in a query specification)

This example searches the table of branch stores (`BRANCHESLIST`) for the ten branches with the highest sales revenues (`SALES`).

```
SELECT "BRANCH-CODE","RGN-CODE","BRANCH-NAME","SALES"
    FROM "BRANCHESLIST"
    ORDER BY "SALES" DESC
    LIMIT 10
```

The underlined portion indicates the `LIMIT` clause.

Example 2 (offset)

This example searches the table of branch stores (`BRANCHESLIST`) for the branches in positions 21 through 30 in terms of sales (`SALES`).

```
SELECT "BRANCH-CODE","RGN-CODE","BRANCH-NAME","SALES"
    FROM "BRANCHESLIST"
    ORDER BY "SALES" DESC
    LIMIT 20,10
```

The underlined portion indicates the `LIMIT` clause.

Example 3 (`LIMIT` clause in a derived table)

This example retrieves 100 rows from the sales history table (`SALESLIST`), and then calculates the total quantity purchased (`PUR-NUM`) for each product code (`PUR-CODE`) in those results.

```
SELECT "PUR-CODE",SUM("PUR-NUM")
    FROM (SELECT * FROM "SALESLIST" LIMIT 100) "SALESLIST"
        GROUP BY "PUR-CODE"
```

The underlined portion indicates the `LIMIT` clause.

The `SELECT` statement above retrieves an arbitrary set of 100 rows from the sales history table (`SALESLIST`), and then determines the results based on them. Because a different set of 100 rows can be retrieved each time it is executed, the `SELECT` statement above can produce different results every time it is executed.

Example 4 (`LIMIT` clause in a derived table)

This example specifies a condition on the date of purchase (`PUR-DATE`) in the sales history table (`SALESLIST`), and then counts the number of rows in the retrieval results. Because the `LIMIT` clause is specified, retrieval stops once the number of rows in the derived table reaches 1,000, and the retrieval results are returned.

```
SELECT COUNT(*)
  FROM (SELECT 1 FROM "SALESLIST"
        WHERE "PUR-DATE" BETWEEN ? AND ? LIMIT 1000) "SALESLIST"("PUR-DATE")
```

The underlined portion indicates the `LIMIT` clause.

By specifying a `LIMIT` clause in a derived table (by fixing the maximum number of rows in a derived table), you are limiting the execution time of the `SELECT` statement. This is useful when you are executing the above `SELECT` statement in order to progressively narrow the search results until you obtain fewer than 1,000 retrieval results. When the execution result is 1,000, it means there are at least 1,000 rows that satisfy the search condition. You can repeatedly execute the `SELECT` statement with different values for the dynamic parameters until you get fewer than 1,000 retrieval results.

Example 5 (`LIMIT` clause in a scalar subquery)

This example searches the sales history table (`SALESLIST`) for the date on which the greatest quantity purchased (`PUR-NUM`) occurred, and returns the corresponding date of purchase (`PUR-DATE`) and product code (`PUR-CODE`).

```
SELECT DISTINCT "PUR-DATE","PUR-CODE"
  FROM "SALESLIST"
    WHERE "PUR-DATE"=(SELECT "PUR-DATE" FROM "SALESLIST"
                        WHERE "PUR-NUM"=(SELECT MAX("PUR-NUM")
                                           FROM "SALESLIST") LIMIT 1)
```

The underlined portion indicates the `LIMIT` clause.

If there is more than one date with the maximum quantity purchased (`PUR-NUM`), the returned date of purchase (`PUR-DATE`) is selected randomly for the retrieval result, which means the `SELECT` statement above can produce different results every time it is executed.

7.10 DEFAULT clause

This section describes the DEFAULT clause.

7.10.1 Specification format and rules for the DEFAULT clause

The DEFAULT clause specifies the default value for a column. The default value for a column is the default value that is stored in the column in any of the circumstances described below.

- When inserting rows with the INSERT statement
  The default value for the column is stored in the following circumstances:
  - DEFAULT is specified in the insertion value
  - DEFAULT VALUES is specified
  - A column name is not specified for the column into which data is to be inserted (unless all column names are omitted)
  - A row is inserted into a viewed table (default values are stored in the columns of the viewed table that do not correspond to the columns of the underlying table)
- When updating column values with the UPDATE statement
  The default value for the column is stored if DEFAULT is specified in the update value.
- When importing data with the adbimport command
  When importing data with the adbimport command, the default value for the column is stored if the field data in the input data file is an empty character string.
  If you want to store the null value rather than the default value for the column, specify NULL for the import option adb_import_null_string.

(1) Specification format

```
DEFAULT-clause ::= DEFAULT default-option
  default-option ::= {literal | CURRENT_DATE | CURRENT_TIME[(p)]
                    | CURRENT_TIMESTAMP[(p)] | CURRENT_USER | NULL}
```

(2) Explanation of specification format

literal:

Specifies the default value for a column in the form of a literal. For details about literals, see 6.3 Literals.

The following table shows the types of literals that can be specified in *default-option* depending on the data type of the column whose default value is to be specified.

Table 7-2: Types of literals that can be specified in default-option depending on the data type of the column whose default value is to be specified

Data type of the column whose default value is to be specified	Literals that can be specified in default-option					
	Numeric literal	Character string literal	Date literal	Time literal	Time stamp literal	Binary literal
Numeric data	Y	N	N	N	N	N

Data type of the column whose default value is to be specified		Literals that can be specified in default-option					
		Numeric literal	Character string literal	Date literal	Time literal	Time stamp literal	Binary literal
Character string data		N	Y#1	N	N	N	N
Datetime data	DATE type	N	Y#2	Y	N	Y	N
	TIME type	N	Y#2	N	Y	N	N
	TIMESTAMP type	N	Y#2	Y	N	Y	N
Binary data		N	N	N	N	N	Y#1

Legend:

Y: Can be specified. However, storage assignment rules apply.#3

N: Cannot be specified.

#1

Character string literals or binary literals of 1,024 bytes or more cannot be specified.

#2

The character string literal must be represented in a relevant predefined input representation. For information about predefined input representations, see 6.3.3 Predefined character-string representations.

#3

For details about the storage assignment rules, see (2) Storage assignments between data types in 6.2.2 Data types that can be converted, assigned, and compared.

For example, because the storage assignment rules are applied, the CREATE TABLE statement will result in an error if the data length of the character string literal specified as the default value for a column exceeds the data length of the column whose DEFAULT clause was specified.

CURRENT_DATE:

The default value for the column will be the date when the INSERT or UPDATE statement is executed, or when the adbimport command is launched.

CURRENT_DATE can be specified for a column of type DATE or TIMESTAMP.

For details about the rules for specifying CURRENT_DATE, see 6.4.1 CURRENT_DATE.

CURRENT_TIME[(*p*)]:

The default value for the column will be the time when the INSERT or UPDATE statement is executed, or when the adbimport command is launched.

Specify the fractional seconds precision (the number of digits to the right of the decimal point) in *p*. If (*p*) is omitted, it is assumed that *p* = 0.

CURRENT_TIME can be specified for a column of type TIME.

For details about the rules for specifying CURRENT_TIME, see 6.4.2 CURRENT_TIME.

CURRENT_TIMESTAMP[(*p*)]:

The default value for the column will be the date and time when the INSERT or UPDATE statement is executed, or when the adbimport command is launched.

Specify the fractional seconds precision (the number of digits to the right of the decimal point) in *p*. If (*p*) is omitted, it is assumed that *p* = 0.

CURRENT_TIMESTAMP can be specified for a column of type DATE or TIMESTAMP.

For details about the rules for specifying CURRENT_TIMESTAMP, see 6.4.3 CURRENT_TIMESTAMP.

CURRENT_USER:

The default value for the column will be the authorization identifier of the user executing the INSERT statement, UPDATE statement, or adbimport command.

CURRENT_USER can be specified for a column of type CHARACTER or VARCHAR.

For details about the rules for specifying CURRENT_USER, see 6.5.1 CURRENT_USER.

NULL:

The default value for the column will be the null value.

NULL cannot be specified for columns having the NOT NULL constraint (the constraint to not allow null values).

Note

- When CURRENT_DATE, CURRENT_TIME[(*p*)], or CURRENT_TIMESTAMP[(*p*)] is specified, the corresponding date and time information is acquired by the HADB server.
- When you store the default value for a column on multiple rows by using a single SQL statement, if you specify CURRENT_DATE, the same date is stored on all rows. If you specify CURRENT_TIME[(*p*)], the same time is stored on all rows. If you specify CURRENT_TIMESTAMP[(*p*)], the same date and time is stored on all rows.

(3) Rules

1. If the DEFAULT clause is omitted, the default value for the column will be the null value.
2. When storing data in a column where a default value is specified, storage assignment rules apply. For example, if CURRENT_DATE is specified for a TIMESTAMP type column, 00:00:00 is assigned to the time portion in accordance with the assignment rules. For details about the assignment rules, see (2) Storage assignments between data types in 6.2.2 Data types that can be converted, assigned, and compared.
3. The precision of the fractional seconds acquired by CURRENT_TIME(*p*) or CURRENT_TIMESTAMP(*p*) depends on the capabilities of the hardware. For example, if you specify CURRENT_TIME(12), depending on the hardware you are using, you might not be able to acquire 12 digits of fractional seconds precision.

   Example:

   10:35:55.123456000000

   As shown above, if only six digits of fractional seconds precision can be acquired, the 7th and subsequent digits will be 0.

(4) Example

Example

Define a sales history table (SALESLIST) using the DEFAULT clause to set the default value of the date of purchase (PUR-DATE) column.

```
CREATE FIX TABLE "SALESLIST"
       ("USERID" CHAR(6),
        "PUR-CODE" CHAR(4),
        "PUR-NUM" SMALLINT,
        "PUR-DATE" DATE DEFAULT CURRENT_DATE)
    IN "DBAREA01"
    PCTFREE=20
    CHUNK=200
```

The underlined portion indicates the DEFAULT clause.

7.11 Table reference

This section describes table references.

7.11.1 Specification format for table references

A table reference, specified in the FROM clause, specifies the table from which to retrieve data.

If you want to retrieve data from a table that is joined to itself, a correlation name can also be specified.

(1) Specification format

```
table-reference::={table-primary|joined-table}

  table-primary::={table-name [[AS] correlation-name][index-specification]
                 |query-name [[AS] correlation-name]
                 |derived-table [[AS] correlation-name [(derived-column-list)]]
                 |table-function-derived-table [AS] correlation-name [(table-function-
column-list)]
                 |(joined-table)}

    derived-table::={table-subquery|(table-value-constructor)}
    derived-column-list::=column-name[,column-name]...

    table-function-derived-table::=TABLE(system-defined-function)
    table-function-column-list::=column-name data-type[,column-name data-type]...
```

(2) Explanation of specification format

table-name:

Specifies the table from which to retrieve data. For rules on specifying a table name, see (2) Table name specification format in 6.1.5 Qualifying a name.

To retrieve data from a dictionary table or system table, specify the schema name MASTER.

If an archivable multi-chunk table is specified, accesses to the location table and system table (STATUS_CHUNKS) occur. At this time, locked resources are secured for the system table (STATUS_CHUNKS). For details about locks, see *Locking* in the *HADB Setup and Operation Guide*.

[AS] *correlation-name*:

Specifies a name assigned to separately identify a table for one of the following purposes:

- To join a table to itself
- To reference a column of the same table inside a subquery

Note the following points:

- When specifying a table function derived table, a correlation name is required.
- To specify the same scope variable multiple times in one FROM clause, specify correlation names so that each scope variable is able to uniquely identify the column specification it qualifies.
- The correlation name specified in one FROM clause must be different from all scope variables specified in that clause. The name must also be different from the table identifiers of the scope variables. For details about the effective scope of scope variables, see 6.8 Scope variables.
- The retrieval results will be the same regardless of whether AS is specified.

- If the correlation name of a derived table is not specified, the correlation name is automatically assigned in the following format:

  ##ADD_DRVTBL_*xxxxxxxxxx*

  In the preceding format, *xxxxxxxxxx* is a 10-digit integer. This correlation name is displayed in the access path execution results.

  For table references with the same effective scope, do not specify a table name or correlation name that begins with ##ADD_DRVTBL_.

  > **Note**
  >
  > If you do so, the HADB server might automatically assign a correlation name that is the same as a name that you specified.

index-specification:

Specifies a B-tree index or text index to be used when retrieving from a base table. Alternatively, it specifies that the use of a B-tree index or text index is to be suppressed. For details about index specifications, see 7.14 Index specification.

query-name:

Specifies a query name. For details about query names, see (a) WITH-clause in (2) Explanation of specification format in 7.1.1 Specification format and rules for query expressions.

derived-table:

Specifies a derived table in the format of a table subquery or table value constructor. For details about subqueries, see 7.3 Subqueries. For details about table value constructors, see 7.17 Table value constructors.

A *derived table* is a table that is derived as a result of a table subquery or table value constructor. The *n*-th column of a table subquery or table value constructor becomes the *n*-th column of a derived table.

A query specification that contains a derived table will be converted to an equivalent query specification that does not contain the derived table.

Assume that the (user-specified or automatically assigned) correlation name of the derived table is the derived query name, and the query expression of the derived table is the derived query expression. In this case, the derived query expression is treated as an internal derived table, following the rules for derived table expansion. For the rules for derived table expansion, see 7.30 Internal derived tables.

In addition, note the following concerning *derived-table*:

- If the correlation name of a derived table is not specified, the scope variable of that derived table has the effective scope but has no name (the HADB server internally generates a name, which users cannot check). Therefore, if there are two or more table references that have the same column name in the same effective scope, explicitly specify correlation names.

  Example of an SQL statement that generates an error:

  ```
  SELECT "C1" FROM "T1",(SELECT "C1" FROM "T1")
  ```

  For the underlined column (C1), there are multiple table references that have the same column name in the same effective scope (table T1 and the derived table). In this case, it is impossible to identify whether the underlined column (C1) is column C1 of table T1 or column C1 of the derived table. Therefore, the SQL statement will result in an error. In such a case, to reference a column of a derived table, specify a correlation name for the derived table, and qualify the column name with that correlation name. The following shows examples.

  Example of a correct SQL statement:

  ```
  SELECT "DT1"."C1" FROM "T1",(SELECT "C1" FROM "T1") AS "DT1"
  ```

- You cannot specify the row interface (ROW) for a derived table.

derived-column-list:

Specify the column name of each column of the derived table. Specify *derived-column-list* in the following format:

column-name`[,` *column-name*`]...`

The column names of the table derived by a query specification vary depending on whether *derived-column-list* is specified. For the rules concerning derived column names, see 6.9 Derived column names.

In addition, note the following concerning *derived-column-list*:

- If *derived-column-list* is omitted, the column names derived from the results of the table subquery must be unique.
- The column names in *derived-column-list* must be unique.
- Do not specify a character string in the `EXP`*nnnn*`_NO_NAME` format as a column name in *derived-column-list*. Such a column name might duplicate a derived column name that is automatically set by HADB. In this format, *nnnn* is an unsigned integer in the range from `0000` to `9999`.
- If *derived-column-list* is specified, the number of column names in *derived-column-list* must be the same as the number of columns in the derived table.
- Make sure that the number of columns specified in *derived-column-list* does not exceed 1,000.
- Make sure that the number of columns derived by table subqueries or table value constructors does not exceed 1,000.

table-function-derived-table:

```
table-function-derived-table::=TABLE(system-defined-function)
```

A *table function derived table* is a collection of data in table format derived by means of a system-defined function. For details about system-defined functions, see 7.15 System-defined functions.

The rules for specifying a table function derived table are as follows:

- To specify a table function derived table as a table reference, specify the table function derived table's correlation name.
- You cannot specify the row interface (`ROW`) for a table function derived table.

table-function-column-list:

```
table-function-column-list::=column-name data-type[,column-name data-type]...
```

Specifies the name and data type of each column in the table function derived table.

The rules for specifying a table function column list are as follows:

- If you specify the `ADB_AUDITREAD` function for a table function derived table, you cannot specify a table function column list.
- If you specify the `ADB_CSVREAD` function for a table function derived table, you must specify a table function column list.
- For the specification format of each data type, see Table 3-8: Data types that can be specified (CREATE TABLE statement).
- The column names in a table function column list must be unique.
- Do not specify a character string in the `EXP`*nnnn*`_NO_NAME` format as a column name in a table function column list. Such a column name might duplicate a derived column name that is automatically set by HADB. In this format, *nnnn* is an unsigned integer in the range from `0000` to `9999`.
- The number of columns in a table function column list must not exceed 1,000.
- You cannot specify `VARCHAR`-type data whose length exceeds 32,000 bytes for a table function column list.

For rules on derived column names, see (4) In the case of a table function derived table in 6.9.2 Decision rules for derived column names in query results.

joined-table:

Specifies a joined table. For details on joined tables, see 7.12 Joined tables.

(3) Examples

The following examples illustrate table references.

Example 1

From the sales history table (SALESLIST), this example retrieves the customer ID (USERID), product code (PUR-CODE), and date of purchase (PUR-DATE) for customers who purchased products on or after September 6, 2011.

```
SELECT "USERID","PUR-CODE","PUR-DATE"
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2011-09-06'
```

The underlined portion indicates the table reference.

Example 2

Search the dictionary table (SQL_INDEXES) to find the names of indexes (INDEX_NAME) that are defined for the sales history table (SALESLIST).

```
SELECT "INDEX_NAME"
    FROM "MASTER"."SQL_INDEXES"
        WHERE "TABLE_NAME"='SALESLIST'
```

The underlined portion indicates the table reference. To search the dictionary table, you must qualify the table name with the schema name MASTER.

Example 3

Search the dictionary table (SQL_INDEXES) to find the names of indexes (INDEX_NAME) that are defined for the sales history table (SALESLIST) using IDX as a correlation name.

```
SELECT "IDX"."INDEX_NAME"
    FROM "MASTER"."SQL_INDEXES" AS "IDX"
        WHERE "IDX"."TABLE_NAME"='SALESLIST'
```

The underlined portion indicates the table reference.

Example 4

Retrieve the customer ID (USERID), product code (PUR-CODE), customer name (NAME), and sex (SEX) from the sales history table (SALESLIST) and customer table (USERSLIST), joined together with the customer ID column (USERID) as the search condition.

```
SELECT "SALESLIST"."USERID","PUR-CODE","NAME","SEX"
    FROM ("SALESLIST" JOIN "USERSLIST"
          ON "USERSLIST"."USERID"="SALESLIST"."USERID")
```

The underlined portion indicates the table reference.

Example 5

Extract the following data from a CSV file (/dir/file.csv.gz) compressed in GZIP format:

- Customer ID (USERID)
- Customer name (NAME)
- Age (AGE)

```
SELECT "USERID","NAME","AGE"
    FROM TABLE(ADB_CSVREAD(MULTISET ['/dir/file.csv.gz'],
                          'COMPRESSION_FORMAT=GZIP;'))
         AS "USERLIST" ("USERID" CHAR(5),
                        "NAME" VARCHAR(100),
                        "AGE" INTEGER,
                        "COUNTRY" VARCHAR(100),
                        "INFORMATION" VARBINARY(10))
```

The underlined portion indicates the table reference.

7.12 Joined tables

This section describes joined tables.

7.12.1 Specification format and rules for joined tables

This subsection explains the methods (Cartesian product, inner join, and outer join) for specifying joined tables. Joined tables are specified in table references.

(1) Specification format

```
joined-table ::= {cross-join|qualified-join|(joined-table)}


  cross-join ::= table-reference CROSS JOIN table-primary

  qualified-join ::= table-reference [{INNER|{LEFT|RIGHT|FULL} [OUTER]}] JOIN [join-m
ethod-specification] table-reference join-specification
    join-specification ::= ON search-condition
```

(2) Explanation of specification format

- *cross-join*

```
cross-join ::= table-reference CROSS JOIN table-primary
```

Specify this to obtain the Cartesian product of the *table-reference* specified on the left side and the *table-primary* specified on the right side. For details about table references, see 7.11 Table reference. For details about *table-primary*, see 7.11.1 Specification format for table references.

Note that when * is specified in the selection expression of a query specification, the columns in the retrieval results will be arranged according to the order of the columns from *table-reference* on the left first and then the columns from *table-primary* on the right.

The cross join is illustrated in the following example.

Example:

■ USERSLIST

Customer ID (USERID)	Customer name (NAME)
U00555	Mike Johnson
U00358	Nancy White
U00212	Maria Gomez

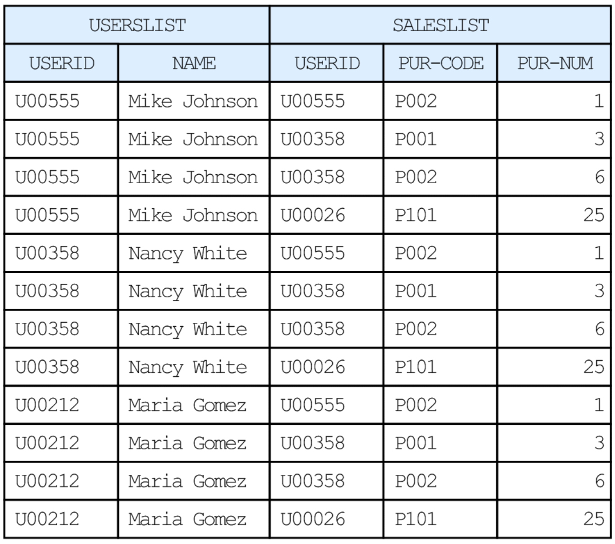
■ SALESLIST

Customer ID (USERID)	Product code (PUR-CODE)	Quantity purchased (PUR-NUM)
U00555	P002	1
U00358	P001	3
U00358	P002	6
U00026	P101	25

SELECT statement to be executed

```
SELECT * FROM "USERSLIST" CROSS JOIN "SALESLIST"
```

Retrieval results



USERSLIST		SALESLIST		
USERID	NAME	USERID	PUR-CODE	PUR-NUM
U00555	Mike Johnson	U00555	P002	1
U00555	Mike Johnson	U00358	P001	3
U00555	Mike Johnson	U00358	P002	6
U00555	Mike Johnson	U00026	P101	25
U00358	Nancy White	U00555	P002	1
U00358	Nancy White	U00358	P001	3
U00358	Nancy White	U00358	P002	6
U00358	Nancy White	U00026	P101	25
U00212	Maria Gomez	U00555	P002	1
U00212	Maria Gomez	U00358	P001	3
U00212	Maria Gomez	U00358	P002	6
U00212	Maria Gomez	U00026	P101	25

Each row in `USERLIST` is combined with every row in `SALESLIST`.

- *qualified-join*

```
qualified-join ::= table-reference [{INNER | {LEFT | RIGHT | FULL} [OUTER]}] JOIN
[join-method-specification] table-reference join-specification
  join-specification ::= ON search-condition
```

Specify this to perform an inner join or outer join.

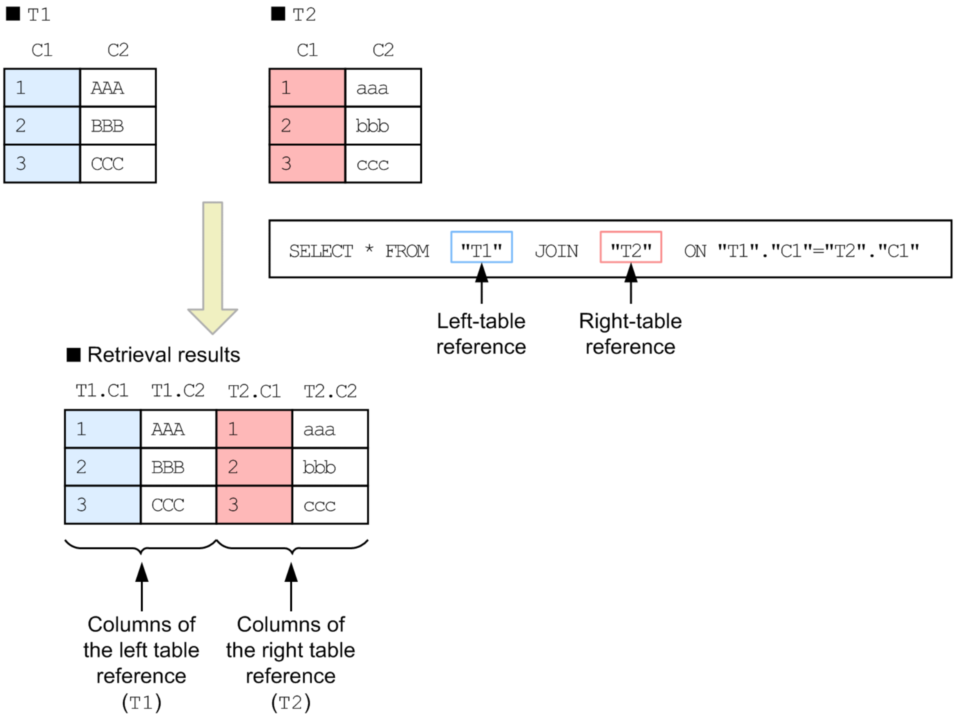
When `INNER JOIN` is specified, the operation is called an inner join, and when `LEFT OUTER JOIN`, `RIGHT OUTER JOIN` or `FULL OUTER JOIN` is specified, the operation is called an outer join.

table-reference:

Specifies a table or a joined table to be joined. For details about table references, see 7.11 Table reference.

When `*` is specified in the selection expression of a query specification, the columns in the retrieval results will be arranged according to the order of the columns from the table references on the left first and then from the table references on the right. This is illustrated in the following example.

Example:



■ T1

C1	C2
1	AAA
2	BBB
3	CCC

■ T2

C1	C2
1	aaa
2	bbb
3	ccc

```
SELECT * FROM "T1" JOIN "T2" ON "T1"."C1"="T2"."C1"
```

■ Retrieval results

T1.C1	T1.C2	T2.C1	T2.C2
1	AAA	1	aaa
2	BBB	2	bbb
3	CCC	3	ccc

Left-table reference
Right-table reference
Columns of the left table reference (T1)
Columns of the right table reference (T2)

`[INNER] JOIN`:

The joined table will consist of the rows in the Cartesian product of the tables referenced by the left and right table references for which the *search-condition* specified in *join-specification* is true.

For an example of `INNER JOIN`, see 7.12.2 Inner join using INNER JOIN.

`LEFT [OUTER] JOIN`:

The joined table will be the union of the following rows:

- The rows in the Cartesian product of the tables referenced by the left and right table references for which the *search-condition* specified in *join-specification* is true (the same results as when `INNER JOIN` is specified).
- The rows in the Cartesian product of the tables referenced by the left and right table references such that *search-condition* is false for the rows of the left table and the rows of the right table are assigned null values.

For an example of `LEFT OUTER JOIN`, see 7.12.3 Outer join using LEFT OUTER JOIN.

`RIGHT [OUTER] JOIN`:

The joined table will be the union of the following rows:

- The rows in the Cartesian product of the tables referenced by the left and right table references for which the *search-condition* specified in *join-specification* is true (the same results as when `INNER JOIN` is specified).
- The rows in the Cartesian product of the tables referenced by the left and right table references such that *search-condition* is false for the rows of the right table and the rows of the left table are assigned null values.

For an example of `RIGHT OUTER JOIN`, see 7.12.4 Outer join using RIGHT OUTER JOIN.

`FULL [OUTER] JOIN`:

The joined table will be the union of the following rows:

- The rows in the Cartesian product of the tables referenced by the left and right table references for which the *search-condition* specified in *join-specification* is true (the same results as when `INNER JOIN` is specified).

- The rows in the Cartesian product of the tables referenced by the left and right table references such that *search-condition* is false for the rows of the left table and the rows of the right table are assigned null values.
- The rows in the Cartesian product of the tables referenced by the left and right table references such that *search-condition* is false for the rows of the right table and the rows of the left table are assigned null values.

For an example of `FULL OUTER JOIN`, see 7.12.5 Outer join using FULL OUTER JOIN.

join-method-specification:

Specifies the method for joining the left and right table references. For details, see 7.13 Join method specification.

Note that *join-method-specification* does not normally need to be specified. If *join-method-specification* is omitted, HADB determines the join method automatically.

join-specification:

```
join-specification ::= ON search-condition
```

Specifies the conditions for joining the two table references.

`ON` *search-condition*:

Specifies a search condition. For details on search conditions, see 7.18 Search conditions.

Each column specification in the search condition must be one of the following:

- A column included in the two table references being joined
- An external reference column

For details about external reference columns, see (a) Common rules for subqueries in (4) Rules in 7.3.1 Specification format and rules for subqueries.

If you qualify a column name from *search-condition* in a table specification, any column from a table with a correlation name must be qualified with the correlation name.

Subqueries are not permitted in the `ON` search condition of a joined table with `FULL OUTER JOIN` specified as the joined table mode.

- (*joined-table*)

  To specify the join order of the tables, enclose the joined tables in parentheses.

(3) Rules

1. When `LEFT OUTER JOIN` is specified, the `NOT NULL` constraint does not apply to the results of the right table reference (null values are allowed).
2. When `RIGHT OUTER JOIN` is specified, the `NOT NULL` constraint does not apply to the result of the left table reference (null values are allowed).
3. When `FULL OUTER JOIN` is specified, the `NOT NULL` constraint does not apply to the results of both the left and right table references (null values are allowed).
4. Up to 63 outer joins with `FULL OUTER JOIN` specified as the joined table mode can be specified in an SQL statement. If the table reference to be joined is a viewed table, an internal derived table is generated according to the query expression specified in the `CREATE VIEW` statement. The limit on the maximum number of `FULL OUTER JOIN` clauses is applied to this internal derived table.
5. If `FULL OUTER JOIN` is specified, a derived table is generated. For the derived table, the HADB server automatically assigns a correlation name in the following format:

   ```
   ##DRVTBL_xxxxxxxxxx
   ```

   In the preceding format, *xxxxxxxxxx* is a 10-digit integer.

6. If hash join is selected as the table joining method and a hash filter is applied to hash join processing, a hash filter area of an appropriate size is required. The size of the hash filter area is specified by using the `adb_sql_exe_hashflt_area_size` operand in the server definition or client definition.

7. The HADB server sometimes converts `INNER JOIN` or `CROSS JOIN` to an equivalent comma join when executing an SQL statement. For details about a comma join, see (2) Explanation of specification format in 7.5.1 Specification format and rules for FROM clauses.

(4) Examples

Example 1 (example of **INNER JOIN**)

From the customer table (`USERSLIST`) and sales history table (`SALESLIST`), retrieve a list of customers (customer ID and name) who purchased product code (`PUR-CODE`) `P001`, eliminating duplicates.

```
SELECT DISTINCT "USERSLIST"."USERID","NAME"
    FROM "USERSLIST" INNER JOIN "SALESLIST"
           ON "USERSLIST"."USERID"="SALESLIST"."USERID"
        WHERE "SALESLIST"."PUR-CODE"='P001'
```

The underlined portion indicates the joined table (inner join).

■ Columns retrieved as results

USERID	NAME
U00212	Maria Gomez
U00358	Nancy White

■ Tables that columns are retrieved from

USERSLIST

USERID	NAME
U00555	Mike Johnson
U00358	Nancy White
U00212	Maria Gomez
U00687	Taro Tanaka
U00869	Bob Clinton

SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2012-12-03
U00358	P001	1	2012-12-04
U00555	P002	5	2012-12-06
U00212	P003	10	2012-12-03
U00358	P003	2	2012-12-05
U00358	P002	6	2012-12-07
U00212	P002	12	2012-12-05
U00687	P002	8	2012-12-06
U00687	P003	5	2012-12-07
U00212	P001	6	2012-12-05
U00358	P001	9	2012-12-03
U00358	P002	3	2012-12-04

Example 2 (example of **LEFT OUTER JOIN**)

From the product table (`PRODUCTLIST`) and sales history table (`SALESLIST`), determine the total number of sales in December 2012 for each product.

```
SELECT "PRODUCTLIST"."PUR-NAME",SUM("SALESLIST"."PUR-NUM") AS "SUM"
    FROM "PRODUCTLIST" LEFT OUTER JOIN "SALESLIST"
       ON "PRODUCTLIST"."PUR-CODE"="SALESLIST"."PUR-CODE"
       AND "SALESLIST"."PUR-DATE" BETWEEN DATE'2012-12-01'
                                AND DATE'2012-12-31'
    GROUP BY "PRODUCTLIST"."PUR-NAME"
```

The underlined portion indicates the joined table (outer join).

■ Columns retrieved as results

PUR-NAME	SUM
File	16
Highlighter	17
Paste	NULL
Pen	37
Scissors	NULL

For products with no sales, the null value is stored for the total number of purchases.

■ Tables from which columns are retrieved

PRODUCTLIST

PUR-CODE	PUR-NAME
P001	File
P002	Pen
P003	Highlighter
P004	Paste
P005	Scissors

SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2012-12-03
U00358	P001	1	2012-12-04
U00555	P002	5	2012-12-06
U00212	P003	10	2012-12-03
U00358	P003	2	2012-12-05
U00358	P002	6	2012-12-07
U00212	P002	12	2012-12-05
U00687	P002	8	2012-12-06
U00687	P003	5	2012-12-07
U00212	P001	6	2012-12-05
U00358	P001	9	2012-12-03
U00358	P002	3	2012-12-04

Example 3 (example of `RIGHT OUTER JOIN`)

From the product table (`PRODUCTLIST`) and sales history table (`SALESLIST`), determine the total number of sales in December 2012 for each product.

```
SELECT "PRODUCTLIST"."PUR-NAME",SUM("SALESLIST"."PUR-NUM") AS "SUM"
    FROM "SALESLIST" RIGHT OUTER JOIN "PRODUCTLIST"
       ON "SALESLIST"."PUR-CODE"="PRODUCTLIST"."PUR-CODE"
       AND "SALESLIST"."PUR-DATE" BETWEEN DATE'2012-12-01'
                                      AND DATE'2012-12-31'
    GROUP BY "PRODUCTLIST"."PUR-NAME"
```

The underlined portion indicates the joined table (outer join).

■ Columns retrieved as results

PUR-NAME	SUM
File	16
Highlighter	17
Paste	NULL
Pen	37
Scissors	NULL

For products with no sales, the null value is stored for the total number of purchases.

■ Tables from which columns are retrieved

PRODUCTLIST

PUR-CODE	PUR-NAME
P001	File
P002	Pen
P003	Highlighter
P004	Paste
P005	Scissors

SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2012-12-03
U00358	P001	1	2012-12-04
U00555	P002	5	2012-12-06
U00212	P003	10	2012-12-03
U00358	P003	2	2012-12-05
U00358	P002	6	2012-12-07
U00212	P002	12	2012-12-05
U00687	P002	8	2012-12-06
U00687	P003	5	2012-12-07
U00212	P001	6	2012-12-05
U00358	P001	9	2012-12-03
U00358	P002	3	2012-12-04

Example 4 (example of `FULL OUTER JOIN`)

From the customer table (`USERSLIST`), product table (`PRODUCTLIST`), and sales history table (`SALESLIST`), retrieve a list of combinations of customer name and product name for the customers who bought products in December 2012, eliminating duplicates.

- For customers without a purchase record, use the null value for the product name (`PUR-NAME`).
- For products with 0 sales, use the null value for the customer name (`NAME`).

```
SELECT DISTINCT "USERSLIST"."NAME","PRODUCTLIST"."PUR-NAME"
    FROM ("USERSLIST" LEFT OUTER JOIN "SALESLIST"
                        ON "USERSLIST"."USERID"="SALESLIST"."USERID"
                        AND "SALESLIST"."PUR-DATE" BETWEEN DATE'2012-12-01'
                                                       AND DATE'2012-12-31')
                      FULL OUTER JOIN "PRODUCTLIST"
                        ON "SALESLIST"."PUR-CODE"="PRODUCTLIST"."PUR-CODE"
```

The underlined portion indicates the joined table (outer join).

■ Columns retrieved as results

NAME	PUR-NAME
Bob Clinton	NULL
Maria Gomez	File
Maria Gomez	Highlighter
Maria Gomez	Pen
Mike Johnson	Pen
Nancy White	File
Nancy White	Highlighter
Nancy White	Pen
Taro Tanaka	Highlighter
Taro Tanaka	Pen
NULL	Paste
NULL	Scissors

For customers with no purchases, the null value is stored for the product name.

For products with no sales, the null value is stored for the customer name.

■ Tables from which columns are retrieved

USERSLIST

USERID	NAME
U00555	Mike Johnson
U00358	Nancy White
U00212	Maria Gomez
U00687	Taro Tanaka
U00869	Bob Clinton

PRODUCTLIST

PUR-CODE	PUR-NAME
P001	File
P002	Pen
P003	Highlighter
P004	Paste
P005	Scissors

SALESLIST

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P002	3	2012-12-03
U00358	P001	1	2012-12-04
U00555	P002	5	2012-12-06
U00212	P003	10	2012-12-03
U00358	P003	2	2012-12-05
U00358	P002	6	2012-12-07
U00212	P002	12	2012-12-05
U00687	P002	8	2012-12-06
U00687	P003	5	2012-12-07
U00212	P001	6	2012-12-05
U00358	P001	9	2012-12-03
U00358	P002	3	2012-12-04

Example 5: (Specifying a subquery in the search condition of a join specification)

From the product table (PRODUCTLIST) and sales history table (SALESLIST), produce a list of customers who purchased the greatest quantity of each product. In the case of products with no sales, the customer ID (USERID) and quantity purchased (PUR-NUM) are assigned null values.

```
SELECT "G"."PUR-NAME","S"."USERID","S"."PUR-NUM"
    FROM "PRODUCTLIST" "G" LEFT JOIN "SALESLIST" "S"
       ON "G"."PUR-CODE"="S"."PUR-CODE"
           AND "S"."PUR-NUM"=(
                              SELECT MAX("SMAX"."PUR-NUM")
                                FROM "SALESLIST" "SMAX"
                                  WHERE "S"."PUR-CODE"="SMAX"."PUR-CODE"
                             )
```

The underlined portion indicates the joined table.

■ Columns retrieved as results

PUR-NAME	USERID	PUR-NUM
File	U00358	3
Pen	U00358	6
Highlighter	U00555	2
Paste	NULL	NULL
Scissors	U00212	2

For products with no sales, the null value is stored for the total number of purchases.

■ Tables that columns are retrieved from

PRODUCTLIST

PUR-CODE	PUR-NAME
P001	File
P002	Pen
P003	Highlighter
P004	Paste
P005	Scissors

SALESLIST

USERID	PUR-CODE	PUR-NUM
U00555	P001	1
U00555	P003	2
U00358	P001	3
U00358	P002	6
U00212	P005	2
U00614	P001	2

7.12.2 Inner join using INNER JOIN

The following is an example of an inner join using `INNER JOIN`.

- **Tables to be retrieved from**

■ USERSLIST

Customer ID (USERID)	Customer name (NAME)
U00555	Mike Johnson
U00358	Nancy White
U00212	Maria Gomez

■ SALESLIST

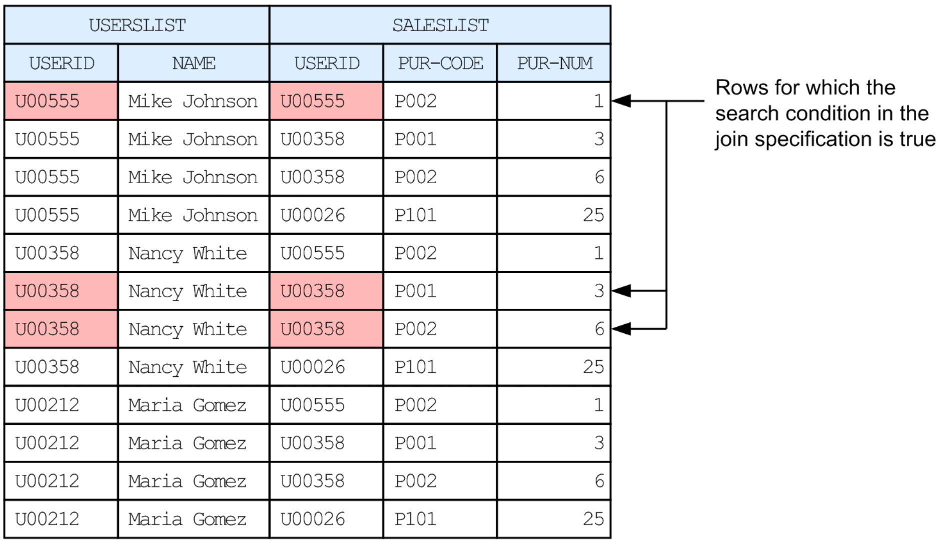
Customer ID (USERID)	Product code (PUR-CODE)	Quantity purchased (PUR-NUM)
U00555	P002	1
U00358	P001	3
U00358	P002	6
U00026	P101	25

- **`SELECT` statement to be run**

```
SELECT * FROM "USERSLIST" INNER JOIN "SALESLIST"
                          ON "USERSLIST"."USERID"="SALESLIST"."USERID"
```

From the Cartesian product of the left and right table references (the Cartesian product of `USERSLIST` and `SALESLIST`), produce a joined table consisting of the rows for which the search condition specified in the join specification (the underlined portion above) is true.

1. Cartesian product of `USERSLIST` and `SALESLIST` (all row combinations)



USERSLIST		SALESLIST		
USERID	NAME	USERID	PUR-CODE	PUR-NUM
U00555	Mike Johnson	U00555	P002	1
U00555	Mike Johnson	U00358	P001	3
U00555	Mike Johnson	U00358	P002	6
U00555	Mike Johnson	U00026	P101	25
U00358	Nancy White	U00555	P002	1
U00358	Nancy White	U00358	P001	3
U00358	Nancy White	U00358	P002	6
U00358	Nancy White	U00026	P101	25
U00212	Maria Gomez	U00555	P002	1
U00212	Maria Gomez	U00358	P001	3
U00212	Maria Gomez	U00358	P002	6
U00212	Maria Gomez	U00026	P101	25

Rows for which the search condition in the join specification is true

Each row of `SALESLIST` is paired with each row of `USERSLIST`.

2. Retrieval results

USERSLIST		SALESLIST		
USERID	NAME	USERID	PUR-CODE	PUR-NUM
U00555	Mike Johnson	U00555	P002	1
U00358	Nancy White	U00358	P001	3
U00358	Nancy White	U00358	P002	6

The joined table consists of the rows within the Cartesian product of `USERSLIST` and `SALESLIST` for which the search condition (`"USERSLIST"."USERID"="SALESLIST"."USERID"`) is true.

The result in this example is a list of customers who have purchased products.

7.12.3 Outer join using LEFT OUTER JOIN

The following is an example of an outer join using `LEFT OUTER JOIN`.

- **Tables to be retrieved from**

■ USERSLIST

Customer ID (USERID)	Customer name (NAME)
U00555	Mike Johnson
U00358	Nancy White
U00212	Maria Gomez

■ SALESLIST

Customer ID (USERID)	Product code (PUR-CODE)	Quantity purchased (PUR-NUM)
U00555	P002	1
U00358	P001	3
U00358	P002	6
U00026	P101	25

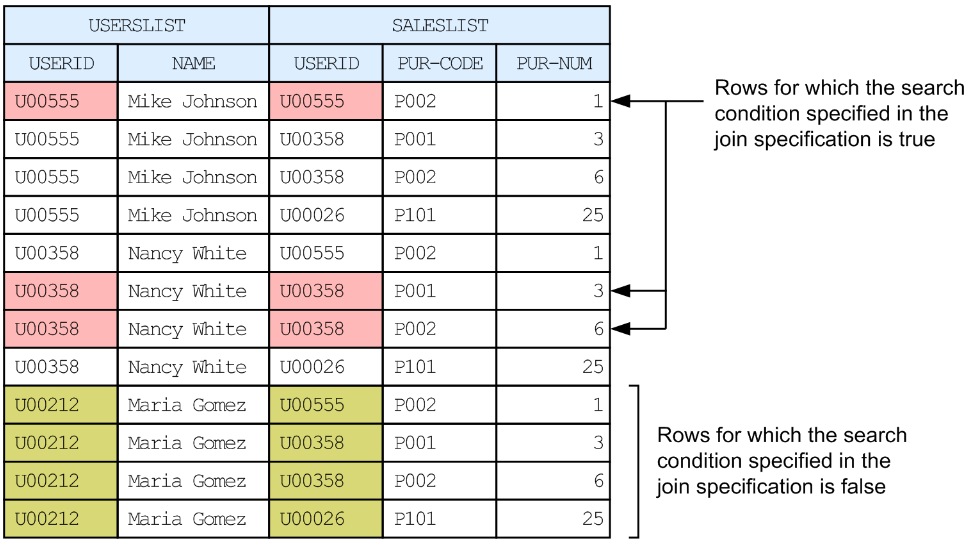
- **`SELECT` statement to be run**

```
SELECT * FROM "USERSLIST" LEFT OUTER JOIN "SALESLIST"
                          ON "USERSLIST"."USERID"="SALESLIST"."USERID"
```

The resulting joined table will be the union of the following rows:

- The rows from the Cartesian product of the left and right table references (the Cartesian product of USERSLIST and SALESLIST) for which the search condition specified in the join specification (the underlined portion above) is true
- The rows from the Cartesian product of the left and right table references (the Cartesian product of USERSLIST and SALESLIST) such that the search condition is false for the rows of the left table and the rows of the right table are assigned null values

1. Cartesian product of USERSLIST and SALESLIST (all row combinations)



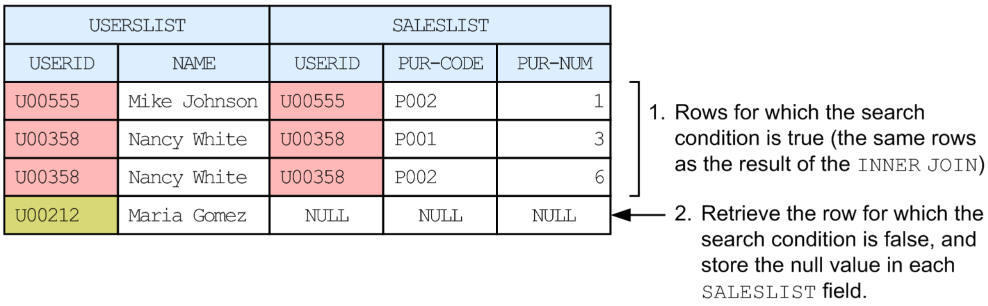
USERSLIST		SALESLIST		
USERID	NAME	USERID	PUR-CODE	PUR-NUM
U00555	Mike Johnson	U00555	P002	1
U00555	Mike Johnson	U00358	P001	3
U00555	Mike Johnson	U00358	P002	6
U00555	Mike Johnson	U00026	P101	25
U00358	Nancy White	U00555	P002	1
U00358	Nancy White	U00358	P001	3
U00358	Nancy White	U00358	P002	6
U00358	Nancy White	U00026	P101	25
U00212	Maria Gomez	U00555	P002	1
U00212	Maria Gomez	U00358	P001	3
U00212	Maria Gomez	U00358	P002	6
U00212	Maria Gomez	U00026	P101	25

Rows for which the search condition specified in the join specification is true

Rows for which the search condition specified in the join specification is false

Each row of SALESLIST is paired with each row of USERSLIST.

2. Retrieval results



USERSLIST		SALESLIST		
USERID	NAME	USERID	PUR-CODE	PUR-NUM
U00555	Mike Johnson	U00555	P002	1
U00358	Nancy White	U00358	P001	3
U00358	Nancy White	U00358	P002	6
U00212	Maria Gomez	NULL	NULL	NULL

1. Rows for which the search condition is true (the same rows as the result of the INNER JOIN)
2. Retrieve the row for which the search condition is false, and store the null value in each SALESLIST field.

The result in this example is a list of the following:

1. Customers who have purchased products (the same results from the INNER JOIN)
2. Customers who have not purchased products (in this case, customer ID U00212)

7.12.4 Outer join using RIGHT OUTER JOIN

The following is an example of an outer join using RIGHT OUTER JOIN.

- **Tables to be retrieved from**

■ USERSLIST

Customer ID (USERID)	Customer name (NAME)
U00555	Mike Johnson
U00358	Nancy White
U00212	Maria Gomez

■ SALESLIST

Customer ID (USERID)	Product code (PUR-CODE)	Quantity purchased (PUR-NUM)
U00555	P002	1
U00358	P001	3
U00358	P002	6
U00026	P101	25

- **SELECT statement to be run**

```
SELECT * FROM "SALESLIST" RIGHT OUTER JOIN "USERSLIST"
                          ON "USERSLIST"."USERID"="SALESLIST"."USERID"
```

The resulting joined table will be the union of the following rows:

- The rows from the Cartesian product of the left and right table references (the Cartesian product of USERSLIST and SALESLIST) for which the search condition specified in the join specification (the underlined portion above) is true
- The rows from the Cartesian product of the left and right table references (the Cartesian product of USERSLIST and SALESLIST) such that the search condition is false for the rows of the right table and the rows of the left table are assigned null values

1. Cartesian product of USERSLIST and SALESLIST (all row combinations)

SALESLIST			USERSLIST	
USERID	PUR-CODE	PUR-NUM	USERID	NAME
U00555	P002	1	U00555	Mike Johnson
U00358	P001	3	U00555	Mike Johnson
U00358	P002	6	U00555	Mike Johnson
U00026	P101	25	U00555	Mike Johnson
U00555	P002	1	U00358	Nancy White
U00358	P001	3	U00358	Nancy White
U00358	P002	6	U00358	Nancy White
U00026	P101	25	U00358	Nancy White
U00555	P002	1	U00212	Maria Gomez
U00358	P001	3	U00212	Maria Gomez
U00358	P002	6	U00212	Maria Gomez
U00026	P101	25	U00212	Maria Gomez

Rows for which the search condition specified in the join specification is true

Rows for which the search condition specified in the join specification is false

Each row of SALESLIST is paired with each row of USERSLIST.

2. Retrieval results

SALESLIST			USERSLIST	
USERID	PUR-CODE	PUR-NUM	USERID	NAME
U00555	P002	1	U00555	Mike Johnson
U00358	P001	3	U00358	Nancy White
U00358	P002	6	U00358	Nancy White
NULL	NULL	NULL	U00212	Maria Gomez

1. Rows for which the search condition is true (the same rows as the result of the INNER JOIN)
2. Retrieve the row for which the search condition is false, and store null values in each SALESLIST column.

The result in this example is a list that includes the following:

1. Customers who have purchased products (the same results from the INNER JOIN)
2. Customers who have not purchased products (in this case, customer ID U00212)

7.12.5 Outer join using FULL OUTER JOIN

The following is an example of an outer join using FULL OUTER JOIN.

- **Tables to be retrieved from**

■ USERSLIST

Customer ID (USERID)	Customer name (NAME)
U00555	Mike Johnson
U00358	Nancy White
U00212	Maria Gomez

■ SALESLIST

Customer ID (USERID)	Product code (PUR-CODE)	Quantity purchased (PUR-NUM)
U00555	P002	1
U00358	P001	3
U00358	P002	6
U00026	P101	25

- **SELECT statement to be run**

```
SELECT * FROM "SALESLIST" FULL OUTER JOIN "USERSLIST"
                          ON "USERSLIST"."USERID"="SALESLIST"."USERID"
```

The resulting joined table will be the union of the following rows:

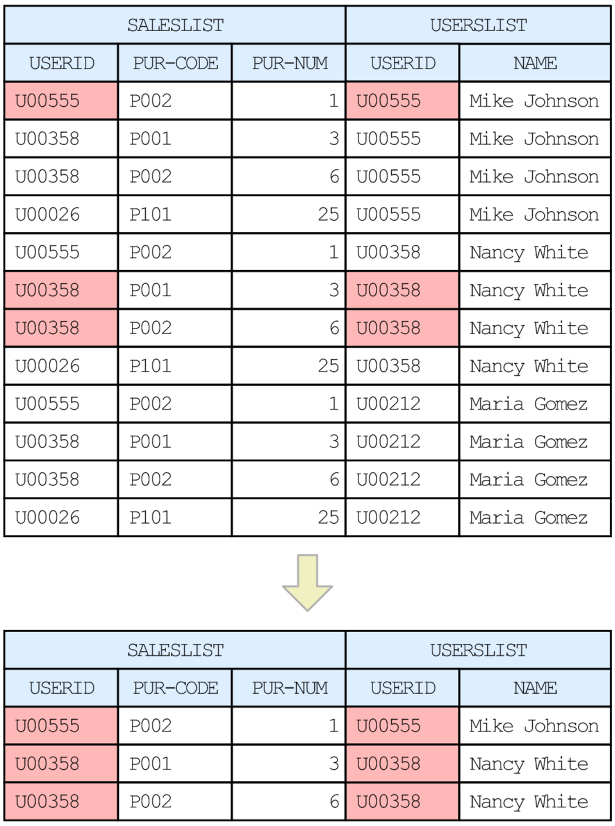
- The rows from the Cartesian product of the left and right table references (the Cartesian product of USERSLIST and SALESLIST) for which the search condition specified in the join specification (the underlined portion above) is true
- The rows from the Cartesian product of the left and right table references (the Cartesian product of USERSLIST and SALESLIST) such that the search condition is false for the rows of the left table and the rows of the right table are assigned null values
- The rows from the Cartesian product of the left and right table references (the Cartesian product of USERSLIST and SALESLIST) such that the search condition is false for the rows of the right table and the rows of the left table are assigned null values

1. Cartesian product of USERSLIST and SALESLIST (all row combinations)

SALESLIST			USERSLIST	
USERID	PUR-CODE	PUR-NUM	USERID	NAME
U00555	P002	1	U00555	Mike Johnson
U00358	P001	3	U00555	Mike Johnson
U00358	P002	6	U00555	Mike Johnson
U00026	P101	25	U00555	Mike Johnson
U00555	P002	1	U00358	Nancy White
U00358	P001	3	U00358	Nancy White
U00358	P002	6	U00358	Nancy White
U00026	P101	25	U00358	Nancy White
U00555	P002	1	U00212	Maria Gomez
U00358	P001	3	U00212	Maria Gomez
U00358	P002	6	U00212	Maria Gomez
U00026	P101	25	U00212	Maria Gomez

Each row of `SALESLIST` is paired with each row of `USERSLIST`.

2. Retrieve the rows for which the search condition is true.



SALESLIST			USERSLIST	
USERID	PUR-CODE	PUR-NUM	USERID	NAME
U00555	P002	1	U00555	Mike Johnson
U00358	P001	3	U00555	Mike Johnson
U00358	P002	6	U00555	Mike Johnson
U00026	P101	25	U00555	Mike Johnson
U00555	P002	1	U00358	Nancy White
U00358	P001	3	U00358	Nancy White
U00358	P002	6	U00358	Nancy White
U00026	P101	25	U00358	Nancy White
U00555	P002	1	U00212	Maria Gomez
U00358	P001	3	U00212	Maria Gomez
U00358	P002	6	U00212	Maria Gomez
U00026	P101	25	U00212	Maria Gomez

SALESLIST			USERSLIST	
USERID	PUR-CODE	PUR-NUM	USERID	NAME
U00555	P002	1	U00555	Mike Johnson
U00358	P001	3	U00358	Nancy White
U00358	P002	6	U00358	Nancy White

From the Cartesian product of `USERSLIST` and `SALESLIST`, this example retrieves the rows for which the search condition specified in the join specification is true (the same rows as the results of specifying `INNER JOIN`).

3. Retrieve the rows such that for each row of the left table reference where the search condition is false, the result of the right table reference is the null value.

SALESLIST			USERSLIST	
USERID	PUR-CODE	PUR-NUM	USERID	NAME
U00555	P002	1	U00555	Mike Johnson
U00358	P001	3	U00555	Mike Johnson
U00358	P002	6	U00555	Mike Johnson
U00026	P101	25	U00555	Mike Johnson
U00555	P002	1	U00358	Nancy White
U00358	P001	3	U00358	Nancy White
U00358	P002	6	U00358	Nancy White
U00026	P101	25	U00358	Nancy White
U00555	P002	1	U00212	Maria Gomez
U00358	P001	3	U00212	Maria Gomez
U00358	P002	6	U00212	Maria Gomez
U00026	P101	25	U00212	Maria Gomez

SALESLIST			USERSLIST	
USERID	PUR-CODE	PUR-NUM	USERID	NAME
U00026	P101	25	NULL	NULL

From the Cartesian product of `USERSLIST` and `SALESLIST`, this example retrieves the rows such that for each row of the left table reference (`SALESLIST`) where the search condition specified in the join specification is false, the result of the right table reference (`USERSLIST`) is the null value. In this example, this yields the row where the value in the `USERID` column of `SALESLIST` is `U00026`.

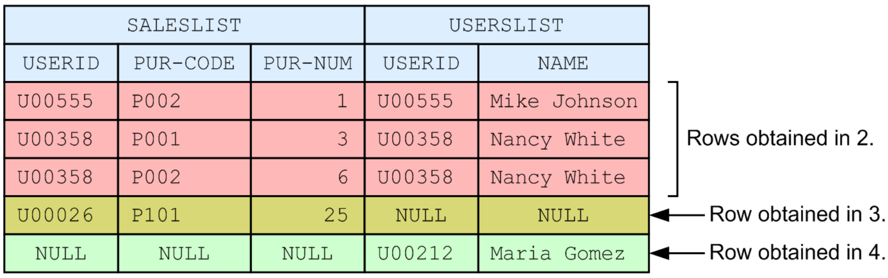
4. Retrieve the rows such that for each row of the right table reference where the search condition is false, the result of the left table reference is the null value.

SALESLIST			USERSLIST	
USERID	PUR-CODE	PUR-NUM	USERID	NAME
U00555	P002	1	U00555	Mike Johnson
U00358	P001	3	U00555	Mike Johnson
U00358	P002	6	U00555	Mike Johnson
U00026	P101	25	U00555	Mike Johnson
U00555	P002	1	U00358	Nancy White
U00358	P001	3	U00358	Nancy White
U00358	P002	6	U00358	Nancy White
U00026	P101	25	U00358	Nancy White
U00555	P002	1	U00212	Maria Gomez
U00358	P001	3	U00212	Maria Gomez
U00358	P002	6	U00212	Maria Gomez
U00026	P101	25	U00212	Maria Gomez

SALESLIST			USERSLIST	
USERID	PUR-CODE	PUR-NUM	USERID	NAME
NULL	NULL	NULL	U00212	Maria Gomez

From the Cartesian product of `USERSLIST` and `SALESLIST`, this example retrieves the rows such that for each row of the right table reference (`USERSLIST`) where the search condition specified in the join specification is false, the result of the left table reference (`SALESLIST`) is the null value. In this example, this yields the row where the value in the `USERID` column of `USERSLIST` is `U00212`.

5. Retrieval results



SALESLIST			USERSLIST	
USERID	PUR-CODE	PUR-NUM	USERID	NAME
U00555	P002	1	U00555	Mike Johnson
U00358	P001	3	U00358	Nancy White
U00358	P002	6	U00358	Nancy White
U00026	P101	25	NULL	NULL
NULL	NULL	NULL	U00212	Maria Gomez

Rows obtained in 2.

Row obtained in 3.

Row obtained in 4.

The result in this example is a list that includes the following:

1. Customers who have purchased products (the same results from the `INNER JOIN`)
2. Customers in the sales history who are not in the customer list (in this case, customer ID `U00026`)
3. Customers who have not purchased products (in this case, customer ID `U00212`)

7.13 Join method specification

This section describes join method specifications.

7.13.1 Specification format and rules for join method specifications

A join method specification specifies the method of joining the specified table references in a joined table. For details about the join methods, see *Table joining methods* in the *HADB Application Development Guide*.

Normally a join method specification is not required. If the join method specification is omitted, HADB determines the join method automatically.

(1) Specification format

```
join-method-specification ::= /*>> BY {NEST|HASH} [({LEFT|RIGHT} FIRST)] <<*/
```

(2) Explanation of specification format

`BY {NEST|HASH}`:

`NEST`:

Specifies a nested loop join as the join method.

`HASH`:

Specifies a hash join as the join method.

`({LEFT|RIGHT} FIRST)`:

`LEFT FIRST`:

Specifies that the outer table is to be the table reference on the left side of the joined table.

`RIGHT FIRST`:

Specifies that the outer table is to be the table reference on the right side of the joined table.

When neither `LEFT FIRST` nor `RIGHT FIRST` is specified, HADB automatically determines which of the joined tables in which the two table references are specified is to be the outer table.

You can check whether the join method specification was applied using the access path information. For details about how to check this, see *Table joining methods* in *Information displayed in the tree view* in the *HADB Application Development Guide*.

(3) Rules

1. If a join method that HADB cannot execute is specified, the join method specification is invalid. When the join method specification is invalid, HADB determines the join method automatically.
2. The character string enclosed in `/*>>` and `<<*/` is not a comment. An error results if you specify something other than a join method specification.

(4) Examples

Example 1

```
SELECT * FROM "T1" INNER JOIN /*>>BY NEST<<*/ "T2"
                   ON "T1"."C1"="T2"."C1"
```

The underlined portion indicates the join method specification.

When the above `SELECT` statement is executed, a nested loop join is used to joined tables `T1` and `T2`. The outer and inner tables are automatically determined by HADB.

Example 2

```
SELECT * FROM "T1" INNER JOIN /*>>BY NEST (LEFT FIRST)<<*/ "T2"
                   ON "T1"."C1"="T2"."C1"
```

The underlined portion indicates the join method specification.

When the above `SELECT` statement is executed, a nested loop join is used to join tables `T1` and `T2`. `T1` is the outer table and `T2` is the inner table.

Example 3

```
SELECT * FROM "T1" INNER JOIN /*>>BY NEST (RIGHT FIRST)<<*/ "T2"
                   ON "T1"."C1"="T2"."C1"
```

The underlined portion indicates the join method specification.

When the above `SELECT` statement is executed, a nested loop join is used to join tables `T1` and `T2`. `T2` is the outer table and `T1` is the inner table.

Example 4

```
SELECT * FROM "T1" INNER JOIN /*>>BY HASH<<*/ "T2"
                   ON "T1"."C1"="T2"."C1"
```

The underlined portion indicates the join method specification.

When the above `SELECT` statement is executed, a hash join is used to join tables `T1` and `T2`. The outer and inner tables are automatically determined by HADB.

Example 5

```
SELECT * FROM "T1" INNER JOIN /*>>BY HASH (LEFT FIRST)<<*/ "T2"
                   ON "T1"."C1"="T2"."C1"
```

The underlined portion indicates the join method specification.

When the above `SELECT` statement is executed, a hash join is used to join tables `T1` and `T2`. `T1` is the outer table and `T2` is the inner table.

Example 6

```
SELECT * FROM "T1" INNER JOIN /*>>BY HASH (RIGHT FIRST)<<*/ "T2"
                   ON "T1"."C1"="T2"."C1"
```

The underlined portion indicates the join method specification.

When the above `SELECT` statement is executed, a hash join is used to join tables `T1` and `T2`. `T2` is the outer table and `T1` is the inner table.

7.14 Index specification

This section describes index specifications.

7.14.1 Specification format and rules for index specifications

An index specification specifies an index to be used when retrieving data from a base table. It can also be specified to suppress use of an index.

Index specifications can only be used with B-tree indexes and text indexes. Range indexes are excluded.

Note that index specifications are usually not necessary, because HADB automatically determines the index to be used when retrieving from a base table. For the rules for determining the indexes to be used for retrieval, see *B-tree indexes and text indexes used during execution of SQL statements* in the *HADB Application Development Guide*.

(1) Specification format

```
index-specification ::= /*>> {WITH INDEX (index-name)|WITHOUT INDEX} <<*/
```

(2) Explanation of specification format

`WITH INDEX (`*index-name*`)`:

Specifies the index to be used when retrieving from the base table specified immediately before the index specification. For rules on specifying an index name, see (3) Index name specification format in 6.1.5 Qualifying a name.

`WITHOUT INDEX`:

Specifies that no index is to be used when retrieving from the base table specified immediately before the index specification. Instead, the table scan method is used for retrieving from the base table. For details about table scans, see *About table scans* in the *HADB Application Development Guide*.

You can check whether the index specification was applied using the access path information. For details about how to check this, see *Index specification* in *Information displayed in the tree view* in the *HADB Application Development Guide*.

(3) Rules

1. You cannot specify an index specification for a viewed table.
2. If you specify the name of a nonexistent index, the index specification is invalid.
3. If both of the following conditions are met, the index specification is invalid.
    - A B-tree index with the null-value exclusion is specified for *index-name*
    - A condition that contains the null value is specified in the search range of the B-tree index
4. Even when a text index is specified for *index-name*, the index specification is invalid if the index is determined to be unusable by HADB. For example, this is the case when you specify a `LIKE` predicate that cannot be evaluated for use by a text index.
5. If the index specification is invalid, HADB will automatically determine the index to be used for retrieval. For details, see *B-tree indexes and text indexes used during execution of SQL statements* in *Designs Related to Improvement of Application Program Performance* in the *HADB Application Development Guide*.

6. The character string enclosed in `/*>>` and `<<*/` is not a comment. An error results if you specify something other than an index specification.

   Example

   ```
   SELECT * FROM "T1"  /*>> comment <<*/
   ```

   In the preceding example, the underlined portion is not treated as a comment. Therefore, the preceding SQL statement results in a syntax error.

7. Conversely, in the following example, the text between `/*` and `*/` is treated as a comment rather than an index specification.

   ```
   SELECT * FROM "T1" /* WITH INDEX ("IDX01") */
   ```

(4) Examples

The following are examples of index specifications.

The examples assume that the following indexes are defined on the employee table (`EMPLOYEE`):

- B-tree index `BTREE_IDX` defined on the column `SCODE`
- Text index `TEXT_IDX` defined on the column `ADDRESS`

Example 1

Retrieve data from the employee table (`EMPLOYEE`) using the B-tree index `BTREE_IDX`.

```
SELECT "NAME" FROM "EMPLOYEE" /*>> WITH INDEX ("BTREE_IDX") <<*/
    WHERE "SCODE" = 'S003' AND "ADDRESS" LIKE '%TOKYO%'
```

The underlined portion shows the index specification.

Example 2

Retrieve data from the employee table (`EMPLOYEE`) using the text index `TEXT_IDX`.

```
SELECT "NAME" FROM "EMPLOYEE" /*>> WITH INDEX ("TEXT_IDX") <<*/
    WHERE "SCODE" = 'S003' AND "ADDRESS" LIKE '%TOKYO%'
```

The underlined portion shows the index specification.

Example 3

Retrieve data from the employee table (`EMPLOYEE`) without using an index.

```
SELECT "NAME" FROM "EMPLOYEE" /*>> WITHOUT INDEX <<*/
    WHERE "SCODE" = 'S003' AND "ADDRESS" LIKE '%TOKYO%'
```

The underlined portion shows the index specification.

Note that if a range index is defined on the `SCODE` column, only the range index will still be used when the above `SELECT` statement is executed. For the conditions on range indexes used during retrieval, see *Range indexes used during execution of SQL statements* in the *HADB Application Development Guide*.

7.15 System-defined functions

This section describes the system-defined functions.

7.15.1 Specification format and rules for system-defined functions

The functions provided by HADB are called *system-defined functions*.

(1) Specification format

```
system-defined-function::={ADB_AUDITREAD-function|ADB_CSVREAD-function}
```

(2) Explanation of specification format

ADB_AUDITREAD-function:

The ADB_AUDITREAD function converts the audit trails in an audit trail file into a dataset in a table format such that the data can be retrieved by the HADB server. For details about the ADB_AUDITREAD function, see 7.15.2 ADB_AUDITREAD function.

ADB_CSVREAD-function:

The ADB_CSVREAD function converts the data in a CSV file into a dataset in a table format such that the data can be retrieved by the HADB server. For details about the ADB_CSVREAD function, see 7.15.3 ADB_CSVREAD function.

(3) Rules

1. A maximum of 1,000 arguments can be specified for a system-defined function.

7.15.2 ADB_AUDITREAD function

Converts the audit trails in an audit trail file into a dataset in a table format such that the data can be retrieved by the HADB server.

Note

- For an overview of the audit trail facility, see *Audit trail facility* in the *HADB Setup and Operation Guide*.
- For details about the operations for searching audit trails, see *Scheduled operations for audit trail facility* in the *HADB Setup and Operation Guide*.

(1) Specification format

```
ADB_AUDITREAD-function::=
    [MASTER.]ADB_AUDITREAD([audit-trail-file-path-name-specification])

        audit-trail-file-path-name-specification::=multiset-value-expression
```

(2) Explanation of specification format

audit-trail-file-path-name-specification:

Specifies the path names of the audit trail files containing the input data for the `ADB_AUDITREAD` function. The path names are specified in the form of a multiset value expression. For details about multiset value expressions, see 7.16 Multiset value expression.

The following rules apply:

- The data type of the result of the multiset value expression must be character string data.
- Absolute paths must be specified for the path names of the audit trail files specified in *audit-trail-file-path-name-specification*.
- Existing files must be specified in the path names of the audit trail files specified in *audit-trail-file-path-name-specification*.

About the multi-node function:

- If the multi-node function is enabled, *audit-trail-file-path-name-specification* cannot be omitted.

■ **Examples of *audit-trail-file-path-name-specification***

In the following examples, the underlined portions are *audit-trail-file-path-name-specification*.

Note that the specification examples shown here are only typical ones. For rules on specifying an audit trail file path name specification, see (b) Rules for audit trail file path name specification in (4) Rules.

Example 1:

```
ADB_AUDITREAD(MULTISET['/audit/adbaud-20170401-123000-159.aud','/audit/adbaud
-20170415-123000-952.aud'])
```

In the preceding example, the path names of two audit trail files are specified. These two audit trail files are used as the input information for the `ADB_AUDITREAD` function.

Example 2:

```
ADB_AUDITREAD(MULTISET['/audit/*.aud'])
```

In the preceding example, the audit trail file path name includes the special character `*`. In this case, all audit trail files stored in the `/audit` directory (files with the extension `aud`) are used as the input information for the `ADB_AUDITREAD` function. However, the current audit trail file is not used as the input information for the `ADB_AUDITREAD` function.

Example 3:

```
ADB_AUDITREAD(MULTISET['/audit1/*.aud','/audit2/*.aud'])
```

In the preceding example, the audit trail files stored in the `/audit1` and `audit2` directories are used as the input information for the `ADB_AUDITREAD` function.

Example 4:

```
ADB_AUDITREAD(MULTISET['/audit/adbaud-201707*.aud','/audit/adbaud-201708*.aud
'])
```

In the preceding example, of the audit trail files stored in the `/audit` directory, the audit trail files that were created in July and August, 2017, are used as the input information for the `ADB_AUDITREAD` function.

Important

For *audit-trail-file-path-name-specification*, you can also specify archive files in which audit trail files have been compressed by using the OS command gzip.

Example:

```
ADB_AUDITREAD(MULTISET['/audit/*.gz'])
```

In the preceding example, the audit trail files that have been compressed in the archive files (with the extension gz) in the /audit directory are used as the input information for the ADB_AUDITREAD function.

■ **If *audit-trail-file-path-name-specification* is omitted**

If *audit-trail-file-path-name-specification* is omitted, the audit trail files in the audit trail directory (the directory specified by the adb_audit_log_path operand in the server definition) are used as the input information for the ADB_AUDITREAD function. However, the following files are not used as the input information for the ADB_AUDITREAD function:

- Current audit trail file
- Files in the subdirectories of the audit trail directory

Example:

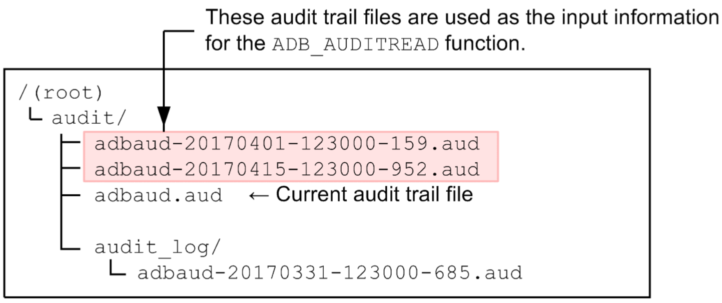
Specification in the server definition

```
adb_audit_log_path = /audit
```

Specification of the ADB_AUDITREAD function

```
ADB_AUDITREAD()
```

If the server definition and the ADB_AUDITREAD function are specified as shown in the preceding example, the audit trail files to be used as the input information for the ADB_AUDITREAD function are as shown in the following figure:



These audit trail files are used as the input information for the ADB_AUDITREAD function.

```
/(root)
 └ audit/
    ├ adbaud-20170401-123000-159.aud
    ├ adbaud-20170415-123000-952.aud
    ├ adbaud.aud    ← Current audit trail file
    │
    └ audit_log/
       └ adbaud-20170331-123000-685.aud
```

Note

If *audit-trail-file-path-name-specification* is omitted, the system assumes that the adb_audit_log_path operand specification in the server definition + /*.aud is specified as a multiset value expression in the *multiset-value-constructor-by-enumeration* format. Note that * is specified as a special character.

(3) Privileges required at execution

To execute the `ADB_AUDITREAD` function, the audit viewer privilege is required.

(4) Rules

(a) Rules for the ADB_AUDITREAD function

1. The `ADB_AUDITREAD` function can be used if the audit trail facility is enabled.
2. When the `ADB_AUDITREAD` function is executed, the audit trails in the audit trail files specified in *audit-trail-file-path-name-specification* are returned as a dataset in a table format. For details about the names of, data types of, and information in the columns of the table-formatted dataset returned by the `ADB_AUDITREAD` function, see *Column structure of table function derived table when retrieving audit trails* in the *HADB Setup and Operation Guide*.
3. If a specified audit trail file contains no audit trail records (other than the header information), the result of the table function derived table for that audit trail file will be an empty set. If a specified audit trail file is a 0-byte file, which contains neither audit trail records nor header information, the SQL statement will result in an error.

   This rule is checked during execution of the SQL statement, not during preprocessing of the SQL statement.

(b) Rules for audit trail file path name specification

1. For *audit-trail-file-path-name-specification*, specify the path names of the audit trail files that are to be used as the input information for the `ADB_AUDITREAD` function. Absolute paths must be specified for the path names of the audit trail files.
2. The file names in the path names of audit trail files can include the following special characters:

   - `*` (asterisk)

     This denotes any character string consisting of 0 or more characters.
   - `?` (question mark)

     This denotes any single character.

   Specification example 1:

   ```
   ADB_AUDITREAD(MULTISET['/audit/*.aud'])
   ```

   In the preceding example, all audit trail files in the `/audit` directory (files with the extension `aud`) are used as the input information for the `ADB_AUDITREAD` function.

   Specification example 2:

   ```
   ADB_AUDITREAD(MULTISET['/audit/adbaud-201704*.aud','/audit/adbaud-201705*.aud'])
   ```

   In the preceding example, any audit trail files having such names as the following are used as the input information for the `ADB_AUDITREAD` function:

   - `/audit/adbaud-20170401-123000-159.aud`
   - `/audit/adbaud-20170415-123000-952.aud`
   - `/audit/adbaud-20170501-123000-599.aud`

Important

The asterisks (*) and question marks (?) included in the names of audit trail files are handled as special characters. The asterisks (*) and question marks (?) included in directory names in the path names of audit trail files are handled as ordinary characters.

Example:

```
ADB_AUDITREAD(MULTISET['/audit*/adbaud-201706*.aud'])
```

In the preceding example, the asterisk in the directory name is handled as an ordinary character. Therefore, /audit* is handled as a directory name. However, the asterisk in the file name is handled as a special character. Therefore, the preceding audit trail file path name specification can denote such files as follows:

- /audit*/adbaud-20170601-123000-159.aud
- /audit*/adbaud-20170602-165522-656.aud

3. If the audit trail file path name specification including special characters denotes no existing audit trail file that can be used as the input information (*audit-trail-file-path-name-specification* results in an empty set), the SQL statement results in an error.
4. If the audit trail file path name specification includes special characters, the following audit trail files are not used as the input information for the ADB_AUDITREAD function:
   - Current audit trail file
   - Audit trail files in the subdirectories of directories included in the audit trail file path name specification
5. If the audit trail file path name specification including special characters denotes 65,536 or more files, the SQL statement results in an error. Note that the current audit trail file, which is not used as input information, is excluded from the files that can be denoted by an audit trail file path name specification that includes special characters.
6. The SQL statement results in an error if the specified files include files other than the following files:
   - Audit trail files
   - Archive files in which audit trail files are compressed by the OS command gzip

   This rule is checked during execution of the SQL statement, not during preprocessing of the SQL statement.
7. The current audit trail file cannot be specified as an audit trail file path name.
8. The spaces at the beginning and end of an audit trail file path name are not handled as the part of the path name.

   Examples:

   'ΔΔΔ/audit/adbaud-20170420-123030-159.aud' → '/audit/adbaud-20170420-123030-159.aud'

   '/audit/adbaud-20170420-123030-159.audΔΔΔ' → '/audit/adbaud-20170420-123030-159.aud'

   'ΔΔΔ/audit/adbaud-20170420-123030-159.audΔΔΔ' → '/audit/adbaud-20170420-123030-159.aud'

   'ΔΔΔ/audit/adbaud-20170420Δ-123030-159.audΔΔΔ' → '/audit/adbaud-20170420Δ-123030-159.aud'

   Δ: Space

Important

Do not specify spaces at the beginning and end of an audit trail file path name. If you specify a path name that begins or ends with spaces, the spaces are removed. Therefore, the specified path name might be changed to an unintended path name.

9. The maximum length of each audit trail file path name is 1,024 bytes. If an audit trail file path name that is 1,025 or more bytes long is specified, the SQL statement results in an error. Note that the path name length check takes place after the HADB server performs the following processing:
   - Deleting any spaces at the beginning and end of audit trail file path names
   - Replacing any audit trail file path names (denoted by the specification including special characters) with their actual path names of audit trail files that are used as input information
10. If the `ADB_AUDITREAD` function is specified in the `CREATE VIEW` statement, the audit trail file path names are not checked when the `CREATE VIEW` statement is executed. The audit trail file path names are checked when an SQL statement in which a defined viewed table is specified is executed. If there are path names that violate the rules, the SQL statement results in an error.

(5) Notes

1. Before HADB administrators can access all directories included in the absolute paths of all audit trail files, set read and execution privileges for those directories. For example, if audit trail files are stored in the `/adbmanager/audit` directory, set read and execution privileges for the `/`, `/adbmanager`, and `/adbmanager/audit` directories so that HADB administrators can access these directories. Also, set read privilege for the audit trail files so that HADB administrators can access them.
2. If an SQL statement in which the `ADB_AUDITREAD` function is specified is executed, the HADB server opens the audit trail files specified by *audit-trail-file-path-name-specification* to read the audit trails. Therefore, during execution of an SQL statement in which the `ADB_AUDITREAD` function is specified, do not move or delete any files specified by *audit-trail-file-path-name-specification*.
3. If the path name of an audit trail file includes special characters, the path names of search-target audit trail files are extracted during the preprocessing of the SQL statement. If audit trail files that are extracted during the preprocessing of the SQL statement do not exist during execution of the SQL statement, the files are not to be searched (the SQL statement does not result in an error).
4. Some of the rules for specifying the `ADB_AUDITREAD` function are checked during execution (rather than the preprocessing) of the SQL statement. The descriptions of the rules to be checked during execution of the SQL statement include the sentence *This rule is checked during execution of the SQL statement, not during preprocessing of the SQL statement*.

(6) Examples

Example 1

Output a list of HADB users who accessed the HADB server in the term from April 1, 2017 to April 30, 2017. Assume that the audit trail files containing the audit trails that were output in the term from April 1, 2017 to April 30, 2017 are stored in the `/audit` directory.

```
SELECT DISTINCT "USER_NAME"
    FROM TABLE(ADB_AUDITREAD(MULTISET['/audit/*.aud'])) "DT"
       WHERE "EXEC_TIME" BETWEEN TIMESTAMP'2017/04/01 00:00:00.000000'
                             AND TIMESTAMP'2017/04/30 23:59:59.999999'
```

In the preceding example, the underlined portion indicates the specification of the `ADB_AUDITREAD` function.

USER_NAME stores the authorization identifier of an HADB user. EXEC_TIME stores the time at which the HADB user performed an operation.

Example 2

Output a list of HADB users who accessed the HADB server in the term from April 1, 2017 to April 30, 2017. Assume that the audit trails that were output in the term from April 1, 2017 to April 30, 2017 are stored in the directory specified for the adb_audit_log_path operand in the server definition.

```
SELECT DISTINCT "USER_NAME"
    FROM TABLE(ADB_AUDITREAD()) "DT"
       WHERE "EXEC_TIME" BETWEEN TIMESTAMP'2017/04/01 00:00:00.000000'
                             AND TIMESTAMP'2017/04/30 23:59:59.999999'
```

In the preceding example, the underlined portion indicates the specification of the ADB_AUDITREAD function.

7.15.3 ADB_CSVREAD function

Converts the data in a CSV file into a table format such that the data can be retrieved by the HADB server.

Note

- For an overview of retrieving data from CSV files, see *Retrieving data from CSV files* in the *HADB Setup and Operation Guide*.
- For details about the operations for retrieving data from CSV files, see *Handling of data retrieval from CSV files* in the *HADB Setup and Operation Guide*.

(1) Specification format

```
ADB_CSVREAD-function ::=
    [MASTER.]ADB_CSVREAD(CSV-file-path-name-specification,function-option-specificati
on)

        CSV-file-path-name-specification ::= multiset-value-expression

        function-option-specification ::= 'function-option[;function-option]...[;]'
          function-option ::= {compression-format-option|specification-column-option
                              |binary-string-format-option|enclosing-character-specif
ication-option
                              |delimiting-character-specification-option}
```

(2) Explanation of specification format

CSV-file-path-name-specification:

Specifies the path names of the CSV files containing the input data for the ADB_CSVREAD function. The path names are specified in the form of a multiset value expression. For details about multiset value expressions, see 7.16 Multiset value expression.

The following rules apply:

- The data type of the result of the multiset value expression must be character string data.
- Absolute paths must be specified for the path names of the CSV files specified in *CSV-file-path-name-specification*.

  This rule is checked during execution of the SQL statement, not during preprocessing of the SQL statement.

- Existing files must be specified in the path names of the CSV files specified in *CSV-file-path-name-specification*. This rule is checked during execution of the SQL statement, not during preprocessing of the SQL statement.

function-option-specification:

Specifies one or more of the following options to the `ADB_CSVREAD` function:

- The compression format option
- The specification column option
- The binary string format option
- The enclosing character specification option
- The delimiting character specification option

The details of each option are described in (3) Compression format option and the following subsections.

The following rules apply:

- *function-option-specification* is specified in the form of a character string literal. For details about the description format of character string literals, see 6.3.2 Description format of literals.
- The entire *function-option-specification* must be enclosed in single quotation marks (`'`).
- If multiple function options are specified, they must be separated by semicolons (`;`).
- Function options can be specified in any order.
- Function options must be unique.
- The half-width lowercase letters specified in function options are treated as half-width uppercase letters. However, for enclosing characters and delimiting characters, half-width lowercase letters and half-width uppercase letters are distinguished.
- A separator can be specified before and after each option and special character (`,`, `-`, `:`, `;`, `=`, `NL`, `CR`, `half-width space`, and `full-width space`).

(3) Compression format option

The compression format option specifies the compression format of the CSV files. The compression format option cannot be omitted.

(a) Specification format

```
COMPRESSION_FORMAT= {GZIP | NONE}
```

(b) Explanation of specification format

`GZIP`:

Specify this keyword if the CSV file is compressed in GZIP format.

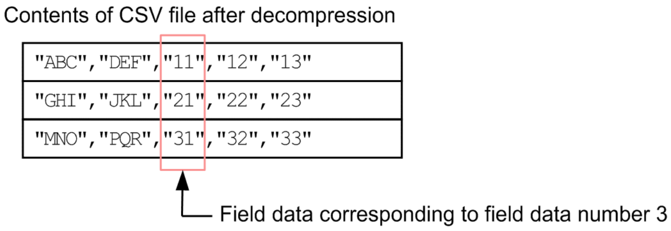
`NONE`:

Specify this keyword if the CSV file is not compressed.

(4) Specification column option

The specification column option specifies the field data numbers of field data in CSV files. *Field data numbers* are numbers representing the order of the field data in the CSV file. Field data number 1 (field 1 data) corresponds to data in the first field in the records, field data number 2 (field 2 data) corresponds to data in the second field, and so on.

Example:



Contents of CSV file after decompression

```
"ABC","DEF","11","12","13"
"GHI","JKL","21","22","23"
"MNO","PQR","31","32","33"
```

The ADB_CSVREAD function extracts the field data corresponding to the field data number specified here.

(a) Specification format

```
FIELD_NUM=filed-data-number-specification[,filed-data-number-specification]...
```

(b) Explanation of specification format

filed-data-number-specification:

Specifies the field data numbers of the field data to be extracted.

To specify multiple field data numbers, separate them with commas (`,`). You can also specify a range of field data numbers such as `1-5`.

Example:

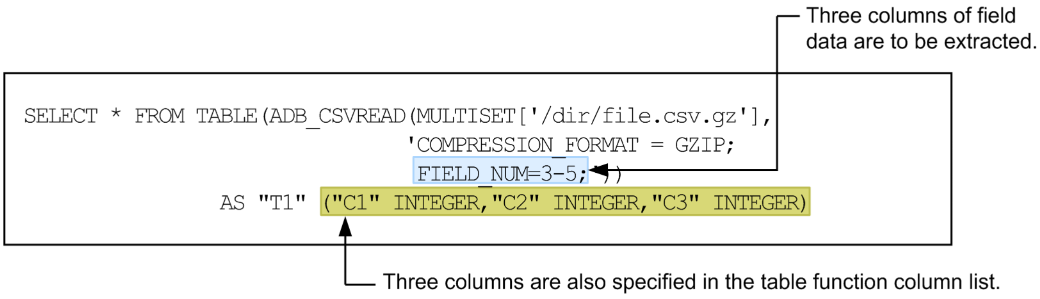
```
FIELD_NUM=3            ...1
FIELD_NUM=1,3,4,6      ...2
FIELD_NUM=1,3-5,8-10   ...3
```

1. The field data in field 3 will be extracted.
2. The field data in fields 1, 3, 4, and 6 will be extracted.
3. The field data in fields 1, 3 to 5, and 8 to 10 will be extracted.

(c) Rules

1. The number of columns of field data to be extracted must be the same as the number of columns in the table function column list.

   Example:



```
SELECT * FROM TABLE(ADB_CSVREAD(MULTISET['/dir/file.csv.gz'],
                    'COMPRESSION_FORMAT = GZIP;
                     FIELD_NUM=3-5;'))
          AS "T1" ("C1" INTEGER,"C2" INTEGER,"C3" INTEGER)
```

Three columns of field data are to be extracted.
Three columns are also specified in the table function column list.

   For details about table function column lists, see (2) Explanation of specification format in 7.11.1 Specification format for table references.

2. Specify field data numbers as unsigned integer literals from 1 to 30,000.
3. Field data numbers must be unique and cannot overlap.

Examples that generate errors:

```
FIELD_NUM=1,2,2 ← Duplication of 2 causes an error.
FIELD_NUM=1,1-3 ← Duplication of 1 causes an error.
```

4. No more than 1,000 columns of field data can be targeted.

   Example that generates an error:

```
FIELD_NUM=1-1001 ← Having too many (1,001) field data columns targeted causes an e
rror.
```

5. An error results if there is no field data corresponding to a field data number. For example, the code fragments below generate an error when there are five columns of field data.

   Examples that generate errors:

```
FIELD_NUM=6
FIELD_NUM=1-7
```

   Because there are only five columns of field data, you cannot specify field data number 6 or higher.

   Note that this rule is checked during execution of the SQL statement, not during preprocessing of the SQL statement.

6. If *specification-column-option* is omitted, the field data numbers from 1 to the number of columns in the table function column list are assumed. In cases where there is no field data corresponding to a field data number, the null value is stored in the table function derived table.

   Example:

```
SELECT * FROM TABLE(ADB_CSVREAD(MULTISET['/dir/file.csv.gz'],
                              'COMPRESSION_FORMAT = GZIP;'))
               AS "T1" ("C1" INTEGER,"C2" INTEGER,"C3" INTEGER,
                        "C4" INTEGER,"C5" INTEGER)
```

   The underlined portion is the table function column list specification.

Contents of CSV file (`/dir/file.csv.gz`) after decompression

"11","12","13"
"21","22","23"
"31","32","33"

Table function derived table

Col. C1	Col. C2	Col. C3	Col. C4	Col. C5
11	12	13	NULL	NULL
21	22	23	NULL	NULL
31	32	33	NULL	NULL

There are three columns of field data in the CSV file, but five columns in the table function column list. As a result, null values are stored in columns `C4` and `C5` of the table function derived table.

(5) Binary string format option

The binary string format option specifies the format of binary data (`BINARY` or `VARBINARY`) in the CSV file.

(a) Specification format

```
BINARY_STRING_FORMAT=filed-data-number-specification:binary-format-specification
                  [,filed-data-number-specification:binary-format-specification]...

  binary-format-specification ::= {HEX | BIN}
```

(b) Explanation of specification format

filed-data-number-specification:

Specifies the field data numbers of the binary data in the CSV file. For the field data number specification rules, see (4) Specification column option.

The field data numbers specified here must be among the field data numbers specified in the specification column option (`FIELD_NUM`).

Example:

```
FIELD_NUM=1-5;BINARY_STRING_FORMAT=1:BIN,4-5:HEX;
```

If the specification column option is omitted, specify integers that are less than or equal to the number of columns in the table function derived table.

binary-format-specification:

Specifies the format of the binary data.

`HEX`:

Specify this when the binary data is in hexadecimal format.

`BIN`:

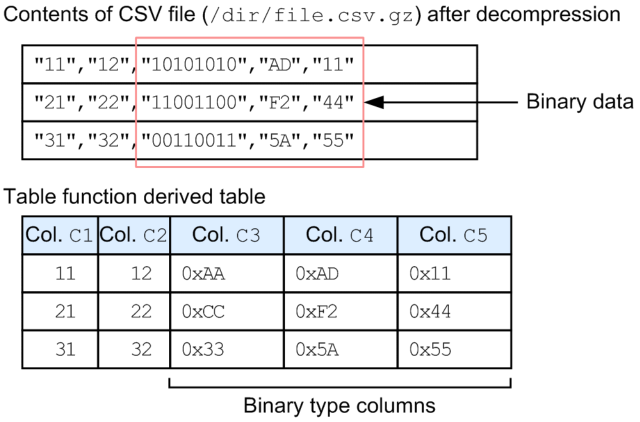
Specify this when the binary data is in binary format.

The following is an example of specifying the binary string format option.

Example:

```
SELECT * FROM TABLE(ADB_CSVREAD(MULTISET['/dir/file.csv.gz'],
                                'COMPRESSION_FORMAT = GZIP;
                                 BINARY_STRING_FORMAT=3:BIN,4:HEX;'))
                AS "T1"  ("C1" INTEGER,"C2" INTEGER,
                          "C3" BINARY(1),"C4" BINARY(1),"C5" BINARY(1))
```

The underlined portion is the binary string format option specification.



Contents of CSV file (`/dir/file.csv.gz`) after decompression

```
"11","12","10101010","AD","11"
"21","22","11001100","F2","44"
"31","32","00110011","5A","55"
```

Table function derived table

Col. C1	Col. C2	Col. C3	Col. C4	Col. C5
11	12	0xAA	0xAD	0x11
21	22	0xCC	0xF2	0x44
31	32	0x33	0x5A	0x55

Explanation

- The field data in columns `C3` to `C5` is binary data.
- Because the binary data in column `C3` is in binary format, `BIN` is specified in the binary format specification.
- Because the binary data in column `C4` is in hexadecimal format, `HEX` is specified in the binary format specification.
- Because the binary data in column `C5` is in hexadecimal format, the binary format specification can be omitted (`HEX` is the default value).

(c) Rules

1. When binary data columns are specified in the table function derived table and the binary string format option is omitted, the following specifications are assumed:
   - The field data numbers corresponding to the binary data in the table function derived table are assumed for the field data number specification
   - `HEX` is assumed for the binary format specification
2. The data types of the columns of the table function derived table corresponding to the field data numbers must be binary (`BINARY` or `VARBINARY`).

(6) Enclosing character specification option

The enclosing character specification option specifies the enclosing character that is used to enclose field data items in the CSV file.

(a) Specification format

```
ENCLOSING_CHAR={enclosing-character | NONE}
```

(b) Explanation of specification format

enclosing-character:

Specifies the enclosing character that is to be used to enclose field data items in the CSV file. You can specify a single-byte character for *enclosing-character*.

Note the following points concerning the enclosing character:

- Characters such as the following are not suitable as the enclosing character because they are likely to overlap with characters in the field data in the CSV file:

  Sign (`+`, `-`), forward slash (`/`), colon (`:`), period (`.`), `|`, `\`, `[`, `]`, `(`, `)`, `{`, `}`, `~`
- Do not specify the same character for *enclosing-character* as the character used for the separator. The separator character will not be recognized as an enclosing character (it is treated as the separator). As a result, if you specify the separator character as the enclosing character, there is a risk of unintended consequences as in the following example.

  Example where a single-byte space, which is the separator, is specified as the enclosing character (Δ represents a single-byte space)

  ```
  '...;ENCLOSING_CHAR=Δ;'
  ```

  In this example, HADB assumes that the semicolon (`;`) is specified as the enclosing character.

`NONE`:

Specify `NONE` if no enclosing character is used in the field data in the CSV file.

Important

Do not specify `NONE` if there are newline characters or the same character as that specified for the delimiting character in the field data. Specifying `NONE` might produce unintended consequences.

- If the field data contains a newline character, the characters to the newline character will be treated as a single line of data.

- If the field data contains the same character as that specified for the delimiting character, it will be treated as a delimiting character, not field data.

(c) Rules

1. If *enclosing-character-specification-option* is omitted, the double quotation mark (`"`) is assumed as the enclosing character.
2. The following characters are not permitted as the enclosing character:
   - Space, tab, asterisk (`*`), newline (`0x0A`), carriage return (`0x0D`)
   - The delimiting character specified in *delimiting-character-specification-option*
3. To define a single quotation mark (`'`) as an enclosing character, specify two single quotation marks. The specification is as follows:

```
ENCLOSING_CHAR=''
```

(7) Delimiting character specification option

The delimiting character specification option specifies the delimiting character that is used to delimit field data items in the CSV file.

(a) Specification format

```
DELIMITER_CHAR={delimiting-character | TAB | SP}
```

(b) Explanation of specification format

delimiting-character:

Specifies the delimiting character that is to be used to delimit field data items in the CSV file. You can specify a single-byte character for *delimiting-character*.

Note the following points concerning the delimiting character:

- Characters such as the following are not suitable as the delimiting character because they are likely to overlap with characters in the field data in the CSV file:

  Sign (`+`, `-`), forward slash (`/`), colon (`:`), period (`.`), `|`, `\`, `[`, `]`, `(`, `)`, `{`, `}`, `~`
- Do not specify the same character for *delimiting-character* as the character used for the separator. The separator character will not be recognized as a delimiting character (it is treated as the separator). As a result, if you specify the separator character as the delimiting character, there is a risk of unintended consequences as in the following example.

  Example where a single-byte space, which is the separator, is specified as the delimiting character (Δ represents a single-byte space)

  ```
  '...;DELIMITER_CHAR=Δ;'
  ```

  In this example, HADB assumes that the semicolon (`;`) is specified as the delimiting character.

`TAB`:

Specify `TAB` when the field data in the CSV file is delimited by tabs.

`SP`:

Specify `SP` when the field data in the CSV file is delimited by spaces.

(c) Rules

1. If *delimiting-character-specification-option* is omitted, the comma (,) is assumed as the delimiting character.
2. The following characters are not permitted as the delimiting character:
   - Alphabetic characters (A to Z, a to z), digits (0 to 9), underscore (_), double quotation marks ("), space, tab, asterisk (*), newline (0x0A), carriage return (0x0D)
   - The enclosing character specified in *enclosing-character-specification-option*
3. To define a single quotation mark (') as a delimiting character, specify two single quotation marks. The specification is as follows:

```
DELIMITER_CHAR=''
```

(8) Rules

(a) Rules for the ADB_CSVREAD function

If the result of the multiset value expression specified in the CSV file path name specification is the empty set, the result of the table function derived table will be the empty set.

(b) Rules for CSV files

1. Each CSV file must be one of the following types:
   - Files compressed in GZIP format by using the gzip command of the OS
   - Output data files exported in GZIP format by using the adbexport command
   - CSV files that are not compressed

   This rule is checked during execution of the SQL statement, not during preprocessing of the SQL statement.
2. The HADB administrator must have read privileges for the CSV files. Grant the HADB administrator read and execute privileges to the directories where the CSV files are stored.

   This rule is checked during execution of the SQL statement, not during preprocessing of the SQL statement.
3. Leading or trailing spaces around CSV file path names are removed before the files are processed.

   Examples:

```
'ΔΔΔ/dir/file.csv.gz' → '/dir/file.csv.gz'
'/dir/file.csv.gzΔΔΔ' → '/dir/file.csv.gz'
'ΔΔΔ/dir/file.csv.gz ΔΔΔ' → '/dir/file.csv.gz'
'ΔΔΔ/dir/fiΔ le.csv.gzΔΔΔ' → '/dir/fiΔle.csv.gz'
```

   Δ: Space

   > **Important**
   >
   > Do not specify a CSV file path name that begins or ends with spaces. If you specify a path name that begins or ends with spaces, the spaces are removed. Therefore, the specified path name might be changed to an unintended path name.

4. The length of the path name of the CSV file must not exceed 510 bytes, excluding leading and trailing spaces around the path name.

   This rule is checked during execution of the SQL statement, not during preprocessing of the SQL statement.

(c) Rules for CSV file formats

1. Each line of the CSV file corresponds to one row of the table function derived table. Lines are terminated with the newline character `X'0A'` (LF), `X'0D0A'` (CRLF), or `X'00'`.
2. Specify the delimiting character to delimit field data items.
3. A character string surrounded by the enclosing character is treated as field data.
4. The data in the CSV file must use the character encoding specified in the environment variable `ADBLANG`.
5. Do not specify the `EOF` control character in the CSV file.
6. When specifying an enclosing character, specify the delimiting character and enclosing character contiguously, with no spaces between them. Spaces between the delimiting character and enclosing character will be treated as field data. As a result, the enclosing character might be treated as part of the field data, or an error might be generated due to invalid specification of the enclosing character.

   Note that this rule is checked during execution of the SQL statement, not during preprocessing of the SQL statement.
7. To specify the enclosing character inside field data, write it twice in a row.

   Example when the enclosing character is a single quotation mark (`'`):

   `'AB''CD'` (field data) → `AB'CD` (data stored in the table function derived table)
8. When specifying the enclosing character as the first character of field data (excluding leading single-byte spaces or tabs), do not omit the first enclosing character.

   Example when the enclosing character is a single quotation mark (`'`):

   `'''AB'` (field data) → `'AB` (data stored in the table function derived table)
9. To specify the delimiting character inside field data, you must surround the field data with the enclosing character. Otherwise, the character will be treated as a delimiting character rather than part of the field data, which might cause an error due to the fact that the specified field no longer exists.

   Examples with double quotation marks (`"`) as the enclosing character and the comma (`,`) as the delimiting character:

   ```
   1,"foo,bar",3
   ```

   In the above example, three columns of field data are recognized: `1`, `foo,bar`, and `3`.

   ```
   1,foo,bar,3
   ```

   In the above example, four columns of field data are recognized: `1`, `foo`, `bar`, and `3`.

   Note that this rule is checked during execution of the SQL statement, not during preprocessing of the SQL statement.
10. The table below shows examples of field data character strings and the corresponding data stored in the table function derived table. In these examples, the comma (`,`) is used as the delimiting character.

Field data character string	Data stored in the table function derived table	
	With double quotation marks (") specified in the enclosing character specification option	With NONE specified in the enclosing character specification option
`ABC,DEF`	• `ABC` • `DEF`	• `ABC` • `DEF`
`"ABC""","DEF"`	• `ABC"` • `DEF`	• `"ABC"""` • `"DEF"`
`"ABC,DEF"`	• `ABC,DEF`	• `"ABC` • `DEF"`
`"ABC,DEF`	Error	• `"ABC`

Field data character string	Data stored in the table function derived table	
	With double quotation marks (") specified in the enclosing character specification option	With NONE specified in the enclosing character specification option
		• DEF

11. The field data in the CSV file is converted to the data corresponding to the data type of the respective column of the table function derived table. The data types of the columns of the table function derived table must therefore be compatible with the description format of the field data. For details about the field data description rules, see (4) Storage assignment to a table function derived table (in the case of the ADB_CSVREAD function) in 6.2.2 Data types that can be converted, assigned, and compared.

    Note that this rule is checked during execution of the SQL statement, not during preprocessing of the SQL statement.

(9) Notes

1. When an SQL statement in which the ADB_CSVREAD function is specified is executed, the HADB server opens the CSV file to read data. The CSV file must therefore not be edited during execution of the SQL statement.
2. No field data is extracted into any columns specified in the table function column list that do not affect the retrieval results (the columns not used in the query). The specifications pertaining to such columns are automatically removed from the SQL statement.

   Example:

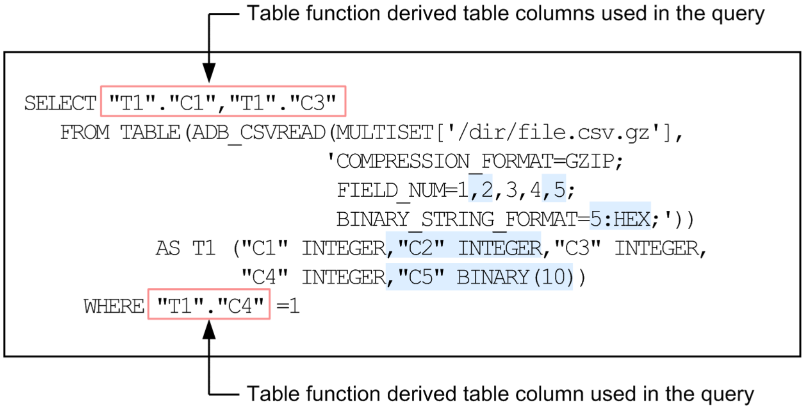


Table function derived table columns used in the query
Table function derived table column used in the query

```
SELECT "T1"."C1","T1"."C3"
   FROM TABLE(ADB_CSVREAD(MULTISET['/dir/file.csv.gz'],
                       'COMPRESSION_FORMAT=GZIP;
                        FIELD_NUM=1,2,3,4,5;
                        BINARY_STRING_FORMAT=5:HEX;'))
           AS T1 ("C1" INTEGER,"C2" INTEGER,"C3" INTEGER,
                  "C4" INTEGER,"C5" BINARY(10))
     WHERE "T1"."C4" =1
```

   In the above example, columns C2 and C5 do not affect the retrieval results. Before execution, the SELECT statement is therefore converted to a statement in which the specifications pertaining to columns C2 and C5 are removed. Specifically, the portions that are shaded in blue are deleted when the SELECT statement is run.

Note

- The following specifications are targeted for removal:
  - Columns specified in the table function column list
  - Specifications of field data numbers in the specification column option
  - Binary string format option specifications
- Once the specifications pertaining to the extraneous columns are removed, only the field data for the remaining columns is targeted for extraction. Furthermore, only the field data for the remaining columns is subject to the rules pertaining to CSV files described above.
- If the specifications of all the columns in the table function derived table are targeted for removal, the specifications of all the columns corresponding to the field data numbers specified in the specification column option will be removed, except for the lowest-numbered one.

3. Some of the rules for specifying the ADB_CSVREAD function are checked during execution of the SQL statement (not during preprocessing of the SQL statement). The descriptions of these rules above include the sentence *This rule is checked during execution of the SQL statement, not during preprocessing of the SQL statement*.

(10) Examples

Example 1

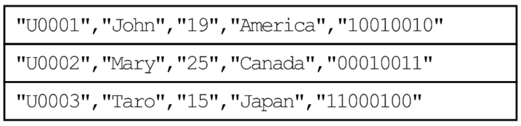
Extract the following data from a CSV file (/dir/file.csv.gz) compressed in GZIP format:

- Customer ID (USERID)
- Customer name (NAME)
- Age (AGE)

```
SELECT "USERID","NAME","AGE"
    FROM TABLE(ADB_CSVREAD(MULTISET ['/dir/file.csv.gz'],
                           'COMPRESSION_FORMAT=GZIP;'))
         AS "USERSLIST" ("USERID" CHAR(5),
                          "NAME" VARCHAR(100),
                          "AGE" INTEGER,
                          "COUNTRY" VARCHAR(100),
                          "INFORMATION" VARBINARY(10))
```

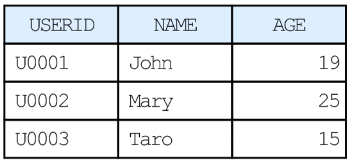
The underlined portion indicates the specification of the ADB_CSVREAD function.

Contents of CSV file (/dir/file.csv.gz) after decompression



"U0001","John","19","America","10010010"
"U0002","Mary","25","Canada","00010011"
"U0003","Taro","15","Japan","11000100"

Retrieval results



USERID	NAME	AGE
U0001	John	19
U0002	Mary	25
U0003	Taro	15

Example 2

Extract the following data from a CSV file (/dir/file.csv.gz) compressed in GZIP format:

- Customer name (NAME)
- Country of origin (COUNTRY)
- Various bit flags (INFORMATION)

```
SELECT "NAME","COUNTRY",BIN("INFORMATION")
    FROM TABLE(ADB_CSVREAD(MULTISET ['/dir/file.csv.gz'],
                           'COMPRESSION_FORMAT=GZIP;
                            FIELD_NUM=2,4,5;
                            BINARY_STRING_FORMAT=5:BIN;
                            ENCLOSING_CHAR=";
                            DELIMITER_CHAR=,;'))
         AS "USERSLIST" ("NAME" VARCHAR(100),
            "COUNTRY" VARCHAR(100),
            "INFORMATION" VARBINARY(10))
```

The underlined portion indicates the specification of the ADB_CSVREAD function.

Contents of CSV file (/dir/file.csv.gz) after decompression

"U0001","John","19","America","10010010"
"U0002","Mary","25","Canada","00010011"
"U0003","Taro","15","Japan","11000100"

Retrieval results

NAME	COUNTRY	INFORMATION
John	America	10010010
Mary	Canada	00010011
Taro	Japan	11000100

Example 3

Extract the following data from a CSV file (/dir/file.csv):

- Customer ID (USERID)
- Customer name (NAME)
- Age (AGE)

```
SELECT "USERID","NAME","AGE"
    FROM TABLE(ADB_CSVREAD(MULTISET ['/dir/file.csv'],
                           'COMPRESSION_FORMAT=NONE;'))
         AS "USERSLIST" ("USERID" CHAR(5),
                         "NAME" VARCHAR(100),
                         "AGE" INTEGER,
                         "COUNTRY" VARCHAR(100),
                         "INFORMATION" VARBINARY(10))
```

The underlined portion indicates the specification of the ADB_CSVREAD function.

Contents of CSV file (/dir/file.csv)

"U0001","John","19","America","10010010"
"U0002","Mary","25","Canada","00010011"
"U0003","Taro","15","Japan","11000100"

Retrieval results

USERID	NAME	AGE
U0001	John	19
U0002	Mary	25
U0003	Taro	15

7.16 Multiset value expression

This section describes multiset value expressions.

7.16.1 Specification format and rules for multiset value expressions

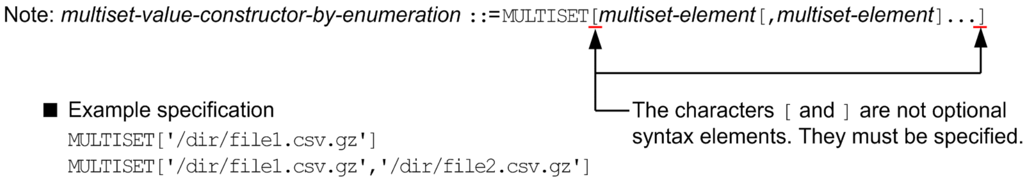
A multiset value expression is used to collect multiple element values into a single data set. A multiset value expression can be specified in the following locations:

- *audit-trail-file-path-name-specification* in the `ADB_AUDITREAD` function
  For details about the `ADB_AUDITREAD` function, see 7.15.2 ADB_AUDITREAD function.
- *CSV-file-path-name-specification* in the `ADB_CSVREAD` function
  For details about the `ADB_CSVREAD` function, see 7.15.3 ADB_CSVREAD function.

(1) Specification format

```
multiset-value-expression ::= {multiset-value-constructor-by-enumeration | multiset-v
alue-constructor-by-query}

  multiset-value-constructor-by-enumeration ::= MULTISET[multiset-element[,multiset-e
lement]...]
  multiset-value-constructor-by-query ::= MULTISET table-subquery
```


Note: *multiset-value-constructor-by-enumeration* ::=`MULTISET[`*multiset-element*`[,`*multiset-element*`]...]`

The characters [and] are not optional syntax elements. They must be specified.

■ Example specification

```
MULTISET['/dir/file1.csv.gz']
MULTISET['/dir/file1.csv.gz','/dir/file2.csv.gz']
```

(2) Explanation of specification format

Important

To specify a multiset value expression in the `ADB_AUDITREAD` function, specify *multiset-value-constructor-by-enumeration*. You cannot specify *multiset-value-constructor-by-query*.

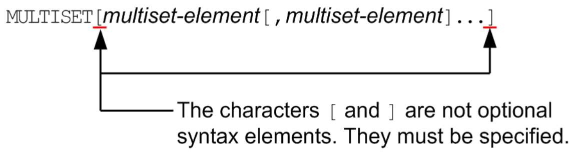
To specify a multiset value expression in the `ADB_CSVREAD` function, note the following:

- To specify individual CSV file names in the `ADB_CSVREAD` function, specify *multiset-value-constructor-by-enumeration*.
- To use a table subquery to determine the CSV file names to be specified in the `ADB_CSVREAD` function, specify *multiset-value-constructor-by-query*.

- *multiset-value-constructor-by-enumeration*

  MULTISET[*multiset-element*[,*multiset-element*]...]:

Important



MULTISET[*multiset-element*[,*multiset-element*]...]

The characters [and] are not optional syntax elements. They must be specified.

■ To specify *multiset-value-constructor-by-enumeration* in the ADB_AUDITREAD function

For *multiset-element*, specify in the character string literal format the path names of the audit trail files to be specified in the ADB_AUDITREAD function. For details about character string literals, see 6.3 Literals.

The following is an example:

```
MULTISET['/audit/adbaud-201707*.aud','/audit/adbaud-201708*.aud']
```

The following rules apply:

- A maximum of 1,000 multiset elements (path names of audit trail files) can be specified.

■ To specify *multiset-value-constructor-by-enumeration* in the ADB_CSVREAD function

multiset-element specifies the path name of a CSV file to be specified in the ADB_CSVREAD function in the form of a character string literal. For details about character string literals, see 6.3 Literals.

The following is an example:

```
MULTISET['/dir/file1.csv.gz','/dir/file2.csv.gz','/dir/file3.csv.gz']
```

The example above specifies three CSV files.

The following rules apply:

- No more than 1,000 multiset elements (CSV file path names) can be specified.

- *multiset-value-constructor-by-query*

  MULTISET *table-subquery*:

Specifies the path names of the CSV files to be specified in the ADB_CSVREAD function in the form of a table subquery. For details about table subqueries, see 7.3 Subqueries.

The following is an example:

```
MULTISET (SELECT "FILE_NAME" FROM "FILELIST"
              WHERE "FILE_DATE" BETWEEN '2012/01/01' AND '2012/12/31')
```

The above example specifies CSV file names (FILE_NAME) for which the FILE_DATE column in the file management table (FILELIST) is between 2012/01/01 and 2012/12/31.

The following rules apply:

- The result of the table subquery must be one column.
- The table subquery cannot contain an external reference column.

  Example that generates an error:

  The underlined portion indicates the external reference column specification.

```
SELECT * FROM "T0"
  WHERE EXISTS (SELECT * FROM "T1",
                  TABLE(ADB_CSVREAD(MULTISET (SELECT "T2"."C1"
                                                FROM "T2"
```

```
                                                  WHERE "T2"."C2" = "T0"."C2"
),
                                   'COMPRESSION_FORMAT=GZIP;'))
                  AS "TF1" ("TFC1" INTEGER,"TFC2" VARCHAR(32)))
```

(3) Examples

Example 1: To specify the path names of audit trail files in the `ADB_AUDITREAD` function

Output a list of HADB users who accessed the HADB server in the term from April 1, 2017 to April 30, 2017. Assume that the audit trails that were output in the term from April 1, 2017 to April 30, 2017 are stored in the `/audit` directory.

```
SELECT DISTINCT "USER_NAME"
    FROM TABLE(ADB_AUDITREAD(MULTISET['/audit/*.aud'])) "DT"
       WHERE "EXEC_TIME" BETWEEN TIMESTAMP'2017/04/01 00:00:00.000000'
                             AND TIMESTAMP'2017/04/30 23:59:59.999999'
```

In the preceding example, the underlined portion indicates a multiset value expression (*multiset-value-constructor-by-enumeration*).

Example 2: To specify the path names of CSV files in the `ADB_CSVREAD` function

Extract the following data from the GZIP-compressed CSV files `/dir/file1.csv.gz`, `/dir/file2.csv.gz`, and `/dir/file3.csv.gz`:

- Customer ID (`USERID`)
- Customer name (`NAME`)
- Age (`AGE`)

```
SELECT "USERID","NAME","AGE"
    FROM TABLE(ADB_CSVREAD(MULTISET ['/dir/file1.csv.gz','/dir/file2.csv.gz','/dir
/file3.csv.gz'],
                          'COMPRESSION_FORMAT=GZIP;'))
         AS "USERSLIST" ("USERID" CHAR(10),"NAME" VARCHAR(100),"AGE" INTEGER)
```

In the preceding example, the underlined portion indicates a multiset value expression (*multiset-value-constructor-by-enumeration*).

Example 3: To use a table subquery to specify the path names of CSV files in the `ADB_CSVREAD` function

Extract customer information data that was registered in 2010. When the data is extracted, the following conditions hold:

- The customer information data is stored in CSV-format files.
- The CSV files are compressed in GZIP format.
- The CSV files are managed in the CSV file management table (`FILELIST`)
- The absolute path name (`FILE_NAME`) of each CSV file and the date each file was registered (`FILE_DATE`) are stored in the CSV file management table.

```
SELECT "USERID","NAME","AGE"
    FROM TABLE(ADB_CSVREAD(MULTISET (SELECT "FILE_NAME" FROM "FILELIST"
                                       WHERE "FILE_DATE" BETWEEN '2010/01/01'
                                                             AND '2010/12/31'),
                          'COMPRESSION_FORMAT=GZIP;'))
         AS "USERSLIST" ("USERID" CHAR(10),"NAME" VARCHAR(100),"AGE" INTEGER)
```

In the preceding example, the underlined portion indicates a multiset value expression (*multiset-value-constructor-by-query*).

7.17 Table value constructors

This section describes table value constructors.

7.17.1 Specification format and rules for table value constructors

For a table value constructor, specify the rows that make up a derived table (a set of row value constructors).

(1) Specification format

```
table-value-constructor ::= VALUES row-value-constructor[,row-value-constructor]...

  row-value-constructor ::= (row-value-constructor-element[,row-value-constructor-ele
ment]...)
    row-value-constructor-element ::= {value-specification | scalar-function-CAST | s
calar-function-CONVERT}
```

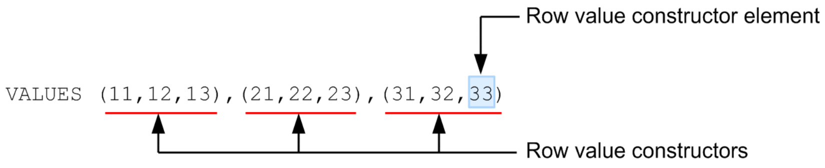
(2) Explanation of specification format

row-value-constructor:

```
row-value-constructor ::= (row-value-constructor-element[,row-value-constructor-el
ement]...)
  row-value-constructor-element ::= {value-specification | scalar-function-CAST |
scalar-function-CONVERT}
```

For a row value constructor, specify one or more row value constructor elements. The value of each row value constructor element becomes the value of each column on a row of a derived table.

Examples:



```
VALUES (11,12,13),(21,22,23),(31,32,33)
```

value-specification:

Specify row value constructor elements in the form of a value specification. For details about value specifications, see 7.21 Value specification.

scalar-function-CAST:

Specify row value constructor elements by using the scalar function `CAST`. For details about the scalar function `CAST`, see 8.12.3 CAST.

The following rules apply:

- For the data to be converted, only `NULL` or a dynamic parameter can be specified.

scalar-function-CONVERT:

Specify row value constructor elements by using the scalar function `CONVERT`. For details about the scalar function `CONVERT`, see 8.12.5 CONVERT.

The following rules apply:

- For the data to be converted, only `NULL` or a dynamic parameter can be specified.

- A format specification cannot be specified.

(3) Rules

1. Make sure that each row value constructor has the same number of row value constructor elements.

   Example of correct specification: `VALUES (11,12,13),(21,22,23),(31,32,33)`

   Example of incorrect specification: `VALUES (11,12,13),(21,22),(31,32,33,34)`

2. The i-th row value constructor elements of all row value constructors must have data types that can be compared mutually. For details about data types that can be compared, see (1) Data types that can be compared in 6.2.2 Data types that can be converted, assigned, and compared.

   Example of correct specification: `VALUES (11,12,13),(21.1,22.2,23.3),(1.0E+1,1.0E+2,1.0E+3)`

   Example of incorrect specification: `VALUES (11,12,13),('AB','CD',23)`

   Note, however, that the following items of data cannot be compared:

   - Date data and the predefined input representation of a date
   - Time data and the predefined input representation of a time
   - Time stamp data and the predefined input representation of a time stamp

3. The data type and length of the result for the i-th column derived by a table value constructor is determined by the data type of the i-th row value constructor element of each row value constructor. For details, see 7.20.2 Data types of the results of value expressions.
4. A maximum of 30,000 row value constructors can be specified.
5. The maximum total number of table value constructors and query specifications in one SQL statement is 1,024.
6. A maximum of 1,000 row value constructor elements can be specified in each row value constructor.
7. The dynamic parameter cannot be specified by itself as a row value constructor element.

(4) Examples

Example 1

In this example, you run the `SELECT` statement with table value constructors specified.

```
SELECT "C1","C2","C3" FROM (VALUES (11,12,13),
                                   (21,22,23)
                           ) AS "V1"("C1","C2","C3")
```

The underlined portion is the specification of table value constructors.

Example of execution results

C1	C2	C3
11	12	13
21	22	23

Example 2

In this example, you retrieve a list of customers (customer IDs and names) who have purchased a product whose product code (`PUR-CODE`) is `P001` (excluding duplicates) from the sales history table (`SALESLIST`) and customer table (`USERSLIST`) derived by table value constructors.

```
SELECT DISTINCT "USERSLIST"."USERID","NAME"
  FROM "SALESLIST"
         INNER JOIN
         (VALUES('U001','Maria'),('U002','Nancy')) AS "USERSLIST"("USERID","NAME")
```

```
            ON "USERSLIST"."USERID"="SALESLIST"."USERID"
        WHERE "SALESLIST"."PUR-CODE"='P001'
```

The underlined portion is the specification of table value constructors.

Example 3

In this example, you insert multiple data items into the customer table (`USERSLIST`).

```
INSERT INTO "USERSLIST"("USERID","AGE")
    SELECT * FROM (VALUES('USER001',10),('USER002',20))
```

The underlined portion is the specification of table value constructors.

7.18 Search conditions

This section describes search conditions.

7.18.1 Specification format and rules for search conditions

Search conditions specify criteria for retrieving data. A logical operation is performed based on the specified search conditions, and the system retrieves only those rows for which the result of the evaluation of the search conditions is `TRUE`. Search conditions can be specified in the following locations:

- `WHERE` clause
- `HAVING` clause
- `CASE` expression
- `ON` search condition of a joined table

(1) Specification format

```
search-condition ::= {[NOT] {(search-condition)|predicate|logical-value-specification
}
              |search-condition OR {(search-condition)|predicate|logical-value-specif
ication}
              |search-condition AND {(search-condition)|predicate|logical-value-speci
fication}}

  logical-value-specification ::= {TRUE|FALSE}
```

(2) Explanation of specification format

`NOT`:

If `NOT` is specified, values that do not satisfy the search conditions become the target of retrieval. For example, if you specify `NOT "USERID"='U00358'`, `USERID`s other than `U00358` are retrieved.

search-condition:

To specify multiple search conditions, connect the search conditions with `AND` or `OR`. A mixture of `AND`s and `OR`s can be specified. The meanings of `AND` and `OR` are as follows:

- *search-condition-1* `AND` *search-condition-2*

  Rows that satisfy both *search-condition-1* and *search-condition-2* will be subject to retrieval.

- *search-condition-1* `OR` *search-condition-2*

  Rows that satisfy either *search-condition-1* or *search-condition-2* will be subject to retrieval.

predicate:

For details about predicates, see 7.19 Predicates.

logical-value-specification:

`TRUE`: If `TRUE` is specified for *logical-value-specification*, the result of logical value specification is true.

`FALSE`: If `FALSE` is specified for *logical-value-specification*, the result of logical value specification is false.

The following are examples of specifying search conditions.

Examples:

C1, C2, and C3 are column names.

- Specification examples using comparison predicates

```
"C1">=100
"C1"=?
"C2"=CURRENT_DATE
SUBSTR("C3",2,3)='150'
```

- Specification examples using the IN predicate, BETWEEN predicate, LIKE predicate, and NULL predicate

```
"C1" IN (10,20)
"C1" BETWEEN 100 AND 200
"C3" LIKE 'M%'
"C3" IS NULL
```

- Examples specifying multiple search conditions

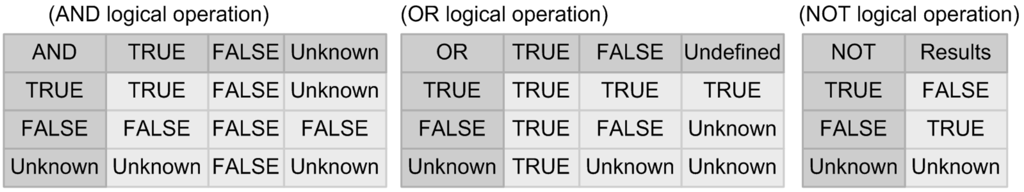
```
"C1"=100 AND "C2">=DATE'2011-09-06'
"C1" IN (10,20) AND "C2">=DATE'2011-09-06'
"C1"=10 OR "C1"=20
"C2">=DATE'2011-09-04' AND ("C1"=10 OR "C2"=20)
```

The order of evaluation of search conditions is: items inside parentheses, NOT, AND, OR.

(3) Rules

1. A maximum of 255 logical operations can be specified in an SQL search condition.
2. The following figure shows the results of performing each logical operation.

Figure 7-3: Results of performing logical operations



(AND logical operation)

AND	TRUE	FALSE	Unknown
TRUE	TRUE	FALSE	Unknown
FALSE	FALSE	FALSE	FALSE
Unknown	Unknown	FALSE	Unknown

(OR logical operation)

OR	TRUE	FALSE	Undefined
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	Unknown
Unknown	TRUE	Unknown	Unknown

(NOT logical operation)

NOT	Results
TRUE	FALSE
FALSE	TRUE
Unknown	Unknown

(4) Examples

Search conditions are illustrated in the following examples.

Example 1

From the sales history table (SALESLIST), this example retrieves the customer ID (USERID), product code (PUR-CODE), and date of purchase (PUR-DATE) for customers who purchased product code P001 or P003 on or after September 4, 2011.

```
SELECT "USERID","PUR-CODE","PUR-DATE"
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2011-09-04'
        AND ("PUR-CODE"='P001' OR "PUR-CODE"='P003')
```

The underlined portion indicates the search conditions specified in the WHERE clause.

Example 2

Using the data in the sales history table (SALESLIST), this example determines the sum and average of the quantities purchased on or after September 3, 2011 by product code (PUR-CODE).

Furthermore, retrieve only the product codes for which the sum of the quantities purchased is 20 or fewer.

```
SELECT "PUR-CODE",SUM("PUR-NUM"),AVG("PUR-NUM")
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2011-09-03'
        GROUP BY "PUR-CODE"
        HAVING SUM("PUR-NUM")<=20
```

The underlined portion indicates the search condition specified in the `HAVING` clause.

Example 3

Insert a row from the products table (`PRODUCTLIST`) into the new products table (`PRODUCTLIST_NEW`). When inserting the row, change the product prices (`PRICE`) as follows:

- If the product code (`PCODE`) is `P001`: reduce the price by 10%
- If the product code is `P002`: reduce the price by 20%
- Otherwise: reduce the price by 30%

```
INSERT INTO "PRODUCTLIST_NEW"("PCODE","PRICE")
    SELECT "PCODE",CASE WHEN "PCODE"='P001' THEN "PRICE"*0.9
                        WHEN "PCODE"='P002' THEN "PRICE"*0.8
                        ELSE "PRICE"*0.7
                   END
        FROM "PRODUCTLIST"
```

The underlined portions indicate the search conditions specified in the `CASE` expression.

7.19 Predicates

The following lists the predicates that can be used. This section describes the functions and specification formats of these predicates.

- `BETWEEN` predicate
- `EXISTS` predicate
- `IN` predicate
- `LIKE` predicate
- `LIKE_REGEX` predicate
- `NULL` predicate
- Comparison predicate
- Quantified predicate

These predicates can be specified in search conditions.

7.19.1 BETWEEN predicate

The `BETWEEN` predicate is used to search for data within a specific range.

(1) Specification format

```
BETWEEN-predicate ::= value-expression-1 [NOT] BETWEEN value-expression-2 AND value-e
xpression-3
```

(2) Explanation of specification format

value-expression-1:

Specifies the column to be evaluated by the `BETWEEN` predicate. It is specified in the form of a value expression. For details about value expressions, see 7.20 Value expression.

`NOT`:

If `NOT` is specified, values that do not satisfy the conditions specified following `BETWEEN` will become the target of retrieval.

`BETWEEN` *value-expression-2* `AND` *value-expression-3*:

Specify a search range by specifying the lower and upper limits. These are specified in the form of value expressions.

Specify the lower limit of the search range in *value-expression-2*, and the upper limit of the search range in *value-expression-3*.

(3) Evaluation of the predicate

The `BETWEEN` predicate is `TRUE` for those rows that satisfy the following condition:

value-expression-2 <= *value-expression-1* `AND` *value-expression-1* <= *value-expression-3*

The following `BETWEEN` predicates are equivalent:

- *value-expression-1* NOT BETWEEN *value-expression-2* AND *value-expression-3*
- NOT (*value-expression-1* BETWEEN *value-expression-2* AND *value-expression-3*)

(4) Rules

1. For *value-expression-1*, you cannot specify a value expression that is composed solely of a dynamic parameter.
2. The data types that can be specified for *value-expression-1*, *value-expression-2*, and *value-expression-3* are numeric data, character string data, and datetime data.
3. Specify *value-expression-1, value-expression-2* and *value-expression-3* such that the data types of the results of the three value expressions are all data types that can be compared. For details about data types that can be compared, see (1) Data types that can be compared in 6.2.2 Data types that can be converted, assigned, and compared.

   However, if you specify the predefined input representation of a date, time, or time stamp for *value-expression-1*, you cannot specify datetime data for *value-expression-2* and *value-expression-3*. For information about predefined input representations, see 6.3.3 Predefined character-string representations.

(5) Examples

Example 1

From the sales history table (SALESLIST), this example retrieves the customer ID (USERID), product code (PUR-CODE), and date of purchase (PUR-DATE) for customers who purchased products from September 4, 2011 to September 5, 2011.

```
SELECT "USERID","PUR-CODE","PUR-DATE"
  FROM "SALESLIST"
    WHERE "PUR-DATE" BETWEEN DATE'2011-09-04' AND DATE'2011-09-05'
```

The underlined portion indicates the BETWEEN predicate.

Example 2

From the sales history table (SALESLIST), this example retrieves the customer ID (USERID), product code (PUR-CODE), and date of purchase (PUR-DATE) for customers who purchased products on dates excluding September 4, 2011 to September 5, 2011.

```
SELECT "USERID","PUR-CODE","PUR-DATE"
  FROM "SALESLIST"
    WHERE "PUR-DATE" NOT BETWEEN DATE'2011-09-04' AND DATE'2011-09-05'
```

The underlined portion indicates the BETWEEN predicate.

7.19.2 EXISTS predicate

The EXISTS predicate is used to determine whether a table subquery result is zero rows (the empty set).

(1) Specification format

```
EXISTS-predicate ::= EXISTS table-subquery
```

(2) Explanation of specification format

table-subquery:

For details about table subqueries, see 7.3 Subqueries.

(3) Evaluation of the predicate

If the table subquery returns one or more rows, the result of the `EXISTS` predicate is `TRUE`. If the table subquery returns zero rows (the empty set), result of the `EXISTS` predicate is `FALSE`. The following table shows the results of the `EXISTS` predicate.

Table 7-3: Results of the EXISTS predicate

No.	Number of rows in the table subquery results	Result of the EXISTS predicate
1	One or more rows	`TRUE`
2	Zero rows (the empty set)	`FALSE`

(4) Rules

If `*` or *table-specification*`.*` is specified in the selection list in a table subquery, it makes sense to specify a column in the table specified by the table reference in the table subquery. On the other hand, it is not appropriate to specify a set function here.

(5) Example

Example

From the sales history table (`SALESLIST`) and product list table (`PRODUCTSLIST`), this example retrieves information on products for which there have been sales.

```
SELECT * FROM "PRODUCTSLIST"
    WHERE EXISTS(SELECT * FROM "SALESLIST"
            WHERE "SALESLIST"."PUR-CODE"="PRODUCTSLIST"."PUR-CODE")
```

The underlined portion indicates the `EXISTS` predicate.

7.19.3 IN predicate

The `IN` predicate is used for retrieving data that satisfies any one condition value out of multiple condition values.

(1) Specification format

```
IN-predicate ::= value-expression-1 [IS] [NOT] IN {(value-expression-2[,value-express
ion-3]...)|table-subquery}
```

(2) Explanation of specification format

value-expression-1:

Specifies the column to be evaluated by the `IN` predicate. It is specified in the form of a value expression. For details about value expressions, see 7.20 Value expression.

`IS`:

`IS` can be omitted. The results will be the same regardless of whether it is specified.

`NOT`:

If `NOT` is specified, values that do not match the value expressions specified following `IN` will become the target of retrieval.

`IN (`*value-expression-2*`[,` *value-expression-3*`]...)`:

The condition values are specified in the form of value expressions. If any of the specified condition values match the result of *value-expression-1*, the `IN` predicate is `TRUE`.

`IN` *table-subquery*:

Specifies a table subquery. For details about table subqueries, see 7.3 Subqueries.

Note that if you specify a table subquery in an `IN` predicate, a work table might be created. If the size of the work table DB area where the work table is created has not been estimated correctly, it might result in performance degradation. For details about estimating the size of the work table DB area, see the *HADB Setup and Operation Guide*. For details about work tables, see *Considerations when executing an SQL statement that creates work tables* in the *HADB Application Development Guide*.

(3) Evaluation of the predicate

The `IN` predicate is `TRUE` for those rows that satisfy the following condition:

- If the result of *value-expression-1* matches the result of any of the value expressions following `IN`
- If the result of *value-expression-1* matches any row from the result of the table subquery

If `NOT` is specified, the `IN` predicate is `TRUE` for those rows that satisfy the following condition:

- If the result of *value-expression-1* does not match the results of any of the value expressions following `IN`
- If the result of *value-expression-1* does not match any row from the result of the table subquery

(4) Rules

(a) Rules for specifying IN (value-expression-2 [, value-expression-3]...)

1. You cannot specify a dynamic parameter alone for *value-expression-1*.
2. There can be a maximum of 30,000 value expressions following `IN`.
3. Specify each value expression such that the data type of the result of *value-expression-1* can be compared to the data types of the results of the value expressions following `IN`. For details about data types that can be compared, see (1) Data types that can be compared in 6.2.2 Data types that can be converted, assigned, and compared.

   However, if you specify the predefined input representation of a date, time, or time stamp for *value-expression-1*, you cannot specify datetime data for *value-expression-2* and subsequent value expressions. For information about predefined input representations, see 6.3.3 Predefined character-string representations.
4. If the result of *value-expression-1* is a null value, the result of the `IN` predicate is undefined. Also, if the results of value expression 2 and all subsequent value expressions are null values, the result of the `IN` predicate is undefined.

(b) Rules for specifying IN table-subquery

1. If a table subquery returns zero rows (empty set), the result of the `IN` predicate is false. However, if `NOT` is specified, the result evaluates to true.

2. When you specify a table subquery, specify only one selection expression in the table subquery.
3. The IN predicate with a table subquery specified is rewritten by HADB into a quantified predicate (=ANY specification) and then processed.

(5) Examples

Example 1

From the sales history table (SALESLIST), this example retrieves the customer ID (USERID), product code (PUR-CODE), and date of purchase (PUR-DATE) for customers who purchased product code P001 or P003 on or after September 5, 2011.

```
SELECT "USERID","PUR-CODE","PUR-DATE"
    FROM "SALESLIST"
       WHERE "PUR-CODE" IN ('P001','P003')
       AND "PUR-DATE">=DATE'2011-09-05'
```

The underlined portion indicates the IN predicate.

Example 2

From the sales history table (SALESLIST), this example retrieves the customer ID (USERID), product code (PUR-CODE), and quantity purchased (PUR-NUM), but excludes customers whose customer ID (USERID) is U00212 or U00358.

```
SELECT "USERID","PUR-CODE","PUR-NUM"
    FROM "SALESLIST"
       WHERE "USERID" NOT IN ('U00212','U00358')
```

The underlined portion indicates the IN predicate.

Example 3

From the customer table (USERSLIST) and sales history table (SALESLIST), retrieve information on customers who purchased product code (PUR-CODE) P001.

```
SELECT * FROM "USERSLIST"
    WHERE "USERID" IN(SELECT "USERID" FROM "SALESLIST"
                            WHERE "PUR-CODE"='P001')
```

The underlined portion indicates the IN predicate.

7.19.4 LIKE predicate

The LIKE predicate is used to retrieve data that contains a specific character string.

(1) Specification format

```
LIKE-predicate ::= match-value [NOT] LIKE pattern-character-string [ESCAPE escape-cha
racter]

   match-value ::= value-expression
   pattern-character-string ::= value-expression
   escape-character ::= value-expression
```

(2) Explanation of specification format

match-value:

Specifies the data to be retrieved in the form of a value expression. For details about value expressions, see 7.20 Value expression.

You must specify CHAR or VARCHAR type data for *match-value*.

NOT:

If NOT is specified, values that do not match the specified pattern character string will become the target of retrieval.

pattern-character-string:

Specifies the pattern character string in the form of a value expression. For details about value expressions, see 7.20 Value expression.

You must specify CHAR or VARCHAR type data for *pattern-character-string*.

The special characters _ (underscore) and % (percent) can be specified in the pattern character string. The special character _ denotes any single character, and % denotes a character string of zero or more characters. These special characters can be used in searches such as the following.

- Five-character character strings that begin with UN: 'UN___'
- Character strings that include OR: '%OR%'

When _ or % appear in a pattern character string, they are considered special characters and are not treated as normal characters. If you want _ or % to be treated as a normal character, you must specify an escape character.

ESCAPE *escape-character*:

Specifies an escape character in the form of a value expression. For details about value expressions, see 7.20 Value expression.

Specify one byte of CHAR or VARCHAR type data for *escape-character*.

When you specify an escape character, special characters in the pattern character string (specifically, special characters immediately following the escape character) can be treated as normal characters.

Examples specifying the special character %:

```
LIKE 'ABC@%'                ...1
LIKE 'ABC@%' ESCAPE '@'     ...2
```

In 1, because % is treated as a special character, character strings beginning with 'ABC@' will be retrieved. In 2, because % is treated as a normal character, character strings beginning with 'ABC%' will be retrieved.

For details about how pattern character strings are handled when an escape character is specified, see (4) How pattern character strings are handled when an escape character is specified.

(3) Examples of pattern character strings

Typical examples of pattern character strings are given in the following table.

Table 7-4: Typical examples of pattern character strings

No.	Item	Pattern character string	Meaning	Pattern character string specification example	Pattern-matched character strings
1	Starts-with match	*xxx*%	The leading portion of the character string is *xxx*.	'ACT%'	Character strings beginning with ACT, such as ACT, ACTOR, and ACTION.

No.	Item	Pattern character string	Meaning	Pattern character string specification example	Pattern-matched character strings
2	Ends-with match	%*xxx*	The trailing portion of the character string is *xxx*.	`'%ING'`	Character strings ending with `ING`, such as `ING`, `BEING`, and `HAVING`.
3	Contains match	%*xxx*%	The character string contains *xxx* at some position.	`'%OR%'`	Character strings containing `OR`, such as `OR`, `MORE`, and `COLOR`.
4	Full match	*xxx*	The character string is equal to *xxx*.	`'EQUAL'`	`EQUAL`
5	Partial match	_..._*xxx*_..._	• A particular portion of the character string is equal to *xxx*; the other portions of the character string contain any characters. • _ denotes any single character.	`'_I_'`	Three-letter character strings, in which the second character is `I`, such as `BIT`, `HIT`, and `KIT`.
6				`'_ _T_ _ _ _'`	Seven-letter character strings, in which the third character is `T`, such as `HITACHI`.
7	Other	*xxx*%*yyy*	The leading portion of the character string is *xxx* and the trailing portion is *yyy*.	`'O%N'`	Character strings that begin with `O` and end with `N` such as `ON`, `OWN`, and `ORIGIN`.
8		%*xxx*%*yyy*%	The character string contains *xxx* at some position and *yyy* 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`.
9		*xxx*_..._*yyy*%	• The leading portion of the character string is *xxx* and the trailing portion is *yyy*. • _ denotes any single character.	`'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`.

Notes:

- *xxx* and *yyy* denote any characters strings that do not include the characters `%` or `_`.
- Because a space is regarded as a character for comparison purposes, a comparison with data that has trailing spaces generates a `FALSE` result.

(4) How pattern character strings are handled when an escape character is specified

This subsection describes how pattern character strings are handled when an escape character is specified. In the examples below, the escape character is the at mark (`@`).

1. When the escape character is immediately followed by a special character, the special character is treated as a normal character.

   Example 1:

   ```
   LIKE 'AB@%C%' ESCAPE '@'
   ```

In this case, because the special character after the `@` mark is treated as a normal character, character strings starting with `AB%C` will be retrieved, such as `AB%C` and `AB%CDE`.

Example 2:

```
LIKE 'AB@_C%' ESCAPE '@'
```

In this case, because the special character after the `@` mark is treated as a normal character, character strings starting with `AB_C` will be retrieved, such as `AB_C` and `AB_CDE`.

2. When the escape character is immediately followed by a normal character, the escape character is skipped.

   Example:

   ```
   LIKE 'ABC@D' ESCAPE '@' → equivalent to LIKE 'ABCD'
   ```

   In this case, the `@` mark is skipped.

3. Two consecutive escape characters are treated as a single normal character.

   Example 1:

   ```
   LIKE 'AB@@C' ESCAPE '@' → equivalent to LIKE 'AB@C'
   ```

   In this case, the two at marks (`@@`) are treated as the single normal `@` mark.

   Example 2:

   ```
   LIKE 'AB@@@C' ESCAPE '@' → equivalent to LIKE 'AB@C'
   ```

   In this case, the first two at marks (`@@`) are treated as the single normal `@` mark. The third `@` mark is skipped because the character after it is a normal character.

   Example 3:

   ```
   LIKE 'AB@@@@C' ESCAPE '@' → equivalent to LIKE 'AB@@C'
   ```

   In this case, the first two at marks (`@@`) are treated as the single normal `@` mark. The next two at marks (the third and fourth `@` marks) are also treated as the single normal `@` mark.

   Example 4:

   ```
   LIKE 'AB@@C@%D%' ESCAPE '@'
   ```

   In this case, character strings starting with `AB@C%D` will be retrieved, such as `AB@C%D` and `AB@C%DE`.

4. When no character follows the escape character, the escape character is skipped.

   Example:

   ```
   LIKE 'ABC@' ESCAPE '@' → equivalent to LIKE 'ABC'
   ```

(5) Evaluation of the predicate

If *match-value* matches the pattern in *pattern-character-string* the result is `TRUE`; otherwise, it is `FALSE`.

If `NOT` is specified, and *match-value* does not match the pattern in *pattern-character-string*, the result is `TRUE`; otherwise, it is `FALSE`.

If the result of *match-value* or *pattern-character-string* has a null value, the result of the predicate is unknown.

If the length of *match-value* is 0 bytes or 0 characters, the result of the `LIKE` predicate will be `TRUE` only in the following cases:

- If the pattern character string has a length of 0 bytes or 0 characters
- If the pattern character string is specified as a dynamic parameter, and its input value is `'%'`
- If the pattern character string is specified as the literal `'%'`

In addition, if the length of *pattern-character-string* is 0 bytes or 0 characters, the result of the `LIKE` predicate will be `TRUE` only if the length of *match-value* is 0 bytes or 0 characters.

(6) Rules

(a) Rules for match-value

1. The special characters underscore (_) and percent sign (`%`) in *match-value* must be specified as single-byte (minimum byte) characters.
2. If a dynamic parameter is specified by itself for *match-value*, the assumed data type of the dynamic parameter will be `VARCHAR(32000)`.

(b) Rules for pattern-character-string

1. The length of the pattern character string includes the special characters _ and `%`.
2. If a percent sign (`%`) is not specified in *pattern-character-string*, and the length of the data in *match-value* is different from the length of the pattern character string, the predicate is not `TRUE`.
3. If you specify a dynamic parameter by itself for *pattern-character-string*, the assumed data type and data length of the dynamic parameter will be as shown in the following table.

Table 7-5: Assumed data type and data length of the dynamic parameter (when you specify a dynamic parameter by itself for pattern-character-string)

Condition	**Assumed data type of dynamic parameter**	**Assumed data length of dynamic parameter**
Escape character not specified	`VARCHAR` type	Data length of the result of match-value

Escape character specified	`VARCHAR` type	• If the data length of the result of match value is no more than 32,000 bytes
		MIN(*data-length-of-the-result-of-match-value* × 2, 32,000)
		• If the data length of the result of match value is 32,001 bytes or more
		MIN(*data-length-of-the-result-of-match-value* × 2, 64,000)

(c) Rules for escape characters

1. If *escape-character* has a length of 0 bytes or 0 characters, it is treated as if no escape character was specified.

   Examples:

   `LIKE 'ABC' ESCAPE ''` → equivalent to `LIKE 'ABC'`

   `LIKE 'ABC' ESCAPE ?` → equivalent to `LIKE 'ABC'` when `NULL` is specified for the dynamic parameter
2. If a dynamic parameter is specified by itself for *escape-character*, the assumed data type of the dynamic parameter is `VARCHAR(1)`. In this case the actual length of the data must be 1 byte.
3. Identification of escape characters in the pattern character string is performed character-by-character rather than byte-by-byte.

4. The following table shows the range of character code points that can be specified for the escape character.

Table 7-6: Range of character code points that can be specified for the escape character

Value specified in the environment variable ADBLANG	Range of character code points that can be specified for the escape character
UTF8 (Unicode (UTF-8))	0x00 to 0x7F#
SJIS (Shift-JIS)	0x00 to 0xFF

#

Does not include the Shift-JIS backslash (\: 0x5C) and swung dash (~: 0x7E) characters when they are represented as multi-byte characters in UTF-8.

(7) Examples

Example 1

This example retrieves the customer ID (USERID) and name (NAME) of customers whose name begins with M.

```
SELECT "USERID","NAME"
    FROM "USERSLIST"
        WHERE "NAME" LIKE 'M%'
```

The underlined portion indicates the LIKE predicate.

Example 2

This example retrieves the customer ID (USERID) and name (NAME) of female customers whose name does not begin with M.

```
SELECT "USERID","NAME"
    FROM "USERSLIST"
        WHERE "NAME" NOT LIKE 'M%'
        AND SEX='F'
```

The underlined portion indicates the LIKE predicate.

Example 3

This example searches the product column (GOODS) in the sales table (T_SALES) for products that begin with the character strings in the pattern column (PATTERN) in the pattern table (T_PATTERN).

```
SELECT "A"."GOODS" FROM "T_SALES" AS "A","T_PATTERN" AS "B"
    WHERE "A"."GOODS" LIKE "B"."PATTERN" + '%'
```

The underlined portion indicates the LIKE predicate.

Sales table (T_SALES)

Product ID (ID)	Product (GOODS)	Sales (SALES)
01	NOTE01	10000
02	BOOK02	15000
03	PENCIL03	20000
04	ERA04	5000
05	PEN05	50000
06	NOTE02	800

Pattern table (T_PATTERN)

Product ID (ID)	Pattern (PATTERN)
01	NOTE
02	PEN

Retrieval results

GOODS
NOTE01
PENCIL03
PEN05
NOTE02

Example 4

This example searches the sales table (T_SALES) and retrieves the product name (GOODS) and sales amount (SALES) for products meeting the following conditions:

- The name of the product includes an underscore (_)
- The product is associated with branch code (BRANCH_CODE) A001

Because underscore (_) is a special character, @ is specified as the escape character so that _ will be treated as a normal character.

```
SELECT "GOODS","SALES" FROM "T_SALES"
    WHERE "GOODS" LIKE '%@_%' ESCAPE '@'
    AND "BRANCH_CODE"='A001'
```

Sales table (T_SALES)

Product ID (ID)	Product (GOODS)	Sales amount (SALES)	Branch code (BRANCH_CODE)
01	NOTE_01	10000	A001
02	BOOK_01	15000	A003
03	PEN	20000	A001
04	NOTE_02	5000	A002
05	PENCIL	50000	A003
06	BOOK_02	800	A001

Retrieval results

GOODS	SALES
NOTE_01	10000
BOOK_02	800

7.19.5 LIKE_REGEX predicate

Use the LIKE_REGEX predicate to search data by using a regular expression.

(1) Specification format

```
LIKE_REGEX-predicate ::= match-value [NOT] LIKE_REGEX
                         regular-expression-string [FLAG {I | IGNORECASE}]
  match-value ::= value-expression
  regular-expression-string ::= character-string-literal
```

(2) Explanation of specification format

match-value:

Specifies the data to be retrieved in the form of a value expression. For details about value expressions, see 7.20 Value expression.

You must specify CHAR or VARCHAR type data for match-value.

NOT:

If you specify NOT, character strings that do not include any character string elements represented by the specified regular expression string are retrieved.

*regular-expression-string***:**

Specify a regular expression in the form of a character string literal. For details about character string literals, see 6.3 Literals.

The regular expression string must be no more than 1,024 bytes long.

Specify the regular expression string in the following format:

```
regular-expression::={[regular-item]|regular-expression vertical-bar regular-expre
ssion}

regular-item::={regular-factor|regular-item regular-factor}

regular-factor::={regular-primary|regular-primary *|regular-primary +|regular-prim
ary ?
             |regular-primary repetition-factor}

repetition-factor::=left-curly-bracket lower-limit [,[upper-limit]]right-curly-bra
cket

regular-primary::={character-specifier|character-class|.|^|$|regular-character-set
|(regular-expression)}

character-specifier::={non-escaped-character|escaped-character}

regular-character-set::={[character-list...]|[^character-list...]}

character-list::={character-specifier|character-specifier - character-specifier|re
gular-character-set-identifier-specification}

regular-character-set-identifier-specification::=[:regular-character-set-identifie
r:]
```

The following table describes the regular expression rules.

Table 7-7: Regular expression rules

No.	Regular expression		Meaning
1	Character specifier		Means a character string of length 1 (unit: characters).
2	`.` (period)		Means any character of length 1 (unit: characters).
3	`^` (caret)		Means the beginning of a match value. For a match value that includes a line break, this symbol does not mean the beginning of the line after the line break. If a caret (`^`) is specified in a pair of square brackets that enclose a regular character set, this pattern means any characters other than the listed characters.
4	`$` (dollar sign)		Means the end of a match value. For a match value that includes a line break, this symbol does not mean the end of the line before the line break.
5	regular-primary`*`		Means zero or more repetitions of the preceding regular primary.
6	regular-primary`+`		Means one or more repetitions of the preceding regular primary.
7	regular-primary`?`		Means zero or one repetition of the preceding regular primary.
8	regular-expression | regular-expression		Means the regular expression that is specified to the left or right of the vertical bar (|).
9	regular-primary`{`*n*`}` regular-primary`{`*n*`,`*m*`}` regular-primary`{`*n*`,}`		Means a repetition of the preceding regular primary. The following describes the specification patterns: `{`*n*`}`: The preceding regular primary is repeated *n* times. `{`*n*`,`*m*`}`: The preceding regular primary is repeated by the number of times in the range from *n* to *m*. `{`*n*`,}`: The preceding regular primary is repeated at least *n* times.
10	upper-limit		An integer in the range from 0 to 256.
11	lower-limit		An integer in the range from 0 to 256.
12	`[`character-list...`]`		Means any of the listed characters.
13	`[^`character-list...`]`		Means any characters other than the listed characters.
14	character-specifier-1 - character-specifier-2		Means any character in the range (of character codes) from character-specifier-1 to character-specifier-2. If an option that ignores case (`FLAG I` or `FLAG IGNORECASE` is specified), this pattern means the following, in addition to the half-width letters in the specified range of character codes: the corresponding half-width uppercase letter for any half-width lowercase letter in the specified range, and the corresponding half-width lowercase letter for any half-width uppercase letter in the specified range.
15	regular-character-set-identifier	`alpha`	Means any half-width uppercase letter (other than `\`, `@`, and `#`) or half-width lowercase letter. The meaning is the same as `[a-zA-Z]`.
16		`upper`	Means any half-width uppercase letter (other than `\`, `@`, and `#`). The meaning is the same as `[A-Z]`.
17		`lower`	Means any half-width lowercase letters. The meaning is the same as `[a-z]`.
18		`digit`	Means any number. The meaning is the same as `[0-9]`.
19		`alnum`	Means any half-width uppercase letter (other than `\`, `@`, and `#`), half-width lowercase letter, or number. The meaning is the same as `[a-zA-Z0-9]`.
20		`space`	Means a half-width space, tab, carriage return, linefeed, vertical tab, or page break character.
21		`blank`	Means a half-width space or tab.

No.	Regular expression		Meaning
22		cntrl	Means a control character. Specifically, this means 0x7f or any of the character codes in the range from 0x00 to 0x1f.
23		graph	Means any of the character codes in the range from 0x21 to 0x7e.
24		print	Means any of the character codes in the range from 0x20 to 0x7e.
25		punct	Means a single-byte symbolic character whose code is 0x7e or lower. The meaning is the same as [!-/\:-@\[-`\{-~].
26		xdigit	Means a hexadecimal character. The meaning is the same as [a-fA-F0-9].
27	character-class	\d	Means any number. The meaning is the same as [0-9].
28		\D	Means any character that is not a number. The meaning is the same as [^0-9].
29		\w	Means any half-width uppercase letter (other than \, @, and #), any half-width lowercase letter, any number, or an underscore (_). The meaning is the same as [a-zA-Z0-9_].
30		\W	Means any character that is not a half-width uppercase letter (other than \, @, and #), half-width lowercase letter, number, or underscore (_). The meaning is the same as [^a-zA-Z0-9_].
31		\s	Means a half-width space, tab, carriage return, linefeed, vertical tab, or page break character.
32		\S	Means any character that is not a half-width space, tab, carriage return, linefeed, vertical tab, or page break character.
33		\A	Means the beginning of a match value.
34		\Z	Means the end of a match value.

FLAG {I|IGNORECASE}:

Specify this option to perform a search that ignores the difference between half-width uppercase letters (other than \, @, and #) and half-width lowercase letters.

Specifications of I and IGNORECASE are equivalent.

Note that if the character encoding that is used on the HADB server is Shift-JIS, this option cannot be specified.

(3) Regular expression specification examples

The following shows typical regular expression specification examples.

Table 7-8: Typical regular expression specification examples

No.	Method	Pattern	Meaning	Example	Matched string
1	Forward match	^*nnn*	Begins with *nnn*	^ACT	ACT, ACTOR, ACTION, and other character strings that begin with ACT
2	Backward match	*nnn*$	Ends with *nnn*	ING$	ING, BEING, HAVING, and other character string that end with ING
3	Partial match	*nnn*	Includes *nnn* in any place	Sun	Sun, Sunday, Sundays, and other character strings that include Sun
4	Exact match	^*nnn*$	Equals to *nnn*	^EQUAL$	EQUAL

No.	Method	Pattern	Meaning	Example	Matched string
5	Middle match	.*nnn*.	Includes *nnn* that follows any character and is followed by any character	.I.	BIT, HIT, KIT, and other three-character strings whose second character is I
6	One or more repetitions	*mmm*[0-9]+ or *mmm*[[:digit:]]+	Includes *mmm* in any place, and *mmm* is followed by any number	KFAA[0-9]+ or KFAA[[:digit:]]+	KFAA123, KFAA11104-E, KFAA11901-E, and other character strings that begin with KFAA that is followed by any number
7	Selection of some characters	^*mmm*.*(n|o) or ^*mmm*.*[no]	Begins with *mmm* and contains n or o at the *i*-th character (*i* is a numeric value)	^KFAA.*(W|E) or ^KFAA.*[WE]	KFAA20008-W, KFAA11901-E, and other character strings that begin with KFAA, followed by W or E
8	*n* repetitions	*mmm*{*n*}	Begins with *mmm*, the last character of which is repeated *n* times	123{3}	12333

(4) Evaluation of predicates

If the specified match value includes an element of the character string set that is represented by the regular expression string, the predicate evaluates to true. In other cases, the predicate evaluates to false. Note that if the length of the regular expression string is 0, the predicate evaluates to true when the match value is not a null value.

If NOT is specified, the predicate evaluates to true when the specified match value does not include any string elements that are represented by the regular expression string. In other cases, the predicate evaluates to false. Note that if the length of the regular expression string is 0, the predicate evaluates to false when the match value is not a null value.

If the match value is a null value, the predicate will have no value.

(5) Rules

(a) Rules pertaining to the match value

1. If a dynamic parameter is specified by itself as the match value, the assumed data type of the dynamic parameter will be VARCHAR(32000).

(b) Rules pertaining to the escape character

1. If a backslash (\) is included in a regular expression string, the backslash (\) is treated as an escape character.
2. A special character that follows the escape character is treated as an ordinary character. The special characters that can be escaped are as follows:
   - . (period)
   - * (asterisk)
   - + (plus sign)
   - ? (question mark)
   - | (vertical bar)
   - ((left parenthesis)
   -) (right parenthesis)

- `{` (left curly bracket)
- `}` (right curly bracket)
- `[` (left square bracket)
- `]` (right square bracket)
- `\` (backslash)
- `-` (minus sign)#
- `:` (colon)#
- `^` (caret)
- `$` (dollar sign)

#: Handled as a special character only if specified in a character list.

3. If the escape character is followed by an ordinary character, the escape character is skipped.
4. If the escape character is followed by no character, the escape character is skipped.
5. Two consecutive escape characters are treated as a single ordinary character.

(6) Considerations for performance

If a text index has been defined, the literal character in the regular expression is used to filter the pages by the text index. Therefore, in the same way as the `LIKE` predicate or scalar function `CONTAINS`, if the literal character in the regular expression is short simple text, such as `a` or `0`, the effect of page filtering is lowered. Also, if the number of patterns that are represented by meta characters (parentheses, brackets, and quantifiers) increases, the text index is not used during a search because text-index-based page filtering takes time.

Therefore, you can improve the effect of text-index-based page filtering if you do not use meta characters to represent patterns. For example, when you search for the strings HADB and HiRDB, you can specify `H(A|iR)DB` or `HADB | HiRDB` as a search condition. In this case, the latter expression provides more effective text-index-based page filtering.

This also applies to repetition factors. For example, `(abc){1,3}` and `abc | abcabc | abcabcabc` have the same meaning. In this case, the latter expression provides more effective text-index-based page filtering.

(7) Example

Example

In this example, you search the data in column `MSG` of table `T_MSG` for the rows that contain a character string that begins with `KFAA30` followed by a three-digit number, and ends with `-E`.

```
SELECT * FROM "T_MSG"
    WHERE "MSG" LIKE_REGEX 'KFAA30[0-9]{3}-E'
```

The underlined portion is the specification of the `LIKE_REGEX` predicate.

The preceding `LIKE_REGEX` predicate specification matches, for example, the string `KFAA30101-E`.

7.19.6 NULL predicate

The `NULL` predicate is used to search for null values. For details about null values, see 6.7 Null value.

(1) Specification format

```
NULL-predicate ::= value-expression IS [NOT] NULL
```

(2) Explanation of specification format

value-expression:

Specifies the column to be evaluated by the NULL predicate. It is specified in the form of a value expression. For details about value expressions, see 7.20 Value expression.

If a dynamic parameter is specified by itself, the assumed data type of the dynamic parameter will be VARCHAR(32000).

NOT:

If NOT is specified, rows that are not the null value will become the target of retrieval.

(3) Evaluation of the predicate

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.

(4) Examples

Example 1

This example retrieves the customer IDs (USERID) from the customer table (USERSLIST) where the name (NAME) is the null value.

```
SELECT "USERID"
    FROM "USERSLIST"
        WHERE "NAME" IS NULL
```

The underlined portion indicates the NULL predicate.

Example 2

This example retrieves the customer IDs (USERID) from the customer table (USERSLIST) where the name (NAME) is not the null value.

```
SELECT "USERID"
    FROM "USERSLIST"
        WHERE "NAME" IS NOT NULL
```

The underlined portion indicates the NULL predicate.

7.19.7 Comparison predicate

Comparison predicates can be specified in search conditions. The following example illustrates a comparison predicate.

Example:

From the sales history table (SALESLIST), this example retrieves the customer ID (USERID), product code (PUR-CODE), and date of purchase (PUR-DATE) for customers who purchased products on or after September 6, 2011.

```
SELECT "USERID","PUR-CODE","PUR-DATE"
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2011-09-06'
```

Explanation

- The underlined portion indicates the comparison predicate.
- >= is called a *comparison operator*.
- The terms on the left and right of a comparison operator are called *comparison operands*. In this example, the comparison operands are `PUR-DATE` (a column name) and `DATE'2011-09-06'` (a literal).

(1) Specification format

```
comparison-predicate ::= comparison-operand-1 comparison-operator comparison-operand-
2

  comparison-operand ::= value-expression
  comparison-operator ::= {=|<>|!=|^=|<|<=|>|>=}
```

(2) Explanation of specification format

comparison-operand-1, *comparison-operand-2*:

A comparison operand specifies a value such as a column name or literal. Comparison operands must be specified as value expressions. For details about value expressions, see 7.20 Value expression.

comparison-operator:

The comparison operator is one of =, <>, !=, ^=, <, <=, >, or >=. The following table lists the meaning of each operator.

Table 7-9: Meaning of comparison operators

No.	Comparison operator	Meaning
1	*X* = *Y*	*X* and *Y* are equal
2	*X* <> *Y* *X* != *Y* *X* ^= *Y*	*X* and *Y* are not equal
3	*X* < *Y*	*X* is less than *Y*
4	*X* <= *Y*	*X* is less than or equal to *Y*
5	*X* > *Y*	*X* is greater than *Y*
6	*X* >= *Y*	*X* is greater than or equal to *Y*

Legend:

X and *Y*: Comparison operands

(3) Evaluation of the predicate

A comparison is `TRUE` if the comparison operands on the left and right of the comparison operator satisfy the comparison condition.

It is unknown if either of the comparison operands is the null value.

(4) Rules

1. The data types of the results of *comparison-operand-1* and *comparison-operand-2* must be data types that can be compared. For details about data that can be compared, see (1) Data types that can be compared in 6.2.2 Data types that can be converted, assigned, and compared.
2. When comparing numeric data, if the data being compared are of different data types, the comparison is performed using the data type with the wider range, as determined by the following hierarchy:

   ```
   DOUBLE PRECISION > DECIMAL > INTEGER > SMALLINT
   ```

3. If the result of a comparison operand is the null value, the comparison result is unknown.
4. If you specify binary type data as the value expression of the comparison operands, you must specify =, <>, !=, or ^= as the comparison operator.
5. A comparison operand composed of only a dynamic parameter cannot appear on both sides of the comparison operator.
   - Example of a specification that is not permitted: `?=?`
   - Examples of a permissible specifications: `C1=?`, `?=10`
6. For *comparison-operand-2* you cannot specify the scalar function `CONTAINS`.
7. If you specify the scalar function `CONTAINS` for *comparison-operand-1*, specify > as the comparison operator. In this case, specify `0` for *comparison-operand-2*.

(5) Example

Example

From the sales history table (`SALESLIST`), this example retrieves the customer ID (`USERID`), product code (`PUR-CODE`), and date of purchase (`PUR-DATE`) for customers who purchased product code `P001` or `P003` on or after September 4, 2011.

```
SELECT "USERID","PUR-CODE","PUR-DATE"
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2011-09-04'
        AND ("PUR-CODE"='P001' OR "PUR-CODE"='P003')
```

The underlined portions show the comparison predicates.

7.19.8 Quantified predicate

Quantified predicates are used to compare the result of a value expression to the result of a table subquery.

(1) Specification format

```
quantified-predicate ::= value-expression{=|<>|!=|^=|<|<=|>|>=}{{ANY|SOME}|ALL}table-subquery
```

(2) Explanation of specification format

value-expression:

Specifies, in the form of a value expression, the column to be evaluated by the quantified predicate. For details about value expressions, see 7.20 Value expression.

ANY or SOME:

If there is at least one row in the results from the table subquery that satisfies the comparison with *value-expression*, the result of the quantified predicate is TRUE.

The results are the same regardless of whether you use ANY or SOME.

ALL:

If either of the following conditions is met, the result of the quantified predicate is TRUE:

- If all the rows in the results of the table subquery satisfy the comparison with *value-expression*
- If the result of the table subquery is zero rows (the empty set)

table-subquery:

For details about table subqueries, see 7.3 Subqueries.

(3) Evaluation of the predicate

(a) When ANY or SOME is specified

- If there is at least one row in the results from the table subquery that satisfies the comparison with *value-expression*, the result of the quantified predicate is TRUE.
- If either of the following conditions is met, the result of the quantified predicate is FALSE:
  - If all of the rows in the results of the table subquery fail to satisfy the comparison with *value-expression*
  - If the result of the table subquery is zero rows (the empty set)
- Otherwise, the result is unknown.

The following table shows the result of a quantified predicate in which ANY or SOME is specified.

Table 7-10: Result of a quantified predicate in which ANY or SOME is specified

No.	Results of comparison to every row in the table subquery		Result of the quantified predicate
1	Some TRUE rows		TRUE
2	No TRUE rows	Some undefined	Undefined
3		No undefined	FALSE
4	0 rows (the empty set)		FALSE

(b) When ALL is specified

- If either of the following conditions is met, the result of the quantified predicate is TRUE:
  - If all the rows in the results of the table subquery satisfy the comparison with *value-expression*
  - If the result of the table subquery is zero rows (the empty set)
- If any of the rows in the results of the table subquery fail to satisfy the comparison with *value-expression*, the result of the quantified predicate is FALSE.

- Otherwise, the result is undefined.

The following table shows the result of a quantified predicate in which ALL is specified.

Table 7-11: Result of a quantified predicate in which ALL is specified

No.	Results of comparison to every row in the table subquery		Result of the quantified predicate
1	Some FALSE rows		FALSE
2	No FALSE rows	Some undefined	Undefined
3		No undefined	TRUE
4	0 rows (the empty set)		TRUE

(4) Rules

1. There must be exactly one column in the results of the table subquery.
2. When binary data is specified in *value-expression*, the only comparison operators allowed are =, <>, !=, and ^=.

(5) Notes

1. When you specify a quantified predicate, a work table might be created. If the size of the work table DB area where the work table is created has not been estimated correctly, it might result in performance degradation. For details about estimating the size of the work table DB area, see the *HADB Setup and Operation Guide*. For details about work tables, see *Considerations when executing an SQL statement that creates work tables* in the *HADB Application Development Guide*.
2. If a quantified predicate (=ANY or =SOME specification) is specified, HADB performs deduplication of the table subquery results.

(6) Example

Example

From the customer table (USERSLIST) and sales history table (SALESLIST), this example retrieves information on customers who purchased product code (PUR-CODE) P001.

```
SELECT * FROM "USERSLIST"
    WHERE "USERID"=ANY(SELECT "USERID" FROM "SALESLIST"
                            WHERE "PUR-CODE"='P001')
```

The underlined portion indicates the quantified predicate.

7.20 Value expression

This section describes value expressions.

7.20.1 Specification format and rules for value expressions

In SQL statements, values can be specified in the form of an expression using items such as column names, literals, set functions, scalar functions, window functions, `CASE` expressions, arithmetic operations (`+`, `-`, `*`, and `/`), and concatenation operations (`+`, `||`). Such specifications are called *value expressions*. Examples of value expressions are given below.

Examples:

- `"C1"`, which specifies a single column name
- `'HADB'`, `100`, and `DATE'2011-09-06'`, which specify single literals
- `"C1"+10`, which uses a column name and an arithmetic operation
- `"C1"||"C2"`, which uses column names and a concatenation operation
- `MAX("C1")/2`, which uses a set function and an arithmetic operation

(1) Specification format

```
value-expression ::= {numeric-value-expression | character-value-expression | datetim
e-value-expression | binary-value-expression}


  numeric-value-expression ::= {value-expression-primary | arithmetic-operation}
  character-value-expression ::= {value-expression-primary | concatenation-operation}
  datetime-value-expression ::= {value-expression-primary | datetime-operation}
  binary-value-expression ::= {value-expression-primary | concatenation-operation}

    value-expression-primary ::= {(value-expression) | column-specification | value-s
pecification | set-function
                     | scalar-function | window-function | CASE-expression
                     | labeled-duration | scalar-subquery}
```

(2) Explanation of specification format

arithmetic-operation:

For details about arithmetic operations, see 7.25 Arithmetic operations.

concatenation-operation:

For details about concatenation operations, see 7.26 Concatenation operations.

datetime-operation:

For details about datetime operations, see 7.27 Datetime operations.

column-specification:

For details about column specifications, see (5) Column specification format in 6.1.5 Qualifying a name.

value-specification:

For details about value specifications, see 7.21 Value specification.

set-function:

For details about set functions, see 7.22 Set functions.

scalar-function:

For details about scalar functions, see 8. Scalar Functions.

window-function:

For details about window functions, see 7.23 Window functions.

CASE-expression:

For details about `CASE` expressions, see 7.29 CASE expression.

labeled-duration:

For details about labeled durations, see 7.28 Labeled duration.

scalar-subquery:

For details about scalar subqueries, see 7.3 Subqueries.

(3) Rules

1. A maximum of 500 total scalar operations and set functions can be specified in a value expression. If a column specified in a value expression is a column from a viewed table, derived table, or query name, the total number of value expressions after expanding the value expression it is based on cannot exceed 10,000.

   Note that in certain circumstances, depending on how a viewed table, derived table, or query name is specified, a value expression might be added to it. For the circumstances under which the value expression is added, see 7.30.6 When the scalar function CONVERT is added to an internal derived table.

   Scalar operation is a general term for the following operations that can be specified in a value expression:

   - Arithmetic operation
   - Concatenation operation
   - Datetime operation
   - Scalar function
   - Window function
   - `CASE` expression

2. When the scalar operations listed below are nested, the upper limit on nesting is 15 levels. If a column specified in a value expression is a column from a viewed table or derived table, after expanding the value expression they are based on, make sure that the nesting depth of the scalar operations does not exceeded 15 levels.

   Note that in certain circumstances, depending on how a viewed table or derived table is specified, a value expression might be added to it. For the circumstances under which the value expression is added, see 7.30.6 When the scalar function CONVERT is added to an internal derived table.

   Even if different scalar operations are combined, the upper limit on nesting remains a total of 15 levels.

   - Scalar function
   - Window function
   - `CASE` expression

   The examples below illustrate how nesting levels are counted.

   Example 1: The scalar function `SUBSTR` nested 15 times

```
SUBSTR(SUBSTR(SUBSTR(SUBSTR(SUBSTR(
SUBSTR(SUBSTR(SUBSTR(SUBSTR(SUBSTR(
SUBSTR(SUBSTR(SUBSTR(SUBSTR(SUBSTR(
SUBSTR("C1",1),1),1),1),1),1),1),1),1),1),1),1),1),1),1),1)
```

Example 2: `CASE` expressions within `CASE` expressions, nested two levels deep

```
CASE WHEN
    CASE WHEN
                CASE WHEN "C1">100
                    THEN  "C1"-100
                    ELSE "C1"
                END >100
            THEN "C1"-100
            ELSE "C1"
        END >100
    THEN "C1"-100
    ELSE "C1"
END
```

Example 3: A mixture of a `CASE` expression and the scalar function `SUBSTR`, with a maximum nesting level of 2

```
SUBSTR(CASE WHEN "C1">100 THEN SUBSTR("C2",1,10)
                          ELSE SUBSTR("C2",1,5)
       END,1,5)
```

3. If an overflow occurs during any of these operations, an SQL error is generated.
4. The order in which scalar operations are evaluated obeys the following priority hierarchy:
    - Items in parentheses
    - `*` or `/`
    - +, −, or `||`
5. The table below shows the conditions under which value expressions are equivalent to literals. However, note that the data type and data length of the result of the value expression will be the data type and data length derived from each component value expression rather than the data type and data length of the literal.

Table 7-12: Conditions under which value expressions are equivalent to literals

No.	Type of value expression		Conditions under which the value expression is equivalent to a literal
1	Arithmetic operation		When you specify literals for the first and second operands
2	Concatenation operation		
3	Datetime operation		When all of the following conditions are satisfied: • A literal is specified for the first operand. • The second operand is a labeled duration, and a literal is specified for the *value-expression-primary* specified in the labeled duration. • Literals are specified for the *value-expression-primary* instances that are multiplied or divided (only when multiplying or dividing labeled durations)
4	Scalar functions	`ABS`	When you specify a literal for the target data
5		`ACOS`	
6		`ASCII`	When you specify a literal for the target data, except in the following case: • When you specify data whose actual length is 0 bytes or 0 characters for the target data
7		`ASIN`	When you specify a literal for the target data
8		`ATAN`	
9		`ATAN2`	When you specify literals for both target data items

No.	Type of value expression		Conditions under which the value expression is equivalent to a literal
10		BIN	When you specify a literal for the target data
11		BITAND	When you specify literals for both target data items
12		BITLSHIFT	When you specify literals for the target data and the number of bits to shift
13		BITNOT	When you specify a literal for the target data
14		BITOR	When you specify literals for both target data items
15		BITRSHIFT	When you specify literals for the target data and the number of bits to shift
16		BITXOR	When you specify literals for both target data items
17		CAST	When you specify a literal for the conversion target data, except in the following case: • When you specify conversion of a character string literal whose actual length is 0 bytes into something other than character string data
18		CEIL	When you specify a literal for the target data
19		CHR	When you specify a literal for the target data, except in the following case: • When you specify a negative integer value for the target data
20		COALESCE	When there is target data specified for at least one argument, and you specify literals for all the target data
21		CONCAT	When you specify literals for both target data items
22		CONVERT	When you specify a literal for the conversion target data, except in the following cases: • When you specify conversion of a character string literal whose actual length is 0 bytes into something other than character string data • When you specify the format specification
23		COS	When you specify a literal for the target data
24		COSH	
25		DATEDIFF	When you specify literals for the start date and end date
26		DAYOFWEEK	When you specify a literal for the target data
27		DAYOFYEAR	
28		DEGREES	When you specify a literal for the angle
29		EXP	When you specify a literal for the exponent
30		EXTRACT	When you specify a literal for the source data
31		FLOOR	When you specify a literal for the target data
32		GETAGE	When you specify literals for the date of birth and reference date
33		HEX	When you specify a literal for the target data
34		LASTDAY	When you specify a literal for the date data
35		LEFT	When you specify literals for the source character string data and extraction length, except in the following case: • When you specify a negative value for the extraction length
36		LENGTH	When you specify a literal for the target data
37		LENGTHB	

No.	Type of value expression		Conditions under which the value expression is equivalent to a literal
38		LN	
39		LOG	When you specify literals for the base and target data
40		LOWER	When you specify a literal for the character string data to be converted
41		LPAD	When you specify literals for the target data, number of characters, and padding character string, except in the following case: • When you specify a negative value for the number of characters
42		LTRIM	When you specify literals for the target data and the characters to be removed
43		MOD	When you specify literals for the dividend and divisor
44		PI	Always treated as a literal.
45		POWER	When you specify literals for the target data and exponent
46		RADIANS	When you specify a literal for the angle
47		REPLACE	When either of the following conditions is met: • When you specify literals for the target data, character string to be replaced, and replacement character string • When you specify literals for the target data and character string to be replaced, and omit the replacement character string
48		RIGHT	When you specify literals for the source character string data and extraction length, except in the following case: • When you specify a negative value for the extraction length
49		ROUND	• Mathematical function ROUND: When you specify literals for the target data and number of digits • Datetime function ROUND: When you specify a literal for the datetime data
50		RPAD	When you specify literals for the target data, number of characters, and padding character string, except in the following case: • When you specify a negative value for the number of characters
51		RTRIM	When you specify literals for the target data and the characters to be removed
52		SIGN	When you specify a literal for the target data
53		SIN	
54		SINH	
55		SQRT	
56		SUBSTR	When you specify literals for the source character string data, start position, and extraction length, except in the following case: • When you specify a negative value for the extraction length
57		SUBSTRB	When you specify literals for the source binary data, start position, and number of bytes to extract (except in the following case) • When you specify a negative value for the number of bytes to extract
58		TAN	When you specify a literal for the target data
59		TANH	
60		TRANSLATE	When you specify literals for the target data, characters to replace, and replacement characters
61		TRIM	When you specify literals for the target data and the characters to be removed

No.	Type of value expression		Conditions under which the value expression is equivalent to a literal
62		TRUNC	• Mathematical function TRUNC: When you specify literals for the target data and number of digits • Datetime function TRUNC: When you specify a literal for the datetime data
63		UPPER	When you specify a literal for the character string data to be converted

7.20.2 Data types of the results of value expressions

The data type of the result is determined by the data types of the specified value expressions for CASE expressions, columns derived from the results of set operations, columns derived by table value constructors, and the following scalar functions:

- COALESCE
- DECODE
- GREATEST
- LEAST
- LTDECODE
- NULLIF
- BITAND
- BITOR
- BITXOR

Example 1: CASE expression

```
CASE
  WHEN search-condition THEN value-expression-1
  WHEN search-condition THEN value-expression-2
  ELSE value-expression-3
END
```

The data type of the result of the CASE expression is determined by the data types of value expressions 1 through 3.

Example 2: Scalar function GREATEST

```
GREATEST (value-expression-1, value-expression-2, value-expression-3)
```

The data type of the result of the scalar function GREATEST is determined by the data types of value expressions 1 through 3.

This section describes the rules for determining data type of the result of a value expression.

(1) If the data type of the value expression is character string data

- CASE expressions and the scalar functions COALESCE, DECODE, GREATEST, LEAST, LTDECODE, and NULLIF
  - If there is at least one value expression whose data type is VARCHAR, the data type of the result will be VARCHAR.
  - The data length of the result will be the data length of the value expression whose data length is longest.

- Columns derived as a result of a set operation, and columns derived by table value constructors
  - If all of the value expressions are CHAR type, and all the data lengths are the same, the data type of the result will be CHAR. Otherwise, the data type of the result will be VARCHAR.
  - The data length of the result will be the data length of the value expression whose data length is longest.

(2) If the data type of the value expression is numeric data

When the value expressions have numeric data types, the data type of the result is determined as shown in the following table.

Table 7-13: Relationship between the data type of the value expressions and the data type of the result (when the value expressions have numeric data types)

No.	Data type of value expression N	Data type of value expression N + 1	Data type of the result
1	SMALLINT	SMALLINT	SMALLINT
2		INTEGER	INTEGER
3		DECIMAL	DECIMAL
4		DOUBLE PRECISION	DOUBLE PRECISION
5	INTEGER	SMALLINT	INTEGER
6		INTEGER	
7		DECIMAL	DECIMAL
8		DOUBLE PRECISION	DOUBLE PRECISION
9	DECIMAL	SMALLINT	DECIMAL
10		INTEGER	
11		DECIMAL	
12		DOUBLE PRECISION	DOUBLE PRECISION
13	DOUBLE PRECISION	SMALLINT	DOUBLE PRECISION
14		INTEGER	
15		DECIMAL	
16		DOUBLE PRECISION	

Legend: *N*: An integer greater than or equal to 1

■ **If the data type of the result is DECIMAL**

The precision and scaling are determined as follows. Let *value-expression-1* be DECIMAL(*p1*,*s1*), *value-expression-2* be DECIMAL(*p2*,*s2*), and *value-expression-N* be DECIMAL(*pN*,*sN*).

- precision = MIN(38, *Pmax* + *Smax*)
- scaling = MIN(*Smax*, 38 - *Pmax*)

Pmax = MAX(*p1*-*s1*, *p2* - *s2*, ..., *pN* - *sN*)

Smax = MAX(*s1*, *s2*, ..., *sN*)

If the data type of the value expression is INTEGER, use DECIMAL(20,0) for the calculation. If the data type of the value expression is SMALLINT, use DECIMAL(10,0) for the calculation.

Note that, if the numeric data of the result falls beyond the precision and scaling that are obtained here, the fractional part will be truncated. The following shows examples.

Example 1:

The SELECT statement below is executed with column C1 having type DECIMAL(37,0) and a value of NULL, and with column C2 having type DECIMAL(10,2) and a value of 12345678.12.

```
SELECT COALESCE("C1","C2") FROM "T1"
```

Retrieval results

```
12345678.1
```

In this case, the data type of the execution result of the scalar function COALESCE becomes DECIMAL(38,1), and the decimal digits beyond the scaling will be truncated.

Example 2:

Assume that you execute the following SELECT statement, which contains table DT that is derived by a table value constructor for which 1.1234567890123456789 and 10 are specified as row value constructor elements. In this case, literal 1.1234567890123456789 is treated as DECIMAL(20,19) type, and literal 10 is treated as INTEGER type. Because INTEGER type is treated as DECIMAL(20,0) type, the data type of the result of column C1 derived by the table value constructor will be DECIMAL(38,18). As a result, the decimal digits beyond the scaling will be truncated.

```
SELECT "C1" FROM (VALUES(1.1234567890123456789),(10)) "DT"("C1")
```

Retrieval results

```
1.123456789012345678
10.000000000000000000
```

To prevent truncation of the decimal digits beyond the scaling, you can explicitly specify literal 10 as a decimal literal (as DECIMAL type) as shown later. In this case, literal 1.1234567890123456789 is treated as DECIMAL(20,19) type, and literal 10.0 is treated as DECIMAL(3,1) type. Therefore, the data type of the result of column C1 derived by the table value constructor will be DECIMAL(21,19).

```
SELECT "C1" FROM (VALUES(1.1234567890123456789),(10.0)) "DT"("C1")
```

Retrieval results

```
1.1234567890123456789
10.0000000000000000000
```

(3) If the data type of the value expression is datetime data

When the value expressions have datetime data types, the data type of the result is determined as shown in the following table.

Table 7-14: Relationship between the data type of the value expressions and the data type of the result (when the value expressions have datetime data types)

No.	Data type of value expression N	Data type of value expression N + 1	Data type of the result
1	DATE	DATE	DATE
2		TIMESTAMP	TIMESTAMP#
3	TIME	TIME	TIME

No.	Data type of value expression N	Data type of value expression N + 1	Data type of the result
4	TIMESTAMP	DATE	TIMESTAMP#
5		TIMESTAMP	TIMESTAMP

Legend:

N: An integer greater than or equal to 1

#

DATE type data is converted to TIMESTAMP type data by setting the time portion to 00:00:00.

■ When the fractional seconds precision is specified

When the fractional seconds precision is included in the results of *value-expression-1* to *value-expression-N*, the fractional seconds precision of the result is determined as follows:

Letting the fractional seconds precision of *value-expression-1* be *p1*, the fractional seconds precision of *value-expression-2* be *p2*, and the fractional seconds precision of *value-expression-N* be *pN*, the fractional seconds precision of the result will be MAX(*p1*, *p2*, ..., *pN*).

The data length of the result depends on its fractional seconds precision.

(4) If the data type of the value expression is binary data

When the value expressions have binary data types, the data type of the result is determined as follows.

- CASE expressions, or the scalar functions COALESCE or NULLIF
  - The data type of the result will be VARBINARY.
  - The data length of the result will be the data length of the value expression whose data length is longest.
- Columns derived as a result of a set operation, columns derived by table value constructors, and scalar functions BITAND, BITOR, and BITXOR
  - If all the value expressions are BINARY type, and all the data lengths are the same, the data type of the result will be BINARY. Otherwise, the data type of the result will be VARBINARY.
  - The data length of the result will be the data length of the value expression whose data length is longest.

7.21 Value specification

This section describes value specifications.

7.21.1 Specification format for value specifications

In an SQL statement, the following items can be specified in places where a value specification is allowed:

- A literal
- A dynamic parameter
- The datetime information acquisition function CURRENT_DATE
- The datetime information acquisition function CURRENT_TIME
- The datetime information acquisition function CURRENT_TIMESTAMP
- The user information acquisition function CURRENT_USER

(1) Specification format

```
value-specification ::= {literal|dynamic-parameter|CURRENT_DATE|CURRENT_TIME
                    |CURRENT_TIMESTAMP|CURRENT_USER}
```

(2) Explanation of specification format

literal:

For details about literals, see 6.3 Literals.

The data type of the result of the value specification will be the data type of the specified literal.

dynamic-parameter:

For details about dynamic parameters, see 6.6 Variables (dynamic parameters).

The data type of the result of the value specification will vary depending on the location where the dynamic parameter is specified.

CURRENT_DATE:

For details about CURRENT_DATE, see 6.4.1 CURRENT_DATE.

The data type of the result of the value specification will be the DATE type.

CURRENT_TIME:

For details about CURRENT_TIME, see 6.4.2 CURRENT_TIME.

The data type of the result of the value specification will be the TIME type.

CURRENT_TIMESTAMP:

For details about CURRENT_TIMESTAMP, see 6.4.3 CURRENT_TIMESTAMP.

The data type of the result of the value specification will be the TIMESTAMP type.

CURRENT_USER:

For details about CURRENT_USER, see 6.5.1 CURRENT_USER.

The data type of the result of the value specification will be VARCHAR type.

(3) Examples

The following examples illustrate value specifications.

Example 1

This example inserts the following data (row) into the sales history table (`SALESLIST`).

- Customer ID (`USERID`): `U00358`
- Product code (`PUR-CODE`): `P003`
- Quantity purchased (`PUR-NUM`): `5`
- Date of purchase (`PUR-DATE`): `2011-09-08`

```
INSERT INTO "SALESLIST"
    ("USERID","PUR-CODE","PUR-NUM","PUR-DATE")
    VALUES('U00358','P003',5,DATE'2011-09-08')
```

The underlined portions show the locations of the value specifications.

Example 2

From the sales history table (`SALESLIST`), this example retrieves the customer ID (`USERID`) and product code (`PUR-CODE`) for customers who purchased products today.

```
SELECT "USERID","PUR-CODE"
    FROM "SALESLIST"
        WHERE "PUR-DATE"=CURRENT_DATE
```

The underlined portion indicates the location of the value specification.

7.22 Set functions

Set functions can be used to calculate aggregate values across multiple rows. The set functions are listed in the following table.

Table 7-15: List of set functions

No.	List of set functions		Description
1	COUNT(*)		Determine the row count (number of results).
2	General set functions	AVG	Determine the average value.
3		COUNT	Determine the row count (number of results).
4		MAX	Determine the maximum value.
5		MIN	Determine the minimum value.
6		SUM	Determine the total value.
7		STDDEV_POP	Determine the standard deviation of a population.
8		STDDEV_SAMP	Determine the standard deviation of a sample.
9		VAR_POP	Determine the variance of a population.
10		VAR_SAMP	Determine the variance of a sample.
11	Inverse distribution functions	MEDIAN	Determine the median value of an ordered set of values. The value might be linearly interpolated.
12		PERCENTILE_CONT	Determine the percentile of an ordered set of values. The value might be linearly interpolated.
13		PERCENTILE_DISC	Determine the percentile of an ordered set of values. A result from the set of values is returned.

7.22.1 COUNT(*)

COUNT(*) determines the number of input rows to the set function.

(1) Specification format

```
set-function-COUNT(*) ::= COUNT(*)
```

(2) Rules

1. The data type of the execution result is INTEGER.
2. If the number of input rows is 0, the result will be 0.

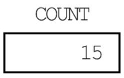
(3) Example

Example

Using the data in the employee table (EMPLIST), this example determines the number of male employees in section S001.

```
SELECT COUNT(*) AS "COUNT"
    FROM "EMPLIST"
        WHERE "SCODE"='S001' AND "SEX"='M'
```

Example of execution results



COUNT
15

7.22.2 AVG

`AVG` determines the average.

(1) Specification format

```
general-set-function-AVG ::= {AVG([ALL] value-expression)|AVG(DISTINCT value-expressi
on)}
```

(2) Explanation of specification format

`AVG([ALL]` *value-expression*`)`:

Determines the average of the results of the value expression. For details about value expressions, see 7.20 Value expression.

`ALL` can be omitted. The results will be the same regardless of whether it is specified.

`AVG(DISTINCT` *value-expression*`)`:

Determines the average of the results of the value expression. For details about value expressions, see 7.20 Value expression.

Duplicate values are only counted once. For example, if the values in the result of the value expression are `2`, `3`, `2`, and `4`, the execution result will be (2 + 3 + 4) ÷ 3 = 3.

(3) Rules

1. Null values are not included in the calculation.
2. When calculating the average, any digits following the significant digits are truncated.
3. In the following cases, the execution result will be a null value.
    - If the number of input rows is 0
    - If the values to be calculated are all null values
4. The following table shows the data type that can be specified in the value expression and the data type of the execution result of the general set function `AVG`.

Table 7-16: Data type that can be specified in the value expression and data type of the execution result of the general set function AVG

No.	Data type that can be specified in the value expression	Data type of the execution result of general set function AVG
1	`INTEGER`	`INTEGER`
2	`SMALLINT`	

No.	Data type that can be specified in the value expression	Data type of the execution result of general set function AVG
3	`DECIMAL(`*m*`,`*n*`)`	`DECIMAL(38,38-`*m*`+`*n*`)`
4	`DOUBLE PRECISION`	`DOUBLE PRECISION`

5. If an overflow occurs during the computation of any set function, an overflow error is generated.

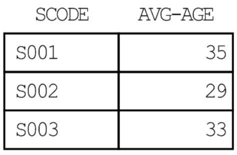
(4) Example

Example

Using the data in the employee table (`EMPLIST`), this example determines the average employee age (`AGE`) in each section (`SCODE`).

```
SELECT "SCODE",AVG("AGE") AS "AVG-AGE"
    FROM "EMPLIST"
        GROUP BY "SCODE"
```

Example of execution results



SCODE	AVG-AGE
S001	35
S002	29
S003	33

7.22.3 COUNT

`COUNT` determines the row count (number of results).

(1) Specification format

```
general-set-function-COUNT ::= {COUNT([ALL] value-expression)|COUNT(DISTINCT value-ex
pression)}
```

(2) Explanation of specification format

`COUNT([ALL]` *value-expression*`)`:

Specify a value expression. For details about value expressions, see 7.20 Value expression.

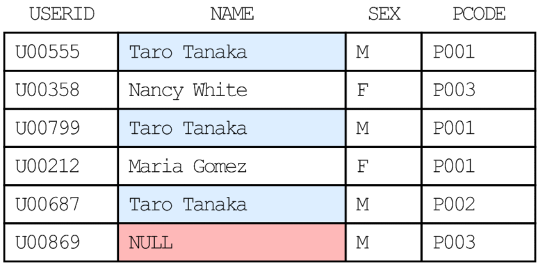
`ALL` can be omitted. The results will be the same regardless of whether it is specified.

`COUNT(DISTINCT` *value-expression*`)`:

Specify a value expression. For details about value expressions, see 7.20 Value expression. For rows containing the same value, the count will exclude duplicates.

The examples below illustrate the difference in execution results for the two specification formats above. In the second example, a `GROUP BY` clause is specified in the `SELECT` statement.

(a) Example 1: Without GROUP BY clause

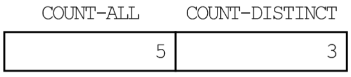


USERID	NAME	SEX	PCODE
U00555	Taro Tanaka	M	P001
U00358	Nancy White	F	P003
U00799	Taro Tanaka	M	P001
U00212	Maria Gomez	F	P001
U00687	Taro Tanaka	M	P002
U00869	NULL	M	P003

Using the employee table (`EMPLIST`) above, this example executes the following `SELECT` statement:

```
SELECT COUNT("NAME") AS "COUNT-ALL",
       COUNT(DISTINCT "NAME") AS "COUNT-DISTINCT"
    FROM "EMPLIST"
```

The results are as follows:



COUNT-ALL	COUNT-DISTINCT
5	3

Explanation

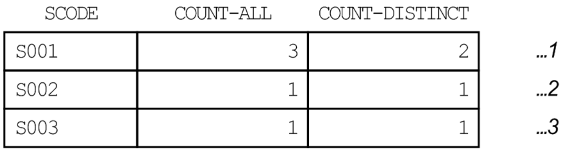
- In the case of `COUNT(NAME)`, duplicates of the same name (`Taro Tanaka`) are counted, but rows with null values are not counted, so the execution result is `5`.
- In the case of `COUNT(DISTINCT NAME)`, duplicates of the same name (`Taro Tanaka`) are not counted, and neither are rows with null values, so the execution result is `3`.

(b) Example 2: With GROUP BY clause

Using the employee table (`EMPLIST`) shown in Example 1, this example executes the following `SELECT` statement:

```
SELECT "SCODE","COUNT("NAME") AS "COUNT-ALL",
       COUNT(DISTINCT "NAME") AS "COUNT-DISTINCT"
    FROM "EMPLIST"
    GROUP BY "SCODE"
```

The results are as follows:



SCODE	COUNT-ALL	COUNT-DISTINCT	
S001	3	2	...1
S002	1	1	...2
S003	1	1	...3

Explanation

1. In the case of `COUNT(NAME)`, because duplicates of the same name (`Taro Tanaka`) are counted, the execution result is `3`. In the case of `COUNT(DISTINCT NAME)`, duplicates of the same name (`Taro Tanaka`) are not counted, so the execution result is `2`.
2. Because there are no duplicate rows, the execution result is `1` is both cases.

3. Because there are no duplicate rows, and rows with null values are not counted, the execution result is `1` is both cases.

(3) Rules

1. You cannot specify binary data for the value expression.
2. Null values are not included in the calculation.
3. The data type of the execution result is `INTEGER`.
4. In the following cases, the execution result will be `0`.
    - If the number of input rows is 0
    - If the values to be calculated are all null values
5. If an overflow occurs during the computation of any set function, an overflow error is generated.

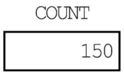
(4) Example

Example

Using the data in the sales history table (`SALESLIST`), this example determines the number of people who purchased products on or after January 1, 2014, counting those who made more than one purchase as a single person.

```
SELECT COUNT(DISTINCT "USERID") AS "COUNT"
    FROM "SALESLIST"
        WHERE "PUR-DATE">=DATE'2014-01-01'
```

Example of execution results



COUNT
150

7.22.4 MAX

`MAX` determines the maximum value.

(1) Specification format

```
general-set-function-MAX ::= {MAX([ALL] value-expression)|MAX(DISTINCT value-expressi
on)}
```

Note: Whichever form is specified, the results will be the same.

(2) Explanation of specification format

`MAX([ALL]` *value-expression*`)`:

Determines the maximum value of the results of the value expression. For details about value expressions, see 7.20 Value expression.

`ALL` can be omitted. The results will be the same regardless of whether it is specified.

`MAX(DISTINCT` *value-expression*`)`:

Determines the maximum value of the results of the value expression. For details about value expressions, see 7.20 Value expression.

(3) Rules

1. Null values are not included in the calculation.
2. In the following cases, the execution result will be a null value.
   - If the number of input rows is 0
   - If the values to be calculated are all null values
3. The following table shows the data type that can be specified in the value expression and the data type of the execution result of the general set function `MAX`.

Table 7-17: Relationship between data type that can be specified in the value expression and data type of the execution result of the general set function MAX

No.	Data type that can be specified in the value expression	Data type of the execution result of general set function MAX
1	`INTEGER`	`INTEGER`
2	`SMALLINT`	`SMALLINT`
3	`DECIMAL(`*m*`,`*n*`)`	`DECIMAL(`*m*`,`*n*`)`
4	`DOUBLE PRECISION`	`DOUBLE PRECISION`
5	`CHARACTER(`*n*`)`	`CHARACTER(`*n*`)`
6	`VARCHAR(`*n*`)`	`VARCHAR(`*n*`)`
7	`DATE`	`DATE`
8	`TIME(`*p*`)`	`TIME(`*p*`)`
9	`TIMESTAMP(`*p*`)`	`TIMESTAMP(`*p*`)`

(4) Examples

Example 1

Using the data in the employee table (`EMPLIST`), this example determines the age (`AGE`) of the oldest male employee.

```
SELECT MAX("AGE") AS "MAX-AGE"
    FROM "EMPLIST"
       WHERE "SEX"='M'
```

Example of execution results

```
MAX-AGE
     63
```

Example 2

Using the data in the employee table (`EMPLIST`), this example determines the age (`AGE`) of the oldest employee in each section (`SCODE`).

```
SELECT "SCODE",MAX("AGE") AS "MAX-AGE"
    FROM "EMPLIST"
        GROUP BY "SCODE"
```

Example of execution results

SCODE	MAX-AGE
S001	55
S002	63
S003	43

7.22.5 MIN

`MIN` determines the minimum value.

(1) Specification format

```
general-set-function-MIN ::= {MIN([ALL] value-expression)|MIN(DISTINCT value-expressi
on)}
```

Note: Whichever form is specified, the results will be the same.

(2) Explanation of specification format

`MIN([ALL]` *value-expression*`)`:

Determines the minimum value of the result of the value expression. For details about value expressions, see 7.20 Value expression.

`ALL` can be omitted. The results will be the same regardless of whether it is specified.

`MIN(DISTINCT` *value-expression*`)`:

Determines the minimum value of the results of the value expression. For details about value expressions, see 7.20 Value expression.

(3) Rules

1. Null values are not included in the calculation.
2. In the following cases, the execution result will be a null value:
   - If the number of input rows is 0
   - If the values to be calculated are all null values
3. The following table shows the data type that can be specified in the value expression and the data type of the execution result of the general set function `MIN`.

Table 7-18: Relationship between data type that can be specified in the value expression and data type of the execution result of the general set function MIN

No.	Data type that can be specified in the value expression	Data type of the execution result of general set function MIN
1	INTEGER	INTEGER
2	SMALLINT	SMALLINT

No.	Data type that can be specified in the value expression	Data type of the execution result of general set function MIN
3	DECIMAL(*m*,*n*)	DECIMAL(*m*,*n*)
4	DOUBLE PRECISION	DOUBLE PRECISION
5	CHARACTER(*n*)	CHARACTER(*n*)
6	VARCHAR(*n*)	VARCHAR(*n*)
7	DATE	DATE
8	TIME(*p*)	TIME(*p*)
9	TIMESTAMP(*p*)	TIMESTAMP(*p*)

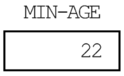
(4) Examples

Example 1

Using the data in the employee table (EMPLIST), this example determines the age (AGE) of the youngest female employee.

```
SELECT MIN("AGE") AS "MIN-AGE"
    FROM "EMPLIST"
       WHERE "SEX"='F'
```

Example of execution results



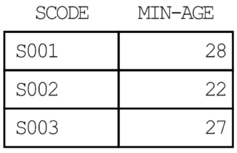
MIN-AGE
22

Example 2

Using the data in the employee table (EMPLIST), this example determines the age (AGE) of the youngest employee in each section (SCODE).

```
SELECT "SCODE",MIN("AGE") AS "MIN-AGE"
    FROM "EMPLIST"
        GROUP BY "SCODE"
```

Example of execution results



SCODE	MIN-AGE
S001	28
S002	22
S003	27

Example 3

Using the data in the employee table (EMPLIST), this example determines the ages (AGE) of the oldest employee and the youngest employee in each section where the age difference does not exceed 20 years.

```
SELECT "SCODE",MAX("AGE") AS "MAX-AGE",MIN("AGE") AS "MIN-AGE"
    FROM "EMPLIST"
        GROUP BY "SCODE"
        HAVING MAX("AGE")-MIN("AGE")<=20
```

Example of execution results

SCODE	MAX-AGE	MIN-AGE
S003	43	27

7.22.6 SUM

SUM determines the sum.

(1) Specification format

```
general-set-function-SUM ::= {SUM([ALL] value-expression)|SUM(DISTINCT value-expressi
on)}
```

(2) Explanation of specification format

SUM([ALL] *value-expression*):

Determines the sum of the result of the value expression. For details about value expressions, see 7.20 Value expression.

ALL can be omitted. The results will be the same regardless of whether it is specified.

SUM(DISTINCT *value-expression*):

Determines the sum of the result of the value expression. For details about value expressions, see 7.20 Value expression.

Duplicate values are only counted once. For example, if the values in the result of the value expression are 2, 3, 2, and 5, the execution result will be 2 + 3 + 5 = 10.

(3) Rules

1. Null values are not included in the calculation.
2. In the following cases, the execution result will be a null value.
   - If the number of input rows is 0
   - If the values to be calculated are all null values
3. The following table shows the data type that can be specified in the value expression and the data type of the execution result of the general set function SUM.

Table 7-19: Data type that can be specified in the value expression and data type of the execution result of the general set function SUM

No.	Data type that can be specified in the value expression	Data type of the execution result of general set function SUM
1	INTEGER	INTEGER
2	SMALLINT	
3	DECIMAL(*m*,*n*)	DECIMAL(38,*n*)
4	DOUBLE PRECISION	DOUBLE PRECISION

4. If an overflow occurs during the computation of any set function, an overflow error is generated.

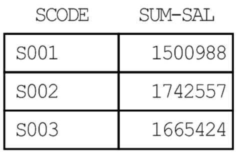
(4) Example

Example

Using the data in the salary table (SALARYLIST), this example determines the sum of the employee salaries (SAL) in each section (SCODE).

```
SELECT "SCODE",SUM("SAL") AS "SUM-SAL"
    FROM "SALARYLIST"
        GROUP BY "SCODE"
```

Example of execution results



SCODE	SUM-SAL
S001	1500988
S002	1742557
S003	1665424

7.22.7 STDDEV_POP

STDDEV_POP determines the standard deviation of a population.

(1) Specification format

```
general-set-function-STDDEV_POP ::= STDDEV_POP(value-expression)
```

(2) Explanation of specification format

value-expression:

Specifies the input values, in the form of a value expression, that make up the population whose standard deviation is to be determined. For details about value expressions, see 7.20 Value expression.

(3) Rules

1. Null values are not included in the calculation.
2. In the following cases, the execution result will be a null value.
   - If the number of input rows is 0
   - If the values to be calculated are all null values
3. The execution result of the general set function STDDEV_POP will be equal to the square root of the general set function VAR_POP.
4. The following table shows the data type that can be specified in the value expression and the data type of the execution result of the general set function STDDEV_POP.

Table 7-20: Data type that can be specified in the value expression and data type of the execution result of the general set function STDDEV_POP

No.	Data type that can be specified in the value expression	Data type of the execution result of general set function STDDEV_POP
1	INTEGER	DOUBLE PRECISION

No.	Data type that can be specified in the value expression	Data type of the execution result of general set function STDDEV_POP
2	`SMALLINT`	
3	`DECIMAL(`*m*`,`*n*`)`	
4	`DOUBLE PRECISION`	

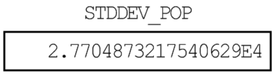
(4) Example

Example

Using the data in the salary table (`SALARYLIST`), this example determines the standard deviation of a population of employee salaries (`SALARY`).

```
SELECT STDDEV_POP("SALARY") AS "STDDEV_POP"
    FROM "SALARYLIST"
```

Example of execution results



```
      STDDEV_POP
  2.7704873217540629E4
```

7.22.8 STDDEV_SAMP

`STDDEV_SAMP` determines the standard deviation of a sample.

(1) Specification format

```
general-set-function-STDDEV_SAMP ::= STDDEV_SAMP(value-expression)
```

(2) Explanation of specification format

value-expression:

Specifies the input values, in the form of a value expression, that make up the sample whose standard deviation is to be determined. For details about value expressions, see 7.20 Value expression.

(3) Rules

1. Null values are not included in the calculation.
2. In the following cases, the execution result will be a null value.
    - If the number of input rows is 0 or 1
    - If the values to be calculated are all null values
3. The execution result of the general set function `STDDEV_SAMP` will be equal to the square root of the general set function `VAR_SAMP`.
4. The following table shows the data type that can be specified in the value expression and the data type of the execution result of the general set function `STDDEV_SAMP`.

Table 7-21: Data type that can be specified in the value expression and data type of the execution result of the general set function STDDEV_SAMP

No.	Data type that can be specified in the value expression	Data type of the execution result of general set function STDDEV_SAMP
1	`INTEGER`	`DOUBLE PRECISION`
2	`SMALLINT`	
3	`DECIMAL(`*m*`,`*n*`)`	
4	`DOUBLE PRECISION`	

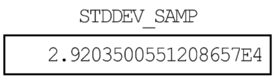
(4) Example

Example

Using the data in the salary table (`SALARYLIST`), this example determines the standard deviation of a sample of employee salaries (`SALARY`).

```
SELECT STDDEV_SAMP("SALARY") AS "STDDEV_SAMP"
    FROM "SALARYLIST"
```

Example of execution results



```
     STDDEV_SAMP
  2.9203500551208657E4
```

7.22.9 VAR_POP

`VAR_POP` determines the variance of a population.

(1) Specification format

```
general-set-function-VAR_POP ::= VAR_POP(value-expression)
```

(2) Explanation of specification format

value-expression:

Specifies the input values, in the form of a value expression, that make up the population whose variance is to be determined. For details about value expressions, see 7.20 Value expression.

(3) Rules

1. Null values are not included in the calculation.
2. In the following cases, the execution result will be a null value.
   - If the number of input rows is 0
   - If the values to be calculated are all null values
3. Letting *N* be the number of input lines, *S1* the sum of the input values, and *S2* the sum of the values obtained by squaring the input values, the result of the general set function `VAR_POP` is calculated as follows:

$(S2 - S1 \times S1 \div N) \div N$

4. The following table shows the data type that can be specified in the value expression and the data type of the execution result of the general set function `VAR_POP`.

Table 7-22: Data type that can be specified in the value expression and data type of the execution result of the general set function VAR_POP

No.	Data type that can be specified in the value expression	Data type of the execution result of general set function VAR_POP
1	`INTEGER`	`DOUBLE PRECISION`
2	`SMALLINT`	
3	`DECIMAL(`*m*`,`*n*`)`	
4	`DOUBLE PRECISION`	

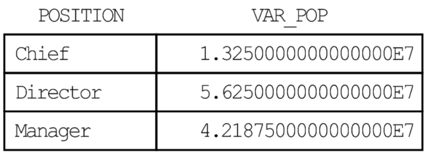
(4) Example

Example

Using the data in the salary table (`SALARYLIST`), this example determines the variance of a population of employee salaries (`SALARY`) by job class (`POSITION`).

```
SELECT "POSITION",VAR_POP("SALARY") AS "VAR_POP"
    FROM "SALARYLIST"
    GROUP BY "POSITION"
    ORDER BY "POSITION"
```

Example of execution results



POSITION	VAR_POP
Chief	1.3250000000000000E7
Director	5.6250000000000000E7
Manager	4.2187500000000000E7

7.22.10 VAR_SAMP

`VAR_SAMP` determines the variance of a sample.

(1) Specification format

```
general-set-function-VAR_SAMP ::= VAR_SAMP(value-expression)
```

(2) Explanation of specification format

value-expression:

Specifies the input values, in the form of a value expression, that make up the sample whose variance is to be determined. For details about value expressions, see 7.20 Value expression.

(3) Rules

1. Null values are not included in the calculation.

2. In the following cases, the execution result will be a null value.
   - If the number of input rows is 0 or 1
   - If the values to be calculated are all null values
3. Letting *N* be the number of input lines, *S1* the sum of the input values, and *S2* the sum of the values obtained by squaring the input values, the result of the general set function VAR_SAMP is calculated as follows:

   $(S2 - S1 \times S1 \div N) \div (N - 1)$
4. The following table shows the data type that can be specified in the value expression and the data type of the execution result of the general set function VAR_SAMP.

Table 7-23: Data type that can be specified in the value expression and data type of the execution result of the general set function VAR_SAMP

No.	Data type that can be specified in the value expression	Data type of the execution result of general set function VAR_SAMP
1	INTEGER	DOUBLE PRECISION
2	SMALLINT	
3	DECIMAL(*m*,*n*)	
4	DOUBLE PRECISION	

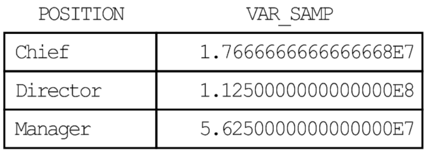
(4) Example

Example

Using the data in the salary table (SALARYLIST), this example determines the variance of a sample of employee salaries (SALARY) by job class (POSITION).

```
SELECT "POSITION",VAR_SAMP("SALARY") AS "VAR_SAMP"
    FROM "SALARYLIST"
    GROUP BY "POSITION"
    ORDER BY "POSITION"
```

Example of execution results



POSITION	VAR_SAMP
Chief	1.7666666666666668E7
Director	1.1250000000000000E8
Manager	5.6250000000000000E7

7.22.11 MEDIAN

MEDIAN determines the median of an ordered set of values. The value might be linearly interpolated.

Note

MEDIAN is an inverse distribution function that gives the same result as specifying the median value (0.5) as the argument (percentile specification) to PERCENTILE_CONT.

Letting ARG1 be the aggregated argument to MEDIAN, MEDIAN is equivalent to the following PERCENTILE_CONT function.

```
PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY ARG1 ASC)
```

(1) Specification format

```
inverse-distribution-function-MEDIAN ::= MEDIAN(value-expression)
```

(2) Explanation of specification format

value-expression:

Specify the input values for obtaining the median in the form of a value expression. For details about value expressions, see 7.20 Value expression.

(3) Rules

1. Null values are not included in the calculation.
2. If the number of input rows is 0, the execution result will be a null value.
3. The following table shows the data type that can be specified in the value expression and the data type of the execution result of the inverse distribution function `MEDIAN`.

Table 7-24: Data type that can be specified in the value expression and data type of the execution result of the inverse distribution function MEDIAN

No.	Data type that can be specified in the value expression	Data type of the execution result of the inverse distribution function MEDIAN
1	`INTEGER`	`DOUBLE PRECISION`
2	`SMALLINT`	
3	`DECIMAL(`*m*`,`*n*`)`	
4	`DOUBLE PRECISION`	

4. `MEDIAN` is calculated by linear interpolation with respect to an ordered set of values. Letting N be the number of input rows, it first calculates the row number $RN = \{1 + 0.5 \times (N - 1)\}$. Then, by linear interpolation between the values of the rows of row numbers CRN = CEIL(RN) and FRN = FLOOR(RN), the execution result of `MEDIAN` is calculated. The result of the calculation is as follows:
   - When CRN=FRN=RN: value of row RN
   - Otherwise: (CRN - RN) × (value of row FRN) + (RN - FRN) × (value of row CRN)

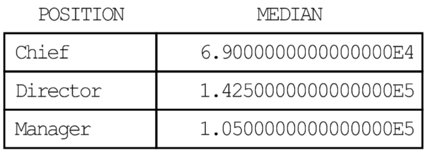
(4) Example

Example 1

Using the data in the salary table (`SALARYLIST`), this example determines the median value (50th percentile) of the employee salaries (`SALARY`) by job class (`POSITION`).

```
SELECT "POSITION",MEDIAN("SALARY") AS "MEDIAN"
    FROM "SALARYLIST"
    GROUP BY "POSITION"
    ORDER BY "POSITION"
```

Example of execution results

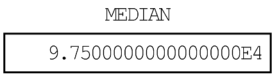


POSITION	MEDIAN
Chief	6.9000000000000000E4
Director	1.4250000000000000E5
Manager	1.0500000000000000E5

Example 2

Using the data in the salary table (SALARYLIST), this example determines the median value (50th percentile) of the employee salaries (SALARY).

```
SELECT MEDIAN("SALARY") AS "MEDIAN"
    FROM "SALARYLIST"
```



MEDIAN
9.7500000000000000E4

7.22.12 PERCENTILE_CONT

PERCENTILE_CONT determines the percentile of an ordered set of values. The value might be linearly interpolated.

(1) Specification format

```
inverse-distribution-function-PERCENTILE_CONT ::= PERCENTILE_CONT(value-specification
) WITHIN-group-specification

  WITHIN-group-specification ::= WITHIN GROUP(ORDER BY sort-specification-list)
```

(2) Explanation of specification format

value-specification:

The value for which the percentile is to be determined, expressed in the form of a value specification. For details about value specifications, see 7.21 Value specification.

The following rules apply:

- The specified value must be between 0 and 1 (data type INTEGER, SMALLINT, or DECIMAL).
- If the null value is specified, the execution result will be a null value.
- If a dynamic parameter is specified by itself, the assumed data type of the dynamic parameter is DECIMAL(3,2).

WITHIN-group-specification:

```
WITHIN-group-specification ::= WITHIN GROUP(ORDER BY sort-specification-list)
```

The WITHIN group specification specifies the data for which the percentile is to be determined and the order of the data. In *sort-specification-list*, specify the data for which the percentile is to be determined as the sort key, and the ordering of the data (ascending or descending) as the order specification. For details about sort specification lists, see 7.24 Sort specification list.

The following rules apply:

- Specification of the null-value sort order is not permitted in the sort specification list in the WITHIN group specification.
- No more than one sort specification is permitted in the sort specification list in the WITHIN group specification.

(3) Rules

1. Null values are not included in the calculation.
2. If the number of input rows is 0, the execution result will be a null value.
3. The following table shows the data types that can be specified in the sort key of the sort specification list and the data type of the execution result of the inverse distribution function `PERCENTILE_CONT`.

Table 7-25: Data types that can be specified in the sort key of the sort specification list and data type of the execution result of the inverse distribution function PERCENTILE_CONT

No.	Data types that can be specified in the sort key of the sort specification list	Data type of the execution result of the inverse distribution function PERCENTILE_CONT
1	`INTEGER`	`DOUBLE PRECISION`
2	`SMALLINT`	
3	`DECIMAL(`*m*`,`*n*`)`	
4	`DOUBLE PRECISION`	

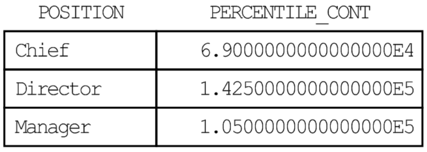
4. `PERCENTILE_CONT` is calculated by linear interpolation with respect to an ordered set of values. Letting *N* be the number of input rows and *P* be the value of the specified argument, it first calculates the row number *RN* = {1 + *P* × (*N* - 1)}. Then, by linear interpolation between the values of the rows of row numbers *CRN* = CEIL(*RN*) and *FRN* = FLOOR(*RN*), the execution result of `PERCENTILE_CONT` is calculated. The result of the calculation is as follows:
   - When *CRN*=*FRN*=*RN*: the value of row *RN*
   - Otherwise: (*CRN* - *RN*) × (value of row *FRN*) + (*RN* - *FRN*) × (value of row *CRN*)

(4) Example

Example 1

Using the data in the salary table (`SALARYLIST`), this example determines the median value (50th percentile) of the employee salaries (`SALARY`) by job class (`POSITION`).

```
SELECT "POSITION",
        PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY "SALARY") AS "PERCENTILE_CONT"
    FROM "SALARYLIST"
    GROUP BY "POSITION"
    ORDER BY "POSITION"
```

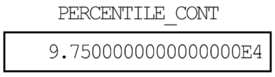


POSITION	PERCENTILE_CONT
Chief	6.9000000000000000E4
Director	1.4250000000000000E5
Manager	1.0500000000000000E5

Example 2

Using the data in the salary table (`SALARYLIST`), this example determines the median value (50th percentile) of the employee salaries (`SALARY`).

```
SELECT PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY "SALARY") AS "PERCENTILE_CONT"
    FROM "SALARYLIST"
```



PERCENTILE_CONT
9.7500000000000000E4

7.22.13 PERCENTILE_DISC

PERCENTILE_DISC determines the percentile of an ordered set of values. It returns a result from the set of values.

(1) Specification format

```
inverse-distribution-function-PERCENTILE_DISC ::= PERCENTILE_DISC(value-specification
) WITHIN-group-specification

  WITHIN-group-specification ::= WITHIN GROUP(ORDER BY sort-specification-list)
```

(2) Explanation of specification format

value-specification:

The value for which the percentile is to be determined, expressed in the form of a value specification. For details about value specifications, see 7.21 Value specification.

The following rules apply:

- The specified value must be between 0 and 1 (data type INTEGER, SMALLINT, or DECIMAL).
- If the null value is specified, the execution result will be a null value.
- If a dynamic parameter is specified by itself, the assumed data type of the dynamic parameter is DECIMAL(3,2).

WITHIN-group-specification:

```
WITHIN-group-specification ::= WITHIN GROUP(ORDER BY sort-specification-list)
```

The WITHIN group specification specifies the data for which the percentile is to be determined and the order of the data. In *sort-specification-list*, specify the data for which the percentile is to be determined as the sort key, and the ordering of the data (ascending or descending) as the order specification. For details about sort specification lists, see 7.24 Sort specification list.

The following rules apply:

- Specification of the null-value sort order is not permitted in the sort specification list in the WITHIN group specification.
- No more than one sort specification is permitted in the sort specification list in the WITHIN group specification.

(3) Rules

1. Null values are not included in the calculation.
2. If the number of input rows is 0, the execution result will be a null value.
3. The following table shows the data types that can be specified in the sort key of the sort specification list and the data type of the execution result of the inverse distribution function PERCENTILE_DISC.

Table 7-26: Data types that can be specified in the sort key of the sort specification list and data type of the execution result of the inverse distribution function PERCENTILE_DISC

No.	Data types that can be specified in the sort key of the sort specification list	Data type of the execution result of the inverse distribution function PERCENTILE_DISC
1	INTEGER	INTEGER
2	SMALLINT	SMALLINT

No.	Data types that can be specified in the sort key of the sort specification list	Data type of the execution result of the inverse distribution function PERCENTILE_DISC
3	DECIMAL(*m*,*n*)	DECIMAL(*m*,*n*)
4	DOUBLE PRECISION	DOUBLE PRECISION
5	CHAR(*n*)	CHAR(*n*)
6	VARCHAR(*n*)	VARCHAR(*n*)
7	DATE	DATE
8	TIME(*p*)	TIME(*p*)
9	TIMESTAMP(*p*)	TIMESTAMP(*p*)

4. PERCENTILE_DISC returns a result from an ordered set of values. If *P* is the value of the specified argument, it sorts the values in the value expression specified in the sort specification list, and then, from among those values, returns the value that is greater than or equal to *P* with the smallest CUME_DIST value with respect to the same sort specification list.

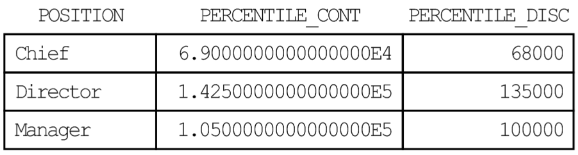
(4) Example

Example

Using the data in the salary table (SALARYLIST), this example determines the median value (50th percentile) of the employee salaries (SALARY) by job class (POSITION).

Both PERCENTILE_CONT and PERCENTILE_DISC can be used to determine the median value.

```
SELECT "POSITION",
        PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY "SALARY" ASC) AS "PERCENTILE_CO
NT",
        PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY "SALARY" ASC) AS "PERCENTILE_DI
SC"
    FROM "SALARYLIST"
    GROUP BY "POSITION"
    ORDER BY "POSITION"
```



POSITION	PERCENTILE_CONT	PERCENTILE_DISC
Chief	6.9000000000000000E4	68000
Director	1.4250000000000000E5	135000
Manager	1.0500000000000000E5	100000

As shown above, the results of PERCENTILE_CONT and PERCENTILE_DISC can differ. This is because PERCENTILE_CONT returns results that are linearly interpolated, while PERCENTILE_DISC returns results only from the set of values upon which calculations are being performed.

7.22.14 Common rules and considerations for set functions

(1) Explanation of terms

1. A general set function in which DISTINCT is specified is called a *DISTINCT set function*. A general set function in which ALL is specified is called an *ALL set function*.
2. The following value expressions are called the *aggregated arguments* of a set function.

- In the case of the inverse distribution functions PERCENTILE_CONT and PERCENTILE_DISC, the value expression specified as the sort key in the WITHIN group specification

  Example:

  ```
  SELECT PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY "C1") FROM "T1"
  ```

  The underlined portion indicates the aggregated argument.

- In the case of set functions other than the inverse distribution functions PERCENTILE_CONT and PERCENTILE_DISC, the value expression specified as the argument to the set function

  Example:

  ```
  SELECT "C1",SUM("C2") FROM "T1" GROUP BY "C1"
  ```

  The underlined portion indicates the aggregated argument.

3. A column specification within an aggregated argument is called an *aggregated column specification*.

   Example:

   ```
   SELECT "C1",SUM("C2"+1) FROM "T1" GROUP BY "C1"
   ```

   The underlined portion indicates the aggregated column specification.

4. A query specification that directly contains a FROM clause that contains a table reference that is referenced by an aggregated column specification is called a *qualified query* of that aggregated column specification.

   Example:

   ```
   SELECT "C1",SUM("T1"."C2") FROM "T1" GROUP BY "C1"
   ```

   The underlined portion (the entire query) indicates the qualified query.

The following example illustrates a qualified query with an external reference.

Example:

```
SELECT "C1" FROM "T1" GROUP BY "C1" HAVING EXISTS   ...[1]
    (SELECT * FROM "T2" WHERE MAX("T1"."C2")>"T2"."C1")
```

Explanation

- The aggregated column specification in the set function MAX("T1"."C2") is "T1"."C2".
- The table referenced by "T1"."C2" is "T1".
- The query specification whose FROM clause directly contains "T1" is the part indicated by [1].
- The qualified query is therefore the query specification in [1].

(2) Common rules

1. A set function can be specified in a selection expression, HAVING clause, or ORDER BY clause that is directly contained in a qualified query of that set function. However, restrictions apply when specifying a set function in an ORDER BY clause. For details about the restrictions, see (2) Rules for specifying value expressions as sort keys in 7.24.2 Rules for specifying a sort specification list in an ORDER BY clause.
2. If the value expression specified as an aggregated argument is not an independent column specification, you cannot specify multiple inverse distribution functions in the same query specification.

   Example of an SQL statement that generates an error:

```
SELECT PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY "C1"+"C2"),
       PERCENTILE_DISC(0.25) WITHIN GROUP (ORDER BY "C1"+"C2")
    FROM "T1"
```

Because the value expression specified as an aggregated argument is not an independent column specification, multiple inverse distribution functions cannot be specified.

3. If you specify multiple inverse distribution functions in the same query specification, the column specifications provided as aggregated arguments must reference the same column.

   Example of an SQL statement that generates an error:

```
SELECT PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY "C1"),
       PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY "C2")
    FROM "T1"
```

The column specifications in inverse distribution functions must reference the same column.

> **Note**
>
> The following is an example of where multiple inverse distribution functions can be specified.
>
> Example:
>
> ```
> SELECT PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY "C1"),
>        PERCENTILE_DISC(0.25) WITHIN GROUP (ORDER BY "GC1")
>     FROM "T1"
>     GROUP BY "C1" AS "GC1"
> ```
>
> Because the column specifications in inverse distribution functions reference the same grouping column, this statement does not result in an error.

4. If you specify multiple inverse distribution functions in the same query specification, make sure that the same order specification is provided for the sort specification in the `WITHIN` group specification in all of the functions.

   Example of an SQL statement that generates an error:

```
SELECT PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY "C1" ASC),
       PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY "C1" DESC)
    FROM "T1"
```

5. You cannot specify a dynamic parameter as the value expression specified for the aggregated argument in a set function.

   Example of an SQL statement that generates an error:

```
SELECT MAX(CASE WHEN "C1">? THEN "C1" ELSE "C1"*100 END) FROM "T1"
```

You cannot specify a dynamic parameter in the underlined portion.

6. You cannot specify a set function, a subquery, a window function, or the RANDOMROW scalar function inside of a set function.

   Example of an SQL statement that generates an error:

```
SELECT SUM(CASE WHEN MAX("C1")>10000 THEN "C1" END) FROM "T1"
```

You cannot specify a set function in the underlined portion.

7. A single query specification can include a maximum of 64 `DISTINCT` set functions with different aggregated arguments (excluding the `DISTINCT` set functions specified in window functions).

8. If you specify something other than a single column specification as the value expression specified for the aggregated argument in a set function, you cannot specify an external reference column in that value expression.

   Example of an SQL statement that generates an error:

   ```
   SELECT SUM("C1") FROM "T1"
      HAVING EXISTS(SELECT * FROM "T2" WHERE AVG("T1"."C2"*1.05)>"C2")
   ```

   You cannot specify an external reference column in the underlined portion.

9. The input to the set function will be the results from the last-specified clause among the `FROM`, `WHERE`, and `GROUP BY` clauses. If a `GROUP BY` clause is specified, the results for each group will be input to the set function.

10. When a set function is used in a window function, the input to the set function will be the set of rows included in the window frame of the current row.

11. You cannot specify an external reference column as the aggregated argument to an inverse distribution function.

12. No more than one sort specification is permitted in the sort specification list specified in an inverse distribution function.

(3) Common considerations

1. If a `DISTINCT` set function or inverse distribution function is specified, a work table might be created. If the size of the work table DB area where the work table is created has not been estimated correctly, it might result in performance degradation. For details about estimating the size of the work table DB area, see the *HADB Setup and Operation Guide*. For details about work tables, see *Considerations when executing an SQL statement that creates work tables* in the *HADB Application Development Guide*.

2. If global hash grouping is applied as the method of grouping performed during deduplication of `DISTINCT` set functions, a derived table might be created. HADB automatically assigns a correlation name in the following format to the derived table:

   ```
   ##DRVTBL_xxxxxxxxxx
   ```

   In the preceding format, *xxxxxxxxxx* is a 10-digit integer.

   For details about global hash grouping, see *Global hash grouping* in *Hash grouping* in the *HADB Application Development Guide*.

3. If a `GROUP BY` clause or `HAVING` clause is specified, no execution results are output for groups where the number of input rows is 0.

7.23 Window functions

Using a window function, you can specify a range of rows derived from the results of a table expression, and then determine aggregated values for the rows in that range.

The window functions are shown in the following table:

Table 7-27: List of window functions

No.	Window function	Description
1	RANK	Determines the ranking of the rows in an ordered set of rows. The ranking values might not be contiguous integer values.
2	DENSE_RANK	Determines the ranking of the rows in an ordered set of rows. The ranking values will be contiguous integer values.
3	CUME_DIST	Determines the relative position of a row in an ordered set of rows. The CUME_DIST of row *R* is the number of rows that are in front of *R* in the window (partition) or that have the same sort key value as *R*, divided by the number of rows in the window (partition) of *R*.
4	ROW_NUMBER	Assigns a unique number to each row in an ordered set of rows.
5	Set function	Determines the value of a set function with respect to a window frame.

7.23.1 Specification format for window functions

(1) Specification format

```
window-function ::= {RANK()
                   | DENSE_RANK()
                   | CUME_DIST()
                   | ROW_NUMBER()
                   | set-function} OVER(window-specification)


  window-specification ::= [window-partition-clause] [window-order-clause]
                           [window-frame-clause]
    window-partition-clause ::= PARTITION BY value-expression[,value-expression]...
    window-order-clause ::= ORDER BY sort-specification-list
    window-frame-clause ::= {ROWS | RANGE} {window-frame-start | window-frame-range}

  window-frame-start ::= {UNBOUNDED PRECEDING
                         |window-frame-value-specification PRECEDING
                         |CURRENT ROW}
  window-frame-range ::= BETWEEN window-frame-start-boundary
                                AND window-frame-end-boundary
    window-frame-start-boundary ::= window-frame-boundary
    window-frame-end-boundary ::= window-frame-boundary
    window-frame-boundary ::= {UNBOUNDED PRECEDING
                              | window-frame-value-specification PRECEDING
                              | CURRENT ROW
                              | window-frame-value-specification FOLLOWING
                              | UNBOUNDED FOLLOWING}
      window-frame-value-specification ::= {unsigned-value-specification | labeled-du
ration}
```

(2) Explanation of specification format

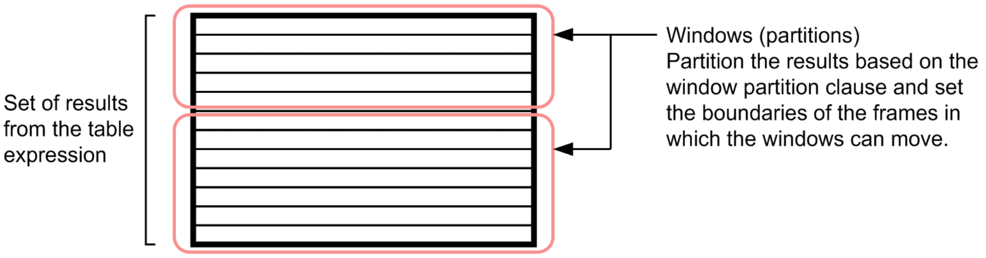
(a) window-partition-clause

```
window-partition-clause ::= PARTITION BY value-expression[,value-expression]...
```

Partitions the results of the table expression using the results of *value-expression*. If *window-partition-clause* is omitted, the result will be a single window (partition) for the entire table expression.

The following figure gives a functional overview of the window partition clause:

Figure 7-4: Functional overview of the window partition clause


Set of results from the table expression
Windows (partitions)
Partition the results based on the window partition clause and set the boundaries of the frames in which the windows can move.

The following rules apply:

- You must specify a value expression that contains a column specification in the window partition clause.
- No more than 16 value expressions can be specified in the window partition clause.
- A column specified in a single column specification in the window partition clause cannot be specified again.
- You cannot specify binary data for the value expressions in the window partition clause.

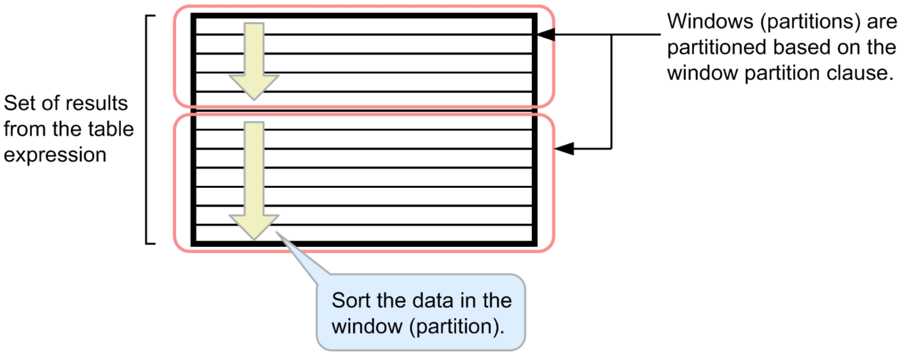
(b) window-order-clause

```
window-order-clause ::= ORDER BY sort -specification-list
```

Specify this to order (sort) the data in a window (partition). For details about the specification format and rules of the sort specification list, see 7.24 Sort specification list.

The following figure gives a functional overview of the window order clause.

Figure 7-5: Functional overview of the window order clause


Set of results from the table expression
Windows (partitions) are partitioned based on the window partition clause.
Sort the data in the window (partition).

The following rules apply:

- The data types that can be specified in the sort key of the sort specification list are shown in the following table.

Table 7-28: Data types that can be specified in the sort key of the sort specification list of the window order clause

Window frame clause specification		Window frame value specification	Data type of sort key			
			Numeric data	Character string data	Datetime data	Binary data
Specified	`ROWS`	--	Y	Y	Y	N
	`RANGE`	Specified	Y	N	Y	N
		Not specified	Y	Y	Y	N
Not specified		--	Y	Y	Y	N

Legend:

Y: Can be specified.

N: Cannot be specified.

--: Not applicable.

- If a dynamic parameter is specified by itself for the sort key of the sort specification list, the assumed data type of the dynamic parameter is `INTEGER`.
- You cannot specify a window order clause when using a `DISTINCT` set function or inverse distribution function as the window function.
- In order to specify `RANK`, `DENSE_RANK`, or `CUME_DIST`, you must specify a window order clause in the window specification.
- If you specify `RANGE` in the window frame clause and a window frame value specification in the window frame boundary, no more than one sort specification is permitted in the sort specification list in the window order clause.

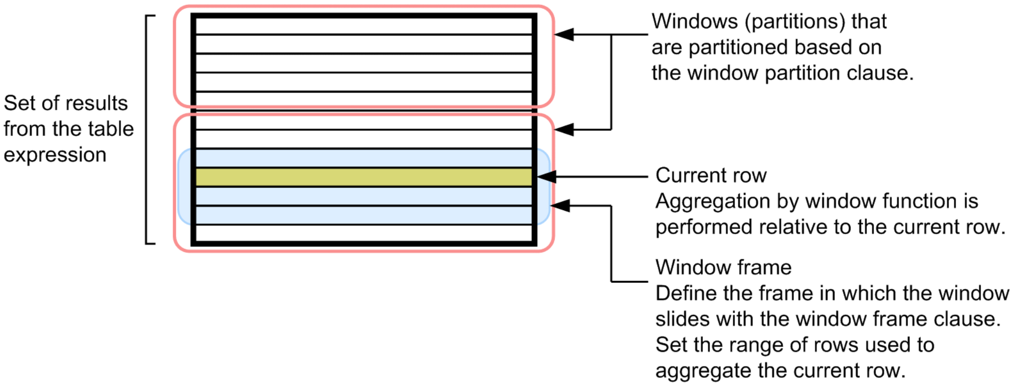
(c) window-frame-clause

```
window-frame-clause ::= {ROWS | RANGE} {window-frame-start | window-frame-range}
```

Specifies a window frame to serve as the aggregation range for the window function.

The following figure gives a functional overview of the window frame clause.

Figure 7-6: Functional overview of the window frame clause


Set of results
from the table
expression
Windows (partitions) that
are partitioned based on
the window partition clause.
Current row
Aggregation by window function is
performed relative to the current row.
Window frame
Define the frame in which the window
slides with the window frame clause.
Set the range of rows used to
aggregate the current row.

If `ROWS` is specified in the window frame clause, a physical row-by-row window frame is used. If `RANGE` is specified, the window frame is implemented as a logical offset (a logical interval such as a datetime).

When *window-frame-clause* is omitted, the range of the window frame will be as follows.

- **When a window order clause is specified**

  The range of the window frame is equivalent to specifying the following window frame range:

  ```
  RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
  ```

  The range of the window function will extend from the first row of the window (partition) to the current row. However, because `RANGE` is assumed, later rows with the same sort key value as the current row will also be included in the aggregation range.

- **When a window order clause is not specified**

  The range of the window function will be the window (partition) containing the current row.

The following rules apply:

- If a window frame clause is specified, either `COUNT(*)` or a general set function (excluding `DISTINCT` set functions) must be used as the window function.
- If you specify a window frame clause other than one of the following that represents all windows (partitions), you must specify a window order clause:
  - `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`
  - `RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`
- If `RANK`, `DENSE_RANK`, `CUME_DIST`, or `ROW_NUMBER` is specified, you cannot specify a window frame clause in the window specification.
- If *window-frame-start* is specified in the window frame clause, the range of the window frame is equivalent to the following:

  ```
  BETWEEN window-frame-start AND CURRENT ROW
  ```

- If a window frame range is specified in the window frame clause, the upper boundary of the window frame is set to *window-frame-start-boundary*, and the lower boundary of the window frame is set to *window-frame-end-boundary*.
- The window frame boundary can be set to one of the following.
  - `UNBOUNDED PRECEDING`:

    The window frame will start from the first row in the window (partition). `UNBOUNDED PRECEDING` can be set for *window-frame-start-boundary*.

  - `UNBOUNDED FOLLOWING`:

    The window frame will end at the last row in the window (partition). `UNBOUNDED FOLLOWING` can be set for *window-frame-end-boundary*.

  - `CURRENT ROW`:

    • If `ROWS` is specified:

    If it is specified in *window-frame-start-boundary*, the window frame will start from the current row. If it is specified in *window-frame-end-boundary*, the window frame will end at the current row.

    • If `RANGE` is specified:

    If it is specified in *window-frame-start-boundary*, the window frame will start from the first row with the same sort key value as the current row. If it is specified in *window-frame-end-boundary*, the window frame will end at the last row with the same sort key value as the current row.

  - *window-frame-value-specification* `PRECEDING` or *window-frame-value-specification* `FOLLOWING`

    • If `ROWS` is specified:

    The value of *window-frame-value-specification* is a physical row offset from the current row. The data type of *unsigned-value-specification* must be `INTEGER`. You cannot specify a labeled duration.

- If `RANGE` is specified:

The value of *window-frame-value-specification* is a logical offset from the sort key value of the current row. The following table shows the data type of the sort key specified in the window order clause, and the unsigned value specification or labeled duration that can be specified.

Table 7-29: Data type of the sort key specified in the window order clause, and the unsigned value specification or labeled duration that can be specified (when RANGE is specified)

Data type of the sort key specified in window-order-clause	Unsigned value specification or labeled duration that can be specified
Numeric data	An unsigned value specification consisting of numeric data
`DATE`	Labeled duration (`YEARS`, `MONTHS`, `DAYS`)
`TIME`	Labeled duration (`HOURS`, `MINUTES`, `SECONDS`, `MILLISECONDS`, `MICROSECONDS`, `NANOSECONDS`, `PICOSECONDS`)
`TIMESTAMP`	Labeled duration (`YEARS`, `MONTHS`, `DAYS`, `HOURS`, `MINUTES`, `SECONDS`, `MILLISECONDS`, `MICROSECONDS`, `NANOSECONDS`, `PICOSECONDS`)

- Certain combinations of window frame boundaries cannot be specified in the window frame range. For example:
  - `UNBOUNDED FOLLOWING` is never permitted for *window-frame-start-boundary*.
  - `UNBOUNDED PRECEDING` is never permitted for *window-frame-end-boundary*.

The following table shows the combinations that can be specified:

Table 7-30: Combinations that can be specified

Window frame start boundary specification	Window frame end boundary specification			
	UNBOUNDED FOLLOWING	CURRENT ROW	window-frame-value-specification PRECEDING	window-frame-value-specification FOLLOWING
`UNBOUNDED PRECEDING`	Y	Y	Y	Y
`CURRENT ROW`	Y	Y	N	Y
window-frame-value-specification `PRECEDING`	Y	Y	Y	Y
window-frame-value-specification `FOLLOWING`	Y	N	N	Y

Legend:

Y: Can be specified.

N: Cannot be specified.

- When you specify a labeled duration for *window-frame-value-specification* in the window frame range, the same labeled duration qualifier must be used for both *window-frame-start-boundary* and *window-frame-end-boundary*. For details about labeled duration qualifiers, see 7.28.1 Specification format and rules for labeled durations.

Example:

```
BETWEEN 2 DAYS PRECEDING AND 1 DAYS PRECEDING
```

- When you specify a labeled duration for *window-frame-value-specification*, only a value specification can be specified for *value-expression-primary* in the labeled duration.
- When you specify a labeled duration for *window-frame-value-specification*, the following value ranges are permitted:

  `YEARS: 0 to 9,998`

  `MONTHS: 0 to 119,987`

  `DAYS: 0 to 3,652,058`

  `HOURS: 0 to 87,649,415`

  `MINUTES: 0 to 5,258,964,959`

  `SECONDS: 0 to 315,537,897,599`

  `MILLISECONDS: 0 to 315,537,897,599,999`

  `MICROSECONDS: 0 to 315,537,897,599,999,999`

  `NANOSECONDS: 0 to 9,223,372,036,854,775,807`

  `PICOSECONDS: 0 to 9,223,372,036,854,775,807`
- An error results if you specify a negative value or null value for *window-frame-value-specification*.
- The following table shows which data type is assumed when a dynamic parameter is specified for *window-frame-value-specification*.

Table 7-31: Assumed data type when the window frame value specification is a dynamic parameter

Window frame specification	Data type of the sort key in the window order clause	Assumed data type of the window frame value specification
RANGE	SMALLINT	SMALLINT
	INTEGER	INTEGER
	DECIMAL	DECIMAL
	DOUBLE PRECISION	DOUBLE PRECISION
	DATE	-- (Only a labeled duration is permitted)
	TIME	
	TIMESTAMP	
ROWS	--	INTEGER

Legend:

--: Not applicable.

7.23.2 Rules for specifying windows (partitions)

This subsection describes how rows are split into windows (partitions) and the order of rows inside the windows (partitions).

(1) How rows are split into windows (partitions)

The set of rows from the table to which the window specification applies is split into windows (partitions) based on the result of the value expression specified in the window partition clause. The rows that make up the windows (partitions) are determined according to the following rules:

- When the result of the value expression in the window partition clause is not the null value

  The windows (partitions) are composed of the rows for which the result of the value expression is the same value. For the comparison rules used to determine if two values are the same, see (1) Data types that can be compared in 6.2.2 Data types that can be converted, assigned, and compared.

- When the result of the value expression in the window partition clause is the null value

  The window (partition) is composed of the rows for which the result of the value expression is the null value.

(2) Order of rows inside the windows (partitions)

The order of rows inside the windows (partitions) is as follows:

- If a window order clause is specified

  The rows are ordered as specified in the sort specification list of the window order clause. For details about sort specification lists, see 7.24 Sort specification list.

- If no window order clause is specified

  No particular ordering can be assumed. The order of rows inside the windows (partitions) is determined by the order in which the rows are actually retrieved.

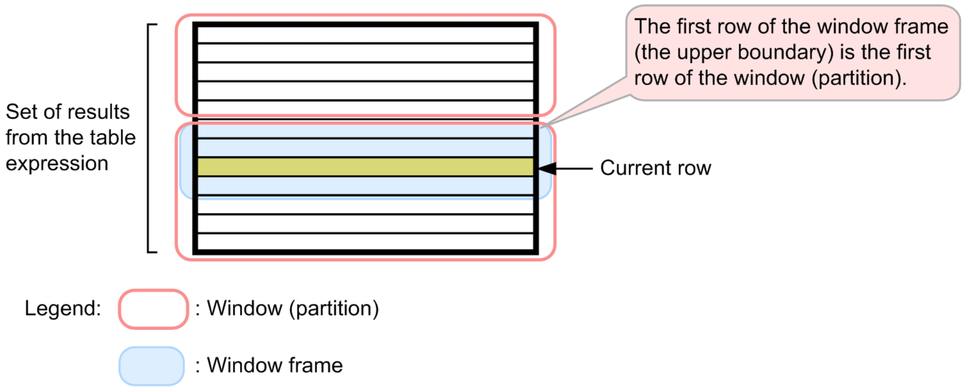
7.23.3 Rules for specifying the window frame (when RANGE is specified in the window frame clause)

When `RANGE` is specified in the window frame clause, the upper and lower boundaries of the window frame are determined as follows.

(1) Upper boundary of the window frame

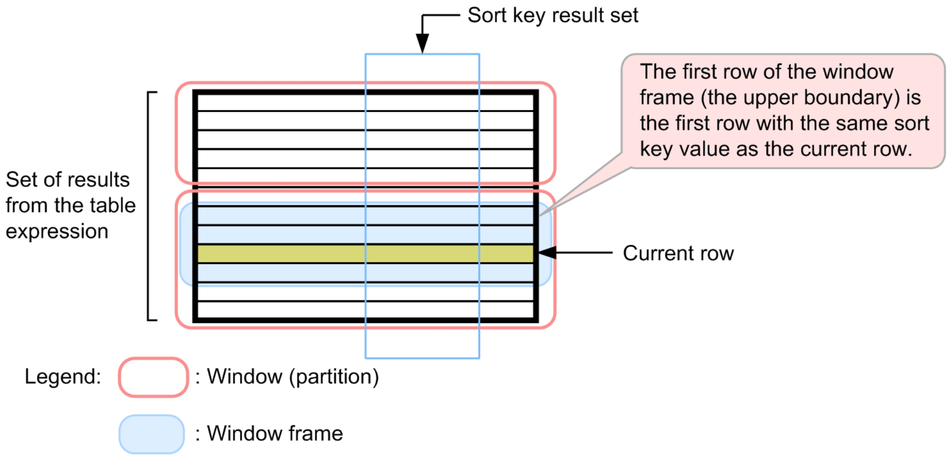
(a) When the window frame start boundary is UNBOUNDED PRECEDING

The first row of the window frame (the upper boundary) will be the first row of the window (partition).


Set of results from the table expression
The first row of the window frame (the upper boundary) is the first row of the window (partition).
Current row
Legend:
: Window (partition)
: Window frame

(b) When the window frame start boundary is CURRENT ROW

The first row of the window frame (the upper boundary) will be the first row with the same sort key value as the current row.


Sort key result set
The first row of the window frame (the upper boundary) is the first row with the same sort key value as the current row.
Set of results from the table expression
Current row
Legend:
: Window (partition)
: Window frame

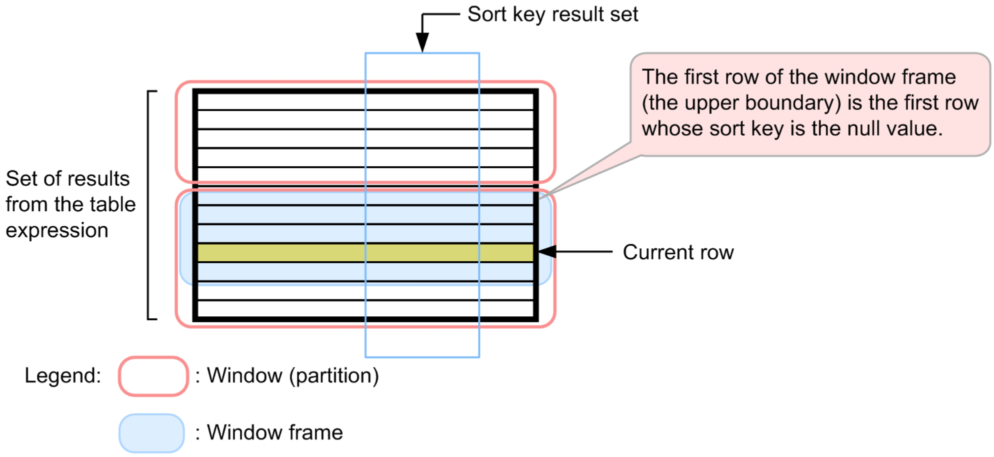
(c) When the window frame start boundary is window-frame-value-specification PRECEDING or window-frame-value-specification FOLLOWING

The upper boundary of the window frame is determined based on the value of the sort key specified in the window order clause.

■ When the sort key value of the current row is the null value

The first row of the window frame (the upper boundary) will be same, regardless of whether *window-frame-value-specification* `PRECEDING` or *window-frame-value-specification* `FOLLOWING` is specified for the window frame start boundary.

The first row of the window frame (the upper boundary) will be the topmost row among the upper rows whose sort key is the null value.


Sort key result set
The first row of the window frame (the upper boundary) is the first row whose sort key is the null value.
Set of results from the table expression
Current row
Legend:
: Window (partition)
: Window frame

■ When the sort key value of the current row is not the null value

The upper boundary of the window frame is determined as follows:

- When the window frame boundary is *window-frame-value-specification* `PRECEDING`

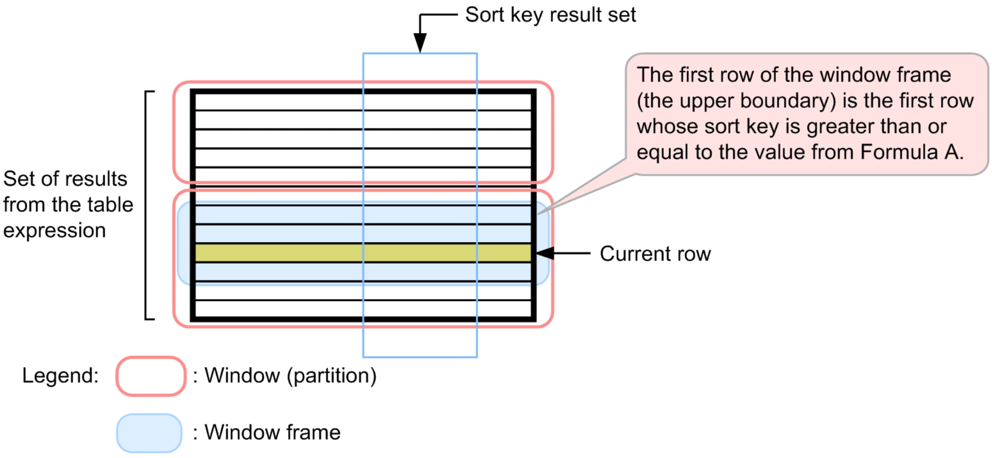
  If the sort order specification is `ASC` (ascending), the first row of the window frame (the upper boundary) will be the first row whose sort key is greater than or equal to the value from Formula A.

  When the sort order specification is `DESC` (descending), the first row of the window frame (the upper boundary) will be the first row whose sort key is less than or equal to the value from Formula A.

  Formula A:

  • When the sort order specification is `ASC` (ascending): *the sort key value of the current row* - *window-frame-value-specification*

• When the sort order specification is `DESC` (descending): *the sort key value of the current row* + *window-frame-value-specification*


Sort key result set
The first row of the window frame (the upper boundary) is the first row whose sort key is greater than or equal to the value from Formula A.
Set of results from the table expression
Current row
Legend:
: Window (partition)
: Window frame

Note: In this example, the sort order specification is `ASC` (ascending).

If the value from Formula A is a value that cannot be represented in the data type of the result, the window frame is determined using the maximum or minimum value that can be represented by the data type of the result.

- When the window frame boundary is *window-frame-value-specification* `FOLLOWING`

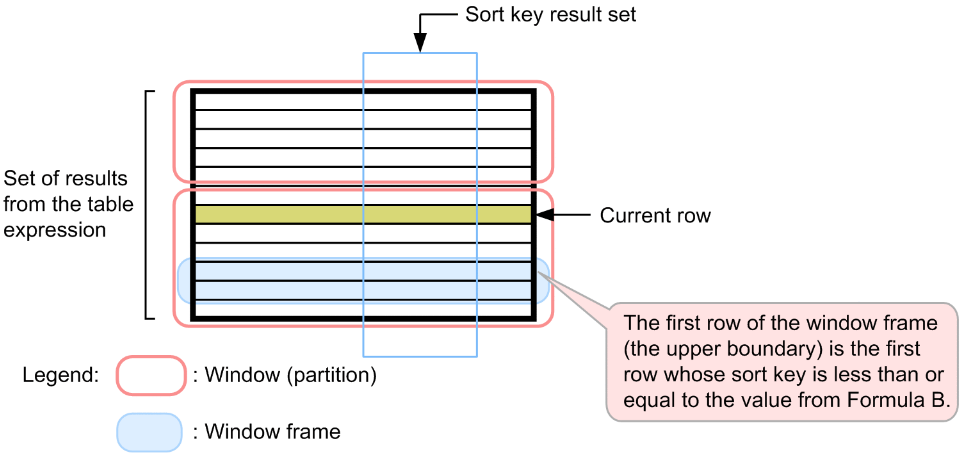
If the sort order specification is `ASC` (ascending), the first row of the window frame (the upper boundary) will be the first row whose sort key is greater than or equal to the value from Formula B.

When the sort order specification is `DESC` (descending), the first row of the window frame (the upper boundary) will be the first row whose sort key is less than or equal to the value from Formula B.

Formula B:

• When the sort order specification is `ASC` (ascending): *the sort key value of the current row* + *window-frame-value-specification*

• When the sort order specification is `DESC` (descending): *the sort key value of the current row* - *window-frame-value-specification*


Sort key result set
Set of results from the table expression
Current row
The first row of the window frame (the upper boundary) is the first row whose sort key is less than or equal to the value from Formula B.
Legend:
: Window (partition)
: Window frame

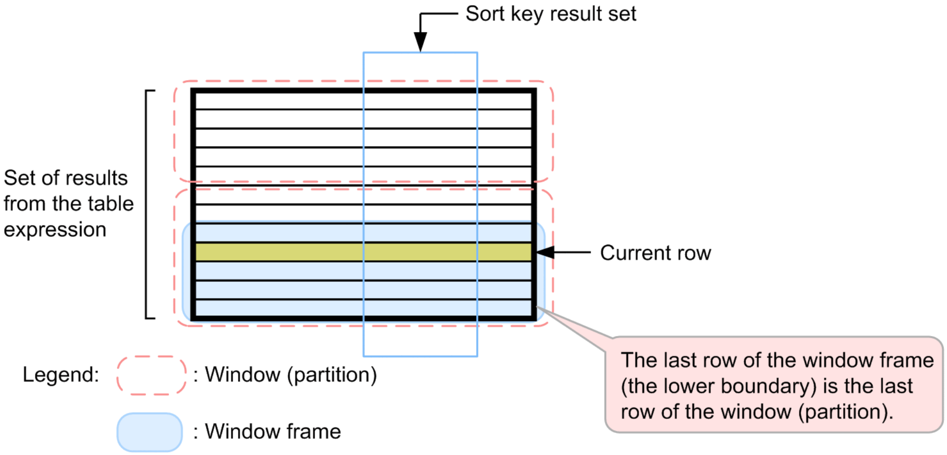
Note: In this example, the sort order specification is `DESC` (descending).

If the value from Formula B is a value that cannot be represented in the data type of the result, the window frame is determined using the maximum or minimum value that can be represented by the data type of the result.

(2) The lower boundary of the window frame

(a) When the window frame end boundary is UNBOUNDED FOLLOWING

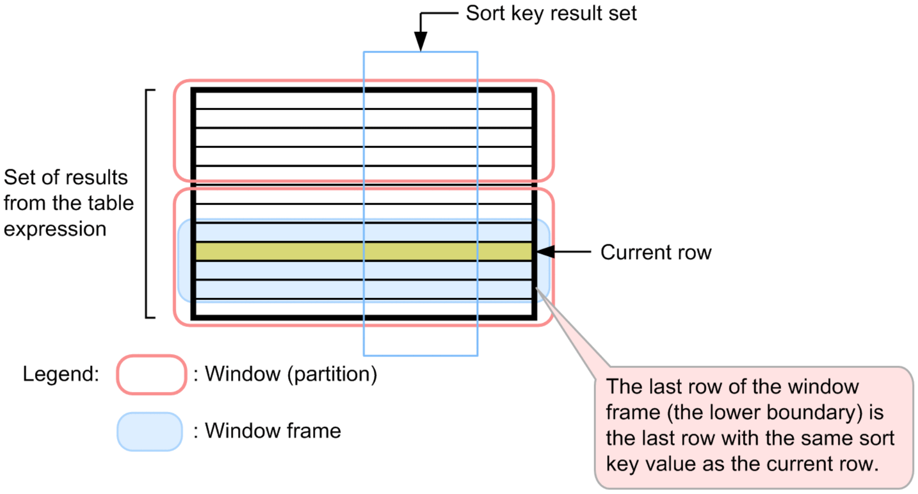
The last row of the window frame (the lower boundary) will be the last row of the window (partition).



Sort key result set
Set of results from the table expression
Current row
Legend:
: Window (partition)
: Window frame
The last row of the window frame (the lower boundary) is the last row of the window (partition).

(b) When the window frame end boundary is CURRENT ROW

The last row of the window frame (the lower boundary) will be the last row with the same sort key value as the current row.



Sort key result set
Set of results from the table expression
Current row
Legend:
: Window (partition)
: Window frame
The last row of the window frame (the lower boundary) is the last row with the same sort key value as the current row.

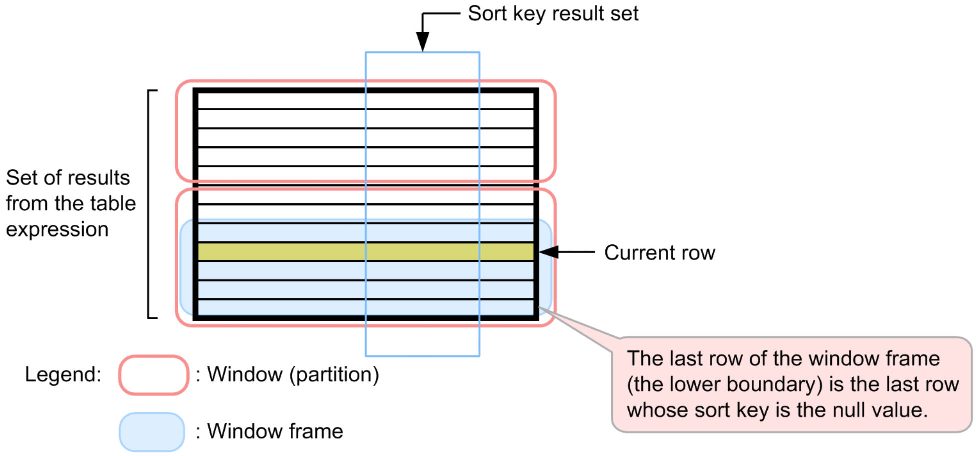
(c) When the window frame end boundary is window-frame-value-specification PRECEDING or window-frame-value-specification FOLLOWING

The lower boundary of the window frame is determined based on the value of the sort key specified in the window order clause.

■ When the sort key value of the current row is the null value

The last row of the window frame (the lower boundary) will be the same, regardless of whether *window-frame-value-specification* `PRECEDING` or *window-frame-value-specification* `FOLLOWING` is specified for the window frame end boundary.

The last row of the window frame (the lower boundary) will be the last row whose sort key is the null value.


Sort key result set
Set of results from the table expression
Current row
The last row of the window frame (the lower boundary) is the last row whose sort key is the null value.
Legend:
: Window (partition)
: Window frame

■ When the sort key value of the current row is not the null value

The lower boundary of the window frame is determined as follows:

- When the window frame boundary is *window-frame-value-specification* `PRECEDING`

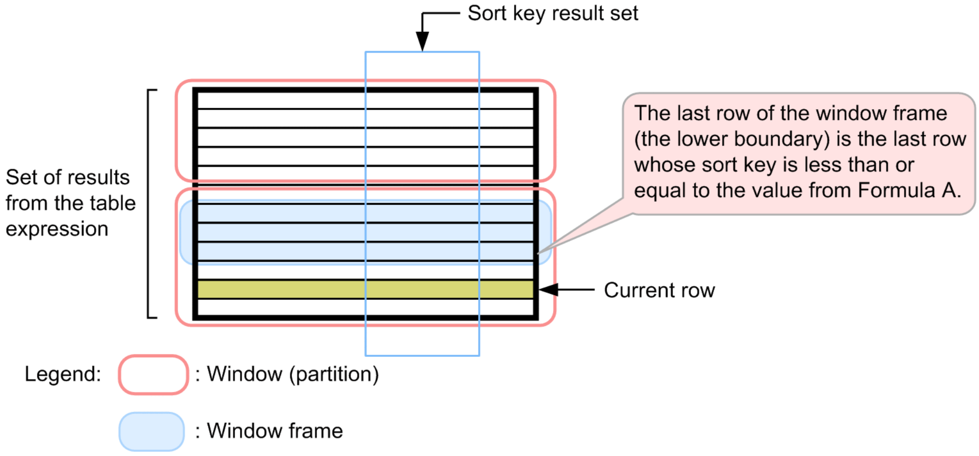
  If the sort order specification is `ASC` (ascending), the last row of the window frame (the lower boundary) will be the last row whose sort key is less than or equal to the value from Formula A.

  When the sort order specification is `DESC` (descending), the last row of the window frame (the lower boundary) will be the last row whose sort key is greater than or equal to the value from Formula A.

  Formula A:

  • When the sort order specification is `ASC` (ascending): *the sort key value of the current row* - *window-frame-value-specification*

  • When the sort order specification is `DESC` (descending): *the sort key value of the current row* + *window-frame-value-specification*


Sort key result set
Set of results from the table expression
The last row of the window frame (the lower boundary) is the last row whose sort key is less than or equal to the value from Formula A.
Current row
Legend:
: Window (partition)
: Window frame

Note: In this example, the sort order specification is `ASC` (ascending).

  If the value from Formula A is a value that cannot be represented in the data type of the result, the window frame is determined using the maximum or minimum value that can be represented by the data type of the result.

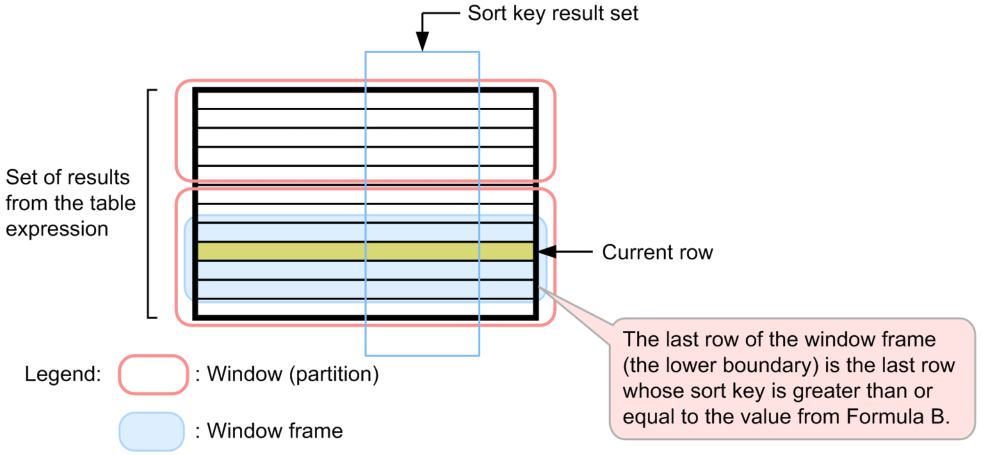
- When the window frame boundary is *window-frame-value-specification* `FOLLOWING`

  If the sort order specification is `ASC` (ascending), the last row of the window frame (the lower boundary) will be the last row whose sort key is less than or equal to the value from Formula B.

  If the sort order specification is `DESC` (descending), the last row of the window frame (the lower boundary) will be the last row whose sort key is greater than or equal to the value from Formula B.

  Formula B:

- When the sort order specification is ASC (ascending): *the sort key value of the current row + window-frame-value-specification*
- When the sort order specification is DESC (descending): *the sort key value of the current row - window-frame-value-specification*


Sort key result set
Set of results from the table expression
Current row
The last row of the window frame (the lower boundary) is the last row whose sort key is greater than or equal to the value from Formula B.
Legend:
: Window (partition)
: Window frame

Note: In this example, the sort order specification is DESC (descending).

If the value from Formula B is a value that cannot be represented in the data type of the result, the window frame is determined using the maximum or minimum value that can be represented by the data type of the result.

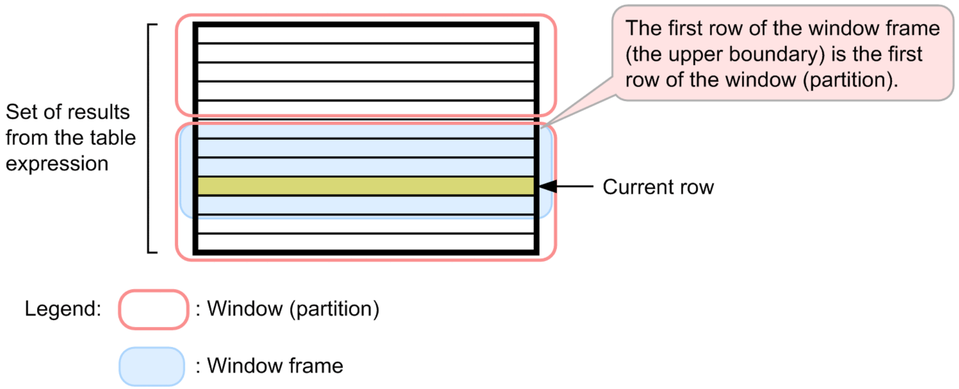
7.23.4 Rules for specifying the window frame (when ROWS is specified in the window frame clause)

When ROWS is specified in the window frame clause, the upper and lower boundaries of the window frame are determined as follows.

(1) The upper boundary of the window frame

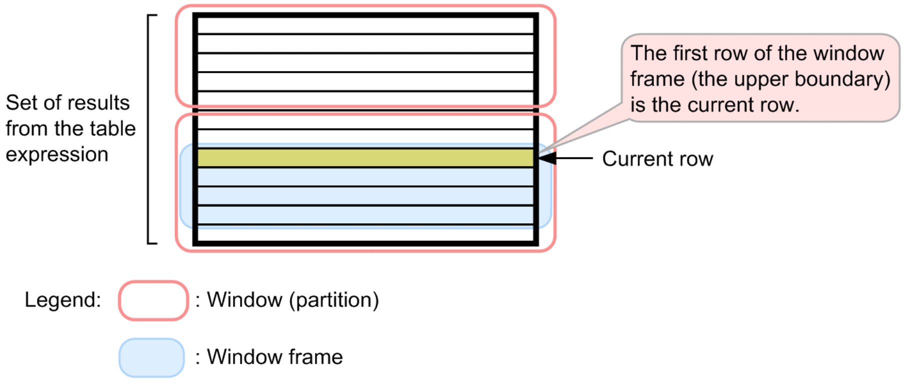
(a) When the window frame start boundary is UNBOUNDED PRECEDING

The first row of the window frame (the upper boundary) will be the first row of the window (partition).


The first row of the window frame (the upper boundary) is the first row of the window (partition).
Set of results from the table expression
Current row
Legend:
: Window (partition)
: Window frame

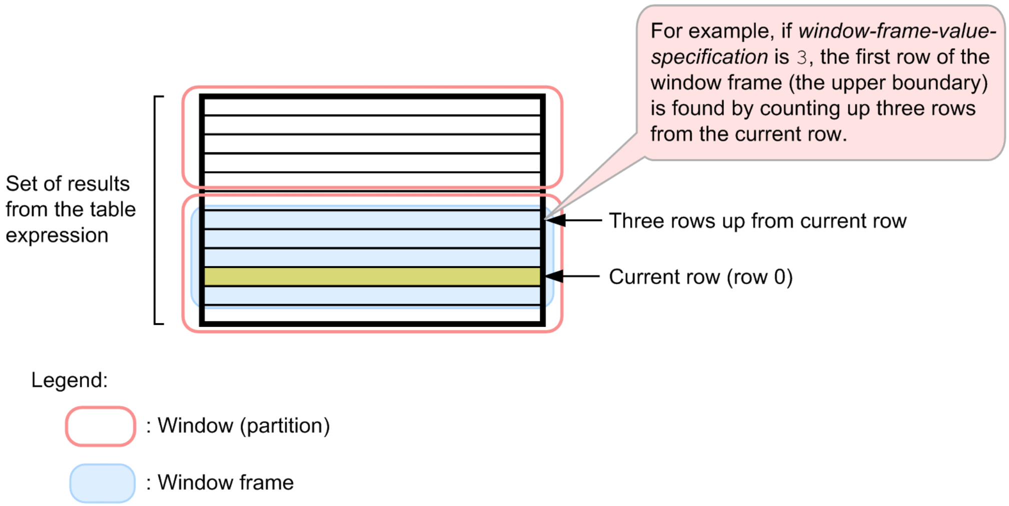
(b) When the window frame start boundary is CURRENT ROW

The first row of the window frame (the upper boundary) will be the current row.


Set of results from the table expression
The first row of the window frame (the upper boundary) is the current row.
Current row
Legend:
: Window (partition)
: Window frame

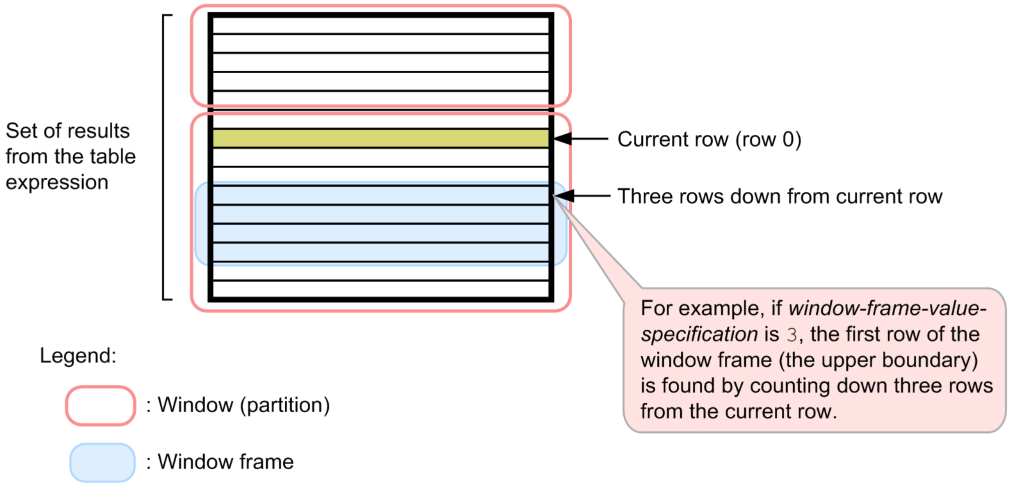
(c) When the window-frame-start-boundary is window-frame-value-specification PRECEDING

The first row of the window frame (the upper boundary) will be the row that is found by counting up from the current row the number of rows in *window-frame-value-specification*.


For example, if *window-frame-value-specification* is 3, the first row of the window frame (the upper boundary) is found by counting up three rows from the current row.
Set of results from the table expression
Three rows up from current row
Current row (row 0)
Legend:
: Window (partition)
: Window frame

(d) When the window frame start boundary is window-frame-value-specification FOLLOWING

The first row of the window frame (the upper boundary) will be the row that is found by counting down from the current row the number of rows in *window-frame-value-specification*.

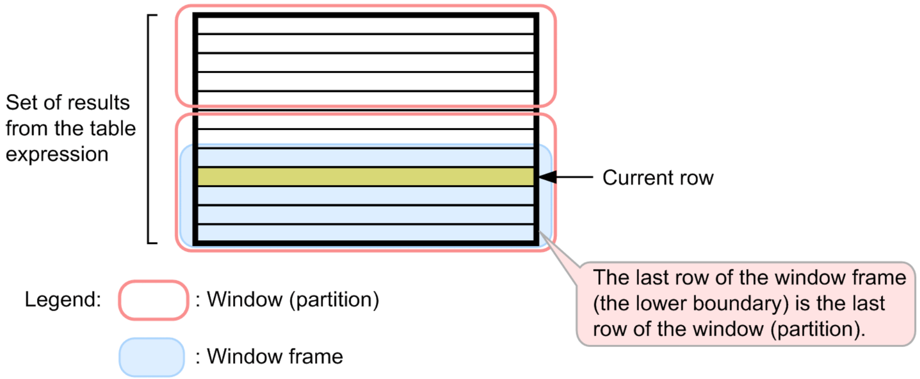

Set of results from the table expression
Current row (row 0)
Three rows down from current row
Legend:
: Window (partition)
: Window frame

For example, if *window-frame-value-specification* is `3`, the first row of the window frame (the upper boundary) is found by counting down three rows from the current row.

(2) The lower boundary of the window frame

(a) When the window frame end boundary is UNBOUNDED FOLLOWING

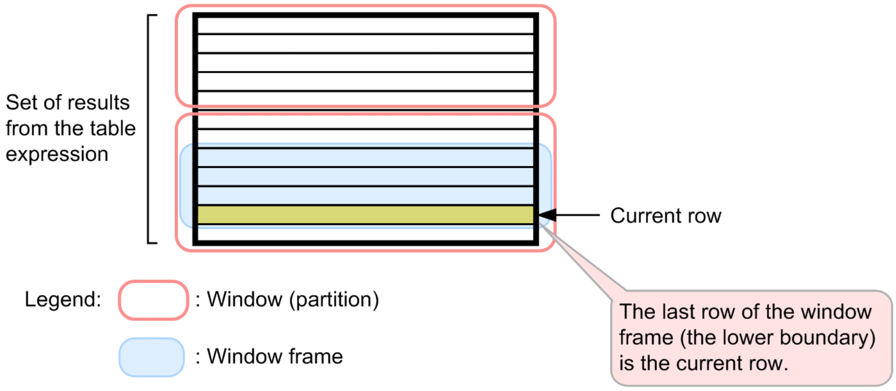
The last row of the window frame (the lower boundary) will be the last row of the window (partition).



The last row of the window frame (the lower boundary) is the last row of the window (partition).

(b) When the window frame end boundary is CURRENT ROW

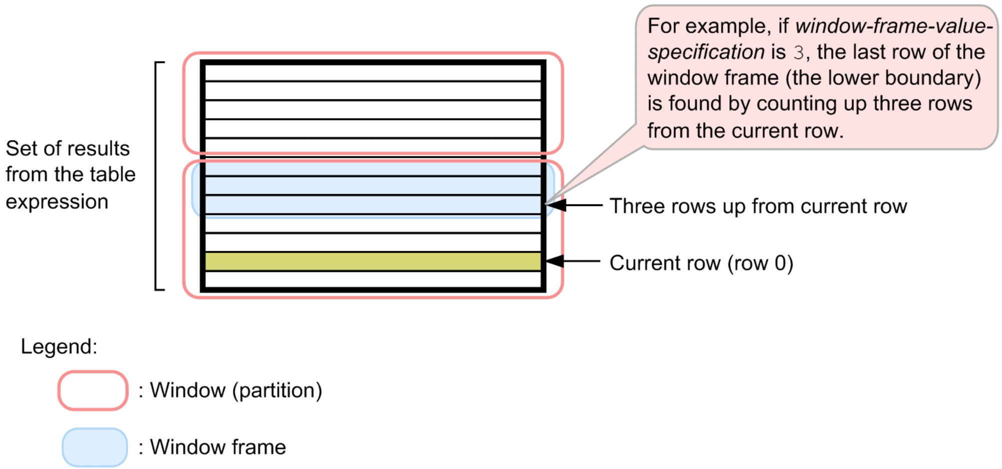
The last row of the window frame (the lower boundary) will be the current row.



The last row of the window frame (the lower boundary) is the current row.

(c) When the window frame end boundary is window-frame-value-specification PRECEDING

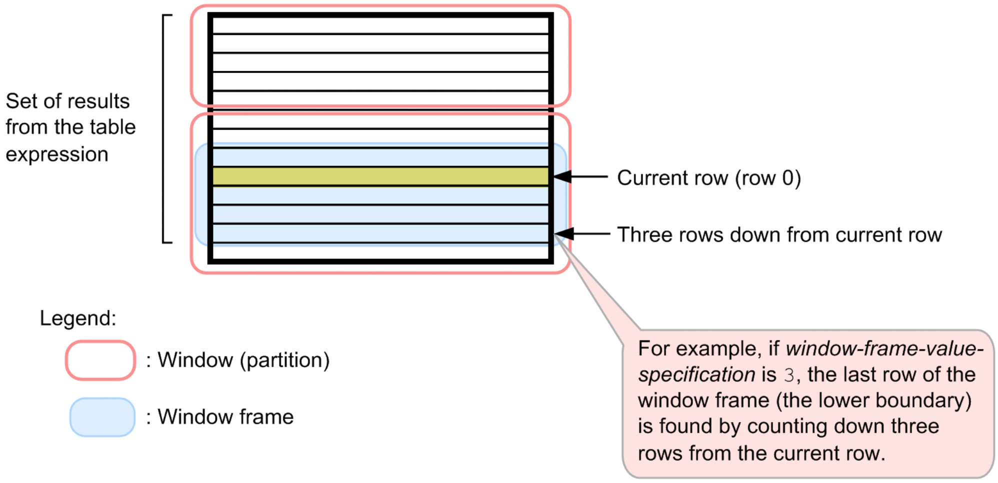
The last row of the window frame (the lower boundary) will be the row that is found by counting up from the current row the number of rows in *window-frame-value-specification*.


Set of results from the table expression
Three rows up from current row
Current row (row 0)
Legend:
: Window (partition)
: Window frame

For example, if *window-frame-value-specification* is `3`, the last row of the window frame (the lower boundary) is found by counting up three rows from the current row.

(d) When the window frame end boundary is window-frame-value-specification FOLLOWING

The last row of the window frame (the lower boundary) will be the row that is found by counting down from the current row the number of rows in *window-frame-value-specification*.


Set of results from the table expression
Current row (row 0)
Three rows down from current row
Legend:
: Window (partition)
: Window frame

For example, if *window-frame-value-specification* is `3`, the last row of the window frame (the lower boundary) is found by counting down three rows from the current row.

7.23.5 Rules and considerations pertaining to window functions

1. Window functions can be specified in selection expressions and `ORDER BY` clauses. However, when specifying a window function in an `ORDER BY` clause, the sort key of the `ORDER BY` clause must be identical to the sort key of the value expressions in the selection expression.
2. You can specify a maximum of eight window functions in a single query specification.
3. You cannot specify a window function, a subquery, or the RANDOMROW scalar function inside of a window function.

4. The execution result of the window function RANK, DENSE_RANK, or ROW_NUMBER will have the data type INTEGER. The execution result of CUME_DIST will have the data type DOUBLE PRECISION. For details about the data type of the execution result of a set function used as a window function, see the description of the set function in 7.22 Set functions.
5. The window function applies to the set of rows derived from the results of the table expression (the results of the FROM clause and WHERE clause). If there are no rows in the results of the table expression, the window function is not executed.
6. You cannot specify ROW and a window function at the same time.
7. If you specify a GROUP BY clause, HAVING clause, or set function, the grouping column must be a column specification that is not positioned as the aggregated argument of a window function or as a set function that is included in a window specification.

   Example:

   ```
   SELECT COUNT("C1") OVER(PARTITION BY SUM("C2")
                             ORDER BY "C1" RANGE UNBOUNDED PRECEDING)
       FROM "T1"
       GROUP BY "C1"
   ```

   In the SQL statement above, column C1, which is specified in the GROUP BY clause, is the grouping column. Because C1, which is specified in COUNT("C1") and ORDER BY "C1", is not the aggregated argument of a set function, it must be specified as the grouping column. On the other hand, C2, which is specified in SUM("C2"), is the aggregated argument of a set function, so it need not be specified as the grouping column.
8. When a window function is specified, a work table might be created. If the size of the work table DB area where the work table is to be created has not been estimated correctly, it might result in a performance degradation. For details about estimating the size of the work table DB area, see the *HADB Setup and Operation Guide*. For details about work tables, see *Considerations when executing an SQL statement that creates work tables* in the *HADB Application Development Guide*.

7.23.6 Examples of using window functions

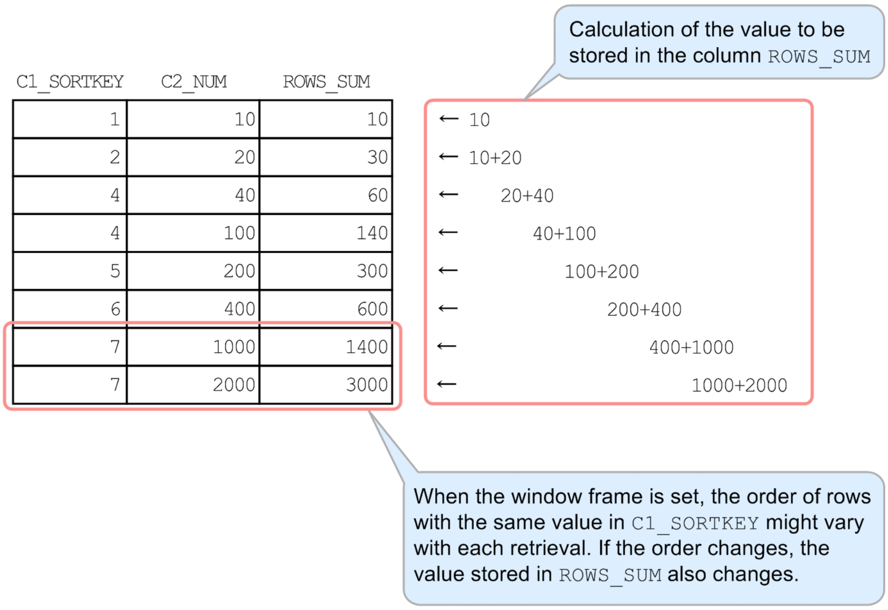
(1) Examples where ROWS or RANGE is specified in the window frame clause

The following examples show SQL statements that determine a moving total, They illustrate the difference between specifying ROWS and RANGE in the window frame clause.

(a) Example using ROWS

```
SELECT "C1_SORTKEY", "C2_NUM",
        SUM("C2_NUM") OVER(ORDER BY "C1_SORTKEY"
                      ROWS BETWEEN 1 PRECEDING AND CURRENT ROW) AS "ROWS_SUM"
    FROM "T1"
    ORDER BY "C1_SORTKEY", "C2_NUM"
```

Example of execution results



C1_SORTKEY	C2_NUM	ROWS_SUM	
1	10	10	← 10
2	20	30	← 10+20
4	40	60	← 20+40
4	100	140	← 40+100
5	200	300	← 100+200
6	400	600	← 200+400
7	1000	1400	← 400+1000
7	2000	3000	← 1000+2000

Calculation of the value to be stored in the column ROWS_SUM

When the window frame is set, the order of rows with the same value in C1_SORTKEY might vary with each retrieval. If the order changes, the value stored in ROWS_SUM also changes.

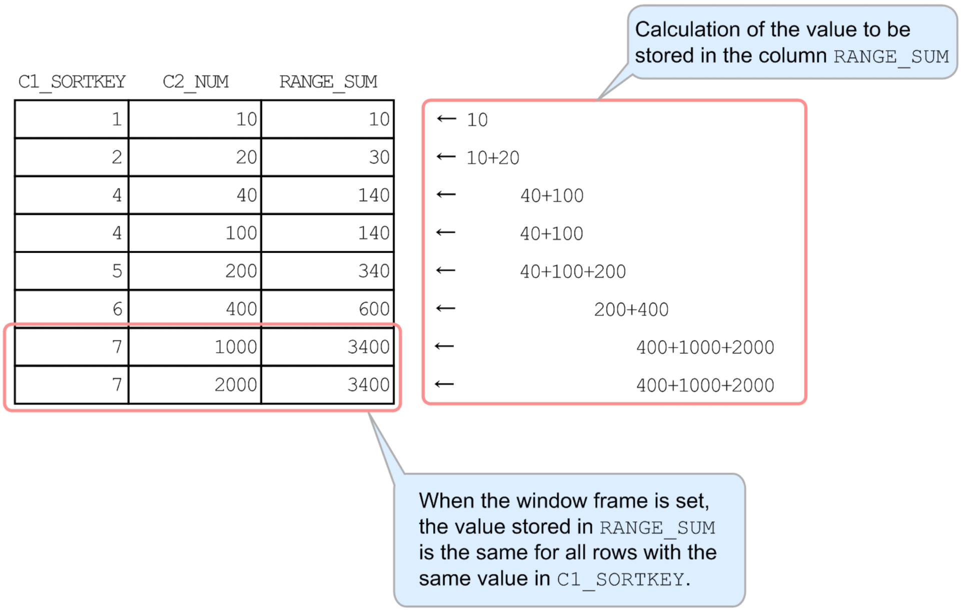
Explanation

- In the above example, the values in C1_SORTKEY are arranged in ascending order when the window frame is set. The window frame is set such that the range of the window function extends from one row above the current row to the current row.
- The ROWS_SUM column stores the sum of the values in column C2_NUM in the rows within the aggregation range.

(b) Example using RANGE

```
SELECT "C1_SORTKEY","C2_NUM",
        SUM("C2_NUM") OVER(ORDER BY "C1_SORTKEY"
                        RANGE BETWEEN 1 PRECEDING AND CURRENT ROW) AS "RANGE_SUM"
    FROM "T1"
    ORDER BY "C1_SORTKEY","C2_NUM"
```

Example of execution results



C1_SORTKEY	C2_NUM	RANGE_SUM	
1	10	10	← 10
2	20	30	← 10+20
4	40	140	← 40+100
4	100	140	← 40+100
5	200	340	← 40+100+200
6	400	600	← 200+400
7	1000	3400	← 400+1000+2000
7	2000	3400	← 400+1000+2000

Calculation of the value to be stored in the column RANGE_SUM

When the window frame is set, the value stored in RANGE_SUM is the same for all rows with the same value in C1_SORTKEY.

Explanation

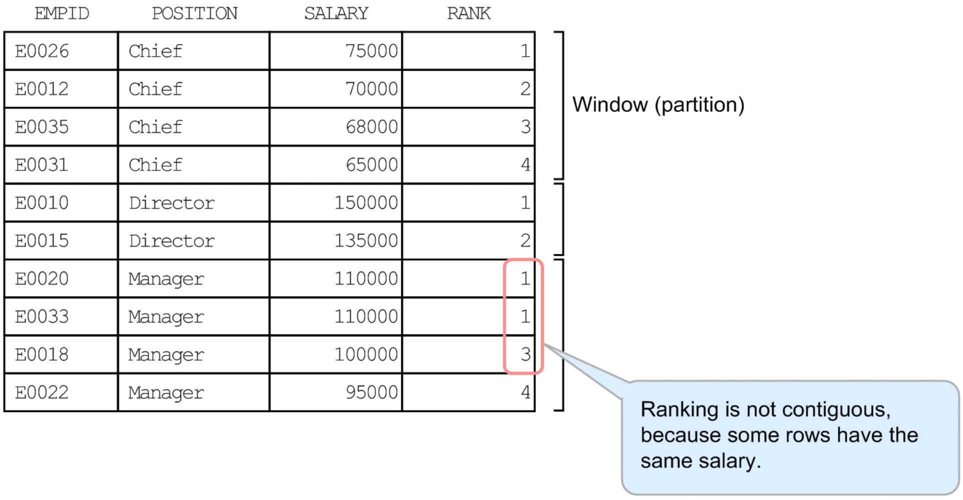
- In the above example, the values in C1_SORTKEY are arranged in ascending order when the window frame is set. The window frame is set such that the range of the window function extends from the row for which the value of C1_SORTKEY is 1 less than the value at the current row, up to the row where it has the same value as the current row.
- The RANGE_SUM column stores the sum of the values in column C2_NUM in the rows within the aggregation range.

(2) Example using RANK

Using the data in the salary table (SALARYLIST), this example ranks employees by salary (SALARY) within each job class (POSITION).

```
SELECT "EMPID","POSITION","SALARY",
        RANK() OVER(PARTITION BY "POSITION" ORDER BY "SALARY" DESC) AS "RANK"
    FROM "SALARYLIST"
    ORDER BY "POSITION","SALARY" DESC,"EMPID"
```

Example of execution results



EMPID	POSITION	SALARY	RANK
E0026	Chief	75000	1
E0012	Chief	70000	2
E0035	Chief	68000	3
E0031	Chief	65000	4
E0010	Director	150000	1
E0015	Director	135000	2
E0020	Manager	110000	1
E0033	Manager	110000	1
E0018	Manager	100000	3
E0022	Manager	95000	4

Window (partition)

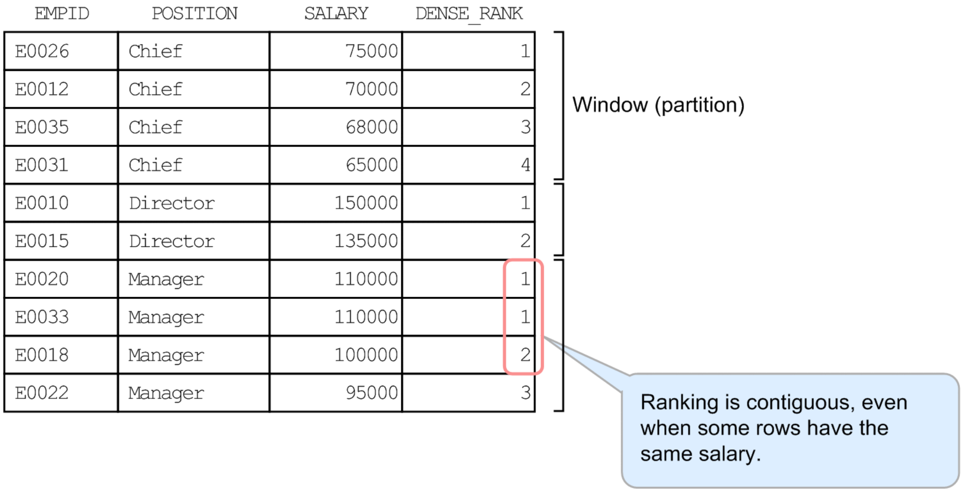
Ranking is not contiguous, because some rows have the same salary.

(3) Example using DENSE_RANK

Using the data in the salary table (SALARYLIST), this example ranks employees by salary (SALARY) within each job class (POSITION).

```
SELECT "EMPID","POSITION","SALARY",
        DENSE_RANK() OVER(PARTITION BY "POSITION" ORDER BY "SALARY" DESC) AS "DENSE_R
ANK"
    FROM "SALARYLIST"
    ORDER BY "POSITION","SALARY" DESC,"EMPID"
```

Example of execution results



EMPID	POSITION	SALARY	DENSE_RANK
E0026	Chief	75000	1
E0012	Chief	70000	2
E0035	Chief	68000	3
E0031	Chief	65000	4
E0010	Director	150000	1
E0015	Director	135000	2
E0020	Manager	110000	1
E0033	Manager	110000	1
E0018	Manager	100000	2
E0022	Manager	95000	3

Window (partition)

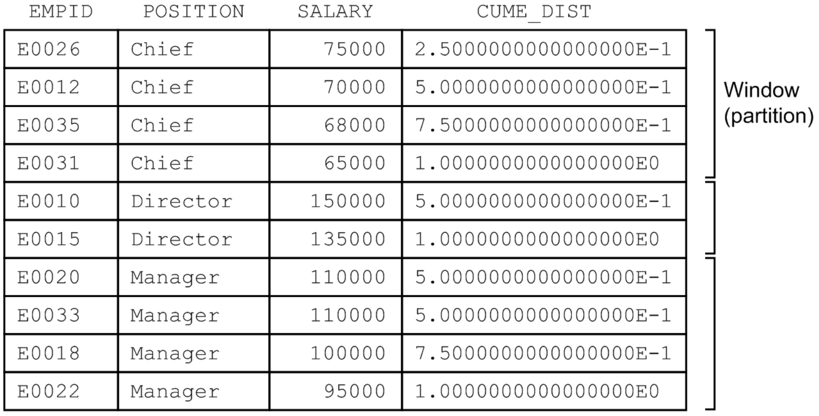
Ranking is contiguous, even when some rows have the same salary.

(4) Example using CUME_DIST

Using the data in the salary table (SALARYLIST), this example determines the relative positions of the employees' salaries (SALARY) within each job class (POSITION).

```
SELECT "EMPID","POSITION","SALARY",
        CUME_DIST() OVER(PARTITION BY "POSITION" ORDER BY "SALARY" DESC) AS "CUME_DIS
T"
    FROM "SALARYLIST"
    ORDER BY "POSITION","SALARY" DESC,"EMPID"
```

Example of execution results



EMPID	POSITION	SALARY	CUME_DIST
E0026	Chief	75000	2.5000000000000000E-1
E0012	Chief	70000	5.0000000000000000E-1
E0035	Chief	68000	7.5000000000000000E-1
E0031	Chief	65000	1.0000000000000000E0
E0010	Director	150000	5.0000000000000000E-1
E0015	Director	135000	1.0000000000000000E0
E0020	Manager	110000	5.0000000000000000E-1
E0033	Manager	110000	5.0000000000000000E-1
E0018	Manager	100000	7.5000000000000000E-1
E0022	Manager	95000	1.0000000000000000E0

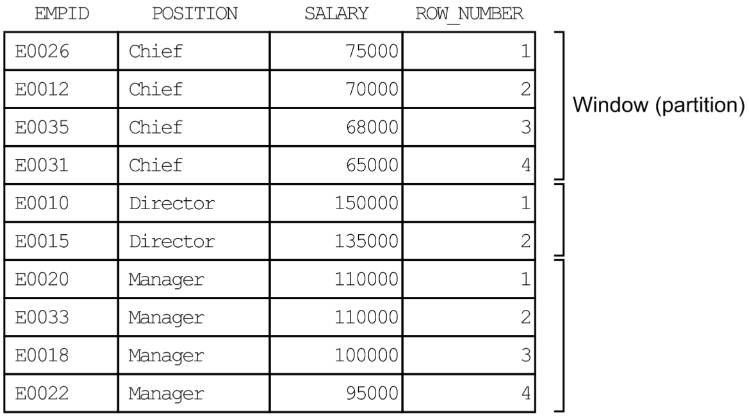
Window (partition)

(5) Example using ROW_NUMBER

Using the data in the salary table (SALARYLIST), this example determines the row numbers in descending order with respect to employee salary (SALARY) within each job class (POSITION).

```
SELECT "EMPID","POSITION","SALARY",
        ROW_NUMBER() OVER(PARTITION BY "POSITION" ORDER BY "SALARY" DESC) AS "ROW_NUM
BER"
    FROM "SALARYLIST"
    ORDER BY "POSITION","SALARY" DESC,"EMPID"
```

Example of execution results



EMPID	POSITION	SALARY	ROW_NUMBER
E0026	Chief	75000	1
E0012	Chief	70000	2
E0035	Chief	68000	3
E0031	Chief	65000	4
E0010	Director	150000	1
E0015	Director	135000	2
E0020	Manager	110000	1
E0033	Manager	110000	2
E0018	Manager	100000	3
E0022	Manager	95000	4

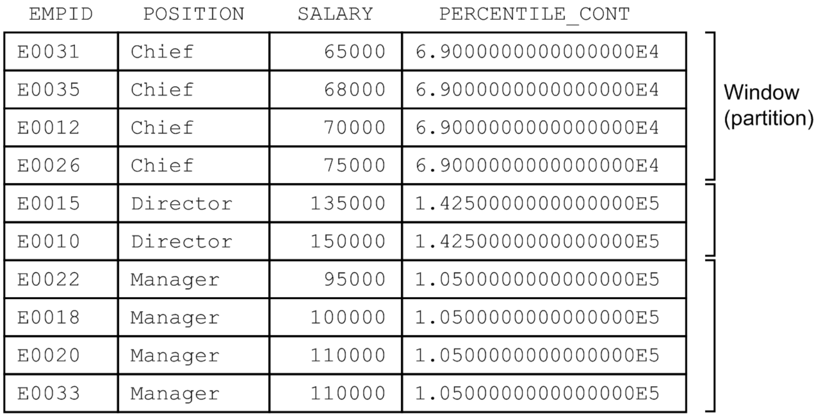
Window (partition)

(6) Example using PERCENTILE_CONT

Using the data in the salary table (SALARYLIST), this example determines the median value (50th percentile) of the employee salaries (SALARY) within each job class (POSITION).

```
SELECT "EMPID","POSITION","SALARY",
        PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY "SALARY")
            OVER(PARTITION BY "POSITION") AS "PERCENTILE_CONT"
    FROM "SALARYLIST"
    ORDER BY "POSITION","SALARY","EMPID"
```

Example of execution results



EMPID	POSITION	SALARY	PERCENTILE_CONT
E0031	Chief	65000	6.9000000000000000E4
E0035	Chief	68000	6.9000000000000000E4
E0012	Chief	70000	6.9000000000000000E4
E0026	Chief	75000	6.9000000000000000E4
E0015	Director	135000	1.4250000000000000E5
E0010	Director	150000	1.4250000000000000E5
E0022	Manager	95000	1.0500000000000000E5
E0018	Manager	100000	1.0500000000000000E5
E0020	Manager	110000	1.0500000000000000E5
E0033	Manager	110000	1.0500000000000000E5

7.24 Sort specification list

This section describes the sort specification list. Sort specification lists are specified in the following locations:

- ORDER BY clause

  See 4.4.1 Specification format and rules for the SELECT statement.
- WITHIN group specification

  See 7.22.12 PERCENTILE_CONT and 7.22.13 PERCENTILE_DISC.
- Window order clause

  See 7.23.1 Specification format for window functions.

7.24.1 Specification format for the sort specification list

The sort specification list is used to specify the sort order of data.

(1) Specification format

```
sort-specification-list ::= sort-specification[,sort-specification]...

  sort-specification ::= sort-key [order-specification] [null-value-sort-order-specif
ication]
    sort-key ::= {value-expression | sort-item-specification-number}
    order-specification ::= {ASC | DESC}
    null-value-sort-order-specification ::= NULLS {FIRST | LAST}
```

(2) Explanation of specification format

- *sort-specification*

```
sort-specification ::= sort-key [order-specification] [null-value-sort-order-speci
fication]
```

A sort specification specifies a sort key, and optionally an order specification and a specification of the null-value sort order.

The following rules apply:

- No more than 64 sort specifications are permitted in a sort specification list specified in an ORDER BY clause or window order clause.
- No more than one sort specification is permitted in a sort specification list specified in a WITHIN group specification (you cannot specify two or more).

- *sort-key*

```
sort-key ::= {value-expression | sort-item-specification-number}
```

The sort key specifies either a value expression or a sort item specification number.

When an integer literal is specified for *sort-key*, it is considered a sort item specification number. When a non-integer literal is specified, it is considered a value expression.

value-expression:

Specifies a sort key in the form of a value expression. For details about value expressions, see 7.20 Value expression.

Note that when multiple sort keys are specified, the sort keys that are specified first take the highest priority when sorting. For example, if `ORDER BY "C1","C2","C3"` is specified, the priority order of the sort keys will be `C1`, `C2`, and then `C3`.

sort-item-specification-number:

Specifies the number of the column that is to be the sort key. For example, if `2` is specified, the sort key will be the second column of the table derived by the query expression body.

Example:

```
SELECT "C1","C2" FROM "T1"
    ORDER BY 2 ASC
```

When the `SELECT` statement above is executed, the sort key will be column `C2`.

The following rules apply:

- The sort item specification number must be an integer literal.
- The sort item specification number must be a value in the range from 1 to the number of columns in the table derived by the query expression body.

  Example:

  ```
  SELECT "C1",SUM("C2"),AVG("C2") FROM "T1"
      GROUP BY "C1" ORDER BY 3 ASC
  ```

  When the above `SELECT` statement is executed, a sort item specification number between `1` and `3` can be specified in the `ORDER BY` clause.
- When there are two or more sort item specification numbers, the ones specified first take the highest priority when sorting. For example, if `ORDER BY 2,3,1` is specified, the priority order for sorting will be column 2, then column 3, and finally column 1.
- You cannot specify a sort item specification number corresponding to `[`*table-specification*`.]ROW`.

  Example of an SQL statement that generates an error:

  ```
  SELECT "C1",ROW FROM "T1" ORDER BY 2
  ```

Important

A sort item specification number is not permitted in a sort specification list in the following locations:

- A `WITHIN` group specification
- A window order clause

- *order-specification*

```
order-specification ::= {ASC | DESC}
```

Specifies whether to sort the results in ascending or descending order. Specify either of the following.

`ASC`: Specify to sort the results in ascending order.

`DESC`: Specify to sort the results in descending order.

If the order specification is omitted, `ASC` is assumed.

- *null-value-sort-order-specification*

```
null-value-sort-order-specification ::= NULLS {FIRST | LAST}
```

Specifies the ordering of the null value when sorting. Specify either of the following.

`NULLS FIRST`: The null value comes first.

`NULLS LAST`: The null value comes last.

If the specification of the null-value sort order is omitted, the null value is ordered as follows:

- If you specify `ASC` for order-specification or if you do not specify order-specification, the null value comes last. This is the same action that is taken when `NULLS LAST` is specified.
- If you specify `DESC` for order-specification, the null value comes first. This is the same action that is taken when `NULLS FIRST` is specified.

Important

You cannot specify the null-value sort order in a sort specification list in a `WITHIN` group specification.

7.24.2 Rules for specifying a sort specification list in an ORDER BY clause

(1) Common rules

1. The sort keys can include a mixture of value expressions and sort item specification numbers.

   Example:

```
SELECT "C1", AVG("C2") FROM "T1"
    GROUP BY "C1"
    ORDER BY "C1" ASC, 2 ASC
```

2. If you specify duplicate sort keys, the first order specification and null-value sort order specification takes precedence.

   Example:

```
SELECT "C1","C2" FROM "T1"
    ORDER BY "C1" ASC NULLS FIRST,"C1" DESC NULLS LAST
```

   In the above case, the underlined portion, which was specified first, takes precedence.

3. If the same column is specified two or more times in the selection list, it cannot be specified as a sort key. For example, the SQL statement below generates an error.

   Example of an SQL statement that generates an error:

```
SELECT "C1","C2","C1" FROM "T1" ORDER BY "C1"
```

4. If a derived column name specified in an `ORDER BY` clause was derived from just a single column specification, it is replaced by that column specification. For example, the following three SQL statements produce the same retrieval results:

```
SELECT "T1"."C1" DR1,"T1"."C2" DR2 FROM "T1" ORDER BY DR1
SELECT "T1"."C1" DR1,"T1"."C2" DR2 FROM "T1" ORDER BY "T1"."C1"
SELECT "T1"."C1" DR1,"T1"."C2" DR2 FROM "T1" ORDER BY 1
```

(2) Rules for specifying value expressions as sort keys

1. You cannot specify a dynamic parameter by itself as a sort key.
2. If you specify a value expression as a sort key (unless the value expression is a column specification only), the sort key cannot include a derived column name (unless it is a derived column consisting of a simple column specification).

■ Example of an SQL statement that generates an error

```
SELECT "C1",SUM("C2") AS "SUMC2" FROM "T1"
    GROUP BY "C1"
    ORDER BY "SUMC2"+1
```

In addition, if a set operation is specified, the sort key cannot include the name of a derived column that was derived by means of the set operation (not even a derived column consisting of a simple column specification).

■ Example of an SQL statement that generates an error

```
SELECT "C1","C2" FROM "T1" UNION ALL SELECT "C1"+"C2","C3" FROM "T1"
    ORDER BY "C1"+"C2"
```

3. If you specify a value expression as the sort key of a `SELECT` statement that includes a set operation, the sort key must be identical to the sort key of the selection expressions in the first query specification.

■ Examples of correct SQL statements

Example 1:

```
SELECT "C1"+"C2","C3" FROM "T1" UNION SELECT "C1","C2" FROM "T2"
    ORDER BY "C1"+"C2"
SELECT "C1"+"C2","C2" FROM "T1" UNION SELECT "C1","C2" FROM "T2"
    ORDER BY "C2"
```

The `SELECT` statements in the above examples can be executed because the sort key value expression is identical to the sort key of the selection expressions in the first query specification.

Example 2:

```
SELECT "C1"+"C2" AS "C1","C2" FROM "T1" UNION SELECT "C1","C2" FROM "T2"
    ORDER BY "C1"
```

When the sort key value expression is a column specification, as in the above example, you can execute the `SELECT` statement by specifying an `AS` clause.

■ Example of an SQL statement that generates an error

```
SELECT "C1","C2" FROM "T1" UNION SELECT "C1"+"C2","C2" FROM "T2"
    ORDER BY "C1"+"C2"
```

The above example generates an error because the sort key value expression is not identical to a selection expression in the first query specification.

4. In a `SELECT` statement with `DISTINCT` specified, the sort key value expression must be identical to one of the selection expressions.

■ Examples of correct SQL statements

Example 1:

```
SELECT DISTINCT "C1"+"C2","C2" FROM "T1" ORDER BY "C1"+"C2"
SELECT DISTINCT "C1"+"C2","C2" FROM "T1" ORDER BY "C2"
```

The `SELECT` statements in the above examples can be executed because the sort key value expression is identical to one of the selection expressions.

Example 2:

```
SELECT DISTINCT "C1"+"C2" AS "C1","C2" FROM "T1" ORDER BY "C1"
```

When the sort key value expression is a column specification, as in the above example, you can execute the `SELECT` statement by specifying an `AS` clause.

■ Example of an SQL statement that generates an error

```
SELECT DISTINCT "C1","C2" FROM "T1" ORDER BY "C1"+"C2"
```

The above example generates an error because the sort key value expression is not identical to a selection expression.

5. If you specify a window function in the sort key, the sort key value expression must be identical to a selection expression.

   ■ Example of a correct SQL statement

   ```
   SELECT SUM("C1") OVER()/100 FROM "T1" ORDER BY SUM("C1") OVER()/100
   ```

   The `SELECT` statement in the above example can be executed because the sort key value expression is also specified as a selection expression.

   ■ Example of an SQL statement that generates an error

   ```
   SELECT SUM("C1") OVER() FROM "T1" ORDER BY SUM("C1") OVER()/100
   ```

   The above example generates an error because the sort key value expression is not identical to a selection expression.

6. You cannot specify a subquery or dynamic parameter in the sort key value expression in the following circumstances:

   - When a set operation is specified
   - In a `SELECT` statement with `DISTINCT` specified

   ■ Example of an SQL statement that generates an error

   ```
   SELECT "C1"+?,"C2" FROM "T1" UNION SELECT "C1","C2" FROM "T2"
       ORDER BY "C1"+?
   ```

7. The name of a table reference specified in the outermost query specification cannot be referenced from a subquery in the `ORDER BY` clause.

   ■ Example of an SQL statement that generates an error

   ```
   SELECT * FROM "T1"
       ORDER BY "C1",(SELECT "C1" FROM "T2" WHERE "C2"="T1"."C2")+"C2"
   ```

8. To specify a set function in the sort key, one of the following conditions must be met:

   (1) A grouping column must be specified in the selection expression of the qualified query of the set function.

   (2) The column specifications included in the sort key value expression must be specified in a grouping column or aggregated argument.

   ■ Examples of correct SQL statements

   ```
   SELECT "C1" FROM "T1" GROUP BY "C1" ORDER BY AVG("C2")
   ```

   The above example meets condition (1).

   ```
   SELECT "C2" FROM "T1" GROUP BY "C1","C2" ORDER BY SUM("C1"),"C2"
   ```

   The above example meets condition (2).

   ■ Example of an SQL statement that generates an error

   ```
   SELECT "C1" FROM "T1" ORDER BY AVG("C2")
   ```

9. If the same value expression is specified for both the sort key and selection expression, sorting is performed by using the value expression specified for the selection expression. The value expression specified for the sort key is not used for sorting.

   Example:

   ```
   SELECT "C1","C2" FROM "T1" ORDER BY "C1"
   SELECT "C1"+"C2","C2" FROM "T1" ORDER BY "C1"+"C2"
   ```

In the above examples, sorting is done using the value expressions in the selection expressions.

10. When the sort key value expression differs from the selection expressions, sort processing is performed using the sort key value expression. However, HADB handles this internally by adding the sort key value expression as a selection expression before performing the sort processing.

    Example:

    ```
    SELECT "C1" FROM "T1" ORDER BY "C2"
    ```

    In the above example, the values in column `C1` are returned in a sorted order that is determined by sorting the values in column `C2`. HADB handles this internally by adding the sort key value expression (`C2`) as a selection expression before performing the sort processing. As a result, rules that restrict value expressions inside the query specification are also applied to the internally-added sort key value expression.

    ■ Example of an SQL statement that generates an error

    ```
    SELECT MEDIAN("C1"*0.5) FROM "T1"
        ORDER BY PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY "C1"*0.5)
    ```

    In the preceding example, although the underscored sort key is internally added to the selection expression, the statement results in an error due to a restriction on inverse distribution functions (a single query specification cannot include multiple inverse distribution functions for which a value expression that is not an independent column specification is specified as an aggregated argument).

7.24.3 Rules for specifying a sort specification list in a WITHIN group specification or window order clause

- Rules for specifying a sort specification list in a `WITHIN` group specification
  1. Set functions are not permitted in a value expression specified as a sort key.
- Rules for specifying a sort specification list in a window order clause
  1. Set functions are not permitted in a value expression specified as a sort key.
  2. If you specify duplicate sort keys, the first order specification and null-value sort order specification takes precedence.

7.24.4 Examples

(1) Examples of specifying a sort specification list in an ORDER BY clause

Example 1 (Specifying one column as the sort key)

This example sorts all the data in the customer table (`USERSLIST`) by customer ID (`USERID`).

```
SELECT "USERID","NAME","SEX"
    FROM "USERSLIST"
    ORDER BY "USERID" ASC
```

The underlined portion indicates the sort specification list.

Example 2 (Specifying multiple columns as sort keys)

This example sorts all the data in the sales history table (`SALESLIST`) by date of purchase (`PUR-DATE`). When the date of purchase is the same, this example sorts by customer ID (`USERID`).

```
SELECT "USERID","PUR-CODE","PUR-NUM","PUR-DATE"
    FROM "SALESLIST"
    ORDER BY "PUR-DATE" ASC,"USERID" ASC
```

The underlined portion indicates the sort specification list.

Example 3 (Specifying a value expression as the sort key)

This example extracts eight characters of data starting at the third character from the beginning of the sales history code (HIS-CODE), and then uses the extracted data as the key to sort all the data in the sales history table (SALESLIST).

```
SELECT * FROM "SALESLIST"
    ORDER BY SUBSTR("HIS-CODE",3,8) ASC
```

The underlined portion indicates the sort specification list.

Example 4 (Specifying a sort item specification number as the sort key)

Using the data in the sales history table (SALESLIST), this example determines the total quantity purchased (PUR-NUM) for each product code (PUR-CODE), and sorts the retrieval results by total quantity purchased.

```
SELECT "PUR-CODE",SUM("PUR-NUM")
    FROM "SALESLIST"
    GROUP BY "PUR-CODE"
    ORDER BY 2 ASC
```

The underlined portion indicates the sort specification list.

Example 5 (Specifying a null-value sort order)

This example sorts all the data in the sales history table (SALESLIST) by date of purchase (PUR-DATE). The rows for which PUR-DATE is the null value come at the top of the sort results.

```
SELECT "USERID","PUR-CODE","PUR-NUM","PUR-DATE"
    FROM "SALESLIST"
    ORDER BY "PUR-DATE" ASC NULLS FIRST
```

The underlined portion indicates the sort specification list.

(2) Example of specifying a sort specification list in a WITHIN group specification

Example

Using the data in the salary table (SALARYLIST), this example determines the median value (50th percentile) of the employee salaries (SALARY).

```
SELECT PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY "SALARY") "PERCENTILE_CONT"
    FROM "SALARYLIST"
```

The underlined portion indicates the sort specification list.

(3) Example of specifying a sort specification list in a window order clause

Example

Using the data in the salary table (SALARYLIST), this example ranks employees by salary (SALARY) within each job class (POSITION).

```
SELECT "EMPID","POSITION","SALARY",
        RANK() OVER(PARTITION BY "POSITION" ORDER BY "SALARY" DESC) "RANK"
```

```
    FROM "SALARYLIST"
    ORDER BY "POSITION","SALARY" DESC,"EMPID"
```

The underlined portion indicates the sort specification list in the window order clause.

7.25 Arithmetic operations

This section describes the types of arithmetic operations and the rules for using them.

7.25.1 Specification format and rules for arithmetic operations

Arithmetic operations can be specified in value expressions.

(1) Specification format

```
arithmetic-operation ::= {term|numeric-value-expression + term|numeric-value-expressi
on - term}

  term ::= {value-expression-primary|numeric-value-expression * value-expression-prim
ary|numeric-value-expression / value-expression-primary}
```

(2) Explanation of specification format

numeric-value-expression:

For details about *numeric-value-expression*, see 7.20.1 Specification format and rules for value expressions.

value-expression-primary:

For details about *value-expression-primary*, see 7.20.1 Specification format and rules for value expressions.

(3) Types of arithmetic operations

The following table lists the types of arithmetic operations.

Table 7-32: Types of arithmetic operations

No.	Arithmetic operation	Meaning	Function
1	+	Addition	Adds the second operand to the first operand.
2	-	Subtraction	Subtracts the second operand from the first operand.
3	*	Multiplication	Multiplies the first operand by the second operand.
4	/	Division	Divides the first operand by the second operand.

For example, if the calculation is 3 + 1, 3 is the first operand, and 1 is the second operand.

(4) Rules

1. Arithmetic operations can only be specified for numeric data (`INTEGER`, `SMALLINT`, `DECIMAL`, or `DOUBLE PRECISION` type data).
2. `INTEGER`, `SMALLINT`, `DECIMAL`, or `DOUBLE PRECISION` type data
3. Arithmetic operations can use a maximum of 500 arithmetic operators (`+`, `-`, `*`, and `/`). If an operand is a value expression with a column from a viewed table, derived table, or query name, the total number of value expressions after expanding the value expression it is based on cannot exceed 10,000.

4. You cannot specify a value expression that is composed solely of a dynamic parameter on both sides of an arithmetic operation (+, -, *, /).
5. If a dynamic parameter is specified in an arithmetic operation, the data type of the dynamic parameter is assumed to be the data type of the other side of the calculation.
6. The NOT NULL constraint does not apply to the execution result (the null value is allowed).
7. If any operand has the null value, the result will also be a null value.

7.25.2 Data types of the results of arithmetic operations

The data type of the result of an arithmetic operation is determined by the data types of the first and second operands. The following table shows the relationship between the data types of the operands and the data type of the result of the operation.

Table 7-33: Relationship between the data types of the operands and the data type of the result of the operation

No.	Data type of the first operand	Data type of the second operand	Data type of the result of the operation
1	SMALLINT	SMALLINT	INTEGER
2		INTEGER	
3		DECIMAL	DECIMAL
4		DOUBLE PRECISION	DOUBLE PRECISION
5	INTEGER	SMALLINT	INTEGER
6		INTEGER	
7		DECIMAL	DECIMAL
8		DOUBLE PRECISION	DOUBLE PRECISION
9	DECIMAL	SMALLINT	DECIMAL
10		INTEGER	
11		DECIMAL	
12		DOUBLE PRECISION	DOUBLE PRECISION
13	DOUBLE PRECISION	SMALLINT	DOUBLE PRECISION
14		INTEGER	
15		DECIMAL	
16		DOUBLE PRECISION	

If the data type of the result of the operation is DECIMAL, the precision and scaling will be as follows:

Let the first operand be DECIMAL(*p1*,*s1*), the second operand be DECIMAL(*p2*,*s2*), and the result of the operation be DECIMAL(*p*,*s*).

- For addition and subtraction
  $p = 1 + \mathrm{MAX}(p1 - s1, p2 - s2) + s$
  $s = \mathrm{MAX}(s1, s2)$

If the result of calculating p is 39 or greater, $p = 38$.

- For multiplication

  $p = p1 + p2$

  $s = s1 + s2$

  If the result of calculating p is 39 or greater, $p = 38$ and $s = \text{MAX}(s1, s2)$.

- For division

  $p = 38$

  $s = \text{MAX}\{0^{\#}, 38 - (p1 - s1 + s2)\}$

  #

  If the value specified for the `adb_sql_prep_dec_div_rs_prior` operand in the server definition or client definition is `FRACTIONAL_PART`, this value is replaced with *s1*.

If the data type is `INTEGER`, change the data type to `DECIMAL(20,0)` during calculation. If the data type is `SMALLINT`, change the data type to `DECIMAL(10,0)` during calculation.

7.25.3 Notes applying when the data type of the division result is DECIMAL

If the data type of the division result is `DECIMAL`, the scaling is determined by the value specified for the `adb_sql_prep_dec_div_rs_prior` operand in the server definition or client definition.

(1) When searching a base table

When base table `T1` that contains the data shown later is searched, the execution result is changed according to the value specified for the `adb_sql_prep_dec_div_rs_prior` operand.

Table `T1`

Column `C1` `DECIMAL(38,4)`	Column `C2` `DECIMAL(10,5)`
30.5256	0.05223

`SELECT` statement to be run

```
SELECT "C1"/"C2" AS "Division Results" FROM "T1"
```

- **If `INTEGRAL_PART` (default) is specified for the `adb_sql_prep_dec_div_rs_prior` operand**

  The data type of the division result of `"C1"/"C2"` will be `DECIMAL(38,0)` because the integral part takes precedence.

  Division result

  ```
  584.
  ```

- **If `FRACTIONAL_PART` is specified for the `adb_sql_prep_dec_div_rs_prior` operand**

  The data type of the division result of `"C1"/"C2"` will be `DECIMAL(38,4)` because the scaling of the first operand takes precedence.

  Division result

  ```
  584.4457
  ```

(2) When searching a viewed table

If the data type of the result of division (arithmetic operation) specified in the query expression body of the CREATE VIEW statement is DECIMAL, the precision and scaling are determined by the following value: the value of the adb_sql_prep_dec_div_rs_prior operand (specified when the viewed table is searched) in the server definition or client definition.

Example:

Note that this example assumes that the contents of table T1 are as follows.

Table T1

Column C1	Column C2
DECIMAL(38,1)	DECIMAL(2,1)
10.1	0.5

Definition of the viewed table

```
CREATE VIEW "VT1"("VC1") AS SELECT "C1"/"C2" FROM "T1"
```

- **If the following SELECT statement is run by specifying INTEGRAL_PART for the adb_sql_prep_dec_div_rs_prior operand**

  ```
  SELECT "VC1" FROM "VT1"
  ```

  Retrieval results

  ```
  20
  ```

  In this case, data type of column VC1 in viewed table VT1 will be DECIMAL(38,0).

- **If the following SELECT statement is run by specifying FRACTIONAL_PART for the adb_sql_prep_dec_div_rs_prior operand**

  ```
  SELECT "VC1" FROM "VT1"
  ```

  Retrieval results

  ```
  20.2
  ```

  In this case, data type of column VC1 in viewed table VT1 will be DECIMAL(38,1).

Note

The data type determined by the value of the adb_sql_prep_dec_div_rs_prior operand (specified when the viewed table is defined) is stored in the column information of the viewed table stored in table SQL_COLUMNS. Therefore, if different values are specified for the adb_sql_prep_dec_div_rs_prior operand during definition and search of the viewed table, the following two data types might not match:

- Data type of the derived columns of the internal derived table that is generated as a result of equivalent exchange of the viewed table
- Data type of the columns of the viewed table stored in table SQL_COLUMNS

Note that if both the first and second operands of division are literals, the precision and scaling of the division result are determined by the value of the `adb_sql_prep_dec_div_rs_prior` operand specified when the viewed table is defined.

7.26 Concatenation operations

Concatenation operations are used to concatenate two character strings or two binary data items. This section describes the types of concatenation operations and the rules for using them.

7.26.1 Specification format and rules for concatenation operations

Concatenation operations can be specified in a value expression.

(1) Specification format

```
concatenation-operation ::= {value-expression-primary
            |character-string-value-expression + value-expression-primary
            |character-string-value-expression || value-expression-primary
            |binary-value-expression + value-expression-primary
            |binary-value-expression || value-expression-primary}
```

(2) Explanation of specification format

value-expression-primary:

For details about *value-expression-primary*, see 7.20.1 Specification format and rules for value expressions.

character-string-value-expression:

For details about *character-string-value-expression*, see 7.20.1 Specification format and rules for value expressions.

binary-value-expression:

For details about *binary-value-expression*, see 7.20.1 Specification format and rules for value expressions.

(3) Types of concatenation operations

The types of concatenation operations are shown in the following table.

Table 7-34: Types of concatenation operations

No.	Concatenation operation	Function
1	+	Concatenate the first operand and the second operand.
2	||	

For example, if the operation is `'ABC'+'DEF'`, the first operand is `'ABC'`, and the second operand is `'DEF'`.

(4) Rules

1. The first and second operands must both be either character string data or binary data.
2. The following table shows the combinations of data types that can be specified in the first and second operands.

Table 7-35: Combinations of data types that can be specified in the first and second operands in a concatenation operation

Data type of the first operand	Data type of the second operand			
	CHAR	VARCHAR	BINARY	VARBINARY
CHAR	Y	Y	N	N
VARCHAR	Y	Y	N	N
BINARY	N	N	Y	Y
VARBINARY	N	N	Y	Y

Legend:

Y: Can be specified.

N: Cannot be specified.

3. Concatenation operations with up to 500 operators (+, ||) can be performed. If an operand is a value expression with a column from a viewed table, derived table, or query name, the total number of value expressions after expanding the value expression it is based on cannot exceed 10,000.
4. You cannot specify a dynamic parameter by itself for the first operand or second operand.
5. The NOT NULL constraint does not apply to the value of the result of the concatenation operation (the null value is allowed).
6. If either the first operand or second operand has the null value, the result of the concatenation operation will be a null value.
7. You cannot concatenate character string data or binary data if the result of the concatenation operation would exceed the maximum length of 32,000 bytes.
8. Spaces at the end of the character string data are also subject to concatenation.

   Example

   If column C1 is type CHAR(5) with a value of 'ABCΔΔ', and column C2 is type VARCHAR(10) with a value of 'XYZ', the following concatenations are performed.

   "C1"+"C2" → 'ABCΔΔXYZ'

   "C2"+"C1" → 'XYZABCΔΔ'

   Legend:

   Δ: Single-byte space

(5) Example

Example 1: Concatenate character string data

This example finds the rows in table T1 for which the result of concatenating the character string data in columns C2 and C3 is 'efg03v03'.

```
SELECT * FROM "T1"
    WHERE "C2"||"C3"='efg03v03'
```

Table T1

Column C1	Column C2	Column C3
CHAR	VARCHAR	VARCHAR
A10101	abc010587	rs3354
A15014	efg03	v03
A31399	hijk99842688	wxyz22725

Retrieval results

A15014	efg03	v03

Example 2: Concatenate binary data

This example finds the rows in table T1 for which the result of concatenating the binary data in columns C2 and C3 is X'ABC1230000DEF456'.

```
SELECT * FROM "T1"
    WHERE "C2"||"C3"=X'ABC1230000DEF456'
```

Table T1

Column C1	Column C2	Column C3
CHAR(6)	BINARY(5)	VARBINARY(10)
A10101	X'0000000000'	X'0000'
A15014	X'ABC1230000'	X'DEF456'
A31399	X'1111111111'	X'DEF4'

Retrieval results

A15014	X'ABC1230000'	X'DEF456'

7.26.2 Data types of the results of concatenation operations

The data type of the result of a concatenation operation is determined by the data types of the first and second operands.

(1) When the operands are character string data

The following table shows the relationship between the data types of the operands and the data type of the result of the operation when the operands are character string data.

Table 7-36: Relationship between the data types of the operands and the data type of the result of the operation (when the operands are character string data)

No.	Data type and data length of the first operand	Data type and data length of the second operand	Data type and data length of the result of the operation
1	CHAR(*m*)	CHAR(*n*)	CHAR(*m*+*n*)
2		VARCHAR(*n*) Actual data length: *L2*	VARCHAR(*m*+*n*) Actual data length: *m*+*L2*
3	VARCHAR(*m*) Actual data length: *L1*	CHAR(*n*)	VARCHAR(*m*+*n*) Actual data length: *L1*+*n*
4		VARCHAR(*n*) Actual data length: *L2*	VARCHAR(*m*+*n*) Actual data length: *L1*+*L2*

Legend:

m: Maximum length of the data in the first operand

n: Maximum length of the data in the second operand

L1: Actual data length of the data in the first operand

L2: Actual data length of the data in the second operand

(2) When the operands are binary data

The following table shows the relationship between the data types of the operands and the data type of the result of the operation when the operands are binary data.

Table 7-37: Relationship between the data types of the operands and the data type of the result of the operation (when the operands are binary data)

No.	Data type and data length of the first operand	Data type and data length of the second operand	Data type and data length of the result of the operation
1	`BINARY(`*m*`)`	`BINARY(`*n*`)`	`BINARY(`*m*+*n*`)`
2		`VARBINARY(`*n*`)` Actual data length: *L2*	`VARBINARY(`*m*+*n*`)` Actual data length: *m*+*L2*
3	`VARBINARY(`*m*`)` Actual data length: *L1*	`BINARY(`*n*`)`	`VARBINARY(`*m*+*n*`)` Actual data length: *L1*+*n*
4		`VARBINARY(`*n*`)` Actual data length: *L2*	`VARBINARY(`*m*+*n*`)` Actual data length: *L1*+*L2*

Legend:

m: Maximum length of the data in the first operand

n: Maximum length of the data in the second operand

L1: Actual data length of the data in the first operand

L2: Actual data length of the data in the second operand

7.27 Datetime operations

This section describes the types of datetime operations and the rules for using them.

7.27.1 Specification format and rules for datetime operations

You can specify datetime operations in value expressions in order to retrieve data based on datetime calculations.

(1) Specification format

```
datetime-operation ::= {value-expression-primary
     |datetime-value-expression + labeled-duration [{*|/}value-expression-primary]
     |datetime-value-expression - labeled-duration [{*|/}value-expression-primary]}
```

(2) Explanation of specification format

value-expression-primary:

For details about *value-expression-primary*, see 7.20.1 Specification format and rules for value expressions.

datetime-value-expression:

For details about *datetime-value-expression*, see 7.20.1 Specification format and rules for value expressions.

labeled-duration:

For details about labeled durations, see 7.28 Labeled duration.

(3) Data types on which datetime operations can be performed

Datetime operations can be performed on `DATE`, `TIME`, and `TIMESTAMP` type data.

Datetime operations can also be performed on character string literals (`CHAR` or `VARCHAR`) that follow the format of the predefined input representations of dates, times, or time stamps. If a character string literal is specified, the datetime operation is performed after converting the character string literal to datetime data.

For details about the predefined input representations of dates, times, and time stamps, see 6.3.3 Predefined character-string representations.

(4) Rules

(a) Common rules

1. When a `DATE` type operation is performed, the data type of the result of the operation will also be `DATE` type.
2. When a `TIME` type operation is performed, the data type of the result of the operation will also be `TIME` type.
3. When a `TIMESTAMP` type operation is performed, the data type of the result of the operation will also be `TIMESTAMP` type.
4. Datetime operations can use a maximum of 500 operators (+ or -). If an operand is a value expression with a column from a viewed table, derived table, or query name, the total number of value expressions after expanding the value expression it is based on cannot exceed 10,000.

5. On the left side of the operator (+ or −), you cannot specify a value expression that consists of only a dynamic parameter.
6. The results of the operation are not subject to the `NOT NULL` constraint (null values are allowed).
7. If an operand has the null value, the result of the operation will also have the null value.
8. Datetime operations are subject to the rules in 7.20 Value expression in addition to the rules listed above.

(b) Rules for performing datetime operations on DATE type data

1. The result of the operation must fall in the range from January 01, 0001 to December 31, 9999.
2. Dates are calculated with the year or month carried over as necessary. The following is an example.

   Example 1

   ```
   DATE'2012-12-31'+2 DAY --> DATE'2013-01-02'
   ```

   Example 2

   ```
   DATE'2013-01-01'-1 DAY --> DATE'2012-12-31'
   ```

3. If an operation results in a nonexistent date in a particular year or month (such as 31 in a 30-day month, February 30, or February 29 in a non-leap year), it will be changed to the last day of that month. The following is an example.

   Example:

   ```
   DATE'2013-03-31'+1 MONTH --> DATE'2013-04-30'
   ```

   When an operation produces a nonexistent date, it is automatically corrected to the last day of that month. As a consequence, if you add some number of months to a certain date, then subtract the same number of months from the resulting date, it does not necessarily return to the original date. The following is an example:

   Example:

   ```
   DATE'2013-03-31'+1 MONTH --> DATE'2013-04-30'
   DATE'2013-04-30'-1 MONTH --> DATE'2013-03-30'
   ```

(c) Rules for performing datetime operations on TIME type data

1. The result of the operation must fall in the range from 00:00:00.000000000000 to 23:59:59.999999999999.
2. When operations are performed on data with different fractional seconds precisions, the higher precision is used, and the lower-precision data is padded with zeros. For example, if the data in a `TIME` type operand has a fractional seconds precision of 0 and the labeled duration is `MILLISECONDS`, the calculation is performed with the fractional seconds precision of the `TIME` type data extended to 3.

(d) Rules for performing datetime operations on TIMESTAMP type data

1. The result of the operation must fall in the range from January 01, 0001 00:00:00.000000000000 to December 31, 9999 23:59:59.999999999999.
2. The methods for calculating the year, month, and date follow the rules in (b) Rules for performing datetime operations on DATE type data.
3. When operations are performed on data with different fractional seconds precisions, the higher precision is used, and the lower-precision data is padded with zeros. For example, if the data in a `TIMESTAMP` type operand has a fractional seconds precision of 0 and the labeled duration is `MILLISECONDS`, the calculation is performed with the fractional seconds precision of the `TIMESTAMP` type data extended to 3.
4. Time stamps are calculated with the day carried over as necessary. This is illustrated in the examples below.

Example 1:

```
TIMESTAMP'2014-02-01 23:59:59'+1 SECOND --> TIMESTAMP'2014-02-02 00:00:00'
```

Example 2:

```
TIMESTAMP'2014-02-02 00:00:00'-1 SECOND --> TIMESTAMP'2014-02-01 23:59:59'
```

Example 3:

```
TIMESTAMP'2013-12-31 23:05:06'+2 HOUR --> TIMESTAMP'2014-01-01 01:05:06'
```

(e) Rules for multiplication and division of labeled durations

1. When multiplying or dividing a labeled duration, the following labeled durations are equivalent:
   - *value-expression-1 labeled-duration-qualifier* * *value-expression-2* → (*value-expression-1**value-expression-2*) *labeled-duration-qualifier*
   - *value-expression-1 labeled-duration-qualifier* / *value-expression-2* → (*value-expression-1*/*value-expression-2*) *labeled-duration-qualifier*

   Examples:

   ```
   C1 DAYS * C2 → (C1*C2) DAYS
   (C1+C2) MINUTES / (C3+C4) → ((C1+C2)/(C3+C4)) MINUTES
   ```
2. The *value-expression-primary* that multiplies or divides the labeled duration must be an integer (`SMALLINT` or `INTEGER` type).
3. If a dynamic parameter is specified by itself for the *value-expression-primary* that multiplies or divides the labeled duration, the assumed data type of the dynamic parameter is `INTEGER`.
4. If the result of the *value-expression-primary* that multiplies or divides the labeled duration is the null value, the result of the labeled duration will be the null value.

(5) Example

Example

Searching the employee table (`EMPLIST`), retrieve the ID (`USERID`) and name (`NAME`) of all employees whose date of hire (`ENT-DAY`) was at least two years ago.

```
SELECT "USERID","NAME" FROM "EMPLIST"
    WHERE "ENT-DAY" <= CURRENT_DATE -2 YEARS
```

The underlined portion indicates the datetime operation, in which the labeled duration is `2 YEARS`.

7.28 Labeled duration

This section describes labeled durations.

7.28.1 Specification format and rules for labeled durations

A labeled duration is used in datetime operations to represent a specific time duration. The format is a numeric value followed by a duration keyword (`YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`, `MICROSECOND`, `NANOSECOND`, `PICOSECOND`). A labeled duration can be specified only in a window frame clause, or as the second operand of an addition or subtraction on datetime data.

(1) Specification format

```
labeled-duration ::= value-expression-primary labeled-duration-qualifier

  labeled-duration-qualifier ::={YEAR[S]|MONTH[S]|DAY[S]
                                 |HOUR[S]|MINUTE[S]|SECOND[S]
                                 |MILLISECOND[S]|MICROSECOND[S]
                                 |NANOSECOND[S]|PICOSECOND[S]}
```

(2) Explanation of specification format

value-expression-primary:

Specify `SMALLINT` or `INTEGER` type data for *value-expression-primary*. For details about the value-expression-primary, see (1) Specification format in 7.20.1 Specification format and rules for value expressions.

labeled-duration-qualifier:

```
labeled-duration-qualifier ::= {YEAR[S] | MONTH[S] | DAY[S]
                              | HOUR[S] | MINUTE[S] | SECOND[S]
                              | MILLISECOND[S] | MICROSECOND[S]
                              | NANOSECOND[S] | PICOSECOND[S]}
```

Specifies one of the following.

`YEAR[S]`:

Expresses a duration in years.

The range of numeric data that can be specified in *value-expression-primary* is -9,998 to 9,998.#

`MONTH[S]`:

Expresses a duration in months.

The range of numeric data that can be specified in *value-expression-primary* is -119,987 to 119,987.#

`DAY[S]`:

Expresses a duration in days.

The range of numeric data that can be specified in *value-expression-primary* is -3,652,058 to 3,652,058.#

`HOUR[S]`:

Expresses a duration in hours.

The range of numeric data that can be specified in *value-expression-primary* is -87,649,415 to 87,649,415.#

`MINUTE[S]`:

Expresses a duration in minutes.

The range of numeric data that can be specified in *value-expression-primary* is -5,258,964,959 to 5,258,964,959.#

`SECOND[S]`:

Expresses a duration in seconds.

The range of numeric data that can be specified in *value-expression-primary* is -315,537,897,599 to 315,537,897,599.#

`MILLISECOND[S]`:

Expresses a duration in milliseconds.

The range of numeric data that can be specified in *value-expression-primary* is -315,537,897,599,999 to 315,537,897,599,999.#

`MICROSECOND[S]`:

Expresses a duration in microseconds.

The range of numeric data that can be specified in *value-expression-primary* is -315,537,897,599,999,999 to 315,537,897,599,999,999.#

`NANOSECOND[S]`:

Expresses a duration in nanoseconds.

The range of numeric data that can be specified in *value-expression-primary* is -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807.#

`PICOSECOND[S]`:

Expresses a duration in picoseconds.

The range of numeric data that can be specified in *value-expression-primary* is -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807.#

#

When a labeled duration in a datetime operation is multiplied, the range given for *value-expression-primary* applies to the value of the product. For example, if you specify `"C1" DAYS *"C2"`, the range that can be specified for `(C1*C2)` is -3,652,058 to 3,652,058.

For the rules concerning multiplication and division in datetime operations that include labeled durations, see (e) Rules for multiplication and division of labeled durations in (4) Rules in 7.27.1 Specification format and rules for datetime operations.

(3) Rules

1. The labeled duration qualifiers that can be specified vary depending on the data type on which the datetime operation is performed, as shown in the following table.

Table 7-38: Labeled duration qualifiers that can be specified

Labeled duration qualifier	**Data type of datetime operation**		
	DATE	**TIME**	**TIMESTAMP**
YEAR	Y	N	Y
MONTH	Y	N	Y
DAY	Y	N	Y
HOUR	N	Y	Y
MINUTE	N	Y	Y
SECOND	N	Y	Y

Labeled duration qualifier	Data type of datetime operation		
	DATE	**TIME**	**TIMESTAMP**
MILLISECOND	N	Y	Y
MICROSECOND	N	Y	Y
NANOSECOND	N	Y	Y
PICOSECOND	N	Y	Y

Legend:

Y: Can be specified.

N: Cannot be specified.

2. If you specify a dynamic parameter for *value-expression-primary*, the assumed data type of the dynamic parameter will be `INTEGER`.
3. The `NOT NULL` constraint does not apply to the result value of the labeled duration (the null value is allowed).
4. If the result of *value-expression-primary* is a null value, the result of the labeled duration will be a null value.
5. Specification of the final `S` in `YEARS`, `MONTHS`, `DAYS`, `HOURS`, `MINUTES`, `SECONDS`, `MILLISECONDS`, `MICROSECONDS`, `NANOSECONDS`, and `PICOSECONDS` is optional, as illustrated in the following example:

   Example: To specify one year:

   `1 YEAR` or `1 YEARS`
6. When you specify a labeled duration for *window-frame-value-specification*, only a value specification can be specified for *value-expression-primary* in the labeled duration.
7. The following table shows the fractional seconds precision that is assumed when you specify a labeled duration qualifier.

Table 7-39: Assumed fractional seconds precision

Specified labeled duration qualifier	Assumed fractional seconds precision
MILLISECOND	3
MICROSECOND	6
NANOSECOND	9
PICOSECOND	12

7.29 CASE expression

This section describes CASE expressions.

7.29.1 Specification format and rules for CASE expressions

A CASE expression returns the result of a specified value expression when a specified search condition is TRUE.

(1) Specification format

```
CASE-expression ::=
    CASE
     {WHEN search-condition THEN {value-expression|NULL}}...
     [ELSE {value-expression|NULL}]
    END
```

(2) Explanation of specification format

WHEN *search-condition*:

Specifies search conditions. For details about search conditions, see 7.18 Search conditions. If the specified search condition is TRUE, the value specified after the THEN is returned as the result.

Note that a maximum of 255 WHEN statements can be specified in a single CASE expression.

THEN {*value-expression*|NULL}:

Specifies, in the form of a value expression, the value to return as the result if the specified search condition is TRUE. Specify NULL if you want to return the null value.

ELSE {*value-expression*|NULL}:

Specifies, in the form of a value expression, the value to return as the result if none of the search conditions specified in the WHEN statement is TRUE. Specify NULL if you want to return the null value.

If the ELSE specification is omitted, it is the same as specifying NULL in the ELSE statement.

(3) Rules

1. If multiple WHEN statements are specified in a CASE expression and more than one search conditions is TRUE, the result of the CASE expression will be the result of the first WHEN statement whose search condition is TRUE.
2. The results of the value expressions specified in THEN and ELSE must be data types that can be compared. For details about data types that can be compared, see (1) Data types that can be compared in 6.2.2 Data types that can be converted, assigned, and compared.

   However, note the following exceptions:

   - Date data cannot be compared to character string data (even to the predefined input representation of a date).
   - Time data cannot be compared to character string data (even to the predefined input representation of a time).
   - Time stamp data cannot be compared to character string data (even to the predefined input representation of a time stamp).

   For information about predefined input representations, see 6.3.3 Predefined character-string representations.

3. The data type and data length of the result of the CASE expression are determined by the data type and data length of the result of the value expression specified in the THEN or ELSE that corresponds to the search condition that was satisfied. For details, see 7.20.2 Data types of the results of value expressions.
4. A value expression must be specified for at least one THEN or the ELSE. You cannot specify NULL for every value expression, as illustrated below.

   Example:

```
CASE
    WHEN "C1"=100 THEN NULL
    ELSE NULL
END
```

5. A dynamic parameter cannot be specified by itself in the value expression in CASE, THEN, or ELSE.
6. The data type of the result of a CASE expression is without the NOT NULL constraint (the null value is allowed).

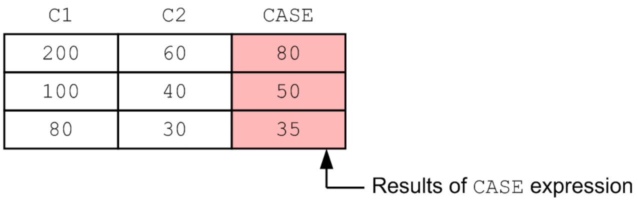
(4) Examples

Example 1

This example shows how to perform the following retrieval from table T1:

- If column C1 is 200: The retrieval result is value of column C2 + 20.
- If column C1 is 100: The retrieval result is value of column C2 + 10.
- If column C1 is a value other than 100 or 200: The retrieval result is value of column C2 + 5

```
SELECT "C1","C2",CASE WHEN "C1"=200 THEN "C2"+20
                      WHEN "C1"=100 THEN "C2"+10
                      ELSE "C2"+5
                 END AS "CASE"
    FROM "T1"
```



C1	C2	CASE
200	60	80
100	40	50
80	30	35

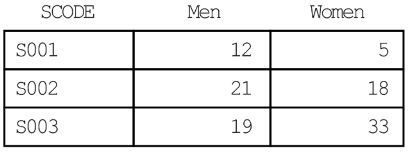
Results of CASE expression

Example 2

This example shows how to search the employee table (EMPLIST), as follows:

- Determine the number of men and number of women in each section (SCODE).

```
SELECT "SCODE",SUM(CASE WHEN "SEX"='M' THEN 1 ELSE 0 END) AS "Men",
        SUM(CASE WHEN "SEX"='F' THEN 1 ELSE 0 END) AS "Women"
    FROM "EMPLIST"
    GROUP BY "SCODE"
```



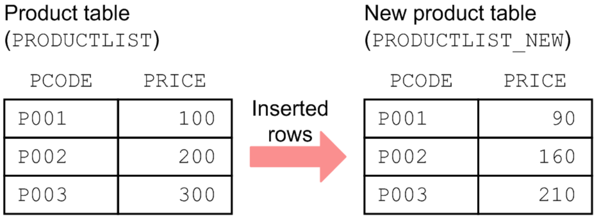
SCODE	Men	Women
S001	12	5
S002	21	18
S003	19	33

Example 3

This example shows how to insert a row from the products table (PRODUCTLIST) into the new products table (PRODUCTLIST_NEW). When inserting the row, change the product prices (PRICE) as follows:

- If the product code (PCODE) is P001: reduce the price by 10%
- If the product code is P002: reduce the price by 20%
- Otherwise: reduce the price by 30%

```
INSERT INTO "PRODUCTLIST_NEW"("PCODE","PRICE")
    SELECT "PCODE",CASE WHEN "PCODE"='P001' THEN "PRICE"*0.9
                        WHEN "PCODE"='P002' THEN "PRICE"*0.8
                        ELSE "PRICE"*0.7
                   END
    FROM "PRODUCTLIST"
```



Product table
(PRODUCTLIST)

PCODE	PRICE
P001	100
P002	200
P003	300

New product table
(PRODUCTLIST_NEW)

PCODE	PRICE
P001	90
P002	160
P003	210

7.30 Internal derived tables

This section gives examples of the use of internal derived tables and explains the rules for derived table expansion.

7.30.1 Examples of using internal derived tables

When you execute a query on a viewed table, HADB derives an internal table based on the CREATE VIEW statement specification that it codes in the FROM clause that specifies the viewed table. This derived table is called an *internal derived table*.

Similarly, an internal derived table is also used when you execute a query using a query name specified in a WITH list element.

The following examples illustrate how internal derived tables are used.

(1) Example 1: Executing a query on a viewed table

When you execute a query on a viewed table, an internal derived table is used in the FROM clause that specifies the viewed table. This is illustrated in the following examples.

Viewed table definition:

```
CREATE VIEW "V1" ("VC1","VC2")
    AS SELECT * FROM "T1" WHERE "C1">100
```

SELECT statement to be executed:

```
SELECT * FROM "V1"
```

When the SELECT statement shown above is executed, the following internal derived table is used.

Internal derived table that is used:

```
SELECT * FROM (SELECT * FROM "T1" WHERE "C1">100) AS "V1" ("VC1","VC2")
```

The underlined portion indicates the internal derived table.

(2) Example 2: Executing a query using a query name in a WITH clause

When you execute a query using a query name in a WITH clause, an internal derived table is used in the FROM clause that specifies the query name. This is illustrated in the following example.

SELECT statement to be executed:

```
WITH "Q1"("QC1","QC2") AS (SELECT * FROM "T1" WHERE "C1">100)
SELECT * FROM "Q1"
```

Here, Q1 is the query name, and the underlined portion indicates the query expression body specified in the WITH clause. When the SELECT statement shown above is executed, the following internal derived table is used.

Internal derived table that is used:

```
SELECT * FROM (SELECT * FROM "T1" WHERE "C1">100) AS "Q1" ("QC1","QC2")
```

The underlined portion indicates the internal derived table.

7.30.2 Derived queries and derived query names

A query expression body that generates a derived table is called a *derived query*. In addition, the name of the table derived from the derived query is called the *derived query name*. The derived table's derived query name is handled as a correlation name.

The derived query and derived query name are identified in each of the following examples.

View definition

```
CREATE VIEW "V1" AS SELECT * FROM "T1" WHERE "C1">100
```

Derived query: underlined portion

Derived query name: `"V1"`

Derived table

```
SELECT "C1","C2"*1.05
   FROM (SELECT "C1","C2" FROM "T1" GROUP BY "C1","C2") "X"
```

Derived query: underlined portion

Derived query name: `"X"`

`WITH` clause query

```
WITH "Q1" AS (SELECT "C1","C2" FROM "T1" GROUP BY "C1","C2")
SELECT "C1","C2"*1.05 FROM "Q1"
```

Derived query: underlined portion

Derived query name: `"Q1"`

7.30.3 Rules for derived table expansion

When you execute a derived query to generate a derived table, it expands the outer query expression body into an equivalent form that does not contain the derived table. This is called a *derived table expansion*. The following examples illustrate derived table expansions.

Example of a query that uses a derived table

```
SELECT "PN1","PR2"*1.05 AS "TXPRICE"
   FROM (SELECT "PNAME","PRICE","PLACE" FROM "STOCK"
         WHERE "PRICE">10000)
      AS "X"("PN1","PR2","PL3")
         WHERE "PL3" IN('Alaska','Arizona')
```

Example of derived table expansion

```
SELECT "PNAME" AS "PN1","PRICE"*1.05 AS "TXPRICE"
   FROM "STOCK"
      WHERE "PRICE">10000 AND "PLACE" IN('Alaska','Arizona')
```

The following shows the expansion rules for derived tables:

1. No internal derived table is expanded when all of the following conditions are met:
   - The same viewed table is specified more than once in a single SQL statement.
   - Any of the following items were specified in the query expression body when the viewed table was defined:

    - SELECT DISTINCT
    - Set operations
    - Subquery
    - Comma join
    - Joined table
    - Viewed table
    - Query name
    - Set function
    - Window function
    - GROUP BY clause
    - HAVING clause
    - Table function derived table
    - Archivable multi-chunk table
    - WHERE clause

   Similarly, no internal derived table is expanded in the following case: the same query name is specified more than once in a single SQL statement, and any of the preceding items are specified within the query expression body specified for a WITH list element of that query name.

2. If a recursive query is specified in a WITH list element, the following rule applies: If a viewed table or query name is specified as a recursive member of the recursive query, no internal derived table is expanded for the viewed table or query name.
3. If a recursive query is included in a WITH list element, no internal derived table is expanded for the query name of the WITH list element.
4. No internal derived table for the query name of a WITH list element is expanded if the list element satisfies either Condition 1 or Condition 2 shown later. Also, in a certain case, a derived table that is derived from a viewed table defined by the CREATE VIEW statement is not expanded. The case is when the WITH clause is specified in the query expression in the CREATE VIEW statement and the query name of a WITH list element that satisfies either of the following conditions is referenced from that query expression:

   Condition 1:

   All of the following conditions are satisfied:

   - Multiple WITH list elements are specified.
   - Any of the following items are specified in the query expression bodies specified in WITH list elements:
     - SELECT DISTINCT
     - Set operations
     - Subquery
     - Comma join
     - Joined table
     - Viewed table
     - Query name
     - Set function
     - Window function

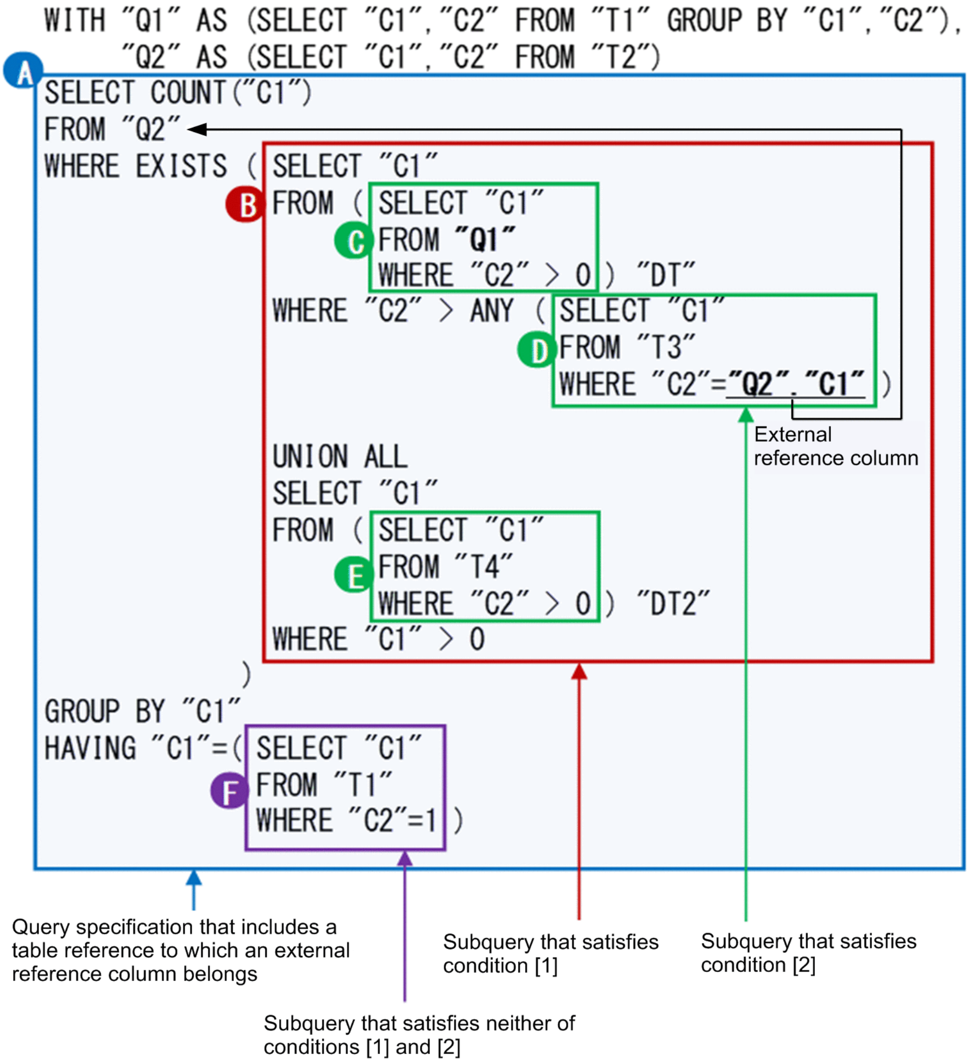
- `GROUP BY` clause
- `HAVING` clause
- Table function derived table
- Archivable multi-chunk table

- Any of the following conditions are satisfied:
  - The query name that corresponds to a `WITH` list element is specified at two or more locations in the SQL statement.
  - The query name that corresponds to a `WITH` list element is specified at one location in the SQL statement, and the query name is specified in a subquery that satisfies either of the following conditions:

    [1] Of the subqueries that are specified in a query specification that includes a table reference to which an external reference column belongs, a subquery that includes a subquery in which that external reference column is specified

    [2] A subquery that is included in the subquery in [1]

    Example of where the query name `Q1` satisfies condition [2]:



```
WITH "Q1" AS (SELECT "C1","C2" FROM "T1" GROUP BY "C1","C2"),
     "Q2" AS (SELECT "C1","C2" FROM "T2")
SELECT COUNT("C1")
FROM "Q2"
WHERE EXISTS (SELECT "C1"
              FROM (SELECT "C1"
                    FROM "Q1"
                    WHERE "C2" > 0) "DT"
              WHERE "C2" > ANY (SELECT "C1"
                                FROM "T3"
                                WHERE "C2"="Q2"."C1")

              UNION ALL
              SELECT "C1"
              FROM (SELECT "C1"
                    FROM "T4"
                    WHERE "C2" > 0) "DT2"
              WHERE "C1" > 0
             )
GROUP BY "C1"
HAVING "C1"=(SELECT "C1"
             FROM "T1"
             WHERE "C2"=1)
```

A
B
C
D
E
F
External reference column
Query specification that includes a table reference to which an external reference column belongs
Subquery that satisfies condition [1]
Subquery that satisfies condition [2]
Subquery that satisfies neither of conditions [1] and [2]

In the preceding example, query B is a subquery that satisfies condition [1]. Queries C to E are a subquery that satisfies condition [2]. Query F is a subquery that satisfies neither condition [1] nor condition [2]. The query name `Q1` satisfies condition [2] because it is specified in query C.

Condition 2:

All of the following conditions are satisfied:

- A WITH list element is specified.
- The same viewed table is specified multiple times in a query expression body that is specified in the WITH list element.

  Alternatively, the query name (other than a WITH clause) that corresponds to the WITH list element is specified multiple times.
- The query name that corresponds to the WITH list element is specified in either of the following subqueries:

  [1] Of the subqueries that are specified in a query specification that includes a table reference to which an external reference column belongs, a subquery that includes a subquery in which that external reference column is specified

  [2] A subquery that is included in the subquery in [1]

5. The following derived tables are not expanded:

- Internal derived table equivalently exchanged from a dictionary table or system table that is specified in the query expression in a CREATE VIEW statement
- Derived table that is derived by a table value constructor
- Derived table that is equivalently exchanged by the FULL OUTER JOIN specification
- Derived table that is equivalently exchanged from an archivable multi-chunk table

  For details about a derived table that is equivalently exchanged from an archivable multi-chunk table, see *Equivalent exchange of SQL statements that search archivable multi-chunk tables* in the *HADB Application Development Guide*.
- Derived table that is equivalently exchanged by the specification of the OR condition

  For details about a derived table that is equivalently exchanged by the specification of the OR condition, see *Equivalent exchange for OR conditions (equivalent exchange to derived tables for which the UNION ALL set operation is specified)* in the *HADB Application Development Guide*.

7.30.4 Conditions under which derived table expansion is not performed

If any of the following conditions are satisfied, derived table expansion is not performed.

1. If, in the outermost query specification for a derived query, you specify a derived query name with SELECT DISTINCT in the FROM clause, and one of the following specifications is in the query specification that directly includes that FROM clause:

- GROUP BY clause, HAVING clause, or set function

  The following is an example:

```
SELECT SUM("C1") FROM (SELECT DISTINCT * FROM "T1") AS "V1"
    GROUP BY "C2"
```

- Table join (including joined table)

  The following is an example:

```
SELECT * FROM (SELECT DISTINCT * FROM "T1") AS "V1","T1"
    WHERE "V1"."C1"="T1"."C1"
```

- Selection expression in which a derived column derived from a value expression that includes a column specification is not specified as a single column specification

The following is an example:

```
SELECT "VC1" FROM (SELECT DISTINCT "C1","C2" FROM "T1") AS "V1"("VC1","VC2")
```

Because derived column `VC2` is not specified in the selection expression, derived table `V1` is not expanded.

```
SELECT "VC1"*1.05,"VC2" FROM (SELECT DISTINCT * FROM "T1") AS "V1"("VC1","VC2")
```

Because derived column `VC1` is not specified as a single column specification in the selection expression, derived table `V1` is not expanded.

- Selection expression containing a value expression that includes scalar function `RANDOM`, scalar function `RANDOM_NORMAL`, scalar function `RANDOMROW`, a scalar subquery, or a window function

  The following is an example:

  ```
  SELECT "VC1","VC2",RANDOM()
      FROM (SELECT DISTINCT "C1","C2" FROM "T1") AS "V1"("VC1","VC2")
  ```

- Selection expression in which a derived column derived from the following value expression is not specified as a single column specification: a value expression that includes scalar function `RANDOM`, scalar function `RANDOM_NORMAL`, scalar function `RANDOMROW`, or a set function, and does not include a column specification

  The following is an example:

  ```
  SELECT "VC1" FROM (SELECT DISTINCT "C1",RANDOM() FROM "T1") AS "V1"("VC1","VC2")
  ```

- Sort key value expression that is not specified as a selection expression

  The following is an example:

  ```
  SELECT * FROM (SELECT DISTINCT * FROM "T1") AS "V1"
      ORDER BY ("C2"+1)
  ```

2. If, in the outermost query specification for a derived query, you specify a derived query name with a `GROUP BY` clause in the `FROM` clause, and one of the following specifications is in the query specification that directly includes that `FROM` clause:

   - A `DISTINCT` set function or inverse distribution function is specified.

     The following is an example:

     ```
     SELECT "C1",SUM(DISTINCT "C2")
         FROM (SELECT "C1","C2" FROM "T1" GROUP BY "C1","C2") AS "V1"
         GROUP BY "C1","C2"
     ```

   - Table join (including joined table)

     The following is an example:

     ```
     SELECT * FROM (SELECT "C1","C2" FROM "T1" GROUP BY "C1","C2")
         AS "V1","T1"
     ```

3. If, all of the following conditions are met: 1) In the outermost query specification for a derived query, the name of a derived query that contains the `GROUP BY` clause is specified for a `FROM` clause. 2) The query specification that directly contains that `FROM` clause also contains the `GROUP BY` clause, the `HAVING` clause, or a set function. 3) Either of the following conditions is met:

   - The number of grouping columns in the query specification for operating a derived query is different from the number of grouping columns in the derived query.

     The following is an example:

```
SELECT "C1","C2" FROM (SELECT "C1","C2" FROM "T1" GROUP BY "C1","C2","C3") AS "V
1"
    GROUP BY "C1","C2"
```

- The derived column derived from the column referenced by a grouping column for a derived query is not specified alone in a grouping column that has a query specification for operating the derived query.

  The following is an example:

```
SELECT "C1","C2"+1 FROM (SELECT "C1","C2" FROM "T1" GROUP BY "C1","C2") AS "V1"
    GROUP BY "C1","C2"+1
```

4. If, in the outermost query specification for a derived query, you specify a derived query name with a GROUP BY clause in the FROM clause including a value expression, and one of the following specifications is in the query specification that directly includes that FROM clause:

   - A grouping column of the derived query that was derived from a value expression that includes the column specification is specified as a column that makes an external reference.

     The following is an example:

```
SELECT * FROM (SELECT "G1" FROM "T1" GROUP BY C1+1 "G1") AS "V1"
    WHERE EXISTS (SELECT * FROM "T2" WHERE "T2"."C1" = "V1"."G1")
```

5. If, in the selection expression of the outermost query specification for a derived query, you specify a derived query name with a value expression that includes a column specification in the FROM clause of that selection expression, and one of the following occurs in the query specification that directly includes that FROM clause:

   - A column of the derived query name that was derived from a value expression that includes a column specification is specified in a selection expression or in the HAVING clause as a grouping column that makes an external reference.

     The following is an example:

```
SELECT "DC1" FROM (SELECT "C1"+1 AS "DC1" FROM "T1") AS "V1"
    GROUP BY "DC1"
    HAVING EXISTS (SELECT * FROM "T2" WHERE "T2"."C1" = "V1"."DC1")
```

   - Multiple inverse distribution functions are specified.

     The following is an example:

```
SELECT MEDIAN("C1")
    FROM (SELECT ABS("C1") AS "C1","C2" FROM "T1") AS "V1"
    HAVING MEDIAN("C1")>100
```

   - A set function argument that makes an external reference to a column of the derived query name that was derived from a value expression that includes the column specification

     The following is an example:

```
SELECT "C1" FROM (SELECT SUBSTR("C1",5) AS "C1","C2" FROM "T1") AS "V1"
    GROUP BY "C1"
    HAVING EXISTS(SELECT * FROM "T1" WHERE MAX("V1"."C1")="C1")
```

6. If, in the selection expression of the outermost query specification for a derived query, a derived query name (for which a value expression that does not include a column specification) is specified in the table reference of a joined table that is on the side filled with null values

   The following is an example:

```
SELECT * FROM "T1" LEFT OUTER JOIN
                    (SELECT SUBSTR('ABC',2) AS "C1","C2" FROM "T2") AS "V1"
                    ON "T1"."C1"="V1"."C1"
```

7. If, in the selection expression of the outermost query specification for a derived query, you specify a derived query name with a value expression that does not include a column specification in the FROM clause of that selection expression, and one of the following occurs in the query specification that directly includes that FROM clause:

   - Multiple inverse distribution functions are specified.

     The following is an example:

     ```
     SELECT MEDIAN("C1")
         FROM (SELECT ABS(100) AS "C1","C2" FROM "T1") AS "V1"
         HAVING MEDIAN("C1")>100
     ```

   - A window function containing a derived query column name derived from a value expression that does not include a column specification

     The following is an example:

     ```
     SELECT "C2",SUM("C2") OVER(ORDER BY "C1")
          FROM (SELECT SUBSTR('ABC',2) AS "C1","C2" FROM "T1") AS "V1"
     ```

8. If, in the selection expression of the outermost query specification of a derived query, you specify, in a FROM clause, the name of a derived query in which a value expression that includes the scalar function RANDOM or RANDOM_NORMAL is specified, and you specify, in the SQL statement, a column derived from the scalar function RANDOM or RANDOM_NORMAL

   The following is an example:

   ```
   SELECT "C1", "C2" FROM (SELECT "C1"+RANDOM() AS "C1", "C2" FROM "T2") AS "V1"
   ```

9. If, all of the following conditions are met: 1) In the selection expression of the outermost query specification for a derived query, the name of a derived query that uses a value expression that includes the scalar function RANDOMCURSOR is specified for a FROM clause. 2) In the query specification that directly contains that FROM clause, the column derived from the value expression that includes scalar function RANDOMCURSOR for a derived query is specified in an item other than a selection expression and ORDER BY clause.

   The following is an example:

   ```
   SELECT "C1","C2"
       FROM (SELECT "C1"+RANDOMCURSOR(1,10,20) AS "C1","C2" FROM "T2") AS "V1"
         WHERE "C1">1000
   ```

10. If, all of the following conditions are met: 1) In the selection expression of the outermost query specification for a derived query, the name of a derived query that uses a value expression that includes the scalar function RANDOMROW is specified for a FROM clause. 2) The query specification that directly contains that FROM clause also contains the following specification:

    - An item (other than a selection expression and ORDER BY clause) in which the column derived from a value expression that includes scalar function RANDOMROW for a derived query is specified

      The following is an example:

      ```
      SELECT "C1","C2"
          FROM (SELECT "C1"+RANDOMROW(1,10,20) AS "C1","C2" FROM "T2") AS "V1"
            WHERE "C1">1000
      ```

    - A set function argument for which the column derived from a value expression that includes scalar function RANDOMROW for a derived query is specified

      The following is an example:

      ```
      SELECT SUM("C1")
          FROM (SELECT "C1"+RANDOMROW(1,10,20) AS "C1","C2" FROM "T2") AS "V1"
      ```

- A window function in which the column derived from a value expression that includes scalar function `RANDOMROW` for a derived query is specified

  The following is an example:

  ```
  SELECT "C1",SUM("C2") OVER(ORDER BY "C1")
      FROM (SELECT "C1", "C2"+RANDOMROW(1,10,20) AS "C2" FROM "T2") AS "V1"
  ```

- The scalar function `RANDOMROW` in which the column derived from a value expression that includes the scalar function `RANDOMROW` for a derived query is specified

  The following is an example:

  ```
  SELECT RANDOMROW(1,"C1","C2")
      FROM (SELECT "C1"+RANDOMROW(1,10,20) AS "C1","C2" FROM "T2") AS "V1"
  ```

- Table join (including joined table)

  The following is an example:

  ```
  SELECT * FROM (SELECT "C1"+RANDOMROW(1,10,20) AS "C1","C2" FROM "T2")
      AS "V1","T1"
  ```

11. If, in the selection expression of the outermost query specification of a derived query, you specify a derived query name with a value expression that includes a dynamic parameter in the `FROM` clause of that selection expression, and you specify a column derived from the dynamic parameter in the query specification.

    The following is an example:

    ```
    SELECT "C1" FROM (SELECT SUBSTR("C1",?) AS "C1","C2" FROM "T1") AS "V1"
    ```

12. If, in the selection expression of the outermost query specification of a derived query, you specify a derived query name that specifies a value expression that includes a scalar subquery in the `FROM` clause of that selection expression.

    The following is an example:

    ```
    SELECT "C1" FROM (SELECT (SELECT "C1" FROM "T2") + 10 AS "C1"
                          FROM "T1") AS "V1"
    ```

13. If, in the selection expression of the outermost query specification of a derived query, you specify a derived query name with a value expression that includes an inverse distribution function in the `FROM` clause of that selection expression, and you specify a column derived from the inverse distribution function in the query specification.

    The following is an example:

    ```
    SELECT "C1" FROM (SELECT MEDIAN("C1") AS "C1",MAX("C2")
                          FROM "T1") AS "V1"
    ```

14. If, all of the following conditions are met: 1) In the selection expression of the outermost query specification for a derived query, the name of a derived query that uses a set function is specified for a `FROM` clause. 2) The query specification that directly contains that `FROM` clause also contains the following specification:

    - The `GROUP BY` clause that includes a value expression, `DISTINCT` set function, or inverse distribution function is specified

      The following is an example:

      ```
      SELECT SUM(DISTINCT "C1")
          FROM (SELECT COUNT("C1") AS "C1" FROM "T1") AS "V1"
          GROUP BY "C1"
      ```

    - Table join (including joined table)

      The following is an example:

```
SELECT * FROM (SELECT COUNT("C1") AS "C1" FROM "T1") AS "V1","T1"
    WHERE "V1"."C1"="T1"."C1"
```

- A set function argument for which a column of the derived query name that was derived from a set function is specified as a column that makes an external reference

  The following is an example:

```
SELECT "C1" FROM (SELECT COUNT("C1") AS "C1" FROM "T1") AS "V1"
    GROUP BY "C1"
    HAVING EXISTS(SELECT * FROM "T2" WHERE MAX("V1"."C1")="C1")
```

15. If a derived query name specifying a window function is specified in a FROM clause in the outermost query specification of a derived query

    The following is an example:

```
SELECT "C2" FROM (SELECT "C1",AVG("C1") OVER(ORDER BY "C2") AS "C2"
                      FROM "T1") AS "V1"
```

16. The `LIMIT` clause is included in the outermost query specification of a derived query.

    The following is an example:

```
SELECT "C1","C2" FROM (SELECT "C1","C2" FROM "T1" LIMIT 10) AS "V1"
```

17. A derived query name for which a comma join is specified is specified for a table reference to a joined table in the outermost query specification of a derived query.

    The following is an example:

```
SELECT "VC1","VC2" FROM (SELECT "T1"."C1","T2"."C1" FROM "T1","T2","T3"
                           WHERE "T1"."C1"="T2"."C1"
                             AND "T2"."C1"="T3"."C1") AS "V1"("VC1","VC2")
                         LEFT JOIN "T3" ON "VC1" = "T3"."C1"
```

18. A derived query is specified for a table reference in `FULL OUTER JOIN`.

    The following is an example:

```
SELECT * FROM (SELECT "C1" FROM "T1") AS "V1"
                FULL OUTER JOIN "T2" ON "V1"."C1"="T2"."C1"
```

19. `VARCHAR`-type data larger than 32,000 bytes is contained in the column derived as a result of a set operation that uses an operand that is the outermost query specification of a derived query.

    The following is an example:

```
SELECT "C1" FROM (SELECT "C1" FROM "T1"
                  UNION
                  SELECT "DEFINE_SOURCE" FROM "MASTER"."SQL_DEFINE_SOURCE"
                 ) AS "V1"
```

20. At least one of the conditions under which an internal derived table in preceding items 1 to 18 is not expanded is satisfied in the following case: the query specification in an operand of a set operation specified in a derived query is assumed to be a derived query.

    The following is an example:

```
SELECT "C1" FROM (SELECT DISTINCT "C1","C2" FROM "T1"
                  UNION ALL
                  SELECT "C1","C2" FROM "T2"
                 ) AS "V1"
```

Assume that SELECT DISTINCT "C1","C2" FROM "T1", which is a query specification in an operand of a set operation, is a derived query. In this case, a condition[#] under which the internal derived table in item 1 is not expanded is satisfied. Therefore, derived table V1 is not expanded.

#

The FROM clause includes the name of a derived query in which SELECT DISTINCT is specified in the outermost query specification, and the FROM clause is directly included in a query specification that satisfies the following condition:

- A derived column derived from a value expression that includes a column specification is not specified as a single column specification in a selection expression.

21. If, in the FROM clause of the outermost query specification for a derived query, you specify a derived query name that specifies only the UNION ALL set operator, and one of the following is specified in the query specification that directly contains that FROM clause:

- GROUP BY clause, HAVING clause, or set function

  The following is an example:

```
SELECT "C1","C2"
    FROM (
          SELECT "C1","C2" FROM "T1"
          UNION ALL SELECT "C1","C2" FROM "T2"
         ) AS "V1"
    GROUP BY "C1","C2"
```

- Window function specified in the selection expression

  The following is an example:

```
SELECT "C1",SUM("C1") OVER(ORDER BY "C2")
    FROM (
          SELECT "C1","C2" FROM "T1"
          UNION ALL SELECT "C1","C2" FROM "T2"
          ) AS "V1"
```

- Table join (including joined table)

  The following is an example:

```
SELECT * FROM (
               SELECT "C1", "C2" FROM "T1"
               UNION ALL SELECT "C1","C2" FROM "T2"
               ) AS "V1","T3"
```

- Sort key that is a derived query name column that is not specified in the selection expression

  The following is an example:

```
SELECT "C1" FROM (
                  SELECT "C1","C2" FROM "T1"
                  UNION ALL SELECT "C1","C2" FROM "T2"
                  ) AS "V1"
    ORDER BY "C2"
```

- Sort key value expression that is not specified as a selection expression

  The following is an example:

```
SELECT "C1" FROM (
                  SELECT "C1","C2" FROM "T1"
                  UNION ALL SELECT "C1","C2" FROM "T2"
                  ) AS "V1"
    ORDER BY ("C1"+1)
```

22. If, in the FROM clause of the outermost query specification for a derived query, you specify a derived query name that specifies a set operator other than UNION ALL, and one of the following is specified in the query specification that directly contains that FROM clause:

- SELECT DISTINCT

  The following is an example:

```
SELECT DISTINCT "C1" FROM (
                           SELECT "C1" FROM "T1"
                           EXCEPT ALL SELECT "C1" FROM "T2"
                           ) AS "V1"
```

- GROUP BY clause, HAVING clause, or set function

  The following is an example:

```
SELECT "C1","C2" FROM (
                       SELECT "C1","C2" FROM "T1"
                       UNION SELECT "C1","C2" FROM "T2"
                       ) AS "V1"
    GROUP BY "C1","C2"
```

- Window function specified in the selection expression

  The following is an example:

```
SELECT "C1",SUM("C1") OVER(ORDER BY "C2") FROM (
                      SELECT "C1","C2" FROM "T1"
                      INTERSECT ALL SELECT "C1","C2" FROM "T2"
                      ) AS "V1"
```

- Table join (including joined table)

  The following is an example:

```
SELECT * FROM (
               SELECT "C1","C2" FROM "T1"
               EXCEPT SELECT "C1","C2" FROM "T2"
               ) AS "V1","T3"
```

- Selection expression containing a value expression that includes scalar function RANDOM, scalar function RANDOM_NORMAL, scalar function RANDOMROW, a scalar subquery, or a window function

  The following is an example:

```
SELECT "VC1","VC2",RANDOM() FROM (
                                  SELECT "C1","C2" FROM "T1"
                                  UNION SELECT "C1","C2" FROM "T2"
                                  ) AS "V1"("VC1","VC2")
```

- Selection expression containing one or more derived query name columns that are not specified as single column specifications

  The following is an example:

```
SELECT "VC1" FROM (
                   SELECT "C1","C2" FROM "T1"
                   INTERSECT SELECT "C1","C2" FROM "T2"
                   ) AS "V1"("VC1","VC2")
```

  Because derived column VC2 is not specified in the selection expression, derived table V1 is not expanded.

  The following is an example:

```
SELECT "VC1"*1.05,"VC2" FROM (
                              SELECT "C1","C2" FROM "T1"
                              UNION SELECT "C1","C2" FROM "T2"
                              ) AS "V1"("VC1","VC2")
```

Because derived column VC1 is not specified as a single column specification in the selection expression, derived table V1 is not expanded.

- Sort key that is a derived query name column that is not specified in the selection expression
  The following is an example:

```
SELECT "C1" FROM (
                  SELECT "C1","C2" FROM "T1"
                  UNION SELECT "C1","C2" FROM "T2"
                  ) AS "V1"
    ORDER BY "C2"
```

- Sort key value expression that is not specified as a selection expression
  The following is an example:

```
SELECT "C1" FROM (
                  SELECT "C1","C2" FROM "T1"
                  UNION SELECT "C1","C2" FROM "T2"
                  ) AS "V1"
    ORDER BY ("C1"+1)
```

7.30.5 Summary of when derived table expansion is performed

The following table summarizes the conditions under which derived table expansion is and is not performed.

Table 7-40: Summary of when derived table expansion is performed (1/2)

Specification of the SQL statement that manipulates a derived query	**Specification of the derived query**								
	SEL_DIST	**GRP**	**GRP_EXP**	**SEL_EXP**	**SEL_NCOL**	**SEL_RAND**	**SEL_RANDCRS**	**SEL_RANDROW**	**SEL_PRM**
SEL_DIST	Y	Y	Y	Y	Y	Y[#5]	Y	Y	N
GRP	N	Y[#6]	Y[#6]	Y[#2]	Y[#2]	Y[#5]	D	D	N
GRP_EXP	N	N	N	Y	Y	Y[#5]	D	D	N
A-FUNC	N	Y[#6, #7]	Y[#6, #7]	Y	Y	Y[#5]	Y[#8]	D	N
D-FUNC	N	N	N	Y	Y	Y[#5]	Y[#8]	D	N
I-FUNC2	N	N	N	D	D	Y[#5]	Y[#8]	D	N
WIN_AGG	N	Y	Y	Y	Y	Y[#5]	Y[#8]	D	N
WIN_PAR	N	Y	Y	Y	D	Y[#5]	Y[#8]	D	N
WIN_ORD	N	Y	Y	Y	Y	Y[#5]	Y[#8]	D	N
SEL_EXP	Y	Y	Y	Y	Y	Y[#5]	Y	Y[#9]	N
SEL_RAND	N	Y	Y	Y	Y	Y[#5]	Y	Y	N
SEL_RANDROW	N	Y	Y	Y	Y	Y[#5]	Y	Y[#9]	N
SEL_SUBQ	N	Y	Y	Y	Y	Y[#5]	Y	Y	N

Specification of the SQL statement that manipulates a derived query	Specification of the derived query								
	SEL_DIST	GRP	GRP_EXP	SEL_EXP	SEL_NCOL	SEL_RAND	SEL_RANDCRS	SEL_RANDROW	SEL_PRM
SEL_WINDOW	N	Y	Y	Y	Y	Y[#5]	Y	Y	N
SEL_CNDRV	N	Y	Y	Y	Y	Y[#5]	Y	Y	N
SEL_NCNDRV	Y[#11]	Y	Y	Y	Y	Y[#5]	Y	Y	N
JOIN	N	N	N	Y	Y	Y[#5]	Y	Y[#5]	N
IN_J_TBL	N	N	N	Y[#4]	N	Y[#5]	Y	Y[#5]	N
J_TBL[#12]	N	N	N	Y	Y	Y[#5]	Y	Y[#5]	N
FJ_TBL	N	N	N	N	N	N	N	N	N
S_KEY	Y	Y	Y	Y	Y	Y[#5]	Y	Y	N
SEXP_KEY	N	Y	Y	Y	Y	Y[#5]	Y	Y[#9]	N
O_REF	Y	Y	D	Y	Y	Y[#5]	D	D	N
O_REF_FUNC	N	N	N	D	D	Y[#5]	D	D	N
Other items	Y	Y	Y	Y	Y	Y[#5]	Y[#8]	Y [#8, #9]	N

Table 7-41: Summary of when derived table expansion is performed (2/2)

Specification of the SQL statement that manipulates a derived query	Specification of the derived query									
	SUBQ	SEL_IFN	FUNC_COL	FUNC_EXP	WINDOW	CJOIN[#12]	LIMIT	U_ALL[#3, #10]	SET_OP[#3, #10]	SETOP_VCH32000	Other items
SEL_DIST	N	N	Y	Y	N	Y	N	Y	Y[#1]	N	Y
GRP	N	N	D	D	N	Y	N	N	N	N	Y
GRP_EXP	N	N	D	D	N	Y	N	N	N	N	Y
A-FUNC	N	N	Y[#7]	Y[#7]	N	Y	N	N	N	N	Y
D-FUNC	N	N	D	D	N	Y	N	N	N	N	Y
I-FUNC2	N	N	D	D	N	Y	N	N	N	N	Y
WIN_AGG	N	N	Y	Y	N	Y	N	N	N	N	Y
WIN_PAR	N	N	Y	Y	N	Y	N	N	N	N	Y
WIN_ORD	N	N	Y	Y	N	Y	N	N	N	N	Y
SEL_EXP	N	N	Y	Y	N	Y	N	Y	Y	N	Y
SEL_RAND	N	N	Y	Y	N	Y	N	Y	N	N	Y
SEL_RANDROW	N	N	Y	Y	N	Y	N	Y	N	N	Y
SEL_SUBQ	N	N	Y	Y	N	Y	N	Y	N	N	Y
SEL_WINDOW	N	N	Y	Y	N	Y	N	Y	N	N	Y
SEL_CNDRV	N	N	Y	Y	N	Y	N	Y	N	N	Y

Specification of the SQL statement that manipulates a derived query	Specification of the derived query										
	SUBQ	SEL_IFN	FUNC_COL	FUNC_EXP	WINDOW	CJOIN#12	LIMIT	U_ALL#3, #10	SET_OP#3, #10	SETOP_VCH32000	Other items
SEL_NCNDRV	N	N	Y	Y	N	Y	N	Y	N	N	Y
JOIN	N	N	N	N	N	Y	N	N	N	N	Y
IN_J_TBL	N	N	N	N	N	N	N	N	N	N	Y
J_TBL#12	N	N	N	N	N	N	N	N	N	N	Y
FJ_TBL	N	N	N	N	N	N	N	N	N	N	N
S_KEY	N	N	Y	Y	N	Y	N	Y	Y	N	Y
SEXP_KEY	N	N	Y	Y	N	Y	N	N	N	N	Y
O_REF	N	N	Y	D	N	Y	N	Y	Y	N	Y
O_REF_FUNC	N	N	N	N	N	Y	N	N	N	N	Y
Other items	N	N	Y	Y	N	Y	N	Y	Y	N	Y

Legend:

Y: Derived table expansion is performed.

D: In general, derived table expansion is performed. However, if you specify a derived column that is derived from the specification of the derived query in a location in which an SQL statement that manipulates a derived query is specified, derived table expansion is not performed.

N: Derived table expansion is not performed.

- A-FUNC: The ALL set function, but only if a column of the derived query name is specified as an aggregated argument
- CJOIN: A comma join
- D-FUNC: A DISTINCT set function, but only if a column of the derived query name is specified as an aggregated argument. Alternatively, only one inverse distribution function in which a column of the derived query name is specified as an aggregated argument.
- I-FUNC2: Two or more inverse distribution functions in which a column of the derived query name is specified as an aggregated argument
- FJ_TBL: A derived query name is specified in a table reference of FULL OUTER JOIN
- FUNC_COL: A set function, but only if column specifications appear as aggregated arguments
- FUNC_EXP: A set function, but only if no column specifications appear as aggregated arguments
- GRP: GROUP BY clause, a HAVING clause, or a set function
- GRP_EXP: GROUP BY clause without column specifications (such as scalar operations)
- IN_J_TBL: The corresponding derived query name for the table reference of a joined table that is on the side filled with null values
- JOIN: Multiple tables
- J_TBL: A derived query name specified in a table reference to a joined table
- LIMIT: A LIMIT clause
- O_REF: A column of the derived query name used as an external reference column

- O_REF_FUNC: A column of the derived query name used as an external reference column in the argument of a set function
- S_KEY: A column of the derived query name specified in the selection expression used as a sort key
- SEL_CNDRV: A derived column derived from a value expression that includes a column specification is not specified as a single column specification in a selection expression.
- SEL_IFN: An inverse distribution function specified in a selection expression
- SEL_DIST: A SELECT DISTINCT clause
- SEL_EXP: Non-column specifications (such as scalar operations) used in a selection expression (even if column specifications are used in value expressions)
- SEL_NCNDRV: A derived column derived from a value expression that does not include a column specification is not specified as a single column specification in a selection expression.
- SEL_NCOL: A value expression consisting of non-column specifications in the selection expression
- SEL_PRM: A dynamic parameter appears in the selection expression.
- SEL_RAND: The selection expression includes the scalar function RANDOM or RANDOM_NORMAL.
- SEL_RANDCRS: The selection expression includes the scalar function RANDOMCURSOR.
- SEL_RANDROW: The selection expression includes the scalar function RANDOMROW.
- SEL_SUBQ: A selection expression includes a scalar subquery.
- SEL_WINDOW: A selection expression includes a window function.
- SET_OP: Set operations specified in cases other than U_ALL
- SETOP_VCH32000: Column of the VARCHAR type larger than 32,000 bytes among the columns derived by the result of a set operation
- SEXP_KEY: A sort key value expression that is not specified as a selection expression

  Example:

  ```
  SELECT "C1"+"C2","C2" AS "DC1" FROM "T1" ORDER BY "C1"/"C2"
  ```

- SUBQ: Subquery included in a selection expression
- U_ALL: Set operations that specify only UNION ALL
- WINDOW: Window function
- WIN_AGG: Derived query name column specified in a set function specified as a window function
- WIN_PAR: Derived query name column specified in a window partition clause in a window function
- WIN_ORD: Derived query name column specified in a window order clause in a window function

#1

Expansion of the derived table is not performed if the set operator that is evaluated last in the set operations specified in the derived query is EXCEPT ALL.

Example where expansion is not performed:

```
SELECT DISTINCT "C1","C2" FROM (
                    (SELECT "C1","C2" FROM "T1"
                     UNION ALL
                     SELECT "C1","C2" FROM "T2")
                     EXCEPT ALL
                     SELECT "C1","C2" FROM "T2") AS "V1"
```

In the SQL statement above, expansion of the derived table is not performed because the set operator that is evaluated last is `EXCEPT ALL`.

Example where expansion is performed:

```
SELECT DISTINCT "C1","C2" FROM (
                    SELECT "C1","C2" FROM "T1"
                    UNION ALL
                   (SELECT "C1","C2" FROM "T2"
                    EXCEPT ALL
                    SELECT "C1","C2" FROM "T2")) AS "V1"
```

In the SQL statement above, expansion of the derived table is performed because the set operator that is evaluated last is `UNION ALL`.

#2

If all of the following conditions are met, derived table expansion is not performed:

1. The derived query name column that was derived from a value expression is specified as a grouping column.
2. The derived query name column in condition 1 is specified in either of the following items and is an external reference column:

- Selection expression

  Example where expansion is not performed:

```
SELECT (SELECT "C1" FROM "T2" WHERE "T2"."C1" = "V1"."DC1") "DC2"
    FROM (SELECT "C1"+1 AS "DC1" FROM "T1") AS "V1"
    GROUP BY "DC1"
```

- `HAVING` clause

  Example where expansion is not performed:

```
SELECT "DC1" FROM (SELECT "C1"+1 AS "DC1" FROM "T1") AS "V1"
    GROUP BY "DC1"
    HAVING EXISTS (SELECT * FROM "T2" WHERE "T2"."C1" = "V1"."DC1")
```

#3

If `FULL OUTER JOIN` is specified in a subquery that is included in a query specification for manipulating a derived table with a set operation specified, expansion of a set operation derived table is not performed.

Example where expansion is not performed:

```
SELECT * FROM (SELECT "C1","C2" FROM "T1"
               UNION ALL
               SELECT "C1","C2" FROM "T2") "DT2"
    WHERE "C1"=ANY(SELECT "X"."C1" FROM "T3" FULL OUTER JOIN "T4"
                                             ON "T3"."C2"> "T4"."C2")
```

#4

If one of the following value expressions is specified in the selection expression of the outermost query specification for a derived query, derived table expansion is not performed:

- Scalar function `COALESCE`
- Scalar function `ISNULL`
- Scalar function `NULLIF`
- Scalar function `NVL`
- Scalar function `DECODE`
- Scalar function `LTDECODE`

- CASE expression

#5

If you specify, in the SQL statement, a derived column to which the details specified in the derived query apply, the derived table is not expanded. If you do not specify such a derived column, the derived table is expanded.

Example where expansion is not performed:

```
SELECT "DC1" FROM (SELECT RANDOM("C1","C2") AS "DC1" FROM "T1") AS "V1"
```

In the SQL statement above, the derived column "DC1" is specified in the SQL statement that manipulates the derived query. Because this derived column corresponds to a column in the selection expression that includes the scalar function RANDOM, which is specified in the derived query, the derived table is not expanded.

#6

If all of the following conditions are met, the derived table is expanded:

- The number of grouping columns in the query specification for operating a derived query is the same as the number of grouping columns in the derived query.
- All derived columns derived from the grouping column specified in a selection expression for a derived query are specified in a grouping column that has a query specification for operating the derived query.

Example where expansion is performed (1):

```
SELECT "C1","C2"
    FROM (SELECT "C1","C2" FROM "T1" GROUP BY "C1","C2") AS "V1"
    GROUP BY "C1","C2"
```

Example where expansion is performed (2):

```
SELECT "C1","DC2"
    FROM (SELECT "C1","C2"+1 AS "DC2" FROM "T1" GROUP BY "C1","C2"+1) AS "V1"
    GROUP BY "C1","DC2"
```

#7

The derived table is expanded when all set functions included in the query specification that operates the derived query satisfy Condition 1 or Condition 2, as follows:

Condition 1:

- The set function included in the query specification that operates the derived query is COUNT(*)#.

#

The set function COUNT (with ALL specified) for which a literal (or a value expression equivalent to a literal) is specified as an argument is replaced by the set function COUNT(*) and is treated as the set function COUNT(*).

Condition 2:

- One of the following set functions is included in the query specification that operates the derived query:
    - Set function MAX with ALL specified
    - Set function MIN with ALL specified
    - Set function SUM with ALL specified
    - Set function AVG with ALL specified
- A derived column consisting of the set functions specified in a derived query is specified as an aggregated argument of the preceding set function.

Example where expansion is performed:

```
SELECT "C1", "C2",SUM("C3")
    FROM (SELECT "C1","C2",COUNT("C3") AS "C3"
             FROM "T1"
             GROUP BY "C1","C2") "V1"
    GROUP BY "C1","C2"
```

#8

If a value expression that includes the derived column subject to the specification of the derived query is specified in an item other than a selection expression and ORDER BY clause, the derived table is not expanded.

#9

If the derived column subject to the specification of the derived query is specified in the scalar function RANDOMROW, the derived table is not expanded.

#10

The derived table is not expanded if at least one of the conditions under which an internal derived table is not expanded is satisfied in the following case: the query specification in an operand of a set operation specified in a derived query is assumed to be a derived query.

#11

The derived table is not expanded if the following derived column is not specified as a single column specification in a selection expression: a derived column derived from a value expression that does not include column specifications and includes any of the following specifications:

- Scalar function RANDOM
- Scalar function RANDOM_NORMAL
- Scalar function RANDOMROW
- Set function

#12

The HADB server might convert INNER JOIN or CROSS JOIN to a comma join. Therefore, if INNER JOIN or CROSS JOIN specified in a derived query is converted to a comma join, it is assumed that a comma join is included in a derived query (CJOIN in the preceding table applies). Also, if a joined table disappears from a query specification in a case where INNER JOIN or CROSS JOIN specified in a query specification that manipulates a derived query is converted to a comma join, it is assumed that no joined table is specified (J_TBL in the preceding table no longer applies).

7.30.6 When the scalar function CONVERT is added to an internal derived table

This section shows the cases in which the scalar function CONVERT is added to an internal derived table. This has the effect of increasing the number of scalar operations by one, and increasing the nesting of scalar operations by one.

- When a set operation is specified in an internal derived table

  The scalar function CONVERT is added to the selection expression so that the result will have the data type of the result of the set operation.

  Example

  - Query using derived tables

    ```
    SELECT "SN","KI"*1.08 AS "TAX" FROM
    (SELECT "NAME","PRICE","ORIGIN" FROM "PRODUCTS_A"
      WHERE "PRICE">10000
    ```

```
 UNION ALL
 SELECT "NAME","PRICE"*0.8,"ORIGIN" FROM "PRODUCTS_B"
  WHERE "PRICE">20000) AS "X"("SN","KI","GE")
WHERE "GE" IN('Tokyo','Osaka')
```

- Expansion of derived tables

```
SELECT CONVERT("NAME",VARCHAR(100)) AS "SN",
       CONVERT("PRICE"*1.08,DEC(23,2)) AS "TAX"
  FROM "PRODUCTS_A"
 WHERE "PRICE">10000 AND
       "ORIGIN" IN('Tokyo','Osaka)
 UNION ALL
SELECT CONVERT("NAME",VARCHAR(100)),
       CONVERT("PRICE"*0.8*1.08,DEC(23,2))
 FROM "PRODUCTS_B"
 WHERE "PRICE">20000 AND
       "ORIGIN" IN('Tokyo','Osaka')
```

- When all of the following conditions are met:
  - An internal derived table that meets Condition 2 in Note #7 in 7.30.5 Summary of when derived table expansion is performed is expanded.
  - The data type of the set function included in the query specification that operates the derived query is different from the data type of the result of the set function specified in the derived query.

  The scalar function CONVERT whose result will have the same data type as the result of the set function included in the query specification that operates the derived query is added to the set function specified in the derived query.

  Example:

  - Query using a derived table (in the case where the data type of the "TEMPERATURE" column is DECIMAL(10,2))

```
SELECT "POINT","DATE",AVG("TEMPERATURE") AS "TEMPERATURE"
  FROM (SELECT "POINT","DATE",MAX("TEMPERATURE") AS "TEMPERATURE"
          FROM "SENSOR_DATA"
            GROUP BY "POINT","DATE") "V1"
    WHERE "DATE" BETWEEN DATE'2018-01-01' AND DATE'2018-12-31'
      GROUP BY "POINT","DATE"
```

  - Expansion of derived tables

```
SELECT "POINT","DATE",CONVERT(MAX("TEMPERATURE"),DECIMAL(38, 30)) AS "TEMPERA
TURE"
  FROM "SENSOR_DATA"
    WHERE "DATE" BETWEEN DATE'2018-01-01' AND DATE'2018-12-31'
      GROUP BY "POINT","DATE"
```

8 Scalar Functions

This chapter describes the functioning, specification formats, and rules of scalar functions.

8.1 List of scalar functions

The following table lists all of the scalar functions.

Table 8-1: List of scalar functions

No.	Category		Name of scalar function	Description
1	Mathematical functions	Trigonometric functions	ACOS	Returns the angle (in radians) that is the inverse cosine of the target data, in the range 0 to π.
2			ASIN	Returns the angle (in radians) that is the inverse sine of the target data, in the range -π/2 to π/2.
3			ATAN	Returns the angle (in radians) that is the inverse tangent of the target data, in the range -π/2 to π/2.
4			ATAN2	Returns the angle (in radians) that is the inverse tangent of *y*/*x*, in the range -π to π.
5			COS	Returns the cosine (COS trigonometric function) of the target data, which must be specified in radians.
6			COSH	Returns the hyperbolic cosine of the target data.
7			DEGREES	Returns the result of converting the specified angle from radians to degrees.
8			PI	Returns the value of π.
9			RADIANS	Returns the result of converting the specified angle from degrees to radians.
10			SIN	Returns the sine (SIN trigonometric function) of the target data, which must be specified in radians.
11			SINH	Returns the hyperbolic sine of the target data.
12			TAN	Returns the tangent (TAN trigonometric function) of the target data, which must be specified in radians.
13			TANH	Returns the hyperbolic tangent of the target data.
14		Exponent and logarithm calculations	EXP	Returns the result of raising the base of the natural logarithm to a power.
15			LN	Returns the natural logarithm of the target data.
16			LOG	Returns the logarithm of the target data (antilogarithm) to the specified base.
17			POWER	Returns the result of raising the target data to a specified power.
18		Numerical calculations	ABS	Returns the absolute value of the target data.
19			CEIL	Returns the smallest integer that is equal to or greater than the target data.
20			FLOOR	Returns the greatest integer that is equal to or less than the value of the target data.
21			MOD	Returns the remainder after dividing the dividend by the divisor.
22			RANDOM	Returns pseudorandom numbers that follow a uniform distribution and are greater than or equal to the value specified

No.	Category		Name of scalar function	Description
				for the minimum value and less than the value specified for the maximum value.
23			RANDOMCURSOR	Returns pseudorandom numbers that follow a uniform distribution and are greater than or equal to the value specified for the minimum value and less than the value specified for the maximum value. If an SQL statement contains multiple RANDOMCURSOR functions for which the same identification number is specified, those functions always return the same values.
24			RANDOMROW	Returns pseudorandom numbers that follow a uniform distribution and are greater than or equal to the value specified for the minimum value and less than the value specified for the maximum value. If a query specification contains multiple RANDOMROW functions for which the same identification number is specified, those functions return the same values for each result row of the query specification.
25			RANDOM_NORMAL	Returns pseudorandom numbers that follow a normal distribution with an average μ and a standard deviation σ.
26			ROUND	Returns the value of the target data rounded to the *n*th digit after the decimal point.
27			SIGN	Returns the sign of the target data (+1 for positive, -1 for negative, 0 for 0).
28			SQRT	Returns the square root of the target data.
29			TRUNC	Returns a value that has been truncated to the specified number of decimal places.
30	Character string functions	Character string operations	CONCAT	Concatenates two character string data items.
31			LEFT	Extracts a substring from a character string starting from the beginning (leftmost position) of the character string data.
32			LPAD	Pads the beginning (left side) of the target data with the padding character string up to the specified number of characters.
33			LTRIM	Removes instances of the specified characters, starting from the beginning of the target character string.
34			RIGHT	Extracts a substring from a character string starting from the end (rightmost position) of the character string data.
35			RPAD	Pads the end (right side) of the target data with the padding character string up to the specified number of characters.
36			RTRIM	Removes instances of the specified characters, starting from the end of the target character string.
37			SUBSTR	Extracts a substring from a character string starting from any position in the character string data.
38			TRIM	Removes instances of the specified characters from the target character string. The characters can be removed in any of the following ways: • Remove the specified characters starting from the beginning of the character string.

No.	Category		Name of scalar function	Description
				• Remove the specified characters starting from the end of the character string. • Remove characters starting from both the beginning and the end of the character string.
39		Acquisition of character string information	CONTAINS	Returns whether character strings that meet the search condition expression are included in the target data.
40			INSTR	Searches the target data for a character string and returns the starting position of the string.
41			LENGTH	Returns the number of characters in the target character string.
42		Character substitution	REPLACE	Replaces any character string in the target data. All instances of the character string to be replaced in the target data are replaced with a replacement character string.
43			TRANSLATE	Replaces any character in the target data.
44		Character string conversion	LOWER	Converts uppercase letters (A to Z) to lowercase letters (a to z) in character string data.
45			UPPER	Converts lowercase letters (a to z) to uppercase letters (A to Z) in character string data.
46	Datetime functions		DATEDIFF	Returns the difference between the start date and time and the end date and time.
47			DAYOFWEEK	Returns the day of the week that the specified date falls on.
48			DAYOFYEAR	Returns the specified date as the number of days elapsed since January 1 of that year.
49			EXTRACT	Extracts a part (year, month, day, hour, minute, or second) from data representing the date and time.
50			GETAGE	Determines a person's age on a reference date given their birth date.
51			LASTDAY	Returns the date or datetime of the last day of the month specified in the datetime data.
52			ROUND	Returns the datetime data rounded to the unit specified in the datetime format.
53			TRUNC	Returns the datetime data truncated to the unit specified in the datetime format.
54	Binary column functions	Binary data operation	CONCAT	Concatenates two binary data items.
55			SUBSTRB	Extracts a substring from binary data starting from any position in the binary data.
56		Bit operations	BITAND	Returns the bitwise logical AND of two binary data items.
57			BITLSHIFT	Returns the value resulting from shifting the bits of a binary data value to the left.
58			BITNOT	Returns the bitwise logical NOT of a binary data item.
59			BITOR	Returns the bitwise inclusive OR of two binary data items.
60			BITRSHIFT	Returns the value resulting from shifting the bits of a binary data value to the right.

No.	Category		Name of scalar function	Description
61			BITXOR	Returns the bitwise exclusive OR of two binary data items.
62	Data conversion functions		ASCII	Returns the character code of the first character of the target data as an integer value.
63			BIN	Converts binary data to a binary string representation (character string data consisting of 0 and 1).
64			CAST	Converts the data type of the data.
65			CHR	Returns the character corresponding to the character code represented by the integer target data.
66			CONVERT	Converts the data type of the data. You can also specify a datetime format or number format to control the conversion. If you specify a datetime format: • When converting datetime data to character string data, you can specify the output format of the character string data after conversion. • When converting character string data to datetime data, you can specify the pre-conversion input format of the character string data. If you specify a number format: • When converting numeric data to character string data, you can specify the output format of the character string data after conversion. • When converting character string data to numeric data, you can specify the input format of the character string data before conversion.
67			HEX	Converts binary data to a hexadecimal string representation (character string data consisting of 0 to 9, and A to F).
68	NULL evaluation functions		COALESCE	Evaluates the specified target data items in the order in which they are specified, and then returns the first non-null value.
69			ISNULL	
70			NULLIF	Compares target data 1 to target data 2 and returns NULL if they are equal, or target data 1 if they are not equal.
71			NVL	Evaluates the specified target data items in the order in which they are specified, and then returns the first non-null value.
72	Information acquisition functions		LENGTHB	Returns the length of the target data in bytes.
73	Comparison functions		DECODE	Compares the values in the target data and the comparison data one at a time, and if there is a match, returns the corresponding return value. If no match is found between the target data and comparison data, this function returns the predefined return value.
74			GREATEST	Returns the greatest value among the specified target data items.
75			LEAST	Returns the smallest value among the specified target data items.
76			LTDECODE	Compares the values in the target data and in the comparison data one at a time, and, if any value in the target data is less than the value in the comparison data, returns the corresponding return value. If no value in the target data is

No.	Category	Name of scalar function	Description
			less than any of the values in the comparison data, this function returns the predefined return value.

8.2 Mathematical functions (trigonometric functions)

This section describes the functions and specification formats of the mathematical functions pertaining to trigonometric functions.

8.2.1 ACOS

Returns the angle (in radians) that is the inverse cosine of the target data, in the range 0 to π.

(1) Specification format

```
scalar-function-ACOS ::= ACOS(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the numeric data whose inverse cosine is to be determined.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the target data. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- Specify a value from -1 to 1 for the target data. Out-of-range values result in an error.
- You cannot specify a dynamic parameter by itself for the target data.

(3) Rules

1. The data type of the execution result is the DOUBLE PRECISION type.
2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If the target data has the null value, the execution result will be a null value.

(4) Example

Example:

Determine the inverse cosine of the values in columns C1 to C3 in table T1.

```
SELECT ACOS("C1"),ACOS("C2"),ACOS("C3")  FROM "T1"
```

Table T1

Column C1	Column C2	Column C3
INTEGER	DECIMAL(3,2)	DOUBLE PRECISION
0	-0.15	2.0000000000000001E-1

Retrieval results

1.5707963267948966E0	1.7213645995715827E0	1.3694384060045659E0

8.2.2 ASIN

Returns the angle (in radians) that is the inverse sine of the target data, in the range -π/2 to π/2.

(1) Specification format

```
scalar-function-ASIN ::= ASIN(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the numeric data whose inverse sine is to be determined.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the target data. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- Specify a value from -1 to 1 for the target data. Out-of-range values result in an error.
- You cannot specify a dynamic parameter by itself for the target data.

(3) Rules

1. The data type of the execution result is the DOUBLE PRECISION type.
2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If the target data has the null value, the execution result will be a null value.

(4) Example

Example:

Determine the inverse sine of the values in columns C1 to C3 in table T1.

```
SELECT ASIN("C1"),ASIN("C2"),ASIN("C3") FROM "T1"
```

Table T1

Column C1	Column C2	Column C3
INTEGER	DECIMAL(3,2)	DOUBLE PRECISION
0	-0.15	2.0000000000000001E-1

Retrieval results

0.0000000000000000E0	-1.5056827277668602E-1	2.0135792079033080E-1

8.2.3 ATAN

Returns the angle (in radians) that is the inverse tangent of the target data, in the range -π/2 to π/2.

(1) Specification format

```
scalar-function-ATAN ::= ATAN(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the numeric data whose inverse tangent is to be determined.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the target data. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for the target data.

(3) Rules

1. The data type of the execution result is the `DOUBLE PRECISION` type.
2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If the target data has the null value, the execution result will be a null value.

(4) Example

Example:

Determine the inverse tangent of the values in columns `C1` to `C3` in table `T1`.

```
SELECT ATAN("C1"),ATAN("C2"),ATAN("C3") FROM "T1"
```

Table `T1`

Column `C1`	Column `C2`	Column `C3`
`INTEGER`	`DECIMAL(3,2)`	`DOUBLE PRECISION`
0	-0.15	2.0000000000000001E-1

Retrieval results

0.0000000000000000E0	-1.4888994760949725E-1	1.9739555984988078E-1

8.2.4 ATAN2

Returns the angle (in radians) that is the inverse tangent of *y*/*x*, in the range -π to π.

In the specification format, *y* is set to *target-data-1*, *x* is set to *target-data-2*, and the quadrant in which the value of the execution result falls is determined by the signs of *y* and *x*.

(1) Specification format

```
scalar-function-ATAN2 ::= ATAN2(target-data-1,target-data-2)

  target-data-1 ::= value-expression
  target-data-2 ::= value-expression
```

(2) Explanation of specification format

target-data-1 and *target-data-2*:

Specifies the numeric data whose inverse tangent of *y*/*x* is to be determined.

The following rules apply:

- Specify *target-data-1* and *target-data-2* in the form of value expressions. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for *target-data-1* and *target-data-2*. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for *target-data-1* or *target-data-2*.

(3) Rules

1. The data type of the execution result is the DOUBLE PRECISION type.
2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If either *target-data-1* or *target-data-2* has a null value, the execution result will be a null value.

(4) Example

Example:

Specify the values of columns C1 and C2 from table T1 for *target-data-1* and *target-data-2*, and then determine the inverse tangent of *y*/*x*.

```
SELECT ATAN2("C1","C2") FROM "T1"
```

Table T1

Column C1	Column C2
DOUBLE PRECISION	DOUBLE PRECISION
2.9999999999999999E-1	2.0000000000000001E-1

Retrieval results

9.8279372324732905E-1

8.2.5 COS

Returns the cosine (COS trigonometric function) of the target data, which must be specified in radians.

(1) Specification format

```
scalar-function-COS ::= COS(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the numeric data whose cosine is to be determined.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the target data. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for the target data.

(3) Rules

1. The data type of the execution result is the `DOUBLE PRECISION` type.
2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If the target data has the null value, the execution result will be a null value.

(4) Example

Example:

Express the value of column `C1` in table `T1` in radians and then determine its cosine.

```
SELECT COS("C1"*PI()/180) FROM "T1"
```

Table `T1`

Column `C1` `INTEGER`
0
60
90

Retrieval results

1.0000000000000000E0
5.0000000000000011E-1
6.1232339957367660E-17

8.2.6 COSH

Returns the hyperbolic cosine of the target data.

(1) Specification format

```
scalar-function-COSH ::= COSH(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the numeric data whose hyperbolic cosine is to be determined.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the target data. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for the target data.

(3) Rules

1. The data type of the execution result is the DOUBLE PRECISION type.
2. If the execution result cannot be represented in the DOUBLE PRECISION type, an overflow error is generated.
3. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
4. If the target data has the null value, the execution result will be a null value.

(4) Example

Example:

Determine the hyperbolic cosine of the values in columns C1 to C3 in table T1.

```
SELECT COSH("C1"),COSH("C2"),COSH("C3") FROM "T1"
```

Table T1

Column C1	Column C2	Column C3
INTEGER	DECIMAL(3,2)	DOUBLE PRECISION
0	-0.15	2.0000000000000001E-1

Retrieval results

1.0000000000000000E0	1.0112711095766704E0	1.0200667556190759E0

8.2.7 DEGREES

Returns the result of converting the specified angle from radians to degrees.

(1) Specification format

```
scalar-function-DEGREES ::= DEGREES(angle)

  angle ::= value-expression
```

(2) Explanation of specification format

angle:

Specifies the angle in radians.

The following rules apply:

- Specify the angle in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the angle. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for the angle.

(3) Rules

1. The data type of the execution result is the `DOUBLE PRECISION` type.
2. If the execution result cannot be represented in the `DOUBLE PRECISION` type, an overflow error is generated.
3. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
4. If the angle has a null value, the execution result will be a null value.

(4) Examples

Example 1:

Convert π to degrees.

```
SELECT DEGREES(PI()) FROM "T1"
```

Retrieval results

1.8000000000000000E2

Example 2:

With the values of columns `C1` and `C2` from table `T1` as input, use the scalar function `ATAN2` to determine the inverse tangent of the angle (in radians), and then use the scalar function `DEGREES` to convert it to degrees.

```
SELECT DEGREES(ATAN2("C1","C2")) FROM "T1"
```

Table `T1`

Col. `C1`	Col. `C2`
DOUBLE PRECISION	DOUBLE PRECISION
1.0000000000000000E0	1.0000000000000000E0

Retrieval results

4.5000000000000000E1

8.2.8 PI

Returns the value of π .

(1) Specification format

```
scalar-function-PI ::= PI()
```

(2) Rules

1. The data type of the execution result is the `DOUBLE PRECISION` type.
2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed). In practice, however, this function always returns the value of π, never the null value.

(3) Example

Example:

Determine the circumference of the circle whose radius is the value of column `C1` in table `T1`.

```
SELECT "C1"*2*PI() FROM "T1"
```

Table T1

Column C1 INTEGER
2
10

Retrieval results

1.2566370614359172E1
6.2831853071795862E1

8.2.9 RADIANS

Returns the result of converting the specified angle from degrees to radians.

(1) Specification format

```
scalar-function-RADIANS ::= RADIANS(angle)

  angle ::= value-expression
```

(2) Explanation of specification format

angle:

Specifies the angle in degrees.

The following rules apply:

- Specify the angle in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the angle. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for the angle.

(3) Rules

1. The data type of the execution result is the DOUBLE PRECISION type.
2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If the angle has a null value, the execution result will be a null value.

(4) Examples

Example 1:

Convert the values (angles) in columns C1 through C3 from table T1 to radians.

```
SELECT RADIANS("C1"),RADIANS("C2"),RADIANS("C3") FROM "T1"
```

Table T1

Col. C1	Col. C2	Col. C3
INTEGER	DECIMAL(3,2)	DOUBLE PRECISION
0	0.15	4.5000000000000000E1

Retrieval results

0.0000000000000000E0	2.6179938779914941E-3	7.8539816339744828E-1

Example 2:

Determine the cosine of the value (angle) in column C1 from table T1.

Because the value in C1 is in degrees, the scalar function RADIANS is used to convert the angle from degrees to radians, and then the scalar function COS is used to determine the cosine.

```
SELECT COS(RADIANS("C1")) FROM "T1"
```

Table T1

Col. C1
DOUBLE PRECISION
6.0000000000000000E1

Retrieval results

5.0000000000000011E-1

In this example, the calculation is COS(60°), but the target data specified in the scalar function COS must be specified in radians.

8.2.10 SIN

Returns the sine (SIN trigonometric function) of the target data, which must be specified in radians.

(1) Specification format

```
scalar-function-SIN ::= SIN(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the numeric data whose sine is to be determined.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the target data. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for the target data.

(3) Rules

1. The data type of the execution result is the `DOUBLE PRECISION` type.
2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If the target data has the null value, the execution result will be a null value.

(4) Example

Example:

Express the value of column `C1` in table `T1` in radians and then determine its sine.

```
SELECT SIN("C1"*PI()/180) FROM "T1"
```

Table `T1`

Column `C1` `INTEGER`
0
60
90

Retrieval results

0.0000000000000000E0
8.6602540378443860E-1
1.0000000000000000E0

8.2.11 SINH

Returns the hyperbolic sine of the target data.

(1) Specification format

```
scalar-function-SINH ::= SINH(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the numeric data whose hyperbolic sine is to be determined.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the target data. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for the target data.

(3) Rules

1. The data type of the execution result is the DOUBLE PRECISION type.
2. If the execution result cannot be represented in the DOUBLE PRECISION type, an overflow error is generated.
3. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
4. If the target data has the null value, the execution result will be a null value.

(4) Example

Example:

Determine the hyperbolic sine of the values in columns C1 to C3 in table T1.

```
SELECT SINH("C1"),SINH("C2"),SINH("C3") FROM "T1"
```

Table T1

Column C1	Column C2	Column C3
INTEGER	DECIMAL(3,2)	DOUBLE PRECISION
0	-0.15	2.0000000000000001E-1

Retrieval results

0.0000000000000000E0	-1.5056313315161265E-1	2.0133600254109402E-1

8.2.12 TAN

Returns the tangent (TAN trigonometric function) of the target data, which must be specified in radians.

(1) Specification format

```
scalar-function-TAN ::= TAN(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the numeric data whose tangent is to be determined.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the target data. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for the target data.

(3) Rules

1. The data type of the execution result is the `DOUBLE PRECISION` type.
2. If the execution result cannot be represented in the `DOUBLE PRECISION` type, an overflow error is generated.
3. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
4. If the target data has the null value, the execution result will be a null value.

(4) Example

Example:

Express the value of column `C1` in table `T1` in radians and then determine its tangent.

```
SELECT TAN("C1"*PI()/180) FROM "T1"
```

Table `T1`

Column `C1` `INTEGER`
0
60
90

Retrieval results

0.0000000000000000E0
1.7320508075688767E0
1.6331239353195370E16

8.2.13 TANH

Returns the hyperbolic tangent of the target data.

(1) Specification format

```
scalar-function-TANH ::= TANH(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the numeric data whose hyperbolic tangent is to be determined.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the target data. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for the target data.

(3) Rules

1. The data type of the execution result is the `DOUBLE PRECISION` type.
2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If the target data has the null value, the execution result will be a null value.

(4) Example

Example:

Determine the hyperbolic tangent of the values in columns `C1` to `C3` in table `T1`.

```
SELECT TANH("C1"),TANH("C2"),TANH("C3") FROM "T1"
```

Table `T1`

Column `C1`	Column `C2`	Column `C3`
`INTEGER`	`DECIMAL(3,2)`	`DOUBLE PRECISION`
0	-0.15	2.0000000000000001E-1

Retrieval results

0.0000000000000000E0	-1.4888503362331798E-1	1.9737532022490401E-1

8.3 Mathematical functions (exponent and logarithm)

This section describes the functions and specification formats of the mathematical functions pertaining to exponents and logarithms.

8.3.1 EXP

Returns the result of raising the base of the natural logarithm to a power.

(1) Specification format

```
scalar-function-EXP ::= EXP(exponent)

  exponent ::= value-expression
```

(2) Explanation of specification format

exponent:

Specifies the exponent.

The following rules apply:

- Specify the exponent in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the exponent. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for the exponent.

(3) Rules

1. The data type of the execution result is the DOUBLE PRECISION type.
2. If the execution result cannot be represented in the DOUBLE PRECISION type, an overflow error is generated.
3. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
4. If the exponent has a null value, the execution result will be a null value.

(4) Example

Example:

Using the values of columns C1 to C3 in table T1 as the exponents, determine the respective powers of the base of the natural logarithm.

```
SELECT EXP("C1"),EXP("C2"),EXP("C3") FROM "T1"
```

Table T1

Column C1 INTEGER	Column C2 DECIMAL(3,2)	Column C3 DOUBLE PRECISION
0	-0.15	2.0000000000000001E-1

Retrieval results

1.0000000000000000E0	8.6070797642505781E-1	1.2214027581601699E0

8.3.2 LN

Returns the natural logarithm of the target data.

(1) Specification format

```
scalar-function-LN ::= LN(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the numeric data whose natural logarithm is to be determined.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the target data. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- Specify a positive value for the target data. Values less than or equal to 0 result in an error.
- You cannot specify a dynamic parameter by itself for the target data.

(3) Rules

1. The data type of the execution result is the DOUBLE PRECISION type.
2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If the target data has the null value, the execution result will be a null value.

(4) Example

Example:

Determine the natural logarithm of the values in columns C1 to C3 in table T1.

```
SELECT LN("C1"),LN("C2"),LN("C3") FROM "T1"
```

Table T1

Column C1	Column C2	Column C3
INTEGER	DECIMAL(3,2)	DOUBLE PRECISION
1	3.15	2.0000000000000001E-1

Retrieval results

0.0000000000000000E0	1.1474024528375417E0	-1.6094379124341003E0

8.3.3 LOG

Given a base and antilogarithm, returns its logarithm.

(1) Specification format

```
scalar-function-LOG ::= LOG(base,target-data)

  base ::= value-expression
  target-data ::= value-expression
```

(2) Explanation of specification format

base:

Specifies the base of the logarithm.

The following rules apply:

- Specify the base in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the base. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a value less than or equal to 0 for the base.
- If you specify `1` for the base, a divide-by-zero error is generated.
- You cannot specify a dynamic parameter by itself for the base.

target-data:

Specify the target data (antilogarithm).

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the target data. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a value less than or equal to `0` for the target data.
- You cannot specify a dynamic parameter by itself for the target data.

(3) Rules

1. The data type of the execution result is the `DOUBLE PRECISION` type.
2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If either the base or target data has a null value, the execution result will be a null value.

4. The execution result of `LOG(`*base*`,`*target-data*`)` is equivalent to `LN(`*target-data*`)/LN(`*base*`)`.

(4) Examples

Example 1:

Determine the common logarithm of the value of column `C1` in table `T1`.

```
SELECT LOG(10,"C1") FROM "T1"
```

Table `T1`

Column `C1` `INTEGER`
10
100
574266

Retrieval results

1.0000000000000000E0
2.0000000000000000E0
5.7591131041977697E0

Example 2:

Determine the logarithm of the value of column `C2` in table `T1`, using the value of column `C1` as the base.

```
SELECT LOG("C1","C2") FROM "T1"
```

Table `T1`

Column `C1` `INTEGER`	Column `C2` `INTEGER`
2	8
3	3
10	1056372

Retrieval results

3.0000000000000000E0
1.0000000000000000E0
6.0238168813585791E0

8.3.4 POWER

Returns the result of raising the target data to a specified power.

(1) Specification format

```
scalar-function-POWER ::= POWER(target-data,exponent)

  target-data ::= value-expression
  exponent ::= value-expression
```

(2) Explanation of specification format

target-data:

Specify the target data whose exponentiation is to be determined.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the target data. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for the target data.

exponent:

Specifies the exponent.

The following rules apply:

- Specify the exponent in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the exponent. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for the exponent.

(3) Rules

1. The data type of the execution result is the `DOUBLE PRECISION` type.
2. If the execution result cannot be represented in the `DOUBLE PRECISION` type, an overflow error is generated.
3. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
4. If either the target data or exponent has a null value, the execution result will be a null value.
5. If you specify a negative value for the target data, you must specify an integer for the exponent.
6. If you specify `0` for the target data, you must specify a positive value for the exponent, or a divide-by-zero error will be generated.

(4) Examples

Example 1:

Determine the square of the value of column `C1` in table `T1`.

```
SELECT POWER("C1",2) FROM "T1"
```

Table T1

Column C1 INTEGER
2
10
-3
5

Retrieval results

4.0000000000000000E0
1.0000000000000000E2
9.0000000000000000E0
2.5000000000000000E1

Example 2:

Determine the power of the target data, where the target data is the value of column C1 and the exponent is the value of column C2 in table T1.

```
SELECT POWER("C1","C2") FROM "T1"
```

Table T1

Column C1 INTEGER	Column C2 INTEGER
2	3
10	8
-3	20
5	-2

Retrieval results

8.0000000000000000E0
1.0000000000000000E8
3.4867844010000000E9
4.0000000000000001E-2

8.4 Mathematical functions (numerical calculations)

This section describes the functions and specification formats of the mathematical functions pertaining to numerical calculations.

8.4.1 ABS

Returns the absolute value of the target data.

(1) Specification format

```
scalar-function-ABS ::= ABS(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the numeric data whose absolute value is to be determined.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the target data. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for the target data.

(3) Rules

1. The data type and data length of the execution result will be the data type and data length of the target data.
2. If the execution results cannot be represented in the data type of the target data, an overflow error is generated.
3. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
4. If the target data has the null value, the execution result will be a null value.

(4) Example

Example:

Determine the absolute values of the values in columns C1 to C3 in table T1.

```
SELECT ABS("C1"),ABS("C2"),ABS("C3") FROM "T1"
```

Table T1

Column C1	Column C2	Column C3
INTEGER	DECIMAL(3,2)	DOUBLE PRECISION
268	7.25	1.0050000000000000E3
-475	-2.28	-3.2550000000000000E2

Retrieval results

268	7.25	1.0050000000000000E3
475	2.28	3.2550000000000000E2

8.4.2 CEIL

Returns the smallest integer that is equal to or greater than the target data.

(1) Specification format

```
scalar-function-CEIL ::= CEIL(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the numeric data to be processed.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the target data. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for the target data.

(3) Rules

1. The data type of the execution result is shown in the following table.

Table 8-2: Data type of the execution result of the scalar function CEIL

Data type of the target data	Data type of the execution result
INTEGER	INTEGER
SMALLINT	SMALLINT
DECIMAL(*p*,*s*)	DECIMAL(*p*,0)
DOUBLE PRECISION	DOUBLE PRECISION

2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If the target data has the null value, the execution result will be a null value.

(4) Example

Example:

Determine the smallest integer that is equal to or greater than the value of column C1 in table T1, and similarly for columns C2 and C3.

```
SELECT CEIL("C1"),CEIL("C2"),CEIL("C3") FROM "T1"
```

Table T1

Column C1 INTEGER	Column C2 DECIMAL(3,2)	Column C3 DOUBLE PRECISION
268	7.25	1.1500000000000000E-3
-475	-2.28	-3.2550000000000003E-2

Retrieval results

268	8.	1.0000000000000000E0
-475	-2.	0.0000000000000000E0

8.4.3 FLOOR

Returns the greatest integer that is equal to or less than the value of the target data.

(1) Specification format

```
scalar-function-FLOOR ::= FLOOR(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the numeric data to be processed.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the target data. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for the target data.

(3) Rules

1. The data type of the execution result is shown in the following table.

Table 8-3: Data type of the execution result of the scalar function FLOOR

Data type of the target data	Data type of the execution result
INTEGER	INTEGER
SMALLINT	SMALLINT

Data type of the target data	Data type of the execution result
DECIMAL(*p*,*s*)	DECIMAL(*p*,0)
DOUBLE PRECISION	DOUBLE PRECISION

2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If the target data has the null value, the execution result will be a null value.

(4) Example

Example:

Determine the greatest integer that is equal to or less than the value of column C1 in table T1, and similarly for columns C2 and C3.

```
SELECT FLOOR("C1"),FLOOR("C2"),FLOOR("C3") FROM "T1"
```

Table T1

Column C1	Column C2	Column C3
INTEGER	DECIMAL(3,2)	DOUBLE PRECISION
268	7.25	1.1500000000000000E-3
-475	-2.28	-3.2550000000000003E-2

Retrieval results

268	7.	0.0000000000000000E0
-475	-3.	-1.0000000000000000E0

8.4.4 MOD

Returns the remainder after dividing the dividend by the divisor.

(1) Specification format

```
scalar-function-MOD ::= MOD(dividend,divisor)

  dividend ::= value-expression
  divisor ::= value-expression
```

(2) Explanation of specification format

dividend:

Specifies the dividend.

The following rules apply:

- Specify the dividend in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the dividend. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for the dividend.

divisor:

Specifies the divisor.

The following rules apply:

- Specify the divisor in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the divisor. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify `0` for the divisor. If you specify `0`, a divide-by-zero error is generated.
- You cannot specify a dynamic parameter by itself for the divisor.

(3) Rules

1. The data type of the execution result is determined by the data types of the dividend and the divisor, as shown in the following table.

Table 8-4: Data type of the execution result of the scalar function MOD

Data type of the dividend	Data type of the divisor	Data type of the execution result
INTEGER	INTEGER	INTEGER
	SMALLINT	SMALLINT
	DECIMAL	DECIMAL
	DOUBLE PRECISION	DOUBLE PRECISION
SMALLINT	INTEGER	INTEGER
	SMALLINT	SMALLINT
	DECIMAL	DECIMAL
	DOUBLE PRECISION	DOUBLE PRECISION
DECIMAL(*p*,0)	INTEGER	INTEGER
	SMALLINT	SMALLINT
	DECIMAL	DECIMAL
	DOUBLE PRECISION	DOUBLE PRECISION
DECIMAL(*p*,*s*) when $s \geq 1$	INTEGER	DECIMAL
	SMALLINT	
	DECIMAL	
	DOUBLE PRECISION	DOUBLE PRECISION
DOUBLE PRECISION	INTEGER	DOUBLE PRECISION
	SMALLINT	
	DECIMAL	
	DOUBLE PRECISION	

Note

If the data type of the execution result is `DECIMAL`, the precision and scaling are determined as follows:

Precision (p) = MIN(py-sy+s, 38)

Scaling (*s*) = MAX(*sx*, *sy*)

When calculating the precision and scaling of the execution result when MOD(*x*,*y*) is specified, let DECIMAL(*px*,*sx*) be the data type of *x* and DECIMAL(*py*,*sy*) be the data type of *y*.

If the data type of *x* or *y* is SMALLINT, use DECIMAL(10,0) for the calculation, and if it is INTEGER, use DECIMAL(20,0).

2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If the dividend or the divisor has a null value, the execution result will be a null value.
4. The execution result will have the same sign as the dividend.
5. If you specify DOUBLE PRECISION type data for the scalar function MOD, beware of calculation errors associated with the DOUBLE PRECISION data type. For example, the execution result of the following is not zero:

```
MOD(5.0E-1,1.0E-1) → 9.999999999999978E-2
```

This is due to the fact that 0.1 does not have a finite binary representation (0.1 and 1.0E-1 are not exactly equal). If you require an exact value for the execution result of the remainder, use the DECIMAL type.

(4) Examples

Example 1:

Determine the remainder after dividing the values of column C1 in table T1 by 3.

```
SELECT MOD("C1",3) FROM "T1"
```

Table T1

Column C1 INTEGER
10
15
-7
-24

Retrieval results

1
0
-1
0

Example 2:

Determine the remainder after dividing the values of column C1 in table T1 by the values of column C2.

```
SELECT MOD("C1","C2") FROM "T1"
```

Table `T1`

Column `C1` `INTEGER`	Column `C2` `DECIMAL(2,1)`
10	4.0
15	5.0
-7	2.3
-24	-5.1

Retrieval results

2.0
0.0
-0.1
-3.6

8.4.5 RANDOM

Returns pseudorandom numbers that follow a uniform distribution and are greater than or equal to the value specified for the minimum value and less than the value specified for the maximum value.

There are some scalar functions, including `RANDOM`, that return pseudorandom numbers. Check the differences in the specifications among those scalar functions that return pseudorandom numbers, and then use the scalar function that is most suitable for your purpose. For details about the differences in the specifications among the scalar functions that return pseudorandom numbers, see (6) List of scalar functions that return pseudorandom numbers.

(1) Specification format

```
scalar-function-RANDOM ::= RANDOM([minimum-value,maximum-value])

  minimum-value ::= value-expression
  maximum-value ::= value-expression
```

(2) Explanation of specification format

minimum-value:

Specifies a minimum value in the range for generating a random number. (The minimum value is included in the range.) If this argument is omitted, *minimum-value* is assumed to be `0`.

The following rules apply:

- Specify *minimum-value* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for *minimum-value*. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- If you specify only a single dynamic parameter for *minimum-value*, the `DOUBLE PRECISION` type is assumed.

maximum-value:

Specifies a maximum value in the range for generating a random number. (The maximum value is not included in the range.) If this argument is omitted, *maximum-value* is assumed to be `1`:

The following rules apply:

- Specify *maximum-value* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for *maximum-value*. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- If you specify only a single dynamic parameter for *maximum-value*, the `DOUBLE PRECISION` type is assumed.

(3) Rules

1. The data type of the execution result is the `DOUBLE PRECISION` type.
2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If *minimum-value* or *maximum-value* is a null value, the execution result will also be a null value.
4. After converting both *minimum-value* and *maximum-value* to the `DOUBLE PRECISION` type, calculate the execution result.
5. If the relationship between value *A* specified for *minimum-value* and value *B* specified for *maximum-value* satisfies "*A* > *B*", *minimum-value* and *maximum-value* are automatically switched. Then, pseudorandom numbers that follow a uniform distribution in a range of values "greater than or equal to *B* and less than *A*" are returned.
6. If you specify the same value for *minimum-value* and *maximum-value*, the execution result is the value specified for *minimum-value*.
7. If the execution result cannot be expressed as the data type specified for the execution result, an overflow error occurs.
8. If you specify `0` for *maximum-value*, `+0` might be returned as the execution result.

(4) Notes

This scalar function is not suitable for use in encryption.

(5) Example

Example

For table `T1`, determine `DOUBLE PRECISION`-type values that follow a uniform distribution, in the range of 1 or more and less than 10.

Note that every time you execute the `SELECT` statement, the values of the execution results change.

```
SELECT RANDOM(1,10) FROM "T1"
```

Table `T1`

Column `C1`
1
2
3
4
5

Search result example

3.4152565556163950E0
5.1485283237583523E0
2.4197853386516375E0
8.0146660601528428E0
6.8577888277555266E0

(6) List of scalar functions that return pseudorandom numbers

In addition to `RANDOM`, the following three scalar functions return pseudorandom numbers:

- `RANDOMCURSOR`
- `RANDOMROW`
- `RANDOM_NORMAL`

The following table describes the differences in the specifications among the preceding scalar functions. Use the scalar function that is most suitable for your purpose.

Table 8-5: List of scalar functions that return pseudorandom numbers

No.	Item	Scalar function RANDOM	Scalar function RANDOMCURSOR	Scalar function RANDOMROW	Scalar function RANDOM_NORMAL
1	Format	`RANDOM`([*minimum-value*,*maximum-value*]) *minimum-value*::=*value-expression* *maximum-value*::=*value-expression*	`RANDOMCURSOR`(*identification-number*[,*minimum-value*,*maximum-value*]) *identification-number*::=*unsigned-integer-literal* *minimum-value*::=*value-specification* *maximum-value*::=*value-specification*	`RANDOMROW`(*identification-number*[,*minimum-value*,*maximum-value*]) *identification-number*::=*unsigned-integer-literal* *minimum-value*::=*value-expression* *maximum-value*::=*value-expression*	`RANDOM_NORMAL`([*average-μ*,*standard-deviation-σ*]) *average-μ*::=*value-expression* *standard-deviation-σ*::=*value-expression*
2	Distribution of pseudorandom numbers	Uniform distribution	Uniform distribution	Uniform distribution	Normal distribution
3	Range of pseudorandom numbers	Value greater than or equal to the specified minimum value and less than the specified maximum value	Value greater than or equal to the specified minimum value and less than the specified maximum value	Value greater than or equal to the specified minimum value and less than the specified maximum value	Value that follows average μ and standard deviation σ

No.	Item	Scalar function RANDOM	Scalar function RANDOMCURSOR	Scalar function RANDOMROW	Scalar function RANDOM_NORMAL
4	Do scalar functions having the same identification number in an SQL statement always return the same value?	--	Y	N	--
5	Do scalar functions having the same identification number in a query specification return the same value for each row?	--	Y	Y	--
6	Possible specification location	*value-expression*	• *value-expression* in a selection expression • ORDER BY clause#	• *value-expression* in a selection expression • ORDER BY clause#	*value-expression*

Legend:

Y: True

N: False

--: Not applicable. No identification number can be specified.

#

The function cannot be specified for the ORDER BY clause in a WITHIN group specification or a window order clause.

8.4.6 RANDOMCURSOR

RANDOMCURSOR returns values in accordance with the following rules:

- This function returns pseudorandom numbers that follow a uniform distribution and are greater than or equal to the value specified for the minimum value and less than the value specified for the maximum value.
- In a retrieval SQL statement, this function always returns the same value while the cursor is open. In an update SQL statement, this function always returns the same value while the SQL statement is being run.
- If an SQL statement contains multiple RANDOMCURSOR functions for which the same identification number is specified, those functions always return the same values.

There are some scalar functions, including RANDOMCURSOR, that return pseudorandom numbers. Check the differences in the specifications among those scalar functions that return pseudorandom numbers, and then use the scalar function that is most suitable for your purpose. For details about the differences in the specifications among the scalar functions that return pseudorandom numbers, see (6) List of scalar functions that return pseudorandom numbers in 8.4.5 RANDOM.

(1) Specification format

```
scalar-function-RANDOMCURSOR::=RANDOMCURSOR(identification-number[,minimum-value,maxi
mum-value])
```

```
identification-number::=unsigned-integer-literal
minimum-value::=value-specification
maximum-value::=value-specification
```

(2) Explanation of specification format

identification-number:

Specifies an integer in the range from `1` to `1000`. If an SQL statement contains multiple `RANDOMCURSOR` functions for which the same identification number is specified, those functions always return the same values.

minimum-value:

Specifies a minimum value in the range for generating a random number. (The minimum value is included in the range.) If both *minimum-value* and *maximum-value* are omitted, *minimum-value* is assumed to be `0`.

The following rules apply:

- Specify *minimum-value* in the form of a value specification. For details about value specifications, see 7.21 Value specification.
- Specify numeric data for *minimum-value*. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- If you specify a dynamic parameter for *minimum-value*, the `DOUBLE PRECISION` type is assumed.

maximum-value:

Specifies a maximum value in the range for generating a random number. (The maximum value is not included in the range.) If both *minimum-value* and *maximum-value* are omitted, *maximum-value* is assumed to be `1`.

The following rules apply:

- Specify *maximum-value* in the form of a value specification. For details about value specifications, see 7.21 Value specification.
- Specify numeric data for *maximum-value*. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- If you specify a dynamic parameter for *maximum-value*, the `DOUBLE PRECISION` type is assumed.

(3) Rules

1. This scalar function can be specified in the following locations:
   - Selection expression in a query specification
   - `ORDER BY` clause (except the `ORDER BY` clause in a `WITHIN` group specification or a window order clause)
2. If you specify multiple instances of `RANDOMCURSOR` with the same identification number in one SQL statement, comply with either of the following rules:
   - Specify the minimum value and maximum value in only one instance of `RANDOMCURSOR`, and omit them in all other instances of `RANDOMCURSOR`.

     Example of a correct SQL statement:

     ```
     SELECT "C1"+RANDOMCURSOR(1,10,20),"C2"+RANDOMCURSOR(1) FROM "T1"
     UNION ALL
     SELECT "C3"+RANDOMCURSOR(1),"C4"+RANDOMCURSOR(1) FROM "T2"
     ```

     Example of an SQL statement that generates an error:

     ```
     SELECT "C1"+RANDOMCURSOR(1,10,20),"C2"+RANDOMCURSOR(1) FROM "T1"
     UNION ALL
     SELECT "C3"+RANDOMCURSOR(1,10,20),"C4"+RANDOMCURSOR(1) FROM "T2"
     ```

- Omit the minimum value and maximum value in all instances of RANDOMCURSOR.

  Example of a correct SQL statement:

  ```
  SELECT "C1"+RANDOMCURSOR(1),"C2"+RANDOMCURSOR(1) FROM "T1"
  UNION ALL
  SELECT "C3"+RANDOMCURSOR(1),"C4"+RANDOMCURSOR(1) FROM "T2"
  ```

3. If an SQL statement contains multiple RANDOMCURSOR functions for which the same identification number is specified, those functions always return the same values.

   Example:

   ```
   SELECT
     "C1"+ RANDOMCURSOR(1,10,20),     ...[a]
     "C2"+ RANDOMCURSOR(1),           ...[a]
     "C3"+ RANDOMCURSOR(2),           ...[b]
     "C4"+ RANDOMCURSOR(2)            ...[b]
   FROM "T1"
   UNION ALL
   SELECT
     "C1"+ RANDOMCURSOR(1),           ...[a]
     "C2"+ RANDOMCURSOR(1),           ...[a]
     "C3"+ RANDOMCURSOR(2,20,30),     ...[b]
     "C4"+ RANDOMCURSOR(2)            ...[b]
   FROM "T2"
   ```

   Explanation:

   a. These are instances of RANDOMCURSOR for which 1 is specified as the identification number. Each instance always returns the same value (a value greater than or equal to 10 and less than 20).

   b. These are instances of RANDOMCURSOR for which 2 is specified as the identification number. Each instance always returns the same value (a value greater than or equal to 20 and less than 30).

4. If the following identification numbers are the same, HADB automatically re-assigns them. Therefore, the SQL statement does not result in an error.

   - Identification number specified when a viewed table is defined (by using the CREATE VIEW statement)
   - Identification number specified in an SQL statement in which the viewed table is specified

   Example:

   Definition of viewed table V1:

   ```
   CREATE VIEW "V1"("VC1","VC2") AS
     SELECT "C1"+RANDOMCURSOR(1,10,20),"C2"+RANDOMCURSOR(1) FROM "T1"
   ```

   SQL statement for searching viewed table V1:

   ```
   SELECT "VC1"+RANDOMCURSOR(1,10,20),"VC2"+RANDOMCURSOR(1) FROM "V1"
   ```

   For the preceding SELECT statements, HADB performs equivalent exchange as follows:

   ```
   SELECT "VC1"+RANDOMCURSOR(1,10,20),"VC2"+RANDOMCURSOR(1)
     FROM (SELECT "C1"+RANDOMCURSOR(2,10,20),"C2"+RANDOMCURSOR(2)
             FROM "T1") "V1"("VC1","VC2")
   ```

   The underlined identification number, 2, is the one that HADB automatically changed from 1. Therefore, this SELECT statement does not result in an error. As shown in the preceding example, HADB automatically changes the identification number that was specified when the viewed table was defined.

5. In one SQL statement, a maximum of 1,000 entities of RANDOMCURSOR identification numbers can be specified. If a viewed table is specified in an SQL statement, the total number of entities of RANDOMCURSOR identification numbers appearing in the SQL statement and the relevant CREATE VIEW statement must not exceed 1,000.
6. The same identification number can be specified for RANDOMCURSOR and RANDOMROW. In this case, each scalar function separately generates and returns a pseudorandom number.

   Example:

```
SELECT RANDOMCURSOR(1,10,20) AS "C1",
       RANDOMROW(1,10,20) AS "C2"
    FROM "T1"
```

   Example of execution results

C1	C2
+1.2475764960039722E+01	+1.7828308131439851E+01
+1.2475764960039722E+01	+1.5309877916946510E+01
+1.2475764960039722E+01	+1.3148733592755859E+01

7. RANDOMCURSOR generates a pseudorandom number when the cursor opens. Therefore, the result changes each time the cursor opens.
8. The data type of the execution result is the DOUBLE PRECISION type.
9. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
10. If *minimum-value* or *maximum-value* is a null value, the execution result will also be a null value.
11. Both *minimum-value* and *maximum-value* are converted to the DOUBLE PRECISION type, and then the execution result is calculated.
12. If the relationship between value *A* specified for *minimum-value* and value *B* specified for *maximum-value* satisfies "*A* > *B*", *minimum-value* and *maximum-value* are automatically switched. Then, pseudorandom numbers that follow a uniform distribution in a range of values "greater than or equal to *B* and less than *A*" are returned.
13. If you specify the same value for *minimum-value* and *maximum-value*, the execution result is the value specified for *minimum-value*.
14. If the execution result cannot be expressed as the data type specified for the execution result, an overflow error occurs.
15. If you specify 0 for *maximum-value*, +0 might be returned as the execution result.

(4) Notes

This scalar function is not suitable for use in encryption.

(5) Example

Example:

Modify the admission date and discharge date so that the following conditions are met:

- The admission and discharge dates are modified so that the hospitalization period does not change. The same number of days is added to both the current admission and discharge dates to produce the new admission and discharge dates.
- A maximum of 6 days is added equally to both the current admission and discharge dates to produce new admission and discharge dates.
- The retrieval results are sorted by the new admission date.

Example:

```
SELECT "PATIENT-ID","ADMISSION-DATE","DISCHARGE-DATE",
       "ADMISSION-DATE"+CAST(RANDOMCURSOR(1,0,7) AS INTEGER) DAY AS "NEW-ADMISSION
-DATE",
       "DISCHARGE-DATE"+CAST(RANDOMCURSOR(1) AS INTEGER) DAY AS "NEW-DISCHARGE-DAT
E"
    FROM "HOSPITALITY-HISTORY"
      ORDER BY "ADMISSION-DATE"+CAST(RANDOMCURSOR(1) AS INTEGER) DAY
```

Example of execution results

PATIENT-ID	ADMISSION-DATE	DISCHARGE-DATE	NEW-ADMISSION-DATE	NEW-DISCHARGE-DATE
U0003	2018-04-01	2018-04-11	2018-04-04	2018-04-14
U0004	2018-05-01	2018-05-11	2018-05-04	2018-05-14
U0001	2018-06-01	2018-06-11	2018-06-04	2018-06-14
U0002	2018-07-01	2018-07-11	2018-07-04	2018-07-14

8.4.7 RANDOMROW

RANDOMROW returns values in accordance with the following rules:

- This function returns pseudorandom numbers that follow a uniform distribution and are greater than or equal to the value specified for the minimum value and less than the value specified for the maximum value.
- If a query specification contains multiple RANDOMROW functions for which the same identification number is specified, those functions return the same values for each result row of the query specification.

There are some scalar functions, including RANDOMROW, that return pseudorandom numbers. Check the differences in the specifications among those scalar functions that return pseudorandom numbers, and then use the scalar function that is most suitable for your purpose. For details about the differences in the specifications among the scalar functions that return pseudorandom numbers, see (6) List of scalar functions that return pseudorandom numbers in 8.4.5 RANDOM.

(1) Specification format

```
scalar-function-RANDOMROW::=RANDOMROW(identification-number[,minimum-value,maximum-va
lue])

  identification-number::=unsigned-integer-literal
  minimum-value::=value-expression
  maximum-value::=value-expression
```

(2) Explanation of specification format

identification-number:

Specifies an integer in the range from 1 to 1000. If a query specification contains multiple RANDOMROW functions for which the same identification number is specified, those functions return the same values for each result row of the query specification.

minimum-value:

Specifies a minimum value in the range for generating a random number. (The minimum value is included in the range.) If both *minimum-value* and *maximum-value* are omitted, *minimum-value* is assumed to be 0.

The following rules apply:

- Specify *minimum-value* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for *minimum-value*. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- If you specify only a single dynamic parameter for *minimum-value*, the `DOUBLE PRECISION` type is assumed.

maximum-value:

Specifies a maximum value in the range for generating a random number. (The maximum value is not included in the range.) If both *minimum-value* and *maximum-value* are omitted, *maximum-value* is assumed to be `1`.

The following rules apply:

- Specify *maximum-value* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for *maximum-value*. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- If you specify only a single dynamic parameter for *maximum-value*, the `DOUBLE PRECISION` type is assumed.

(3) Rules

1. This scalar function can be specified in the following locations:
   - Selection expression in a query specification
   - `ORDER BY` clause (except the `ORDER BY` clause in a `WITHIN` group specification or a window order clause)
2. `RANDOMROW` cannot be specified in a value expression in `RANDOMROW`.

   However, `RANDOMROW` can be specified in a subquery specified in `RANDOMROW`.

   Example of an SQL statement that generates an error:

   ```
   RANDOMROW(1,0,RANDOMROW(1))
   ```

   Example of a correct SQL statement:

   ```
   RANDOMROW(1,0,(SELECT RANDOMROW(1) FROM "T1"))
   ```
3. If you specify multiple `RANDOMROW` functions with the same identification number in one query specification, comply with either of the following rules:
   - Specify the minimum value and maximum value in only one `RANDOMROW` function, and omit them in all other `RANDOMROW` functions.

     Example of a correct SQL statement:

     ```
     SELECT "C1"+RANDOMROW(1,10,20),"C2"+RANDOMROW(1),"C3"+RANDOMROW(1) FROM "T1"
     ```

     Example of an SQL statement that generates an error:

     ```
     SELECT "C1"+RANDOMROW(1,10,20),"C2"+RANDOMROW(1,10,20),"C3"+RANDOMROW(1) FROM "T
     1"
     ```
   - Omit the minimum value and maximum value in all `RANDOMROW` functions.

     Example of a correct SQL statement:

     ```
     SELECT "C1"+RANDOMROW(1),"C2"+RANDOMROW(1),"C3"+RANDOMROW(1) FROM "T1"
     ```
4. `RANDOMROW` generates a pseudorandom number for each result row of the query specification. Therefore, the result changes for each result row of the query specification.

5. All RANDOMROW functions for which the same identification number is specified return the same values for each result row of the query specification.

   Example:

```
SELECT
  "C1"+ RANDOMROW(1,10,20),     ...[a]
  "C2"+ RANDOMROW(1),           ...[a]
  "C3"+ RANDOMROW(2,20,30),     ...[b]
  "C4"+ RANDOMROW(2)            ...[b]
FROM "T1"
UNION ALL
SELECT
  "C1"+ RANDOMROW(1,10,20),     ...[c]
  "C2"+ RANDOMROW(1),           ...[c]
  "C3"+ RANDOMROW(2),           ...[d]
  "C4"+ RANDOMROW(2)            ...[d]
FROM "T2"
```

   Explanation:

   a. These are RANDOMROW functions for which 1 is specified as the identification number. Each function returns the same value (a value greater than or equal to 10 and less than 20) for each result row of the query specification.

   b. These are RANDOMROW functions for which 2 is specified as the identification number. Each function returns the same value (a value greater than or equal to 20 and less than 30) for each result row of the query specification.

   c. These are RANDOMROW functions for which 1 is specified as the identification number. Each function returns the same value (a value greater than or equal to 10 and less than 20) for each result row of the query specification.

   d. These are RANDOMROW functions for which 2 is specified as the identification number. Each function returns the same value (a value greater than or equal to 0 and less than 1) for each result row of the query specification.

6. When the internal derived table shown later is expanded, HADB automatically re-assigns the identification number of the RANDOMROW in the derived query for the internal derived table. This prevents that identification number from coinciding with the identification number of the RANDOMROW in the query specification in which the internal derived table is specified.

   - Internal derived table for which RANDOMROW is specified in the derived query

   Example:

   SQL statement in which a derived table is specified:

```
SELECT "DC1"+RANDOMROW(1,10,20),"DC2"+RANDOMROW(1)
    FROM (SELECT "C1"+RANDOMROW(1,20,30),"C2"+RANDOMROW(1)
              FROM "T1") "DT"("DC1","DC2")
```

   SQL statement in which a derived table is expanded:

```
SELECT "C1"+RANDOMROW(2,20,30)+RANDOMROW(1,10,20),
       "C2"+RANDOMROW(2)+RANDOMROW(1)
    FROM "T1"
```

   The identification numbers of the RANDOMROW functions in the derived table DT are re-assigned as follows:

   - RANDOMROW(1,20,30) → RANDOMROW(2,20,30)
   - RANDOMROW(1) → RANDOMROW(2)

7. Multiple RANDOMROW functions with the same identification number can be specified in different query specifications. Note, however, that the identification number of each RANDOMROW function is treated as a different entity.

8. The maximum number of entities of identification numbers that can be specified in one SQL statement is 1,000. However, if a viewed table is specified in an SQL statement, equivalent exchange is performed to convert the viewed table into a derived table, and then the number of entities of identification numbers is checked.

9. The same identification number can be specified for `RANDOMCURSOR` and `RANDOMROW`. In this case, each scalar function separately generates and returns a pseudorandom number.

   Example:

   ```
   SELECT RANDOMCURSOR(1,10,20) AS "C1",
          RANDOMROW(1,10,20) AS "C2"
       FROM "T1"
   ```

   Example of execution results

C1	C2
+1.2475764960039722E+01	+1.7828308131439851E+01
+1.2475764960039722E+01	+1.5309877916946510E+01
+1.2475764960039722E+01	+1.3148733592755859E+01

10. The data type of the execution result is the `DOUBLE PRECISION` type.

11. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).

12. If *minimum-value* or *maximum-value* is a null value, the execution result will also be a null value.

13. Both *minimum-value* and *maximum-value* are converted to the `DOUBLE PRECISION` type, and then the execution result is calculated.

14. If the relationship between value *A* specified for *minimum-value* and value *B* specified for *maximum-value* satisfies "*A* > *B*", *minimum-value* and *maximum-value* are automatically switched. Then, pseudorandom numbers that follow a uniform distribution in a range of values "greater than or equal to *B* and less than *A*" are returned.

15. If you specify the same value for *minimum-value* and *maximum-value*, the execution result is the value specified for *minimum-value*.

16. If the execution result cannot be expressed as the data type specified for the execution result, an overflow error occurs.

17. If you specify `0` for *maximum-value*, `+0` might be returned as the execution result.

(4) Notes

This scalar function is not suitable for use in encryption.

(5) Example

Example

Modify the admission date and discharge date so that the following conditions are met:

- The admission and discharge dates are modified so that the hospitalization period does not change. The same number of days is added to both the current admission and discharge dates to produce the new admission and discharge dates.
- A maximum of 6 days is added equally to both the current admission and discharge dates to produce new admission and discharge dates. The number of days to be added differs for each patient.
- The retrieval results are sorted by the new admission date.

Example:

```
SELECT "PATIENT-ID","ADMISSION-DATE","DISCHARGE-DATE",
       "ADMISSION-DATE"+CAST(RANDOMROW(1,0,7) AS INTEGER) DAY AS "NEW-ADMISSION-DA
TE",
       "DISCHARGE-DATE"+CAST(RANDOMROW(1) AS INTEGER) DAY AS "NEW-DISCHARGE-DATE"
    FROM "HOSPITALITY-HISTORY"
      ORDER BY "ADMISSION-DATE"+CAST(RANDOMROW(1) AS INTEGER) DAY
```

PATIENT-ID	ADMISSION-DATE	DISCHARGE-DATE	NEW-ADMISSION-DATE	NEW-DISCHARGE-DATE
U0003	2018-04-01	2018-04-11	2018-04-04	2018-04-14
U0004	2018-05-01	2018-05-11	2018-05-06	2018-05-16
U0001	2018-06-01	2018-06-11	2018-06-07	2018-06-17
U0002	2018-07-01	2018-07-11	2018-07-03	2018-07-13

8.4.8 RANDOM_NORMAL

Returns pseudorandom numbers that follow a normal distribution with an average μ, and a standard deviation σ.

There are some scalar functions, including `RANDOM_NORMAL`, that return pseudorandom numbers. Check the differences in the specifications among those scalar functions that return pseudorandom numbers, and then use the scalar function that is most suitable for your purpose. For details about the differences in the specifications among the scalar functions that return pseudorandom numbers, see (6) List of scalar functions that return pseudorandom numbers in `8.4.5 RANDOM`.

(1) Specification format

```
scalar-function-RANDOM_NORMAL ::= RANDOM_NORMAL([average-μ,standard-deviation-σ])

  average-μ ::= value-expression
  standard-deviation-σ ::= value-expression
```

(2) Explanation of specification format

average-μ:

Specifies an average, μ. If this argument is omitted, *average-μ* is assumed to be `0`.

The following rules apply:

- Specify *average-μ* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for *average-μ*. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- If you specify only a single dynamic parameter for *average-μ*, the `DOUBLE PRECISION` type is assumed.

standard-deviation-σ:

Specifies a standard deviation, σ. If the argument is omitted, *standard-deviation-σ* is assumed to be `1` (standard normal distribution).

The following rules apply:

- Specify *standard-deviation-σ* in the form of a value expression. For details about value expressions, see 7.20 Value expression.

- Specify numeric data for *standard-deviation-σ*. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- Specify a value greater than or equal to `0` for *standard-deviation-σ*.
- If you specify only a single dynamic parameter for *standard-deviation-σ*, the `DOUBLE PRECISION` type is assumed.

(3) Rules

1. The data type of the execution result is the `DOUBLE PRECISION` type.
2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If *average-μ* or *standard-deviation-σ* is a null value, the execution result will also be a null value.
4. After converting both *average-μ* and *standard-deviation-σ* to the `DOUBLE PRECISION` type, calculate the execution result.
5. If the execution result can no longer be expressed as the data type of the execution result, an overflow error occurs.

(4) Notes

This scalar function is not suitable for use in encryption.

(5) Example

Example

For table `T1`, determine the `DOUBLE PRECISION`-type values that follow a normal distribution whose *average-μ* is 30 and whose *standard-deviation-σ* is 20.

Note that every time you execute the `SELECT` statement, the values of the execution results change.

```
SELECT RANDOM_NORMAL(30,20) FROM "T1"
```

Table `T1`

Column `C1`
1
2
3
4
5

Search result example

-1.2987575425050977E1
5.1920095220623608E1
2.3649975133459982E1
5.7409734912048918E1
3.2063684046363342E1

8.4.9 ROUND

Returns the value of the target data rounded to the *n*th digit after the decimal point.

For the scalar function ROUND that is used to round datetime data, see 8.9.7 ROUND.

(1) Specification format

```
scalar-function-ROUND ::= ROUND(target-data[,num-digits])

  target-data ::= value-expression
  num-digits ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies numeric data (the value to be rounded to the *n*th digit after the decimal point).

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the target data. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for the target data.

num-digits:

Specifies the number of digits.

The following rules apply:

- Specify *num-digits* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify data of type INTEGER or SMALLINT for *num-digits*.
- If you omit *num-digits*, zero is assumed.
- If you specify a dynamic parameter for *num-digits*, the dynamic parameter will be assumed to be INTEGER type.

(3) Rules

1. The data type of the execution result is shown in the following table.

Table 8-6: Data type of the execution result of the scalar function ROUND

Data type of the target data		Data type of the execution result
INTEGER		INTEGER
SMALLINT		SMALLINT
DECIMAL(*p*,*s*)	when $p \leq 37$	DECIMAL($p + 1$,*s*)
	when $p = 38$	DECIMAL(38,*s*)
DOUBLE PRECISION		DOUBLE PRECISION

2. If the execution results cannot be represented in the data type of the target data, an overflow error is generated.
3. If the data type of the target data is SMALLINT, INTEGER, or DECIMAL, the function returns a value whose fractional part is rounded to $n + 1$ digits.
   Example: ROUND(325.72,1) → 325.70

In the case of negative values, the target data is rounded as follows:

Example 1: `ROUND(-2.3,0)` → `-2.0`

Example 2: `ROUND(-2.7,0)` → `-3.0`

4. If the data type of the target data is `DOUBLE PRECISION`, midpoint values at position *n* + 1 are rounded to the nearest even number.
5. If you specify a negative value for *num-digits*, it rounds the integer part at the specified decimal place.

   Example: `ROUND(325.72,-1)` → `330.00`
6. If you omit *num-digits*, or specify `0`, all decimal places are rounded, and the execution result will be rounded to the integer part (to the ones position).

   Example: `ROUND(325.72,0)` → `326.00`
7. If you specify a number of digits that is outside the range of the target data, it is handled as follows:
   - If you specified a positive value for *num-digits*, no rounding is performed. The value of the original target data is returned unchanged.

     Example 1: `ROUND(0.12,5)` → `0.12`

     Example 2: `ROUND(58,1)` → `58`
   - If you specified a negative value for *num-digits*, it returns `0`.

     Example: `ROUND(58,-5)` → `0`
8. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
9. If *target-data* or *num-digits* has a null value, the execution result will be a null value.

(4) Example

Example:

Round the values of columns `C1` to `C3` in table `T1` as follows.

- Column `C1`: Round to the first decimal place (by rounding off the second decimal place).
- Column `C2`: Round to the hundreds column of the integer part (by rounding off the tens column).
- Column `C3`: Round to the first decimal place, rounding midpoint values to the nearest even number.

This assumes that HADB is running in an environment in which the rounding mode is round-to-even.

```
SELECT ROUND("C1",1),ROUND("C2",-2),ROUND("C3",0) FROM "T1"
```

Table T1

Column C1 DECIMAL(4,3)	Column C2 INTEGER	Column C3 DOUBLE PRECISION
3.123	128	1.1400000000000000E1
-4.089	1051	1.1500000000000000E1
5.000	-565	1.1600000000000000E1
3.123	-1117	1.2400000000000000E1
-4.089	7	1.2500000000000000E1
5.001	-5	1.2600000000000000E1

Retrieval results

3.100	100	1.1000000000000000E1
-4.100	1100	1.2000000000000000E1
5.000	-600	1.2000000000000000E1
3.100	-1100	1.2000000000000000E1
-4.100	0	1.2000000000000000E1
5.000	0	1.3000000000000000E1

8.4.10 SIGN

Returns the sign of the target data (`+1` for positive, `-1` for negative, `0` for zero).

(1) Specification format

```
scalar-function-SIGN ::= SIGN(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the numeric data to be processed.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the target data. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for the target data.

(3) Rules

1. The data type of the execution result is determined based on the data type of the target data, as shown in the following table:

Table 8-7: Data type of the execution result of the scalar function SIGN

No.	Data type of the target data	Data type of the execution result
1	INTEGER	INTEGER

No.	Data type of the target data	Data type of the execution result
2	SMALLINT	SMALLINT
3	DECIMAL(*p*,*s*)	DECIMAL(1,0)
4	DOUBLE PRECISION	DOUBLE PRECISION

2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If the target data has the null value, the execution result will be a null value.

(4) Example

Example:

Determine whether the values of columns C1 to C3 in table T1 are positive, negative, or zero.

```
SELECT SIGN("C1"),SIGN("C2"),SIGN("C3") FROM "T1"
```

Table T1

Column C1	Column C2	Column C3
INTEGER	DECIMAL(3,2)	DOUBLE PRECISION
268	7.25	1.1500000000000000E-3
-475	-2.28	-3.2550000000000003E-2
0	0.00	0.0000000000000000E0

Retrieval results

1	1.	1.0000000000000000E0
-1	-1.	-1.0000000000000000E0
0	0.	0.0000000000000000E0

8.4.11 SQRT

Returns the square root of the target data.

(1) Specification format

```
scalar-function-SQRT ::= SQRT(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the numeric data whose square root is to be determined.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the target data. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- Specify a value greater than or equal to 0 for the target data. Negative values cannot be specified.

- You cannot specify a dynamic parameter by itself for the target data.

(3) Rules

1. The data type of the execution result is the DOUBLE PRECISION type.
2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If the target data has the null value, the execution result will be a null value.

(4) Example

Example:

Determine the square root of the values in columns C1 to C3 in table T1.

```
SELECT SQRT("C1"),SQRT("C2"),SQRT("C3") FROM "T1"
```

Table T1

Column C1	Column C2	Column C3
INTEGER	DECIMAL(3,2)	DOUBLE PRECISION
9	5.15	2.1200000000000000E8

Retrieval results

3.0000000000000000E0	2.2693611435820435E0	1.4560219778561037E4

8.4.12 TRUNC

Returns a value that has been truncated to the specified number of decimal places.

For the scalar function TRUNC that is used to truncate datetime data, see 8.9.8 TRUNC.

(1) Specification format

```
scalar-function-TRUNC ::= TRUNC(target-data[,num-digits])

  target-data ::= value-expression
  num-digits ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the numeric data to be processed.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify numeric data for the target data. For details about numeric data, see (1) Numeric data in 6.2.1 List of data types.
- You cannot specify a dynamic parameter by itself for the target data.

num-digits:

Specifies the number of digits.

The following rules apply:

- Specify *num-digits* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify INTEGER type or SMALLINT type data for *num-digits*.

If the value specified for *num-digits* is positive (*n*), it leaves *n* decimal places of the target data and truncates the decimal places at position *n* + 1 and beyond. If the value specified for *num-digits* is negative (-*n*), it truncates *n* digits from the integer portion of the target data.

If *num-digits* is omitted, the target data is truncated to 0 decimal places.

The following example illustrates the result of executing the scalar function TRUNC.

Example

For numeric data 123.456, truncate everything past the second decimal place.

TRUNC(123.456,2) → 123.450

(3) Rules

1. If you specify a dynamic parameter for *num-digits*, the assumed data type of the dynamic parameter is INTEGER.
2. The length of the data type of the execution result will be the length of the data type of the argument *numeric-data*.
3. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
4. If *target-data* or *num-digits* has a null value, the execution result will be a null value.
5. Because data of the DOUBLE PRECISION type includes an error, you must be careful when using the data for the scalar function TRUNC. For example, you will not obtain expected calculation results in the following case:

```
TRUNC(2.172157E4,2) -> 2.17215600000000001E4
```

   This is due to the fact that, because 0.01 does not have a finite binary representation, 0.01 and 1.0E-02 are not exactly equal. If you want an exact value for the execution result of truncation, use DECIMAL-type data as the target data.

6. The following table shows the data types that can be specified for the argument and the corresponding valid ranges for *num-digits*.

Table 8-8: Data types that can be specified for the argument and valid ranges for num-digits

No.	Data type specified for the argument (numeric data)	Valid range of num-digits
1	INTEGER	-18 to 0
2	SMALLINT	-9 to 0
3	DECIMAL(*m*,*n*)	-(*m* - *n* - 1) to *n*
4	DOUBLE PRECISION	-308 to 323

Legend: *m*, *n*: Positive integers

Notes:

If the value specified for *num-digits* falls outside the valid range, it does not result in an error. For a positive value outside the valid range, truncation does not occur. For a negative value outside the valid range, the result will be 0.

The following examples illustrate the results for different values of *num-digits*.

Example 1:

The following SQL statement is executed on table `T1`, assuming the value of column `C1` is `123456789`, and its type is `INTEGER`:

```
SELECT TRUNC("C1",x) FROM "T1"
```

The results of executing `TRUNC("C1",`*x*`)` for different values of *x* are shown in the following table.

Table 8-9: Execution results for the SQL statement for different values of x

Value of x	Result of TRUNC(C1,x)
`1` or greater	`123456789`
`0`	`123456789`
`-1`	`123456780`
`-8`	`100000000`
`-9` or less	`0`

Example 2:

The following SQL statement is executed on table `T1`, assuming the value of column `C2` is 123.45 and its type is `DECIMAL(5,2)`:

```
SELECT TRUNC("C2",y) FROM "T1"
```

The results of executing `TRUNC("C2",`*y*`)` for different values of *y* are shown in the following table.

Table 8-10: Execution results for the SQL statement for different values of y

Value of y	Result of TRUNC(C2,y)
`3` or greater	`123.45`
`2`	`123.45`
`1`	`123.40`
`0`	`123.00`
`-1`	`120.00`
`-2`	`100.00`
`-3` or less	`0.00`

(4) Examples

Example 1:

Retrieve that data in columns `C2` and `C3` from table `T1` and truncate the decimal portion starting at position 3, leaving 2 decimal places.

```
SELECT TRUNC("C2",2),TRUNC("C3",2) FROM "T1"
```

Table T1

Col. C2	Col. C3
DECIMAL(8,4)	DOUBLE PRECISION
1502.4890	1.3547110000000000E1
217.3538	3.8875659999999999E2
738.6600	6.3456700000000001E3

Retrieval results

1502.4800	1.3539999999999999E1
217.3500	3.8875000000000000E2
738.6600	6.3456700000000001E3

Example 2:

Truncate 2 digits from the integer portion of the data in columns C1 to C3 from table T1.

```
SELECT TRUNC("C1",-2),TRUNC("C2",-2),TRUNC("C3",-2) FROM "T1"
```

Table T1

Col. C1	Col. C2	Col. C3
INTEGER	DECIMAL(8,4)	DOUBLE PRECISION
78543	1502.4890	1.3547110000000000E1
44712	217.3538	3.8875659999999999E2
11475	738.6600	6.3456700000000001E3

Retrieval results

78500	1500.0000	0.0000000000000000E0
44700	200.0000	3.0000000000000000E2
11400	700.0000	6.3000000000000000E3

8.5 Character string functions (character string operations)

This section describes the functions and specification formats of the character string functions pertaining to operations on character strings.

8.5.1 CONCAT

Concatenates two character string data items.

For the scalar function that concatenates binary data, see 8.10.1 CONCAT.

(1) Specification format

```
scalar-function-CONCAT ::= CONCAT(target-data-1,target-data-2)

  target-data-1 ::= value-expression
  target-data-2 ::= value-expression
```

(2) Explanation of specification format

target-data-1 and *target-data-2*:

Specifies the character string data to be concatenated.

The following rules apply:

- Specify *target-data-1* and *target-data-2* in the form of value expressions. For details about value expressions, see 7.20 Value expression.
- Specify CHAR or VARCHAR type data for *target-data-1* and *target-data-2*.
- You cannot specify a dynamic parameter by itself for *target-data-1* and *target-data-2*.

The following example illustrates the result of executing the scalar function CONCAT.

Example

Concatenate the two character strings ABC and XYZ.

CONCAT('ABC','XYZ') → 'ABCXYZ'

(3) Rules

1. The data type and data length of the execution result are shown in the following table.

Table 8-11: Data type and data length of the execution result of the scalar function CONCAT

Data type and data length of target-data-1	Data type and data length of target-data-2	Data type and data length of the execution result
CHAR(*m*)	CHAR(*n*)	CHAR(*m*+*n*)
	VARCHAR(*n*) Actual data length: *L2*	VARCHAR(*m*+*n*) Actual data length: *m*+*L2*
VARCHAR(*m*) Actual data length: *L1*	CHAR(*n*)	VARCHAR(*m*+*n*) Actual data length: *L1*+*n*

Data type and data length of target-data-1	Data type and data length of target-data-2	Data type and data length of the execution result
	`VARCHAR(`*n*`)` Actual data length: *L2*	`VARCHAR(`*m*+*n*`)` Actual data length: *L1*+*L2*

Legend:

m: Maximum length of *target-data-1*

n: Maximum length of *target-data-2*

L1: Actual data length of *target-data-1*

L2: Actual data length of *target-data-2*

2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If either *target-data-1* or *target-data-2* has a null value, the execution result will be a null value.
4. You cannot concatenate *target-data-1* and *target-data-2* if the result of the concatenation operation would exceed the maximum character string length of 32,000 bytes.
5. Spaces at the end of the character string data are also subject to concatenation.

   Example

   If column `C1` is type `CHAR(5)` with a value of `'ABCΔΔ'`, and column `C2` is type `VARCHAR(10)` with a value of `'XYZ'`, the following concatenations are performed.

   `CONCAT("C1","C2")` → `'ABCΔΔXYZ'`

   `CONCAT("C2","C1")` → `'XYZABCΔΔ'`

   Legend:

   Δ: Single-byte space

(4) Example

Example:

Find the rows in table `T1` for which the execution result of concatenating character string data in columns `C2` and `C3` is `efg03v03`.

```
SELECT * FROM "T1"
    WHERE CONCAT("C2","C3")='efg03v03'
```

Table `T1`

Column `C1` `CHAR`	Column `C2` `VARCHAR`	Column `C3` `VARCHAR`
`A10101`	`abc010587`	`rs3354`
`A15014`	`efg03`	`v03`
`A31399`	`hijk99842688`	`wxyz22725`

Retrieval results

`A15014`	`efg03`	`v03`

8.5.2 LEFT

Extracts a substring from a character string starting from the beginning (leftmost position) of the character string data.

(1) Specification format

```
scalar-function-LEFT ::= LEFT(source-character-string-data,extraction-length)

  source-character-string-data ::= value-expression
  extraction-length ::= value-expression
```

(2) Explanation of specification format

source-character-string-data:

Specifies the source character string data.

The following rules apply:

- Specify the source character string data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify `CHAR` or `VARCHAR` type data for the source character string data.
- You cannot specify a dynamic parameter by itself for the source character string data.

extraction-length:

Specifies the number of characters to extract. The specified number of characters will be extracted from the beginning of the source character string data.

The following rules apply:

- Specify the extraction length in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify an integer (data of type `INTEGER` or `SMALLINT`) for the extraction length.
- If a dynamic parameter is specified by itself for the extraction length, the assumed data type of the dynamic parameter will be `INTEGER`.

The following example illustrates the result of executing the scalar function `LEFT`.

Example

Extract three characters from the beginning of the character string `ABCDEF`.

`LEFT('ABCDEF',3)` → `'ABC'`

(3) Rules

1. The data type and data length of the execution result are shown in the following table.

Table 8-12: Data type and data length of the execution result of the scalar function LEFT

Data type and data length of the source character string data	Data type and length of the execution result of the scalar function LEFT
`CHAR(`*n*`)`	`VARCHAR(`*n*`)`
`VARCHAR(`*n*`)`	

Legend:

n: Maximum length of the source character string data

The number of characters that are extracted is determined as follows:

MIN(*extraction length*, *number of characters in the source character string*)

2. If the extraction length is greater than the number of characters in the source character string data, the amount of data returned will be the number of characters in the source character string data.
3. In the following cases, the execution result will be data whose actual length is 0 bytes:
    - If the length of the character string of the execution result is 0
    - If the actual length of the source character string data is 0 bytes or 0 characters
4. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
5. In the following cases, the execution result will be a null value:
    - If either the source character string data or extraction length has a null value
    - If the extraction length has a negative value (the result will be the null value regardless of what is specified for the source character string data)

(4) Example

Example:

Retrieve rows from table `T1` where the data in column `C1` begins with the three-character string `A15`.

```
SELECT * FROM "T1"
    WHERE LEFT("C1",3)='A15'
```

Table `T1`

Column `C1` `CHAR`	Column `C2` `INTEGER`
A10101	300
A15014	1000
A31399	200

Retrieval results

A15014	1000

8.5.3 LPAD

Pads the beginning (left side) of the target data with the padding character string up to the specified number of characters.

(1) Specification format

```
scalar-function-LPAD ::= LPAD(target-data,num-chars[,padding-character-string])

  target-data ::= value-expression
  num-chars ::= value-expression
  padding-character-string ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the character string data to be padded.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify CHAR or VARCHAR type data for the target data.
- You cannot specify a dynamic parameter by itself for the target data.

num-chars:

Specifies the number of characters in the result character string after it is padded.

The following rules apply:

- Specify the number of characters in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify INTEGER or SMALLINT type data for the number of characters.
- If a dynamic parameter is specified as the number of characters, the assumed data type of the dynamic parameter will be INTEGER type.

padding-character-string:

Specifies the character string to be used for padding the beginning (left side) of the target data.

The following rules apply:

- Specify *padding-character-string* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- You must specify CHAR or VARCHAR type data for *padding-character-string*.
- If *padding-character-string* is omitted, its assumed value is a single-byte space character.
- If a dynamic parameter is specified by itself for *padding-character-string*, the assumed data type of the dynamic parameter is VARCHAR(32000).

The following example illustrates the result of executing the scalar function LPAD.

Example

Pad the beginning (left side) of the data in column C1 with the character string 'xyz' repeatedly until the data has a total length of 10 characters.

LPAD("C1",10,'xyz') → 'xyzxyzxABC'

Column C1 has type VARCHAR(20) and contains the character string 'ABC'.

(3) Rules

1. The data type and data length of the execution result are shown in the following table.

Table 8-13: Data type and data length of the execution result of the scalar function LPAD

Data type and data length of the target data	Data type and data length of the execution result
CHAR(*n*)	VARCHAR(*n*)
VARCHAR(*n*)	

Legend:

n: Maximum length of the target data

2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If either of the following conditions are met, the execution result will be a null value:
   - If the target data, number of characters, or padding character string is the null value

- If you specify a negative value for the number of characters

4. If the actual length of the padding character string is 0 bytes or 0 characters, no character string padding will be performed.
5. If the number of characters in the *target-data* character string is greater than the value of *num-chars*, the function returns the specified number of characters from the beginning of the *target-data* character string.

   Example: `LPAD('ABCDE',3,'xy')` → `'ABC'`
6. If a character string of the specified number of characters cannot be represented in the data length of the execution result, the padding characters will be truncated in mid-string. This means that the number of characters in the execution result might be different from the specified number of characters. If you want to obtain a character string with the specified number of characters, use the scalar function `CAST` to change the data length of the target data.

   Examples

   These examples assume that Unicode (UTF-8) is being used as the character encoding, and that the value and data type of column `C1` are as follows:

   • Value of column `C1`: ⅠⅡ

   • Data type of column `C1`: `VARCHAR(10)`

   `LPAD("C1",5,'ⅢⅣⅤ')` → `'ⅢⅠⅡ'`

   In the above example, the data type of the execution result of `LPAD` is `VARCHAR(10)`. Because each character is 3 bytes, the number of characters of the execution result is not the specified number of characters (5).

   `LPAD(CAST("C1" AS VARCHAR(15)),5,'ⅢⅣⅤ')` → `'ⅢⅣⅤⅠⅡ'`

   In the above example, the data type of the execution result of `LPAD` is `VARCHAR(15)`. Because each character is 3 bytes, the number of characters of the execution result is the specified number of characters (5).

(4) Example

Example:

Column `C1` of table `T1` is a column of the `VARCHAR(8)` type. In column `C1`, for each row containing a character string shorter than 8 characters, add an appropriate number of `0`s to the left of each character string so that all rows in the column contain an 8-character string.

```
SELECT LPAD("C1",8,'0') FROM "T1"
```

Table `T1`

Col. `C1` `VARCHAR(8)`
10101A
1501A
9999999A

Retrieval results

0010101A
0001501A
9999999A

8.5.4 LTRIM

Removes instances of the specified characters, starting from the beginning of the target character string.

Proceeding from the beginning of the character string, it removes all character that matches any of the characters targeted for removal, stopping as soon as it encounters a character that is not targeted for removal.

(1) Specification format

```
scalar-function-LTRIM ::= LTRIM(target-data[,chars-to-remove])

  target-data ::= value-expression
  chars-to-remove ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the data from which the characters specified in *chars-to-remove* are to be removed.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify `CHAR` or `VARCHAR` type data for the target data.
- You cannot specify a dynamic parameter by itself for the target data.

chars-to-remove:

Specifies the characters to be removed from the target data.

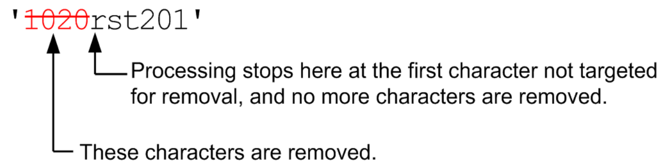
The following rules apply:

- Specify *chars-to-remove* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- You must specify `CHAR` or `VARCHAR` type data for *chars-to-remove*.
- If *chars-to-remove* is omitted, its value is assumed to be a space character.
- If a dynamic parameter is specified by itself for *chars-to-remove*, the assumed data type of the dynamic parameter is `VARCHAR(32000).`

The following examples illustrate the result of executing the scalar function `LTRIM`.

Examples

`LTRIM('1020rst201','012') → 'rst201'`


'~~1020~~rst201'
Processing stops here at the first character not targeted for removal, and no more characters are removed.
These characters are removed.

`LTRIM('aaaadatabaseaaaa','a') → 'databaseaaaa'`

`LTRIM('aabbccdatabase','abc') → 'database'`

`LTRIM('ΔΔΔdatabaseΔ') → 'databaseΔ'`

`LTRIM('database','012') → 'database'`

Legend:

Δ: Single-byte space

(3) Rules

1. The data type and data length of the execution result are shown in the following table.

   Table 8-14: Data type and data length of the execution result of the scalar function LTRIM

Data type and data length of the target data	Data type and data length of the execution result
`CHAR(`*n*`)`	`VARCHAR(`*n*`)`
`VARCHAR(`*n*`)`	

   Legend:

   n: Maximum length of the target data

2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If *target-data* or *chars-to-remove* is the null value, the execution result will be the null value.
4. If the actual length of the target data is 0 bytes or 0 characters, the execution result will be data whose actual length is 0 bytes.
5. If all the target character string data is removed, the execution result will be data whose actual length is 0 bytes.
6. If you specify data whose actual length is 0 bytes or 0 characters for *chars-to-remove*, the execution result will be the target data.

(4) Example

Example:

Remove the numeric prefix from the character string data in column `C2` of table `T1`.

```
SELECT "C1",LTRIM("C2",'0123456789') FROM "T1"
```

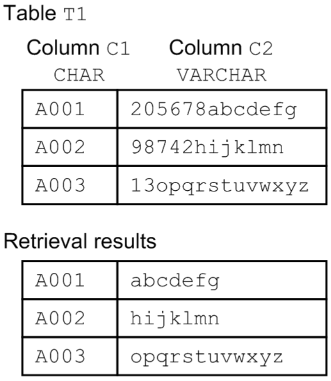


Table `T1`

Column `C1` `CHAR`	Column `C2` `VARCHAR`
`A001`	`205678abcdefg`
`A002`	`98742hijklmn`
`A003`	`13opqrstuvwxyz`

Retrieval results

`A001`	`abcdefg`
`A002`	`hijklmn`
`A003`	`opqrstuvwxyz`

8.5.5 RIGHT

Extracts a substring from a character string starting from the end (rightmost position) of the character string data.

(1) Specification format

```
scalar-function-RIGHT ::= RIGHT(source-character-string-data,extraction-length)

  source-character-string-data ::= value-expression
  extraction-length ::= value-expression
```

(2) Explanation of specification format

source-character-string-data:

Specifies the source character string data.

The following rules apply:

- Specify the source character string data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify `CHAR` or `VARCHAR` type data for the source character string data.
- You cannot specify a dynamic parameter by itself for the source character string data.

extraction-length:

Specifies the number of characters to extract. The specified number of characters will be extracted from the end of the source character string data.

The following rules apply:

- Specify the extraction length in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify an integer (data of type `INTEGER` or `SMALLINT`) for the extraction length.
- If a dynamic parameter is specified by itself for the extraction length, the assumed data type of the dynamic parameter will be `INTEGER`.

The following example illustrates the result of executing the scalar function `RIGHT`.

Example

Extract three characters from the end of the character string `ABCDEF`.

```
RIGHT('ABCDEF',3) → 'DEF'
```

(3) Rules

1. The data type and data length of the execution result are shown in the following table.

Table 8-15: Data type and data length of the execution result of the scalar function RIGHT

Data type and data length of the source character string data	Data type and data length of the execution result of the scalar function RIGHT
`CHAR(`*n*`)`	`VARCHAR(`*n*`)`
`VARCHAR(`*n*`)`	

Legend:

n: Maximum length of the source character string data

The number of characters that are extracted is determined as follows:

MIN(*extraction length*, *number of characters in the source character string*)

2. If the extraction length is greater than the number of characters in the source character string data, the amount of data returned will be the number of characters in the source character string data.
3. In the following cases, the execution result will be data whose actual length is 0 bytes:
   - If the length of the character string of the execution result is 0
   - If the actual length of the source character string data is 0 bytes or 0 characters
4. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).

5. In the following cases, the execution result will be a null value:
   - If either the source character string data or extraction length has a null value
   - If the extraction length has a negative value (the result will be the null value regardless of what is specified for the source character string data)

(4) Example

Example:

Retrieve rows from table `T1` where the data in column `C1` ends with the three-character string `14B`.

```
SELECT * FROM "T1"
    WHERE RIGHT("C1",3)='14B'
```

Table `T1`

Column `C1`	Column `C2`
`CHAR`	`INTEGER`
A10101B	300
A15014B	1000
A31399B	200

Retrieval results

A15014B	1000

8.5.6 RPAD

Pad the end (right side) of the target data with the padding character string up to the specified number of characters.

(1) Specification format

```
scalar-function-RPAD ::= RPAD(target-data,num-chars[,padding-character-string])

  target-data ::= value-expression
  num-chars ::= value-expression
  padding-character-string ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the character string data to be padded.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify `CHAR` or `VARCHAR` type data for the target data.
- You cannot specify a dynamic parameter by itself for the target data.

num-chars:

Specifies the number of characters in the result character string after it is padded.

The following rules apply:

- Specify the number of characters in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify `INTEGER` or `SMALLINT` type data for the number of characters.
- If a dynamic parameter is specified as the number of characters, the assumed data type of the dynamic parameter will be `INTEGER` type.

padding-character-string:

Specifies the character string to be used for padding the end (right side) of the target data.

The following rules apply:

- Specify *padding-character-string* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- You must specify `CHAR` or `VARCHAR` type data for *padding-character-string*.
- If *padding-character-string* is omitted, its assumed value is a single-byte space character.
- If a dynamic parameter is specified by itself for *padding-character-string*, the assumed data type of the dynamic parameter is `VARCHAR(32000)`.

The following example illustrates the result of executing the scalar function `RPAD`.

Example

Pad the end (right side) of the data in column `C1` with the character string `'xyz'` repeatedly until it reaches 10 characters total.

`RPAD("C1",10,'xyz')` → `'ABCxyzxyzx'`

Column `C1` has type `VARCHAR(20)` and holds the character string `'ABC'`.

(3) Rules

1. The data type and data length of the execution result are shown in the following table.

Table 8-16: Data type and data length of the execution result of the scalar function RPAD

Data type and data length of the target data	Data type and data length of the execution result
`CHAR(`*n*`)`	`VARCHAR(`*n*`)`
`VARCHAR(`*n*`)`	

Legend:

n: Maximum length of the target data

2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If either of the following conditions are met, the execution result will be a null value:
   - If the target data, number of characters, or padding character string is the null value
   - If you specify a negative value for the number of characters
4. If the actual length of the padding character string is 0 bytes or 0 characters, no character string padding will be performed.
5. If the number of characters in the *target-data* character string is greater than the value of *num-chars*, the function returns the specified number of characters from the beginning of the *target-data* character string.

   Example: `RPAD('ABCDE',3,'xy')` → `'ABC'`
6. If a character string of the specified number of characters cannot be represented in the data length of the execution result, the padding characters will be truncated in mid-string. This means that the number of characters in the

execution result might be different from the specified number of characters. If you want to obtain a character string with the specified number of characters, use the scalar function `CAST` to change the data length of the target data.

Examples

These examples assume that Unicode (UTF-8) is being used as the character encoding, and that the value and data type of column `C1` are as follows:

- Value of column `C1`: ⅠⅡ
- Data type of column `C1`: `VARCHAR(10)`

`RPAD("C1",5,'ⅢⅣⅤ')` → `'ⅠⅡⅢ'`

In the above example, the data type of the execution result of `RPAD` is `VARCHAR(10)`. Because each character is 3 bytes, the number of characters of the execution result is not the specified number of characters (5).

`RPAD(CAST("C1" AS VARCHAR(15)),5,'ⅢⅣⅤ')` → `'ⅠⅡⅢⅣⅤ'`

In the above example, the data type of the execution result of `RPAD` is `VARCHAR(15)`. Because each character is 3 bytes, the number of characters of the execution result is the specified number of characters (5).

(4) Example

Example:

Column `C1` of table `T1` is a column of the `VARCHAR(8)` type. In column `C1`, for each row containing a character string shorter than 8 characters, add an appropriate number of `0`s to the right of each character string so that all rows in the column contain an 8-character string.

```
SELECT RPAD("C1",8,'0') FROM "T1"
```

Table `T1`

Col. `C1` `VARCHAR(8)`
A10101
A1501
A9999999

Retrieval results

A1010100
A1501000
A9999999

8.5.7 RTRIM

Removes instances of the specified characters, starting from the end of the target character string.

Proceeding from the end of the character string, it removes all character that matches any of the characters targeted for removal, stopping as soon as it encounters a character that is not targeted for removal.

(1) Specification format

```
scalar-function-RTRIM ::= RTRIM(target-data[,chars-to-remove])

  target-data ::= value-expression
  chars-to-remove ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the data from which the characters specified in *chars-to-remove* are to be removed.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify `CHAR` or `VARCHAR` type data for the target data.
- You cannot specify a dynamic parameter by itself for the target data.

chars-to-remove:

Specifies the characters to be removed from the target data.

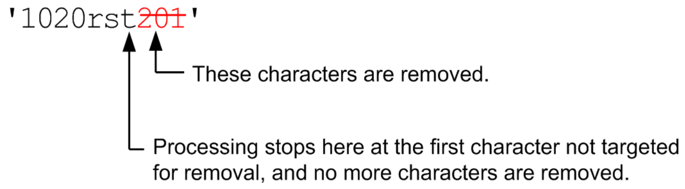
The following rules apply:

- Specify *chars-to-remove* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- You must specify `CHAR` or `VARCHAR` type data for *chars-to-remove*.
- If *chars-to-remove* is omitted, its value is assumed to be a space character.
- If a dynamic parameter is specified by itself for *chars-to-remove*, the assumed data type of the dynamic parameter is `VARCHAR(32000)`.

The following examples illustrate the result of executing the scalar function `RTRIM`.

Examples

```
RTRIM('1020rst201','012') → '1020rst'
```


'1020rst~~201~~'
These characters are removed.
Processing stops here at the first character not targeted for removal, and no more characters are removed.

```
RTRIM('aaaadatabaseaaaa','a') → 'aaaadatabase'
RTRIM('aabbccdatabase','abes') → 'aabbccdat'
RTRIM('ΔΔΔdatabaseΔ') → 'ΔΔΔdatabase'
RTRIM('database','012') → 'database'
```

Legend:

Δ: Single-byte space

(3) Rules

1. The data type and data length of the execution result are shown in the following table.

Table 8-17: Data type and data length of the execution result of the scalar function RTRIM

Data type and data length of the target data	Data type and data length of the execution result
`CHAR(`*n*`)`	`VARCHAR(`*n*`)`
`VARCHAR(`*n*`)`	

Legend:

n: Maximum length of the target data

2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If *target-data* or *chars-to-remove* is the null value, the execution result will be the null value.
4. If the actual length of the target data is 0 bytes or 0 characters, the execution result will be data whose actual length is 0 bytes.
5. If all the target character string data is removed, the execution result will be data whose actual length is 0 bytes.
6. If you specify data whose actual length is 0 bytes or 0 characters for *chars-to-remove*, the execution result will be the target data.

(4) Example

Example:

Remove the numeric suffix from the character string data in column `C2` of table `T1`.

```
SELECT "C1",RTRIM("C2",'0123456789') FROM "T1"
```

Table `T1`

Column `C1` `CHAR`	Column `C2` `VARCHAR`
`A001`	`abcdefg205678`
`A002`	`hijklmn98742`
`A003`	`opqrstuvwxyz13`

Retrieval results

`A001`	`abcdefg`
`A002`	`hijklmn`
`A003`	`opqrstuvwxyz`

8.5.8 SUBSTR

Extracts a substring from a character string starting from any position in the character string data.

(1) Specification format

```
scalar-function-SUBSTR ::= SUBSTR(source-character-string-data, start-position[,extra
ction-length])

  source-character-string-data ::= value-expression
  start-position ::= value-expression
  extraction-length ::= value-expression
```

(2) Explanation of specification format

source-character-string-data:

Specifies the source character string data.

The following rules apply:

- Specify the source character string data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify CHAR or VARCHAR type data for the source character string data.
- You cannot specify a dynamic parameter by itself for the source character string data.

start-position:

Specifies the starting character position from which to extract character string data.

If you specify a value greater than or equal to 0 for the start position, the value represents the position from the beginning of the source character string data. For example, if the start position is 2, the extraction will start at the second character.

If you specify a negative value for the start position, the value represents a position from the end of the source character string data. For example, if the start position is -2, the extraction will start at the second character from the end.

The following rules apply:

- Specify the start position in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify an integer for the start position (INTEGER or SMALLINT type data).
- If you specify 0 for the start position, a start position of 1 is assumed.
- If a dynamic parameter is specified by itself for the start position, the assumed data type of the dynamic parameter will be INTEGER.

extraction-length:

Specifies the number of characters to extract.

The following rules apply:

- Specify the extraction length in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify an integer greater than or equal to 0 (data of type INTEGER or SMALLINT) for the extraction length.
- If no extraction length is specified, when the source character string data is CHAR type, it extracts from the start position to the last character of the defined length.When the source character string data is VARCHAR type, it extracts from the start position to the last character of the actual data.
- If a dynamic parameter is specified by itself for the extraction length, the assumed data type of the dynamic parameter will be INTEGER.

The following examples illustrate the result of executing the scalar function SUBSTR.

Examples

- Extract three characters starting from the second character from the beginning of the character string ABCDEF.

  `SUBSTR('ABCDEF',2,3) → 'BCD'`
- Extract two characters starting from the third character from the end of the character string ABCDEF.

  `SUBSTR('ABCDEF',-3,2) → 'DE'`

(3) Rules

1. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
2. In the following cases, the execution result will be a null value:

- If the extraction length has a negative value (the result will be the null value regardless of what is specified for the source character string data or the start position)
- If the source character string data, start position, or extraction length is a null value

3. The data type and data length of the execution result are shown in the following table.

Table 8-18: Data type and data length of the execution result of the scalar function SUBSTR

Data type and data length of the source character string data	Data type and data length of the execution result
`CHAR(`*n*`)`	`VARCHAR(`*n*`)`
`VARCHAR(`*n*`)`	

Legend:

n: Maximum length of the source character string data

4. The following table shows the number of characters that can be extracted by the scalar function `SUBSTR`.

Table 8-19: Number of characters that can be extracted by the scalar function SUBSTR

Specification of the scalar function SUBSTR		Number of characters that can be extracted
Specification of extraction length	**Value specified for start position**	
Specified	Positive value	MAX{0, MIN(*extraction length*, *number of characters in source character string data* - *start position* + 1)}
	0	MIN(*extraction length*, *number of characters in source character string data*)
	Negative value	MIN(*extraction length*, *absolute value of the start position*, *number of characters in source character string data*)
Omitted	Positive value	MAX(0, *number of characters in source character string data* - *start position* + 1)
	0	*number of characters in source character string data*
	Negative value	MIN(*absolute value of the start position*, *number of characters in source character string data*)

5. In the following cases, the execution result will be data whose actual length is 0 bytes:
   - If the length of the character string of the execution result is 0
   - If the actual length of the source character string data is 0 bytes or 0 characters
   - If the specified start position satisfies either of the following inequalities:
     start position > *number of characters in source character string data*
     start position < -*number of characters in source character string data*

6. If the number of characters in the source character string data, starting from the start position, is less than the extraction length, all of the source character string data, starting from the start position, is returned.

   Example

   `SUBSTR('ABCDEF',5,3)` → `'EF'`

(4) Examples

Example 1:

Retrieve rows from table `T1` where the data in column `C1` contains the three-character substring `150` starting from the second character.

```
SELECT * FROM "T1"
    WHERE SUBSTR("C1",2,3)='150'
```

Table T1

Column C1 CHAR	Column C2 INTEGER
A10101	300
A15014	1000
A31399	200

Retrieval results

A15014	1000

Example 2:

Retrieve rows from table T1 where the data in column C1 contains the two-character substring 01 starting from the second character from the end.

```
SELECT * FROM "T1"
    WHERE SUBSTR("C1",-2,2)='01'
```

Table T1

Column C1 CHAR	Column C2 INTEGER
A10101	300
A15014	1000
A31399	200

Retrieval results

A10101	300

8.5.9 TRIM

Removes instances of the specified characters from the target character string. The characters can be removed in any of the following ways:

- Remove the specified characters starting from the beginning of the character string.
- Remove the specified characters starting from the end of the character string.
- Remove characters starting from both the beginning and the end of the character string.

(1) Specification format

```
scalar-function-TRIM ::= TRIM([{where chars-to-remove
                               |where
                               |chars-to-remove} FROM] target-data)

  where ::= {LEADING|TRAILING|BOTH}
  chars-to-remove ::= value-expression
  target-data ::= value-expression
```

(2) Explanation of specification format

where:

Specifies where to begin the process of removing characters. If this is omitted, `BOTH` is assumed.

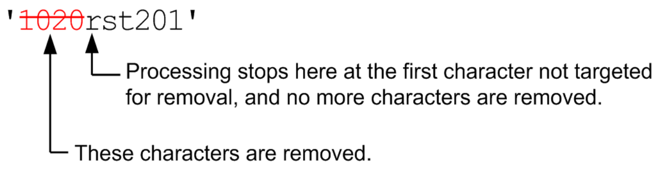
`LEADING`:

When `LEADING` is specified, it removes all characters that match any of the characters specified for removal, proceeding from the beginning of the character string, and stopping as soon as it encounters a character that is not targeted for removal.

The following examples illustrate the execution results when `LEADING` is specified.

Examples

```
TRIM(LEADING '012' FROM '1020rst201') → 'rst201'
```



Processing stops here at the first character not targeted for removal, and no more characters are removed.

These characters are removed.

```
TRIM(LEADING 'a' FROM 'aaaadatabaseaaaa') → 'databaseaaaa'
TRIM(LEADING 'abc' FROM 'aabbccdatabase') → 'database'
TRIM(LEADING FROM 'ΔΔΔdatabaseΔ') → 'databaseΔ'
TRIM(LEADING '012' FROM 'database') → 'database'
```

Legend:

Δ: Single-byte space

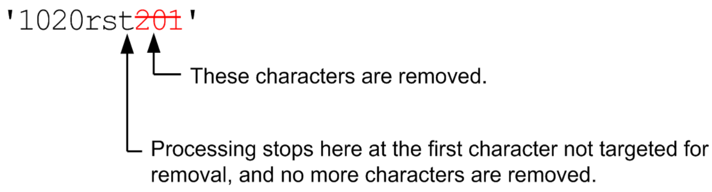
`TRAILING`:

When `TRAILING` is specified, it removes all characters that match any of the characters specified for removal, proceeding from the end of the character string, and stopping as soon as it encounters a character that is not targeted for removal.

The following examples illustrate the execution results when `TRAILING` is specified.

Examples

```
TRIM(TRAILING '012' FROM '1020rst201') → '1020rst'
```



These characters are removed.

Processing stops here at the first character not targeted for removal, and no more characters are removed.

```
TRIM(TRAILING 'a' FROM 'aaaadatabaseaaaa') → 'aaaadatabase'
TRIM(TRAILING 'abes' FROM 'aabbccdatabase') → 'aabbccdat'
TRIM(TRAILING FROM 'ΔΔΔdatabaseΔ') → 'ΔΔΔdatabase'
TRIM(TRAILING '012' FROM 'database') → 'database'
```

Legend:

Δ: Single-byte space

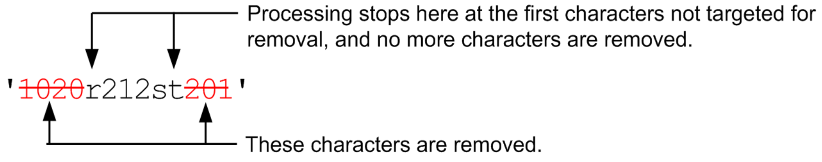
BOTH:

When BOTH is specified, it removes all characters that match any of the characters specified for removal, proceeding from both the beginning and end of the character string, stopping as soon as it encounters a character that is not targeted for removal.

The following examples illustrate the execution results when BOTH is specified.

Examples

```
TRIM(BOTH '012' FROM '1020r212st201') → 'r212st'
```



Processing stops here at the first characters not targeted for removal, and no more characters are removed.

These characters are removed.

```
TRIM(BOTH 'a' FROM 'aaaadatabaseaaaa') → 'database'
TRIM(BOTH 'abces' FROM 'aabbccdatabase') → 'dat'
TRIM(BOTH FROM 'ΔΔΔdatabaseΔ') → 'database'
TRIM(BOTH '012' FROM 'database') → 'database'
```

Legend:

Δ: Single-byte space

chars-to-remove:

Specifies the characters to be removed from the target data.

The following rules apply:

- Specify *chars-to-remove* in the form of a value expression. Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- You must specify CHAR or VARCHAR type data for *chars-to-remove*.
- If *chars-to-remove* is omitted, its value is assumed to be a space character.
- If a dynamic parameter is specified by itself for *chars-to-remove*, the assumed data type of the dynamic parameter is VARCHAR(32000).

target-data:

Specifies the data from which the characters specified in *chars-to-remove* are to be removed.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify CHAR or VARCHAR type data for the target data.
- You cannot specify a dynamic parameter by itself for the target data.

(3) Rules

1. The data type and data length of the execution result are shown in the following table.

Table 8-20: Data type and data length of the execution result of the scalar function TRIM

Data type and data length of the target data	Data type and data length of the execution result
CHAR(*n*)	VARCHAR(*n*)
VARCHAR(*n*)	

Legend: *n*: Maximum length of the target data

2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If *target-data* or *chars-to-remove* is the null value, the execution result will be the null value.
4. If the actual length of the target data is 0 bytes or 0 characters, the execution result will be data whose actual length is 0 bytes.
5. If all the target character string data is removed, the execution result will be data whose actual length is 0 bytes.
6. If you specify data whose actual length is 0 bytes or 0 characters for *chars-to-remove*, the execution result will be the target data.

(4) Example

Example:

Remove the numeric prefix and suffix from the character string data in column C2 of table T1.

```
SELECT "C1",TRIM(BOTH '0123456789' FROM "C2") FROM "T1"
```

Table T1

Column C1	Column C2
CHAR	VARCHAR
A001	581abcdefg205678
A002	611hijklmn98742
A003	32opqrstuvwxyz13

Retrieval results

A001	abcdefg
A002	hijklmn
A003	opqrstuvwxyz

8.6 Character string functions (acquisition of character string information)

This section describes the functions and specification formats of the character string functions pertaining to the acquisition of character string information.

8.6.1 CONTAINS

Returns whether the target data contains any character strings that meet the conditions provided by the search condition expression specification. If the target data contains any character strings that meet the conditions, this function returns `1`. In the other cases, this function returns `0`.

The scalar function `CONTAINS` can be specified in search conditions. However, it cannot be specified in a search condition in the `CASE` expression.

(1) Specification format

```
scalar-function-CONTAINS ::= CONTAINS(target-data,search-condition-expression-specifi
cation)

  target-data ::= value-expression
  search-condition-expression-specification ::= character-string-literal
```

(2) Explanation of specification format

target-data:

Specifies the data to be searched.

The following rules apply:

- Specify *target-data* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- The data type of *target-data* must be `CHAR` or `VARCHAR`.
- You cannot specify a dynamic parameter by itself for *target-data*.

search-condition-expression-specification:

Specifies a search condition.

The following rules apply:

- You must specify *search-condition-expression-specification* in the form of a character string literal. For details about character string literals, see 6.3 Literals.
- All characters other than those in the search string and synonym dictionary name in *search-condition-expression-specification* are assumed to be half-width uppercase.
- Separators cannot be specified.
- For *search-condition-expression-specification*, one of the following four methods can be specified: simple-string specification, notation-correction-search specification, synonym-search specification, and word-context search specification.

```
search-condition-expression-specification ::= {simple-string-specification|notatio
n-correction-search-specification|synonym-search-specification
                          |word-context-search-specification}
```

```
  simple-string-specification ::= "search-character-string"

  notation-correction-search-specification ::= {IGNORECASE(simple-string-specifica
tion)|SORTCODE(simple-string-specification)}

  synonym-search-specification ::= SYNONYM("synonym-dictionary-name",{simple-strin
g-specification|notation-correction-search-specification})

  word-context-search-specification ::= {WORDCONTEXT({simple-string-specification|
notation-correction-search-specification
                                  |synonym-search-specification})
                        |WORDCONTEXT_PREFIX({simple-string-specification|notation-co
rrection-search-specification})}
```

simple-string-specification:

Specifies the search string in the following format:

```
simple-string-specification ::= "search-string"
```

The following shows an example of a simple-string specification.

Example: `"COMPUTER"` or `"computer"`

Enclose the search string (`COMPUTER` or `computer`) in double quotation marks (`"`).

Note the following points:

- The characters in *search-string* are case sensitive.
- To use a double quotation mark (`"`) as an ordinary character in *search-string*, specify two consecutive double quotation marks (`""`).
- If *search-string* is 0-byte character string data, `1` is returned as the execution result. In this case, regardless of *target-data*, the function judges that *target-data* contains *search-string*.

notation-correction-search-specification:

Specify this item when you perform a correction search. For details about correction searches, see *Correction search* in the *HADB Setup and Operation Guide*.

Specify the search string in either of the following formats:

```
notation-correction-search-specification ::= {IGNORECASE(simple-string-specifica
tion) | SORTCODE(simple-string-specification)}
```

- `IGNORECASE(`*simple-string-specification*`)`:

  If `IGNORECASE` is specified, correction search ignores only the difference between half-width uppercase and lowercase letters.

  You can also use the following specification format:

  `I(`*simple-string-specification*`)`

- `SORTCODE(`*simple-string-specification*`)`:

  Specify this item when you perform a correction search.

  You can also use the following specification format:

  `S(`*simple-string-specification*`)`

synonym-search-specification:

Specify this item if you want to search for the synonyms specified as the same synonym group in the synonym dictionary at the same time. The following shows the specification format:

```
synonym-search-specification ::= SYNONYM("synonym-dictionary-name",{simple-strin
g-specification | notation-correction-search-specification})
```

- *synonym-dictionary-name*:
  Specifies the name of the synonym dictionary.

The following shows an example of a synonym-search specification.

Note that this example assumes that the following character strings are registered in a synonym dictionary named `Dictionary1`.

```
PC,personal computer,microcomputer
```

Example 1 (Simple-string specification)

```
SYNONYM("Dictionary1","COMPUTER")
```

In this case, all of the following words registered in the synonym dictionary are used as search strings: `PC`, `personal computer`, and `microcomputer`.

Example 2 (Notation-correction-search specification)

```
SYNONYM("Dictionary1",IGNORECASE("COMPUTER"))
```

In this case, in addition to `PC`, `personal computer`, and `microcomputer`, which are registered in the synonym dictionary, all their variants, such as `pc`, `Personal Computer`, and `Microcomputer`, are used as search strings.

> **Important**
>
> When you register or update a synonym dictionary, if you specify `CASESENSITIVE` (do not create a synonym dictionary that supports correction search) as a notation-correction option, you cannot include notation-correction-search specification in the synonym-search specification.

word-context-search-specification:

Specify this item when you perform a word-context search. For details about word-context searches, see *Word-context search* in the *HADB Setup and Operation Guide*. The following shows the specification format of a word-context search specification:

```
word-context-search-specification ::= {WORDCONTEXT({simple-string-specification|
notation-correction-search-specification
                                       |synonym-search-specification})
                        |WORDCONTEXT_PREFIX({simple-string-specification|notation-co
rrection-search-specification})}
```

To perform word-based complete-match retrieval, specify `WORDCONTEXT`. To perform word-based leading-match search, specify `WORDCONTEXT_PREFIX`.

(3) Rules

1. The scalar function `CONTAINS` can be specified as the comparison operand on the left side of a comparison predicate. For the comparison operator and the right-side comparison operand, specify `>0`.
2. If the character encoding that is used on the HADB server is Shift-JIS, notation-correction-search specification cannot be used.
3. The data type of the execution result will be `INTEGER`.
4. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
5. If the target data is the null value, the execution result will be the null value.

(4) Examples

Assume that you have the table T1 whose column C2 (of the VARCHAR type) contains document information. In the following examples, you use the scalar function CONTAINS to search the document information for specific character strings.

Example 1 (Search using a simple-string specification)

In this example, you retrieve rows whose document information contains the character string COMPUTER.

```
SELECT "C1" FROM "T1"
    WHERE CONTAINS("C2",'"COMPUTER"') > 0
```

In this case, half-width uppercase and lowercase letters are distinguished. Therefore, computer and other variants of the specified string are not treated as search strings.

Example 2 (Correction search)

In this example, you retrieve rows whose document information contains the character string COMPUTER and its variants.

```
SELECT "C1" FROM "T1"
    WHERE CONTAINS("C2",'IGNORECASE("COMPUTER")') > 0
```

In this case, as a result of notation correction, in addition to the rows that contain COMPUTER, the rows that contain computer, Computer, and other similar variants are to be retrieved.

Example 3 (Correction search)

In this example, you retrieve rows whose document information contains máquina and its variants.

```
SELECT "C1" FROM "T1"
    WHERE CONTAINS("C2",'SORTCODE("máquina")') > 0
```

In this case, as a result of notation correction, in addition to the rows that contain máquina, the rows that contain maquina, Maquina, and other variants are to be retrieved.

Example 4 (Synonym search)

Assume that the following character strings are registered in the synonym dictionary Dictionary1 as synonyms: PC, personal computer, and microcomputer. In this example, you search for these character strings in one operation.

```
SELECT "C1" FROM "T1"
    WHERE CONTAINS("C2",'SYNONYM("Dictionary1","PC")') > 0
```

Example 5 (Synonym search + correction search)

Assume that the following character strings are registered in the synonym dictionary Dictionary1 as synonyms: PC, personal computer, and microcomputer. In this example, you search for these character strings in one operation. In this case, a correction search is also performed for each of the character strings registered in the synonym dictionary.

Note that the synonym dictionary Dictionary1 must support correction search.

```
SELECT "C1" FROM "T1"
    WHERE CONTAINS("C2",'SYNONYM("Dictionary1",SORTCODE("PC"))') > 0
```

When the preceding SELECT statement is run, in addition to the character strings registered in the synonym dictionary (PC, personal computer, microcomputer), their variants, such as pc, Personal Computer, and MICROCOMPUTER, are to be retrieved.

Example 6 (Word-based complete-match word-context search)

In this example, you retrieve rows that contain the English word COMPUTER or its variant from the English document stored in column C2.

```
SELECT "C1" FROM "T1"
    WHERE CONTAINS("C2",'WORDCONTEXT(IGNORECASE("COMPUTER"))') > 0
```

In this case, because the correction search (IGNORECASE) is specified, in addition to the rows that contain COMPUTER, the rows that contain computer, Computer, and other similar variants are also retrieved.

Example 7 (Word-based leading-match word-context search)

In this example, you retrieve rows that contain an English word that begins with COMP from the English document stored in column C2.

```
SELECT "C1" FROM "T1"
    WHERE CONTAINS("C2",'WORDCONTEXT_PREFIX("COMP")') > 0
```

In this case, the rows that contain a word such as COMPUTER, COMPUTERS, or COMPANY are also retrieved.

8.6.2 INSTR

Searches the target data for a character string and returns the starting position of the string.

A starting position at which to begin the search can also be specified.

For example, using this function, you can find the starting position of the character string 'ABC' in the target data, or even find the starting position of the third occurrence of 'ABC'.

(1) Specification format

```
scalar-function-INSTR ::= INSTR(target-data,search-character-string[,search-start-pos
ition[,nth-occurrence]])

  target-data ::= value-expression
  search-character-string ::= value-expression
  search-start-position ::= value-expression
  nth-occurrence ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the target data in which to search for the character string.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify CHAR or VARCHAR type data for the target data.
- You cannot specify a dynamic parameter by itself for the target data.

search-character-string:

Specifies the character string to search for.

The following rules apply:

- Specify *search-character-string* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify `CHAR` or `VARCHAR` type data for *search-character-string*.
- You cannot specify a dynamic parameter by itself for *search-character-string*.

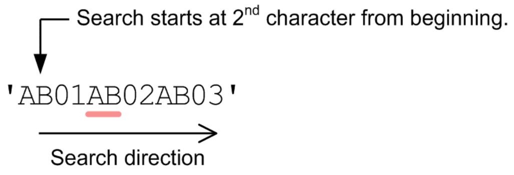
search-start-position:

Specifies the starting character position to begin searching the target data.

- **If you specify a positive integer for *search-start-position***

  The search starts at that position in the target data and proceeds in the forward direction (left to right). For example, if you specify `2` for *search-start-position*, the search starts at the second character of the target data and proceeds in the forward direction (left to right).

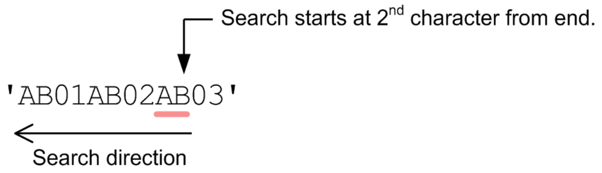
  Example: `INSTR('AB01AB02AB03','AB',2)` → `5`



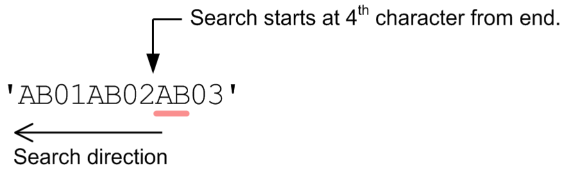
- **If you specify a negative integer for *search-start-position***

  The search starts at that position from the end of the target data and proceeds backwards (right to left). For example, if you specify `-2` for *search-start-position*, the search starts at the second character from the end of the target data and proceeds backwards (right to left).

  Example 1: `INSTR('AB01AB02AB03','AB',-2)` → `9`



  Example 2: `INSTR('AB01AB02AB03','AB',-4)` → `9`



  In the above example, the search begins at `A`, which is immediately followed by `B`. The execution results is therefore `9`.

The following rules apply:

- Specify *search-start-position* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify `INTEGER` or `SMALLINT` type data for *search-start-position*.
- If *search-start-position* is omitted, `1` is assumed.
- If you specify a dynamic parameter by itself for *search-start-position*, the assumed data type of the dynamic parameter is `INTEGER`.

nth-occurrence:

Specifies which occurrence of the character string to search for. For example, if you specify `3` for *nth-occurrence*, it returns the starting position of the third occurrence of the character string in the target data.

The following rules apply:

- Specify *nth-occurrence* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify a positive integer for *nth-occurrence*.
- Specify `INTEGER` or `SMALLINT` type data for *nth-occurrence*.
- If *nth-occurrence* is omitted, `1` is assumed.
- If you specify a dynamic parameter by itself for *nth-occurrence*, the assumed data type of the dynamic parameter is `INTEGER`.

The following examples illustrate the result of executing the scalar function `INSTR`.

Examples

- `INSTR('AB01AB02AB03','AB') → 1`

  This example returns `1` because the search character string `'AB'` is found at the first position in the target data.
- `INSTR('AB01AB02AB03','AB',3) → 5`

  In this example, the search begins at the third character from the beginning of the target data. It returns `5` because the search character string `'AB'` is found at the fifth position in the target data.
- `INSTR('AB01AB02AB03','AB',3,2) → 9`

  In this example, the search begins at the third character from the beginning of the target data. Furthermore, because *nth-occurrence* is `2`, it returns the starting position of the second occurrence of `'AB'`, which in this case is `9` because the second occurrence of `'AB'` starts at the ninth character of the target data.
- `INSTR('AB01AB02AB03','AB',-2,3) → 1`

  In this example, the search begins at the second character from the end of the target data. Furthermore, because *nth-occurrence* is `3`, it returns the starting position of the third occurrence of `'AB'`, which in this case is `1` because the third occurrence of `'AB'` starts at the first character of the target data.

(3) Rules

1. The value of the execution result is expressed in units of number of characters.
2. Regardless of the value of *search-start-position*, the value returned as the execution result will be the position of the occurrence of the character string as counted from the beginning of the target data (from the left).
3. If the specified character string is not found, `0` is returned as the value of the execution result.
4. The data type of the execution result is the `INTEGER` type.
5. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
6. In the following cases, the execution result will be a null value:
   - If either *target-data*, *search-character-string*, *search-start-position*, or *nth-occurrence* has a null value
   - If you specify `0` for *search-start-position*
   - If you specify `0` or a negative value for *nth-occurrence*
7. If either *target-data* or *search-character-string* has an actual length of 0 bytes or 0 characters, the value of the execution result will be `0`, except in the following cases:

- If either *target-data* or *search-character-string* has a null value
- If you specify `0` for *search-start-position*
- If you specify `0` or a negative value for *nth-occurrence*

8. The character strings *search-character-string* and *target-data* are compared character-by-character until either the *n*th occurrence (specified in *nth-occurrence*) of *search-character-string* is found, or the end of *target-data* is reached.

(4) Example

Example:

From the email addresses stored in column `C1` of table `T1`, extract the character string preceding the `@` part of each address.

```
SELECT LEFT("C1",INSTR("C1",'@')-1) FROM "T1"
```

Table `T1`

Col. `C1` VARCHAR
abcdefg@hhhh.co.jp
abcdefghijk@iiii.co.jp
lmn@bbbb.co.jp

Retrieval results

abcdefg
abcdefghijk
lmn

8.6.3 LENGTH

Returns the number of characters in the target character string.

(1) Specification format

```
scalar-function-LENGTH ::= LENGTH(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the data whose length in characters is to be counted.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify `CHAR` or `VARCHAR` type data for the target data.
- You cannot specify a dynamic parameter by itself for the target data.

(3) Rules

1. The data type of the execution result is the `INTEGER` type.
2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If the target data has the null value, the execution result will be a null value.
4. If the actual length of the target data is 0 bytes or 0 characters, the execution result will be `0`.

(4) Example

Example:

Determine the number of characters in the data in columns `C1` and `C2` from table `T1`.

In this case, the character encoding being used is Unicode (UTF-8).

```
SELECT LENGTH("C1"),LENGTH("C2") FROM "T1"
```

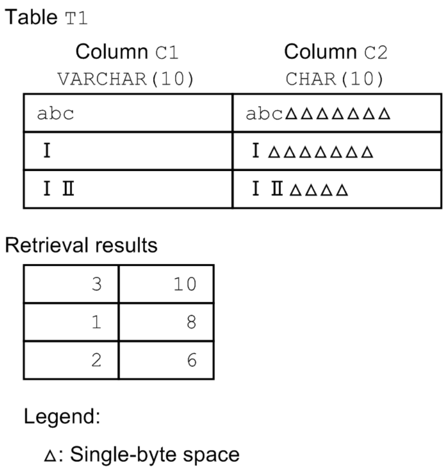


Table T1

Column C1 VARCHAR(10)	Column C2 CHAR(10)
abc	abc△△△△△△△
Ⅰ	Ⅰ△△△△△△△
ⅠⅡ	ⅠⅡ△△△△

Retrieval results

3	10
1	8
2	6

Legend:

△: Single-byte space

In the above example, each space is counted as one character.

8.7 Character string functions (Character substitution)

This section describes the functions and specification formats of the character string functions pertaining to character substitution.

8.7.1 REPLACE

Replaces any character string in the target data. All instances of the character string to be replaced in the target data are replaced with a replacement character string.

(1) Specification format

```
scalar-function-REPLACE ::= REPLACE(target-data,character-string-to-replace[,replacem
ent-character-string])

  target-data ::= value-expression
  character-string-to-replace ::= value-expression
  replacement-character-string ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the target data.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify CHAR or VARCHAR type data for the target data.
- You cannot specify a dynamic parameter by itself for the target data.

character-string-to-replace:

Specifies the character string to be replaced.

The following rules apply:

- Specify *character-string-to-replace* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- You must specify CHAR or VARCHAR type data for *character-string-to-replace*.
- If a dynamic parameter is specified by itself for *character-string-to-replace*, the assumed data type of the dynamic parameter is VARCHAR(32000).

replacement-character-string:

Specifies the replacement character string.

The following rules apply:

- Specify *replacement-character-string* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- You must specify CHAR or VARCHAR type data for *replacement-character-string*.
- If a dynamic parameter is specified by itself for *replacement-character-string*, the assumed data type of the dynamic parameter is VARCHAR(32000).

- If *replacement-character-string* is omitted, data whose actual length is 0 bytes is assumed.

The following example illustrates the result of executing the scalar function `REPLACE`.

Example

Replace all instances of the character string `BCD` in the target data with `YZ`.

`REPLACE('ABCDEBCD','BCD','YZ')` → `'AYZEYZ'`

(3) Rules

1. The data type and data length of the execution result are shown in the following table.

   Table 8-21: Data type and data length of the execution result of the scalar function REPLACE

Data type and data length of the target data	Data type and data length of the execution result
`CHAR(`*n*`)`	`VARCHAR(`*n*`)`
`VARCHAR(`*n*`)`	

   Legend: *n*: Maximum length of the target data

2. An error results if, after the replacement, the data length of the execution result is exceeded. If you want to increase the data length of the execution result, use the scalar function `CAST` to change the data length of the target data.

   Examples

   These examples assume that column `C1` has type `VARCHAR(5)` and contains the character string `'ABCD'`.

   `REPLACE("C1",'AB','WXYZ')` → `Error`

   The above example results in an error because the data length of the result of executing `REPLACE` is too large for `VARCHAR(5)`.

   `REPLACE(CAST("C1" AS VARCHAR(10)),'AB','WXYZ')` → `'WXYZCD'`

   The above example does not result in an error because the data length of the result of executing `REPLACE` can fit in `VARCHAR(10)`.

3. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
4. If the target data, character string to replace, or replacement character string is the null value, the execution result will be the null value.
5. If all the characters in the target data are deleted as a result of the replacement, the actual data length of the execution result will be 0 bytes.
6. If you specify data whose actual length is 0 bytes or 0 characters for the character string to replace, no characters in the target data are replaced.
7. If you specify data whose actual length is 0 bytes or 0 characters for the replacement character string, all instances of the character string to replace are removed from the target data.

(4) Example

Example:

This example assumes that column `C1` (`CHAR` type) of table `T1` holds dates in the format *YYYY*`.`*MM*`.`*DD* (where *YYYY* is the year, *MM* is the month, and *DD* is the day).

All instances of `2013` are replaced with `2014`.

```
SELECT REPLACE("C1",'2013','2014') FROM "T1"
```

Table `T1`

Column `C1` `CHAR(10)`
`2012.10.18`
`2013.01.25`
`2013.02.05`

Retrieval results

`2012.10.18`
`2014.01.25`
`2014.02.05`

8.7.2 TRANSLATE

Replaces any character in character string data.

(1) Specification format

```
scalar-function-TRANSLATE ::= TRANSLATE(target-data,characters-to-replace,replacement
-characters)

  target-data ::= value-expression
  characters-to-replace ::= value-expression
  replacement-characters ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the target data.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify `CHAR` or `VARCHAR` type data for the target data.
- You cannot specify a dynamic parameter by itself for the target data.

characters-to-replace:

Specifies the characters to be replaced.

The following rules apply:

- Specify *characters-to-replace* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- If you specify the same character more than once in *characters-to-replace*, it uses the character that was specified first.
- If a dynamic parameter is specified by itself for *characters-to-replace*, the assumed data type of the dynamic parameter is `VARCHAR(32000)`.

replacement-characters:

Specifies the replacement characters.

The following rules apply:

- Specify *replacement-characters* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- If a dynamic parameter is specified by itself for *replacement-characters*, the assumed data type of the dynamic parameter is `VARCHAR(32000)`.

> **Tip**
>
> To replace multiple characters, align the characters in the same positions in *characters-to-replace* and *replacement-characters*. For example, to replace `A` with `a`, `B` with `b`, and `C` with `c`, specify `'ABC'` for *characters-to-replace*, and `'abc'` for *replacement-characters*.

The following example illustrates the result of executing the scalar function `TRANSLATE`.

Example

In a character string, replace `A` with `a`, `B` with `b`, and `C` with `c`.

`TRANSLATE('AXBYCZ','ABC','abc')` → `'aXbYcZ'`

(3) Rules

1. The data type and data length of the execution result are shown in the following table.

   Table 8-22: Data type and data length of the execution result of the scalar function TRANSLATE

Data type and data length of the target data	Data type and data length of the execution result
`CHAR(`*n*`)`	`VARCHAR(`*n*`)`
`VARCHAR(`*n*`)`	

   Legend:

   n: Maximum length of the target data

2. An error results if, after the replacement, the data length of the execution result is exceeded. If you want to increase the data length of the execution result, use the scalar function `CAST` to change the data length of the target data.

   Examples

   These examples assume that Unicode (UTF-8) is the character encoding, and that column `C1` has type `VARCHAR(5)` and holds the character string `'ABC'`.

   `TRANSLATE("C1",'ABC','ⅠⅡⅢ')`→`Error`

   This example generates an error because the type `VARCHAR(5)` is insufficient to store the data length of the execution result of `TRANSLATE`.

   `TRANSLATE(CAST("C1" AS VARCHAR(9)),'ABC','ⅠⅡⅢ')`→`'ⅠⅡⅢ'`

   This time there is no error because the data length of the execution result of `TRANSLATE` is `VARCHAR(9)`.

3. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).

4. If *target-data*, *characters-to-replace*, or *replacement-characters* is the null value, the execution result will be the null value.

5. In the target data, the characters in *characters-to-replace* are replaced with the characters in *replacement-characters*. If no characters are specified in *characters-to-replace*, no characters in the target data are replaced.

6. When *characters-to-replace* is longer than *replacement-characters*, the extra characters in *characters-to-replace* are deleted from the target data if they are present.

   Example: `TRANSLATE('ABCD','ABC','ab')` → `'abD'`

7. When *characters-to-replace* is shorter than *replacement-characters*, the extra characters in *replacement-characters* are ignored.

   Example: `TRANSLATE('ABCD','AB','abc')` → `'abCD'`

8. If all the characters in the target data are deleted as a result of the replacement, the actual data length of the execution result will be 0 bytes.

(4) Example

Example:

Translate the format of the dates that are stored in column `C1` (type `CHAR`) in table `T1` from *YYYY*`.`*MM*`.`*DD* to *YYYY*`/`*MM*`/`*DD*, where *YYYY* is the year, *MM* is the month, and *DD* is the day.

```
SELECT TRANSLATE("C1",'.','/') FROM "T1"
```

Table `T1`

Col. `C1` `CHAR(10)`
2014.01.25
2014.02.05

Retrieval results

2014/01/25
2014/02/05

8.8 Character string functions (character string conversion)

This section describes the functions and specification formats of the character string functions pertaining to character string conversion.

8.8.1 LOWER

Converts character string data from uppercase (A to Z) to lowercase (a to z). Single- and double-byte letters are supported.

(1) Specification format

```
scalar-function-LOWER ::= LOWER(character-string-data-to-convert)

  character-string-data-to-convert ::= value-expression
```

(2) Explanation of specification format

character-string-data-to-convert:

Specifies the character string data to convert.

The following rules apply:

- Specify *character-string-data-to-convert* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify CHAR or VARCHAR type data for *character-string-data-to-convert*.
- You cannot specify a dynamic parameter by itself for *character-string-data-to-convert*.

The following table shows the character encodings and character ranges that are converted.

Table 8-23: Character encodings and character ranges that are converted by the scalar function LOWER

Character encoding	Range of characters to be converted	Range of characters post-conversion	Single-byte/ double-byte
Unicode (UTF-8)	A (0x41) to Z (0x5a)	a (0x61) to z (0x7a)	Single-byte characters
	Ａ (0xefbca1) to Ｚ (0xefbcba)	ａ (0xefbd81) to ｚ (0xefbd9a)	Double-byte characters
Shift-JIS	A (0x41) to Z (0x5a)	a (0x61) to z (0x7a)	Single-byte characters
	Ａ (0x8260) to Ｚ (0x8279)	ａ (0x8281) to ｚ (0x829a)	Double-byte characters

The following example illustrates the result of executing the scalar function LOWER.

Example

Convert the uppercase letters in the character string aBc123XyZ to lowercase.

LOWER('aBc123XyZ') → 'abc123xyz'

(3) Rules

1. The length of the data type of *character-string-data-to-convert* becomes the length of the data type of the execution result.
2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If *character-string-data-to-convert* has the null value, the execution result will be a null value.

(4) Example

Example:

Convert the data in the NAME column of the employee table (EMPLIST) to all lowercase.

```
SELECT "USERID",LOWER("NAME")
   FROM "EMPLIST"
```

EMPLIST

USERID	NAME	SEX	SCODE
U00555	Taro Tanaka	M	S001
U00358	Nancy White	F	S003
U00212	Maria Gomez	F	S001
U00687	Taro Tanaka	M	S002
U00869	NULL	M	S003

Retrieval results

U00555	taro tanaka
U00358	nancy white
U00212	maria gomez
U00687	taro tanaka
U00869	NULL

8.8.2 UPPER

Converts character string data from lowercase (a to z) to uppercase (A to Z). Single- and double-byte letters are supported.

(1) Specification format

```
scalar-function-UPPER ::= UPPER(character-string-data-to-convert)

  character-string-data-to-convert ::= value-expression
```

(2) Explanation of specification format

character-string-data-to-convert:

Specifies the character string data to convert.

The following rules apply:

- Specify *character-string-data-to-convert* in the form of a value expression. For details about value expressions, see 7.20 Value expression.

- Specify `CHAR` or `VARCHAR` type data for *character-string-data-to-convert*.
- You cannot specify a dynamic parameter by itself for *character-string-data-to-convert*.

The following table shows the character encodings and character ranges that are converted.

Table 8-24: Character encodings and character ranges that are converted by the scalar function UPPER

Character encoding	Range of characters to be converted	Range of characters post-conversion	Single-byte/ double-byte
Unicode (UTF-8)	`a` (`0x61`) to `z` (`0x7a`)	`A` (`0x41`) to `Z` (`0x5a`)	Single-byte characters
	`ａ` (`0xefbd81`) to `ｚ` (`0xefbd9a`)	`Ａ` (`0xefbca1`) to `Ｚ` (`0xefbcba`)	Double-byte characters
Shift-JIS	`a` (`0x61`) to `z` (`0x7a`)	`A` (`0x41`) to `Z` (`0x5a`)	Single-byte characters
	`ａ` (`0x8281`) to `ｚ` (`0x829a`)	`Ａ` (`0x8260`) to `Ｚ` (`0x8279`)	Double-byte characters

The following example illustrates the result of executing the scalar function `UPPER`.

Example

Convert the lowercase letters in the character string `aBc123XyZ` to uppercase.

`UPPER('aBc123XyZ') → 'ABC123XYZ'`

(3) Rules

1. The length of the data type of *character-string-data-to-convert* becomes the length of the data type of the execution result.
2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If *character-string-data-to-convert* has the null value, the execution result will be a null value.

(4) Example

Example:

Convert the data in the `NAME` column of the employee table (`EMPLIST`) to all uppercase.

```
SELECT "USERID",UPPER("NAME")
   FROM "EMPLIST"
```

EMPLIST

USERID	NAME	SEX	SCODE
U00555	Taro Tanaka	M	S001
U00358	Nancy White	F	S003
U00212	Maria Gomez	F	S001
U00687	Taro Tanaka	M	S002
U00869	NULL	M	S003

Retrieval results

U00555	TARO TANAKA
U00358	NANCY WHITE
U00212	MARIA GOMEZ
U00687	TARO TANAKA
U00869	NULL

8.9 Datetime functions

This section describes the functions and specification formats of the datetime functions.

8.9.1 DATEDIFF

Returns the difference between the start date and time and the end date and time.

(1) Specification format

```
scalar-function-DATEDIFF ::= DATEDIFF(datetime-unit,start-datetime,end-datetime)

  datetime-unit ::= {YEAR|QUARTER|MONTH|WEEK|DAY
                    |DAYOFYEAR|HOUR|MINUTE|SECOND
                    |MILLISECOND|MICROSECOND|NANOSECOND|PICOSECOND}
  start-datetime ::= value-expression
  end-datetime ::= value-expression
```

(2) Explanation of specification format

datetime-unit:

Specifies the unit to be used when determining the difference between *start-datetime* and *end-datetime*. Specify one of the following values:

- `YEAR`

  Specify this to determine the difference in years between *start-datetime* and *end-datetime*.

  Examples

  `DATEDIFF(YEAR,'2011-05-05','2013-07-10')` → 2

  `DATEDIFF(YEAR,'2013-05-05','2013-07-10')` → 0

  `DATEDIFF(YEAR,'2012-12-31 23:59:59','2013-01-01 00:00:00')` → 1

- `QUARTER`

  Specify this to determine the difference in quarters between *start-datetime* and *end-datetime*. Quarters are calculated as three-month periods beginning January 1.

  • First quarter: January 1 to March 31

  • Second quarter: April 1 to June 30

  • Third quarter: July 1 to September 30

  • Fourth quarter: October 1 to December 31

  Examples

  `DATEDIFF(QUARTER,'2013-01-05','2013-07-10')` → 2

  `DATEDIFF(QUARTER,'2013-01-05','2013-03-10')` → 0

  `DATEDIFF(QUARTER,'2012-12-31 23:59:59','2013-01-01 00:00:00')` → 1

- `MONTH`

  Specify this to determine the difference in months between *start-datetime* and *end-datetime*.

  Examples

  `DATEDIFF(MONTH,'2013-01-05','2013-07-10')` → 6

  `DATEDIFF(MONTH,'2013-01-05','2013-01-10')` → 0

DATEDIFF(MONTH,'2012-12-31 23:59:59','2013-01-01 00:00:00') → 1

- WEEK

  Specify this to determine the difference in weeks between *start-datetime* and *end-datetime*. Weeks are calculated as beginning on Sunday.

  Examples

  DATEDIFF(WEEK,'2013-07-05','2013-07-10') → 1

  This example returns 1 because the week changes on July 7, 2013, which is a Sunday.

  DATEDIFF(WEEK,'2012-12-30','2013-01-01') → 0

  This example returns 0 because December 30, 2012 is a Sunday, so the week does not change.

- DAY

  Specify this to determine the difference in days between *start-datetime* and *end-datetime*.

  Examples

  DATEDIFF(DAY,'2013-07-05','2013-07-10') → 5

  DATEDIFF(DAY,'2013-07-05 08:02:25','2013-07-05 17:55:18') → 0

  DATEDIFF(DAY,'2012-12-31 23:59:59','2013-01-01 00:00:00') → 1

- DAYOFYEAR

  Specify this to determine the difference between *start-datetime* and *end-datetime* in terms of cumulative number of days. It returns the same result as when DAY is specified.

  Examples

  DATEDIFF(DAYOFYEAR,'2013-07-05','2013-07-10') → 5

  DATEDIFF(DAYOFYEAR,'2013-07-05 08:02:25','2013-07-05 17:55:18') → 0

  DATEDIFF(DAYOFYEAR,'2012-12-31 23:59:59','2013-01-01 00:00:00') → 1

- HOUR

  Specify this to determine the difference in hours between *start-datetime* and *end-datetime*.

  Examples

  DATEDIFF(HOUR,'2013-07-10 08:02:25','2013-07-10 11:37:55') → 3

  DATEDIFF(HOUR,'2013-07-10 08:02:25','2013-07-10 08:45:15') → 0

  DATEDIFF(HOUR,'2012-12-31 23:59:59','2013-01-01 00:00:00') → 1

- MINUTE

  Specify this to determine the difference in minutes between *start-datetime* and *end-datetime*.

  Examples

  DATEDIFF(MINUTE,'2013-07-10 08:02:25','2013-07-10 08:07:25') → 5

  DATEDIFF(MINUTE,'2013-07-10 08:02:25','2013-07-10 08:02:32') → 0

  DATEDIFF(MINUTE,'2012-12-31 23:59:59','2013-01-01 00:00:00') → 1

- SECOND

  Specify this to determine the difference in seconds between *start-datetime* and *end-datetime*.

  Examples

  DATEDIFF(SECOND,'2013-07-10 08:02:25','2013-07-10 08:02:33') → 8

  DATEDIFF(SECOND,'2012-12-31 23:59:59','2013-01-01 00:00:00') → 1

- MILLISECOND

  Specify this to determine the difference in milliseconds (1/1,000 seconds) between *start-datetime* and *end-datetime*.

Examples

`DATEDIFF(MILLISECOND,'08:02:25.000','08:02:25.003')` → `3`

`DATEDIFF(MILLISECOND,'08:02:24.000','08:02:25.001')` → `1001`

`DATEDIFF(MILLISECOND,'08:02:25.000000','08:02:25.003111')` → `3`

- `MICROSECOND`

  Specify this to determine the difference in microseconds (1/1,000,000 seconds) between *start-datetime* and *end-datetime*.

  Example

  `DATEDIFF(MICROSECOND,'08:02:25.000000','08:02:25.000012')` → `12`

- `NANOSECOND`

  Specify this to determine the difference in nanoseconds (1/1,000,000,000 seconds) between *start-datetime* and *end-datetime*.

  Example

  `DATEDIFF(NANOSECOND,'08:02:25.000000000','08:02:25.000000123')` → `123`

- `PICOSECOND`

  Specify this to determine the difference in picoseconds (1/1,000,000,000,000 seconds) between *start-datetime* and *end-datetime*.

  Example

  `DATEDIFF(PICOSECOND,'08:02:25.000000000000','08:02:25.000000000003')` → `3`

start-datetime:

Specifies the start datetime.

The following rules apply:

- Specify *start-datetime* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- The data type of *start-datetime* must be `DATE`, `TIME`, `TIMESTAMP`, `CHAR`, or `VARCHAR`. In the case of `CHAR` or `VARCHAR`, you must specify a character string literal that adheres to the predefined input representation formats. For details about the predefined input representations, see 6.3.3 Predefined character-string representations.
- You cannot specify a dynamic parameter by itself for *start-datetime*.

end-datetime:

Specifies the end datetime.

The following rules apply:

- Specify *end-datetime* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- The data type of *end-datetime* must be `DATE`, `TIME`, `TIMESTAMP`, `CHAR`, or `VARCHAR`. In the case of `CHAR` or `VARCHAR`, you must specify a character string literal that adheres to the predefined input representation formats. For details about the predefined input representations, see 6.3.3 Predefined character-string representations.
- You cannot specify a dynamic parameter by itself for *end-datetime*.

(3) Rules

1. For the execution result, the value obtained by subtracting *start-datetime* from *end-datetime* is returned. If *end-datetime* is earlier than *start-datetime*, a negative value is returned.

2. When the hour, minutes, and seconds are missing, for example when *start-datetime* is `DATE` type and *end-datetime* is `TIMESTAMP` type, the hour, minutes, and seconds are assumed to be `00:00:00`. When the fractional seconds are missing, all the missing digits are assumed to be 0.

   Example: `DATEDIFF(SECOND,'2013-07-10','2013-07-10 00:00:07')` → `7`

   In the example above, *start-datetime* is assumed to be `2013-07-10 00:00:00`.

3. If you specify a `DATE` type, `TIMESTAMP` type, predefined character-string representation of a date, or a predefined character-string representation of a time stamp for *start-datetime*, you must also specify a `DATE` type, `TIMESTAMP` type, predefined character-string representation of a date, or a predefined character-string representation of a time stamp for *end-datetime*.

4. If you specify a `TIME` type or a predefined character-string representation of a time for *start-datetime*, you must also specify a `TIME` type or a predefined character-string representation of a time for *end-datetime*.

5. If you specify a `TIME` type or a predefined character-string representation of a time for *start-datetime* and *end-datetime*, and `YEAR`, `QUARTER`, `MONTH`, `DAYOFYEAR`, `DAY`, or `WEEK` for *datetime-unit*, the value of the execution result will be 0.

6. The data type of the execution result is the `INTEGER` type. An error results if the execution result exceeds the range that can be represented by the `INTEGER` type. For the range that can be represented by the `INTEGER` type, see (1) Numeric data in 6.2.1 List of data types.

7. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).

8. If *start-datetime* or *end-datetime* has a null value, the execution result will be a null value.

(4) Example

Example:

Determine the difference in days between the datetime data in columns `C1` and `C2` from table `T1`.

```
SELECT DATEDIFF(DAY,"C1","C2") FROM "T1"
```

Table `T1`

Column `C1` (`DATE` type)	Column `C2` (`DATE` type)
2013-06-12	2013-06-20
2013-05-31	2013-06-21
2013-06-20	2013-06-18

Retrieval results

8
21
-2

8.9.2 DAYOFWEEK

Returns the day of the week that the specified date falls on. Note that the first day of the week is Sunday.

(1) Specification format

```
scalar-function-DAYOFWEEK ::= {DAYOFWEEK|DOW}(target-data)

  target-data ::= value-expression
```

Note: DOW can be used as an abbreviated form for DAYOFWEEK.

(2) Explanation of specification format

target-data:

Specifies the data representing the day.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- The data type of the target data must be DATE, TIMESTAMP, CHAR, or VARCHAR. In the case of CHAR or VARCHAR, you must specify a character string literal that adheres to the predefined input representation formats. For details about the predefined input representations, see 6.3.3 Predefined character-string representations.
- You cannot specify a dynamic parameter by itself for the target data.

The following example illustrates the result of executing the scalar function DAYOFWEEK.

Example

Return an integer value indicating the day of the week that September 12, 2012 falls on.

DAYOFWEEK(DATE'2012-09-12') → 4

September 12, 2012 is a Wednesday, so it returns 4.

(3) Rules

1. The data type of the execution result is INTEGER.
2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If the target data has the null value, the execution result will be a null value.
4. The relationship between the value of the execution result and the day of the week is shown in the following table.

Table 8-25: Relationship between the value of the execution result and the day of the week

Value of execution result	Day of week
1	Sunday
2	Monday
3	Tuesday
4	Wednesday
5	Thursday
6	Friday
7	Saturday

(4) Example

Example:

Return an integer value that indicates the day of the week for the data in column C2 in table T1.

```
SELECT "C1",DAYOFWEEK("C2") FROM "T1"
```

Table T1

Column C1 CHAR	Column C2 DATE
A001	2011-12-28
A002	2012-01-21
A003	2012-02-23

Retrieval results

A001	4
A002	7
A003	5

8.9.3 DAYOFYEAR

Returns the specified date as the number of days elapsed since January 1 of that year.

(1) Specification format

```
scalar-function-DAYOFYEAR ::= {DAYOFYEAR|DOY}(target-data)

  target-data ::= value-expression
```

Note: DOY can be used as an abbreviated form for DAYOFYEAR.

(2) Explanation of specification format

target-data:

Specifies the data representing the day.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- The data type of the target data must be DATE, TIMESTAMP, CHAR, or VARCHAR. In the case of CHAR or VARCHAR, you must specify a character string literal that adheres to the predefined input representation formats. For details about the predefined input representations, see 6.3.3 Predefined character-string representations.
- You cannot specify a dynamic parameter by itself for the target data.

The following example illustrates the result of executing the scalar function DAYOFYEAR.

Example

Return the number of days elapsed in the year associated with the date January 15, 2013.

DAYOFYEAR(DATE'2013-01-15') → 15

(3) Rules

1. The value of the execution result will be an integer value from 1 to 366 representing the number of days elapsed since January 1 of that year.
2. The data type of the execution result will be `INTEGER`.
3. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
4. If the target data has the null value, the execution result will be a null value.

(4) Example

Example:

Determine the number of days elapsed in the year associated with target data in column `C2` of table `T1`.

```
SELECT "C1",DAYOFYEAR("C2") FROM "T1"
```

Table `T1`

Column `C1` `CHAR`	Column `C2` `DATE`
A001	2013-01-03
A002	2013-02-01
A003	2013-12-31
A004	2012-12-31

Retrieval results

A001	3
A002	32
A003	365
A004	366

8.9.4 EXTRACT

Extracts a part (year, month, day, hour, minute, or second) from data representing the date and time.

(1) Specification format

```
scalar-function-EXTRACT ::= EXTRACT(extraction-part FROM source-data)

  extraction-part ::= {YEAR|MONTH|DAY|HOUR|MINUTE|SECOND}
  source-data ::= value-expression
```

(2) Explanation of specification format

extraction-part:

Specifies the part to be extracted from the source data. Specify one of the values listed below. Note that `HOUR`, `MINUTE`, and `SECOND` can be specified only when the source data contains data that represents time.

- `YEAR`

  Specify this to extract the year part from the source data. The range of values of the execution result is `1` to `9999`.

- `MONTH`

Specify this to extract the month part from the source data. The range of values of the execution result is 1 to 12.

- DAY

  Specify this to extract the day part from the source data. The range of values of the execution result is 1 to 31.

- HOUR

  Specify this to extract the hour part from the source data. The range of values of the execution result is 0 to 23.

- MINUTE

  Specify this to extract the minute part from the source data. The range of values of the execution result is 0 to 59.

- SECOND

  Specify this to extract the second part from the source data. The range of values of the execution result varies depending on the fractional seconds precision of the source data, as shown in the following table.

Table 8-26: Range of values of the execution result of the scalar function EXTRACT (when SECOND is specified as the extraction part)

Fractional seconds precision of the source data	Range of values of the execution result
0	0 to 59
3	0.000 to 59.999
6	0.000000 to 59.999999
9	0.000000000 to 59.999999999
12	0.000000000000 to 59.999999999999

source-data:

Specifies the source data to be extracted from.

The following rules apply:

- Specify the source data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- If you specify YEAR, MONTH, or DAY for *extraction-part,* the data type of *source-data* must be DATE, TIMESTAMP, CHAR, or VARCHAR. In the case of CHAR or VARCHAR, you must specify a character string literal that adheres to the format of the predefined input representation of a date or time stamp. For details about predefined input representations, see 6.3.3 Predefined character-string representations.
- If you specify HOUR, MINUTE, or SECOND for *extraction-part*, the data type of *source-data* must be TIME, TIMESTAMP, CHAR, or VARCHAR. In the case of CHAR or VARCHAR, you must specify a character string literal that adheres to the format of the predefined input representation of a time or time stamp. For details about predefined input representations, see 6.3.3 Predefined character-string representations.
- You cannot specify a dynamic parameter by itself for the source data.

The following example illustrates the result of executing the scalar function EXTRACT.

Example

Extract the year part of from the DATE type data DATE'2012-03-15'.

```
EXTRACT(YEAR FROM DATE'2012-03-15') → 2012
```

(3) Rules

1. If you specify anything other than SECOND for *extraction-part*, the data type of the execution result will be INTEGER.
2. If you specify SECOND for *extraction-part*, the data type of the execution result varies depending on the fractional seconds precision of the source data, as shown in the following table.

Table 8-27: Data type of the execution result of the scalar function EXTRACT (when SECOND is specified as the extraction part)

Fractional seconds precision of the source data	Data type of the execution result
0	INTEGER
3	DECIMAL(5,3)
6	DECIMAL(8,6)
9	DECIMAL(11,9)
12	DECIMAL(14,12)

3. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
4. If the source data has a null value, the execution result will be a null value.

(4) Examples

Example 1:

Retrieve data from table T1 where the year in column C2 is 2012.

```
SELECT "C1","C2" FROM "T1"
    WHERE EXTRACT(YEAR FROM "C2")=2012
```

Table T1

Column C1 CHAR	Column C2 DATE
A001	2011-12-28
A002	2012-01-21
A003	2012-02-23

Retrieval results

A002	2012-01-21
A003	2012-02-23

Example 2:

Delete all of the rows from table T1 where the month in column C2 is not March.

```
DELETE FROM "T1"
    WHERE EXTRACT(MONTH FROM "C2")<>3
```

Table T1

Column C1	Column C2
CHAR	DATE
A001	2012-01-15
A002	2012-02-21
A003	2012-03-09

Retrieval results

A003	2012-03-09

8.9.5 GETAGE

Determines a person's age on a reference date given their birth date.

(1) Specification format

```
scalar-function-GETAGE ::= GETAGE(birth-date,reference-date)

  birth-date ::= value-expression
  reference-date ::= value-expression
```

(2) Explanation of specification format

birth-date:

Specifies the person's birth date.

The following rules apply:

- Specify *birth-date* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- The data type of *birth-date* must be DATE, TIMESTAMP, CHAR, or VARCHAR. In the case of CHAR or VARCHAR, you must specify a character string literal that adheres to the format of the predefined input representation of a date or time stamp. For details about predefined input representations, see 6.3.3 Predefined character-string representations.
- If a dynamic parameter is specified by itself for *birth-date*, the assumed data type of the dynamic parameter is DATE.

reference-date:

Specifies the reference date for calculating the person's age.

The following rules apply:

- Specify *reference-date* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- The data type of *reference-date* must be DATE, TIMESTAMP, CHAR, or VARCHAR. In the case of CHAR or VARCHAR, you must specify a character string literal that adheres to the format of the predefined input representation of a date or time stamp. For details about predefined input representations, see 6.3.3 Predefined character-string representations.
- If a dynamic parameter is specified by itself for *reference-date*, the assumed data type of the dynamic parameter is DATE.

The following example illustrates the result of executing the scalar function GETAGE.

Example

Determine the age on September 30, 2014 of a person born on January 15, 1986.

`GETAGE(DATE'1986-01-15',DATE'2014-09-30')` → `28`

(3) Rules

1. The data type of the execution result is `INTEGER`.
2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If *birth-date* or *reference-date* is the null value, the execution result will be the null value.
4. If *reference-date* is earlier than *birth-date*, the execution result will be `0`.
5. The scalar function `GETAGE` returns a person's age on the reference date. The same day one year after the birth date counts as 1 year old. Note that a birth date of February 29 is treated as March 1 in non-leap years.

(4) Example

Example:

Using the data in the employees table (`EMPLIST`), determine the number of employees 30 years of age or older as of January 1, 2015. The column `BIRTH` holds the employees' birth dates.

```
SELECT COUNT(*) FROM "EMPLIST"
    WHERE GETAGE("BIRTH",DATE'2015-01-01')>=30
```

Retrieval results

```
         247
```

8.9.6 LASTDAY

Returns the date or datetime of the last day of the month specified in the datetime data.

(1) Specification format

```
scalar-function-LASTDAY ::= {LASTDAY|LAST_DAY}(datetime-data)

  datetime-data ::= value-expression
```

(2) Explanation of specification format

datetime-data:

Specifies the datetime data to be processed.

The following rules apply:

- Specify *datetime-data* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- The data type of *datetime-data* must be `DATE`, `TIMESTAMP`, `CHAR`, or `VARCHAR`. In the case of `CHAR` or `VARCHAR`, you must specify a character string literal that adheres to the format of the predefined input representation of a date or time stamp. For details about predefined input representations, see 6.3.3 Predefined character-string representations.

- You cannot specify a dynamic parameter by itself for *datetime-data*.

(3) Rules

1. The data type of the execution result will be as follows.
   - If *datetime-data* is a DATE type or a predefined input representation of a date (CHAR or VARCHAR type), the data type of the execution result will be DATE.
   - If *datetime-data* is a TIMESTAMP type or a predefined input representation of a time stamp (CHAR or VARCHAR type), the data type of the execution result will be TIMESTAMP.
2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If *datetime-data* is the null value, the execution result will be the null value.
4. If *datetime-data* is a TIMESTAMP type or a predefined input representation of a time stamp (CHAR or VARCHAR type), the values that were entered for the hours, minutes, seconds, and fractional seconds parts are returned unchanged.

   Example

   LASTDAY(TIMESTAMP'2014-07-03 15:30:45.123') → '2014-07-31 15:30:45.123'

(4) Example

Example:

Determine the date of the last day of the month of the datetime data in column C2 of table T1.

```
SELECT "C1",LASTDAY("C2") FROM "T1"
```

Table T1

Column C1 CHAR	Column C2 DATE
A001	2013-01-03
A002	2013-02-01
A003	2012-02-01

Retrieval results

A001	2013-01-31
A002	2013-02-28
A003	2012-02-29

8.9.7 ROUND

Return the datetime data rounded to the unit specified in the datetime format.

For the scalar function ROUND that is used to round numeric data, see 8.4.9 ROUND.

(1) Specification format

```
scalar-function-ROUND ::= ROUND(datetime-data,datetime-format)

  datetime-data ::= value-expression
  datetime-format ::= literal
```

(2) Explanation of specification format

datetime-data:

Specifies the datetime data to be rounded.

The following rules apply:

- Specify the datetime data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- The data type of the datetime data must be `DATE`, `TIME`, or `TIMESTAMP`.
- You cannot specify a dynamic parameter by itself for the datetime data.

datetime-format:

Specifies the units of the datetime data to be rounded.

The following rules apply:

- Specify a character string literal for *datetime-format*. For details about character string literals, see 6.3 Literals.
- The following table shows the elements that can be specified in the datetime format.

Table 8-28: Elements that can be specified in the datetime format

No.	Element that can be specified in the datetime format	Unit	Description

1	`CC`	Century	If the datetime data is on or after the 51st year of the century, it is rounded up to January 1 00:00:00 of the first year of the next century. If it is on or before the 50th year, it is rounded down to January 1 00:00:00 of the first year of the same century.
			• Example of rounding up
			`ROUND(TIMESTAMP'1951-10-04 15:25:38','CC')`
			→ `TIMESTAMP'2001-01-01 00:00:00'`
			• Example of rounding down
			`ROUND(TIMESTAMP'1950-10-04 15:25:38','CC')`
			→ `TIMESTAMP'1901-01-01 00:00:00'`

2	`YYYY`
	`YYYYN`
	`YY`
	`YYN`
		Year	If the datetime data is on or after July 1, it is rounded up to January 1 00:00:00 of the next year. If it is on or before June 30, it is rounded down to January 1 00:00:00 of the same year.
			• Example of rounding up
			`ROUND(TIMESTAMP'2013-07-01 15:25:38','YYYY')`
			→ `TIMESTAMP'2014-01-01 00:00:00'`
			• Example of rounding down
			`ROUND(TIMESTAMP'2013-06-30 15:25:38','YYYY')`
			→ `TIMESTAMP'2013-01-01 00:00:00'`

3	`Q`	Quarter	If the datetime data is on or after the 16th of the second month of the quarter (February, May, August, or November), it is rounded up to 00:00:00 on the first day of the first month of the next quarter. If it is on or before the 15th, it is rounded down to 00:00:00 on the first day of the first month of the same quarter.
			• Example of rounding up
			`ROUND(TIMESTAMP'2013-11-16 15:25:38','Q')`
			→ `TIMESTAMP'2014-01-01 00:00:00'`
			• Example of rounding down
			`ROUND(TIMESTAMP'2013-11-15 15:25:38','Q')`
			→ `TIMESTAMP'2013-10-01 00:00:00'`

No.	Element that can be specified in the datetime format	Unit	Description
			Quarters are assumed to be three months long, starting on January 1. • First quarter: January 1 - March 31 • Second quarter: April 1 - June 30 • Third quarter: July 1 - September 30 • Fourth quarter: October 1 - December 31

4	`MONTH`
	`MON`
	`MM`
		Month	If the datetime data is on or after the 16th, it is rounded up to 00:00:00 on the first day of the next month. If it is on or before the 15th, it is rounded down to 00:00:00 on the first day of the same month.
			• Example of rounding up
			`ROUND(TIMESTAMP'2014-01-16 15:25:38','MONTH')`
			→ `TIMESTAMP'2014-02-01 00:00:00'`
			• Example of rounding down
			`ROUND(TIMESTAMP'2014-01-15 15:25:38','MONTH')`
			→ `TIMESTAMP'2014-01-01 00:00:00'`

5	`WW`	Week	The first day of the week is assumed to be the day of the week of the first day of the same year.
			If the datetime data is on or after 12:00 noon on the fourth day from the start of the week, it is rounded up to 00:00:00 on the first day of the next week. If it is before 12:00 noon on the fourth day, it is rounded down to 00:00:00 on the first day of the same week.
			• Example of rounding up
			`ROUND(TIMESTAMP'2014-01-04 15:25:38','WW')`
			→ `TIMESTAMP'2014-01-08 00:00:00'`
			Because January 1, 2014 falls on a Wednesday, the first day of the week is taken to be Wednesday. The example is on or after 12:00 noon on the fourth day of that week (Saturday January 4), and so is rounded up to 00:00:00 on the first day of the next week (January 8).
			• Example of rounding down
			`ROUND(TIMESTAMP'2014-01-04 10:25:38','WW')`
			→ `TIMESTAMP'2014-01-01 00:00:00'`
			Because January 1, 2014 falls on a Wednesday, the first day of the week is taken to be Wednesday. The example is before 12:00 noon on the fourth day of that week (Saturday January 4), and so is rounded down to 00:00:00 on the first day of the same week (January 1).

6	`W`	Week	The first day of the week is assumed to be the day of the week of the first day of the same month.
			If the datetime data is on or after 12:00 noon on the fourth day from the start of the week, it is rounded up to 00:00:00 on the first day of the next week. If it is before 12:00 noon on the fourth day, it is rounded down to 00:00:00 on the first day of the same week.
			• Example of rounding up
			`ROUND(TIMESTAMP'2014-02-04 12:25:38','W')`
			→ `TIMESTAMP'2014-02-08 00:00:00'`
			Because February 1, 2014 falls on a Saturday, the first day of the week is taken to be Saturday. The example is on or after 12:00 noon on the fourth day of that week (Tuesday, February 4), and so is rounded up to 00:00:00 on the first day of the next week (February 8).
			• Example of rounding down
			`ROUND(TIMESTAMP'2014-02-04 11:55:38','W')`
			→ `TIMESTAMP'2014-02-01 00:00:00'`
			Because February 1, 2014 falls on a Saturday, the first day of the week is taken to be Saturday. The example is before 12:00 noon on the fourth

No.	Element that can be specified in the datetime format	Unit	Description
			day of that week (Tuesday, February 4), and so is rounded down to 00:00:00 on the first day of the same week (February 1).

7	`DAY`
	`DAYN`
	`DY`
	`DYN`
	`D`
		Week	The first day of the week is defined as Sunday.
			If the datetime data is on or after 12:00 noon on the fourth day (Wednesday) from the start of the week, it is rounded up to 00:00:00 on the first day of the next week. If it is before 12:00 noon on the fourth day, it is rounded down to 00:00:00 on the first day of the same week.
			• Example of rounding up
			`ROUND(TIMESTAMP'2014-02-05 12:25:38','DAY')`
			→ `TIMESTAMP'2014-02-09 00:00:00'`
			February 5, 2014 falls on a Wednesday. Therefore, it is rounded up to 00:00:00 on February 9 (Sunday).
			• Example of rounding down
			`ROUND(TIMESTAMP'2014-02-05 11:55:38','DAY')`
			→ `TIMESTAMP'2014-02-02 00:00:00'`
			February 5, 2014 falls on a Wednesday. Therefore, it is rounded down to 00:00:00 on February 2 (Sunday).

8	`DD`
	`DDD`
		Day	If the datetime data is on or after 12:00 noon, it is rounded up to 00:00:00 on the next day. If it is before 12:00 noon, it is rounded down to 00:00:00 on the same day.
			• Example of rounding up
			`ROUND(TIMESTAMP'2014-01-16 15:25:38','DD')`
			→ `TIMESTAMP'2014-01-17 00:00:00'`
			• Example of rounding down
			`ROUND(TIMESTAMP'2014-01-16 10:25:38','DD')`
			→ `TIMESTAMP'2014-01-16 00:00:00'`

9	`HH`
	`HH12`
	`HH24`
		Hour	If the datetime data is on or after the 30 minute mark, it is rounded up to the beginning of the next hour. If it is on or before the 29 minute mark, it is rounded down to the beginning of the same hour.
			• Example of rounding up
			`ROUND(TIMESTAMP'2014-01-16 15:35:38','HH')`
			→ `TIMESTAMP'2014-01-16 16:00:00'`
			• Example of rounding down
			`ROUND(TIMESTAMP'2014-01-16 15:25:38','HH')`
			→ `TIMESTAMP'2014-01-16 15:00:00'`

10	`MI`	Minute	If the datetime data is on or after the 30 second mark, it is rounded up to the beginning of the next minute. If it is on or before the 29 second mark, it is rounded down to the beginning of the same minute.
			• Example of rounding up
			`ROUND(TIMESTAMP'2014-01-16 15:35:33','MI')`
			→ `TIMESTAMP'2014-01-16 15:36:00'`
			• Example of rounding down
			`ROUND(TIMESTAMP'2014-01-16 15:35:28','MI')`
			→ `TIMESTAMP'2014-01-16 15:35:00'`

11	`SSSSS` `SS`	Second	If the datetime data is on or after the 500 millisecond mark, it is rounded up to the beginning of the next second. If it is before the 500 millisecond mark, it is rounded down to the beginning of the same second. • Example of rounding up `ROUND(TIME'11:59:30.596123','SS')`

No.	Element that can be specified in the datetime format	Unit	Description
			→ TIME'11:59:31.000000' • Example of rounding down ROUND(TIME'11:59:30.488123','SS') → TIME'11:59:30.000000'

- The datetime format must be specified as single-byte character string data. Uppercase and lowercase letters are treated the same.
- In the cases where multiple datetime format elements are listed, the execution results will be the same regardless of which alternative is specified. For example, you can specify YYYY or YYYYN and the execution results will be the same.
- Spaces before and after the datetime format element are ignored.
- The length of the datetime format cannot exceed 64 bytes.

(3) Rules

1. The data type and data length of the execution result are shown in the following table.

Table 8-29: Data type and data length of the execution result of the scalar function ROUND

Data type and data length of the datetime data	Data type and data length of the execution result
DATE	DATE
TIME(*p*)	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP(*p*)

Legend: *p*: Fractional seconds precision

2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If the datetime data has a null value, the execution result will be a null value.
4. When the type of *datetime-data* is DATE but the datetime format is an element for rounding based on time within the day (DDD, DD, HH, HH12, HH24, MI, SSSSS, or SS), the original datetime data is returned unchanged.
5. When you specify TIME type data as the datetime data, you cannot specify the non-time elements in the datetime format (CC, YYYY, YYYYN, YY, YYN, Q, MONTH, MON, MM, WW, W, DAY, DAYN, DY, DYN, D, DDD, and DD).
6. When the type of *datetime-data* is DATE, 00:00:00 is assumed for the time elements. This is why the week is rounded down (not up) in the following example.

   Example

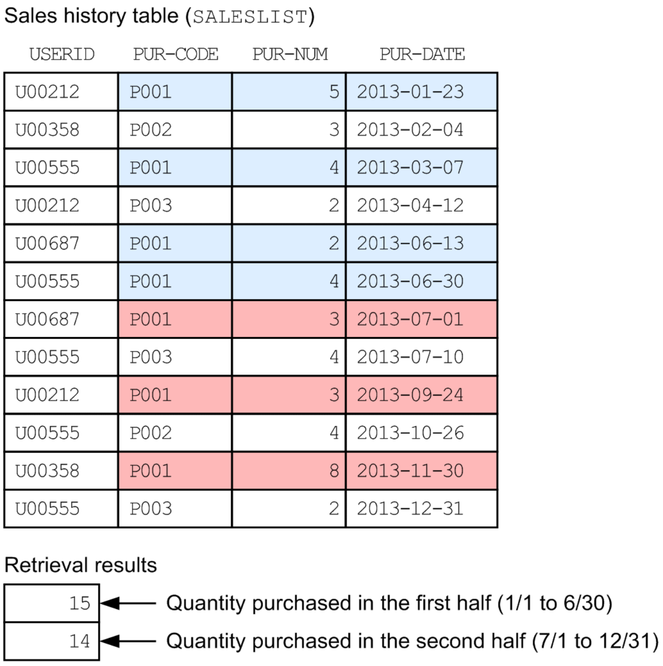
   ROUND(DATE'2013-10-04','W') → DATE'2013-10-01'
7. If the data type of the execution result is DATE, an error occurs if the execution result falls outside the range January 1, 0001 to December 31, 9999.
8. If the data type of the execution result is TIME, an error occurs if the execution result falls outside the range 00:00:00.000000000000 to 23:59:59.999999999999.
9. If the data type of the execution result is TIMESTAMP, an error occurs if the execution result falls outside the range January 1, 0001 00:00:00.000000000000 to December 31, 9999 23:59:59.999999999999.

(4) Example

Example:

From the sales history table (SALESLIST), retrieve the quantities purchased in 2013 of product code (PUR-CODE) P001, and group the results into six-month periods (January 1 to June 30 and July 1 to December 31).

```
SELECT SUM("PUR-NUM") FROM "SALESLIST"
    WHERE "PUR-DATE" BETWEEN DATE'2013-01-01' AND DATE'2013-12-31'
    AND "PUR-CODE"='P001'
    GROUP BY ROUND("PUR-DATE",'YYYY')
```



Sales history table (SALESLIST)

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P001	5	2013-01-23
U00358	P002	3	2013-02-04
U00555	P001	4	2013-03-07
U00212	P003	2	2013-04-12
U00687	P001	2	2013-06-13
U00555	P001	4	2013-06-30
U00687	P001	3	2013-07-01
U00555	P003	4	2013-07-10
U00212	P001	3	2013-09-24
U00555	P002	4	2013-10-26
U00358	P001	8	2013-11-30
U00555	P003	2	2013-12-31

Retrieval results

15
14

Quantity purchased in the first half (1/1 to 6/30)
Quantity purchased in the second half (7/1 to 12/31)

When element YYYY is specified for the datetime format, data from July 1 and later is rounded up, and data from June 30 and earlier is rounded down. The data can therefore be grouped into six-month periods, with 1/1 - 6/30 as the first half and 7/1 - 12/31 as the second half.

8.9.8 TRUNC

Returns the datetime data truncated to the unit specified in the datetime format.

For the scalar function TRUNC that is used to truncate numeric data, see 8.4.12 TRUNC.

(1) Specification format

```
scalar-function-TRUNC ::= TRUNC(datetime-data,datetime-format)

  datetime-data ::= value-expression
  datetime-format ::= literal
```

(2) Explanation of specification format

datetime-data:

Specifies the datetime data to be truncated.

The following rules apply:

- Specify the datetime data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- The data type of the datetime data must be `DATE`, `TIME`, or `TIMESTAMP`.
- You cannot specify a dynamic parameter by itself for the datetime data.

datetime-format:

Specifies the units of the datetime data to be truncated.

The following rules apply:

- Specify a character string literal for datetime format. For details about character string literals, see 6.3 Literals.
- The following table shows the elements that can be specified in the datetime format.

Table 8-30: Elements that can be specified in the datetime format

No.	Element that can be specified in the datetime format	Unit	Description
1	`CC`	Century	The datetime data is rounded down to January 1 00:00:00 of the first year of the same century. **Example** `TRUNC(TIMESTAMP'2014-03-14 15:25:38','CC')` → `TIMESTAMP'2001-01-01 00:00:00'`

2	`YYYY`
	`YYYYN`
	`YY`
	`YYN`
		Year	The datetime data is rounded down to January 1 00:00:00 of the same year.
			Example
			`TRUNC(TIMESTAMP'2014-03-14 15:25:38','YYYY')`
			→ `TIMESTAMP'2014-01-01 00:00:00'`

3	`Q`	Quarter	The datetime data is rounded down to 00:00:00 on the first day of the first month of the same quarter. **Example** `TRUNC(TIMESTAMP'2014-03-14 15:25:38','Q')` → `TIMESTAMP'2014-01-01 00:00:00'` Quarters are assumed to be three months long, starting on January 1. • First quarter: January 1 to March 31 • Second quarter: April 1 to June 30 • Third quarter: July 1 to September 30 • Fourth quarter: October 1 to December 31

4	`MONTH`
	`MON`
	`MM`
		Month	The datetime data is rounded down to 00:00:00 on the first day of the same month.
			Example
			`TRUNC(TIMESTAMP'2014-03-14 15:25:38','MONTH')`
			→ `TIMESTAMP'2014-03-01 00:00:00'`

5	`WW`	Week	The datetime data is rounded down to 00:00:00 on the first day of the same week. The first day of the week is assumed to be the day of the week of the first day of the same year. **Example** `TRUNC(TIMESTAMP'2014-03-14 15:25:38','WW')` → `TIMESTAMP'2014-03-12 00:00:00'` Because January 1, 2014 falls on a Wednesday, the first day of the week is taken to be Wednesday. The datetime data is therefore rounded down to 00:00:00 on March 12, which is the first day (Wednesday) of that week.

No.	Element that can be specified in the datetime format	Unit	Description
6	`W`	Week	The datetime data is rounded down to 00:00:00 on the first day of the same week. The first day of the week is assumed to be the day of the week of the first day of the same month. **Example** `TRUNC(TIMESTAMP'2014-03-14 15:25:38','W')` → `TIMESTAMP'2014-03-08 00:00:00'` Because March 1, 2014 falls on a Saturday, the first day of the week is taken to be Saturday. The datetime data is therefore rounded down to 00:00:00 on March 8, which is the first day (Saturday) of that week.

7	`DAY`
	`DAYN`
	`DY`
	`DYN`
	`D`
		Week	The datetime data is rounded down to 00:00:00 on the first day of the same week. The first day of the week is assumed to be Sunday.
			Example
			`TRUNC(TIMESTAMP'2014-03-14 15:25:38','DAY')`
			→ `TIMESTAMP'2014-03-09 00:00:00'`
			March 14, 2014 falls on a Friday. The datetime data is therefore rounded down to 00:00:00 on March 9, which is the first day (Sunday) of that week.

8	`DD` `DDD`	Day	The datetime data is rounded down to 00:00:00 on the same day. **Example** `TRUNC(TIMESTAMP'2014-03-14 15:25:38','DD')` → `TIMESTAMP'2014-03-14 00:00:00'`

9	`HH`
	`HH12`
	`HH24`
		Hour	The datetime data is rounded down to the beginning of the same hour.
			Example
			`TRUNC(TIMESTAMP'2014-03-14 15:25:38','HH')`
			→ `TIMESTAMP'2014-03-14 15:00:00'`

10	`MI`	Minute	The datetime data is rounded down to the beginning of the same minute. **Example** `TRUNC(TIMESTAMP'2014-03-14 15:25:38','MI')` → `TIMESTAMP'2014-03-14 15:25:00'`
11	`SSSSS` `SS`	Second	The datetime data is rounded down to the beginning of the same second. **Example** `TRUNC(TIME'11:58:31.784','SS')` → `TIME'11:58:31.000'`

- The datetime format must be specified as single-byte character string data. Uppercase and lowercase letters are treated the same.
- In the cases where multiple datetime format elements are listed, specify one of them. The execution results will be the same regardless of which alternative is specified. For example, you can specify `YYYY` or `YYYYN` and the execution results will be the same.
- Spaces before and after the datetime format element are ignored.
- The length of the datetime format cannot exceed 64 bytes.

(3) Rules

1. The data type and data length of the execution result are shown in the following table.

Table 8-31: Data type and data length of the execution result of the scalar function TRUNC

Data type and data length of the datetime data	Data type and data length of the execution result
DATE	DATE
TIME(*p*)	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP(*p*)

Legend: *p*: Fractional seconds precision

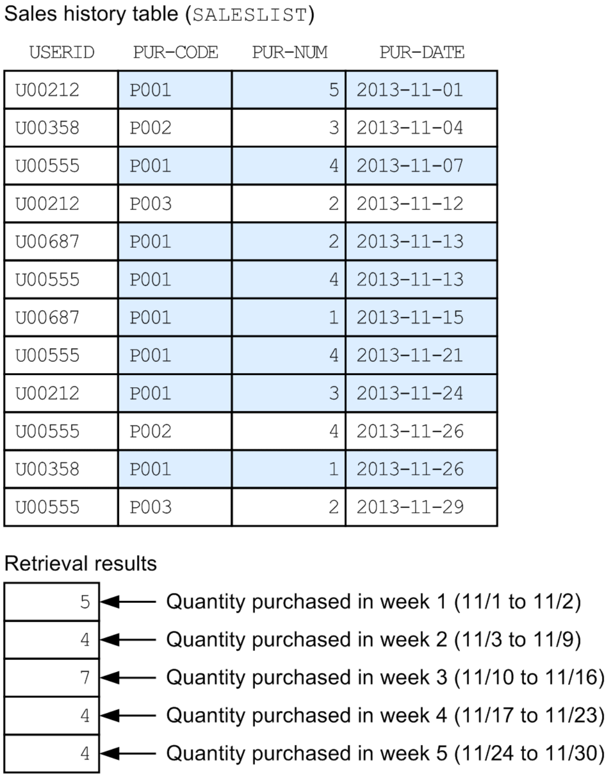
2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If the datetime data has a null value, the execution result will be a null value.
4. When the type of datetime data is DATE but the datetime format is an element for rounding based on the time of day (DDD, DD, HH, HH12, HH24, MI, SSSSS, or SS), the original datetime data is returned unchanged.
5. When you specify TIME type data as the datetime data, you cannot specify the non-time elements in the datetime format (CC, YYYY, YYYYN, YY, YYN, Q, MONTH, MON, MM, WW, W, DAY, DAYN, DY, DYN, D, DDD, and DD).
6. An error results if you specify DAY, DAYN, DY, DYN, or D in the datetime format and the execution result is earlier than January 1, 0001.

(4) Example

Example:

From the sales history table (SALESLIST), retrieve the quantities of product code (PUR-CODE) P001 purchased in November 2013 and group the results by week.

```
SELECT SUM("PUR-NUM") FROM "SALESLIST"
    WHERE "PUR-DATE" BETWEEN DATE'2013-11-01' AND DATE'2013-11-30'
    AND "PUR-CODE"='P001'
    GROUP BY TRUNC("PUR-DATE",'DAY')
```



Sales history table (SALESLIST)

USERID	PUR-CODE	PUR-NUM	PUR-DATE
U00212	P001	5	2013-11-01
U00358	P002	3	2013-11-04
U00555	P001	4	2013-11-07
U00212	P003	2	2013-11-12
U00687	P001	2	2013-11-13
U00555	P001	4	2013-11-13
U00687	P001	1	2013-11-15
U00555	P001	4	2013-11-21
U00212	P001	3	2013-11-24
U00555	P002	4	2013-11-26
U00358	P001	1	2013-11-26
U00555	P003	2	2013-11-29

Retrieval results

5	Quantity purchased in week 1 (11/1 to 11/2)
4	Quantity purchased in week 2 (11/3 to 11/9)
7	Quantity purchased in week 3 (11/10 to 11/16)
4	Quantity purchased in week 4 (11/17 to 11/23)
4	Quantity purchased in week 5 (11/24 to 11/30)

When you specify the element DAY for the datetime format, it groups the quantities purchased by week, using Sunday as the first day of the week.

8.10 Binary column functions (binary data operations)

This section describes the functions and specification formats of the binary column functions that operate on binary data.

8.10.1 CONCAT

Concatenates two binary data items.

For the scalar function that concatenates character data, see 8.5.1 CONCAT.

(1) Specification format

```
scalar-function-CONCAT ::= CONCAT(target-data-1,target-data-2)

  target-data-1 ::= value-expression
  target-data-2 ::= value-expression
```

(2) Explanation of specification format

target-data-1 and *target-data-2*:

Specifies the binary data to be concatenated.

The following rules apply:

- Specify *target-data-1* and *target-data-2* in the form of value expressions. For details about value expressions, see 7.20 Value expression.
- Specify `BINARY` or `VARBINARY` type data for *target-data-1* and *target-data-2*.
- You cannot specify a dynamic parameter by itself for *target-data-1* and *target-data-2*.

The following example illustrates the result of executing the scalar function `CONCAT`.

Example:

Concatenate two binary data items (`X'ABC1230000'` and `X'DEF456'`).

`CONCAT(X'ABC1230000',X'DEF456')` → `X'ABC1230000DEF456'`

(3) Rules

1. The data type and data length of the execution result are shown in the following table.

Table 8-32: Data type and data length of the execution result of the scalar function CONCAT

Data type and data length of target-data-1	Data type and data length of target-data-2	Data type and data length of the execution result
`BINARY(`*m*`)`	`BINARY(`*n*`)`	`BINARY(`*m* + *n*`)`
	`VARBINARY(`*n*`)` Actual data length: *L2*	`VARBINARY(`*m* + *n*`)` Actual data length: *m* + *L2*
`VARBINARY(`*m*`)` Actual data length: *L1*	`BINARY(`*n*`)`	`VARBINARY(`*m* + *n*`)` Actual data length: *L1* + *n*

Data type and data length of target-data-1	Data type and data length of target-data-2	Data type and data length of the execution result
	`VARBINARY(`*n*`)` Actual data length: *L2*	`VARBINARY(`*m* + *n*`)` Actual data length: *L1* + *L2*

Legend:

m: Maximum length of *target-data-1*

n: Maximum length of *target-data-2*

L1: Actual data length of *target-data-1*

L2: Actual data length of *target-data-2*

2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If either *target-data-1* or *target-data-2* has a null value, the execution result will be a null value.
4. You cannot concatenate *target-data-1* and *target-data-2* if the result of the concatenation operation would exceed the maximum binary data length of 32,000 bytes.

8.10.2 SUBSTRB

Extracts a substring from binary data starting from any position in the binary data.

(1) Specification format

```
scalar-function-SUBSTRB ::= SUBSTRB(source-binary-data, start-position[,number-of-byt
es-to-extract])

  source-binary-data ::= value-expression
  start-position ::= value-expression
  number-of-bytes-to-extract ::= value-expression
```

(2) Explanation of specification format

source-binary-data:

Specifies the source binary data.

The following rules apply:

- Specify the source binary data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify `BINARY` or `VARBINARY` type data for the source binary data.
- You cannot specify a dynamic parameter by itself for the source binary data.

start-position:

Specifies the starting byte position from which binary data is to be extracted.

If you specify a value greater than or equal to `0` for the start position, the value represents the position from the beginning of the source binary data. For example, if the start position is `2`, the extraction will start at the second byte.

If you specify a negative value for the start position, the value represents a position from the end of the source binary data. For example, if the start position is `-2`, the extraction will start at the second byte from the end.

The following rules apply:

- Specify the start position in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify an integer for the start position (`INTEGER` or `SMALLINT` type data).
- If you specify `0` for the start position, a start position of 1 is assumed.
- If a dynamic parameter is specified by itself for the start position, the assumed data type of the dynamic parameter will be `INTEGER`.

number-of-bytes-to-extract:

Specifies the length of the binary data to extract.

The following rules apply:

- Specify *number-of-bytes-to-extract* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify an integer greater than or equal to `0` (data of type `INTEGER` or `SMALLINT`) for *number-of-bytes-to-extract*.
- If you specify a dynamic parameter by itself for *number-of-bytes-to-extract*, the assumed data type of the dynamic parameter is `INTEGER`.

The following examples illustrate the result of executing the scalar function `SUBSTRB`.

Examples:

- Extract three bytes starting from the second byte from the beginning of the binary data `X'ABCDEF1234567890'`.

  `SUBSTRB(X'ABCDEF1234567890',2,3)` → `X'CDEF12'`

- Extract two bytes starting from the third byte from the end of the binary data `X'ABCDEF1234567890'`.

  `SUBSTRB(X'ABCDEF1234567890',-3,2)` → `X'5678'`

(3) Rules

1. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
2. In the following cases, the execution result will be a null value:
   - If *number-of-bytes-to-extract* has a negative value (the result will be the null value regardless of what is specified for the source binary data or the start position)
   - If the source binary data, start position, or number of bytes to extract is a null value
3. The data type and data length of the execution result are shown in the following table.

   Table 8-33: Data type and data length of the execution result of the scalar function SUBSTRB

Data type and data length of the source binary data	Data type and data length of the execution result
`BINARY(`*n*`)`	`VARBINARY(`*n*`)`
`VARBINARY(`*n*`)`	

   Legend:

   n: Maximum length of the source binary data
4. The following table shows the number of bytes of binary data that can be extracted by the scalar function `SUBSTRB`.

Table 8-34: Number of bytes of binary data that can be extracted by the scalar function SUBSTRB

Specification of the scalar function SUBSTRB		**Number of bytes of binary data that can be extracted**
Specification of number of bytes to extract	**Value specified for start position**	
Specified	Positive value	MAX{0, MIN (*number of bytes to extract*, *number of bytes in source binary data* - *start position* + 1)}
	0	MIN(*number of bytes to extract*, *number of bytes in source binary data*)
	Negative value	MIN(*number of bytes to extract*, *absolute value of the start position*, *number of bytes in source binary data*)
Omitted	Positive value	MAX(0, *number of bytes in source binary data* - *start position* + 1)
	0	*number of bytes in source binary data*
	Negative value	MIN(*absolute value of the start position*, *number of bytes in source binary data*)

5. In the following cases, the execution result will be data whose actual length is 0 bytes:
   - If the execution result is binary data of length 0
   - If the actual length of the source binary data is 0 bytes
   - If the specified start position satisfies either of the following inequalities:
     start position > *number of bytes in source binary data*
     start position < -*number of bytes in source binary data*

6. If the number of bytes in the source binary data, starting from the start position, is less than the number of bytes to extract, all of the source binary data, starting from the start position, is returned.

   Example:

   ```
   SUBSTRB(X'ABCDEF',2,5) → X'CDEF'
   ```

8.11 Binary column functions (bit operations)

This section describes the functions and specification formats of the binary column functions that perform bit operations.

8.11.1 BITAND

Returns the bitwise logical AND of two binary data items.

(1) Specification format

```
scalar-function-BITAND ::= BITAND(target-data-1,target-data-2)

  target-data-1 ::= value-expression
  target-data-2 ::= value-expression
```

(2) Explanation of specification format

target-data-1 and *target-data-2*:

Specifies the target binary data.

The following rules apply:

- Specify *target-data-1* and *target-data-2* in the form of value expressions. For details about value expressions, see 7.20 Value expression.
- Specify `BINARY` or `VARBINARY` type data for *target-data-1* and *target-data-2*.
- Make sure that *target-data-1* and *target-data-2* have the same data length (if the target data is `BINARY` type), or actual length (if the target data is `VARBINARY` type).
- You cannot specify a dynamic parameter by itself for *target-data-1*.
- If you specify a dynamic parameter by itself for *target-data-2*, the data type and data length of *target-data-1* are assumed for the data type and data length of the dynamic parameter.

The following example illustrates the result of executing the scalar function `BITAND`.

Examples:

Return the bitwise logical AND of two binary data items.

`BITAND(B'01011011',B'01001110')` → `B'01001010'`

`BITAND(B'01011011',X'FF')` → `B'01011011'`

`BITAND(X'0F',X'FF')` → `X'0F'`

(3) Rules

1. The execution result of the scalar function `BITAND` (the value of the *n*th bit) is shown in the following table.

Table 8-35: Execution result of the scalar function BITAND (value of the nth bit)

Value of the nth bit of target-data-1	Value of the nth bit of target-data-2	Execution result of the scalar function BITAND (value of the nth bit)
0	0	0
	1	0

Value of the nth bit of target-data-1	Value of the nth bit of target-data-2	Execution result of the scalar function BITAND (value of the nth bit)
`1`	`0`	`0`
	`1`	`1`

2. The data type and data length of the execution result are determined by the data types and data lengths of *target-data-1* and *target-data-2*. The data type and data length of the execution result of the scalar function `BITAND` are shown in the following table.

Table 8-36: Data type and data length of the execution result of the scalar function BITAND

Data type and data length of target-data-1	Data type and data length of target-data-2	Data type and data length of the execution result
`BINARY(`*m*`)`	`BINARY(`*m*`)`	`BINARY(`*m*`)`
	`VARBINARY(`*Y*`)` Actual length of target data: *m*	`VARBINARY(`*Y*`)` Actual length of target data: *m*
`VARBINARY(`*X*`)` Actual length of target data: *m*	`BINARY(`*m*`)`	`VARBINARY(`*X*`)` Actual length of target data: *m*
	`VARBINARY(`*Y*`)` Actual length of target data: *m*	`VARBINARY(`MAX(*X*,*Y*)`)` Actual length of target data: *m*

Legend:

m: Data length or actual length

X: Data length (when $X \geq m$)

Y: Data length (when $Y \geq m$)

3. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
4. If either *target-data-1* or *target-data-2* has a null value, the execution result will be a null value.
5. If the actual length of *target-data-1* and *target-data-2* is 0 bytes, the execution result will be binary data whose actual length is 0 bytes.

8.11.2 BITLSHIFT

Returns the value resulting from shifting the bits of a binary data value to the left.

(1) Specification format

```
scalar-function-BITLSHIFT ::= BITLSHIFT(target-data,number-of-bits-to-shift)

  target-data ::= value-expression
  number-of-bits-to-shift ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the target binary data.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify BINARY or VARBINARY type data for the target data.
- You cannot specify a dynamic parameter by itself for the target data.

number-of-bits-to-shift:

Specifies the number of bits to shift the binary data.

The following rules apply:

- Specify *number-of-bits-to-shift* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify an integer (data of type INTEGER or SMALLINT) for *number-of-bits-to-shift*.
- If you specify a positive value for *number-of-bits-to-shift*, the return value is the target data shifted to the left.
- If you specify a negative value for *number-of-bits-to-shift*, the return value is the target data shifted to the right.
- If you specify 0 for *number-of-bits-to-shift*, the same binary data as the target data is returned.
- If you specify a dynamic parameter by itself for *number-of-bits-to-shift*, the assumed data type of the dynamic parameter is INTEGER.

The following example illustrates the result of executing the scalar function BITLSHIFT.

Examples:

Return the specified binary data value with its bits shifted to the left.

```
BITLSHIFT(B'01011011',1) → B'10110110'
BITLSHIFT(B'01011011',8) → B'00000000'
BITLSHIFT(B'01011011',0) → B'01011011'
BITLSHIFT(X'0F0F',8) → X'0F00'
```

If you specify a negative value for *number-of-bits-to-shift*, the return value is the target data shifted to the right.

```
BITLSHIFT(B'01011011',-3) → B'00001011'
BITLSHIFT(X'0F0F',-16) → X'0000'
```

(3) Rules

1. The bits vacated by shifting are filled with B'0'.
2. If the data length of the target data (or the actual length if the target data is VARBINARY type) is m, and the number of bits to shift is n, the execution result of BITLSHIFT will be as shown in the following table.

Table 8-37: Execution result of BITLSHIFT

Conditions		Execution result of BITLSHIFT
$n > 0$	$8 \times m \leq n$	Returns binary data in which X'00' is set to the number of bytes in the data length of the target data (or actual length if the target data is VARBINARY type).
	$0 < n < 8 \times m$	Returns binary data in which the target data is shifted n bits to the left.
n=0		The same binary data as the target data is returned.
$n < 0$	$0 < \lvert n\rvert < 8 \times m$	Returns binary data in which the target data is shifted $\lvert n\rvert$ bits to the right.

Conditions		Execution result of BITLSHIFT
	Other than the above	Returns binary data in which X'00' is set to the number of bytes in the data length of the target data (or actual length if the target data is VARBINARY type).

3. The data type and data length of the execution result will be the data type and data length of the target data.
4. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
5. If either the target data or *number-of-bits-to-shift* is the null value, the execution result will be the null value.
6. If the target data is binary data whose actual length is 0 bytes, the execution result will be binary data whose actual length is 0 bytes.

8.11.3 BITNOT

Returns the bitwise logical NOT of a binary data item.

(1) Specification format

```
scalar-function-BITNOT ::= BITNOT(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the target binary data.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify BINARY or VARBINARY type data for the target data.
- You cannot specify a dynamic parameter by itself for the target data.

The following example illustrates the result of executing the scalar function BITNOT.

Examples:

Return the bitwise logical NOT of a binary data item.

BITNOT (B'01011011') → B'10100100'

BITNOT (B'11010001') → B'00101110'

BITNOT (X'0F') → X'F0'

(3) Rules

1. The data type and data length of the execution result will be the data type and data length of the target data.
2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If the target data has a null value, the execution result will be a null value.

4. If the target data is binary data whose actual length is 0 bytes, the execution result will be binary data whose actual length is 0 bytes.
5. The execution result of the scalar function BITNOT (value of the *n*th bit) is shown in the following table.

Table 8-38: Execution result of the scalar function BITNOT (value of the nth bit)

Value of the nth bit of the target data	Execution result of the scalar function BITNOT (value of the nth bit)
0	1
1	0

8.11.4 BITOR

Returns the bitwise inclusive OR of two binary data items.

(1) Specification format

```
scalar-function-BITOR ::= BITOR(target-data-1,target-data-2)

  target-data-1 ::= value-expression
  target-data-2 ::= value-expression
```

(2) Explanation of specification format

target-data-1 and *target-data-2*:

Specifies the target binary data.

The following rules apply:

- Specify *target-data-1* and *target-data-2* in the form of value expressions. For details about value expressions, see 7.20 Value expression.
- Specify BINARY or VARBINARY type data for *target-data-1* and *target-data-2*.
- Make sure that *target-data-1* and *target-data-2* have the same data length (if the target data is BINARY type), or actual length (if the target data is VARBINARY type).
- You cannot specify a dynamic parameter by itself for *target-data-1*.
- If you specify a dynamic parameter by itself for *target-data-2*, the data type and data length of *target-data-1* are assumed for the data type and data length of the dynamic parameter.

The following example illustrates the result of executing the scalar function BITOR.

Examples:

Return the bitwise inclusive OR of two binary data items.

BITOR(B'01011011',B'01001110') → B'01011111'

BITOR(B'01011011',X'FF') → B'11111111'

BITOR(X'0F',X'FF') → X'FF'

(3) Rules

1. The execution result of the scalar function BITOR (value of the *n*th bit) is shown in the following table.

Table 8-39: Execution result of the scalar function BITOR (value of the nth bit)

Value of the nth bit of target-data-1	Value of the nth bit of target-data-2	Execution result of the scalar function BITOR (value of the nth bit)
0	0	0
	1	1
1	0	1
	1	1

2. The data type and data length of the execution result are determined by the data types and data lengths of *target-data-1* and *target-data-2*. The data type and data length of the execution result of the scalar function `BITOR` are shown in the following table.

Table 8-40: Data type and data length of the execution result of the scalar function BITOR

Data type and data length of target-data-1	Data type and data length of target-data-2	Data type and data length of the execution result
`BINARY(`*m*`)`	`BINARY(`*m*`)`	`BINARY(`*m*`)`
	`VARBINARY(`*Y*`)` Actual length of target data: *m*	`VARBINARY(`*Y*`)` Actual length of target data: *m*
`VARBINARY(`*X*`)` Actual length of target data: *m*	`BINARY(`*m*`)`	`VARBINARY(`*X*`)` Actual length of target data: *m*
	`VARBINARY(`*Y*`)` Actual length of target data: *m*	`VARBINARY(`MAX(*X*,*Y*)`)` Actual length of target data: *m*

Legend:

m: Data length or actual length

X: Data length (when $X \geq m$)

Y: Data length (when $Y \geq m$)

3. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).

4. If either *target-data-1* or *target-data-2* has a null value, the execution result will be a null value.

5. If the actual length of *target-data-1* and *target-data-2* is 0 bytes, the execution result will be binary data whose actual length is 0 bytes.

8.11.5 BITRSHIFT

Returns the value resulting from shifting the bits of a binary data value to the right.

(1) Specification format

```
scalar-function-BITRSHIFT ::= BITRSHIFT(target-data,number-of-bits-to-shift)

  target-data ::= value-expression
  number-of-bits-to-shift ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the target binary data.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify `BINARY` or `VARBINARY` type data for the target data.
- You cannot specify a dynamic parameter by itself for the target data.

number-of-bits-to-shift:

Specifies the number of bits to shift the binary data.

The following rules apply:

- Specify *number-of-bits-to-shift* in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify an integer (data of type `INTEGER` or `SMALLINT`) for *number-of-bits-to-shift*.
- If you specify a positive value for *number-of-bits-to-shift*, the return value is the target data shifted to the right.
- If you specify a negative value for *number-of-bits-to-shift*, the return value is the target data shifted to the left.
- If you specify 0 for *number-of-bits-to-shift*, the same binary data as the target data is returned.
- If you specify a dynamic parameter by itself for *number-of-bits-to-shift*, the assumed data type of the dynamic parameter is `INTEGER`.

The following example illustrates the result of executing the scalar function `BITRSHIFT`.

Examples:

Return the specified binary data value with its bits shifted to the right.

```
BITRSHIFT(B'01011011',1)→B'00101101'
BITRSHIFT(B'01011011',8)→B'00000000'
BITRSHIFT(B'01011011',0)→B'01011011'
BITRSHIFT(X'0F0F',8)→X'000F'
```

If you specify a negative value for *number-of-bits-to-shift*, the return value is the target data shifted to the left.

```
BITRSHIFT(B'01011011',-3)→B'11011000'
BITRSHIFT(X'0F0F',-16)→X'0000'
```

(3) Rules

1. The bits vacated by shifting are filled with `B'0'`.
2. If the data length of the target data (or the actual length if the target data is `VARBINARY` type) is *m*, and the number of bits to shift is *n*, the execution result of `BITRSHIFT` will be as shown in the following table.

Table 8-41: Execution result of BITRSHIFT

Conditions		Execution result of BITRSHIFT
$n > 0$	$8 \times m \leq n$	Returns binary data in which `X'00'` is set to the number of bytes in the data length of the target data (or actual length if the target data is `VARBINARY` type).

Conditions		Execution result of BITRSHIFT
	$0 < n < 8 \times m$	Returns binary data in which the target data is shifted *n* bits to the right.
n=0		The same binary data as the target data is returned.
$n < 0$	$0 < \lvert n \rvert < 8 \times m$	Returns binary data in which the target data is shifted $\lvert n \rvert$ bits to the left.
	Other than the above	Returns binary data in which `X'00'` is set to the number of bytes in the data length of the target data (or actual length if the target data is `VARBINARY` type).

3. The data type and data length of the execution result will be the data type and data length of the target data.
4. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
5. If either the target data or *number-of-bits-to-shift* is the null value, the execution result will be the null value.
6. If the target data is binary data whose actual length is 0 bytes, the execution result will be binary data whose actual length is 0 bytes.

8.11.6 BITXOR

Returns the bitwise exclusive OR of two binary data items.

(1) Specification format

```
scalar-function-BITXOR ::= BITXOR(target-data-1,target-data-2)

  target-data-1 ::= value-expression
  target-data-2 ::= value-expression
```

(2) Explanation of specification format

target-data-1 and *target-data-2*:

Specifies the target binary data.

The following rules apply:

- Specify *target-data-1* and *target-data-2* in the form of value expressions. For details about value expressions, see 7.20 Value expression.
- Specify `BINARY` or `VARBINARY` type data for *target-data-1* and *target-data-2*.
- Make sure that *target-data-1* and *target-data-2* have the same data length (if the target data is `BINARY` type), or actual length (if the target data is `VARBINARY` type).
- You cannot specify a dynamic parameter by itself for *target-data-1*.
- If you specify a dynamic parameter by itself for *target-data-2*, the data type and data length of *target-data-1* are assumed for the data type and data length of the dynamic parameter.

The following example illustrates the result of executing the scalar function `BITXOR`.

Examples:

Return the bitwise exclusive OR of two binary data items.

`BITXOR(B'01011011',B'01001110')` → `B'00010101'`

```
BITXOR(B'01011011',X'FF') → B'10100100'
BITXOR(X'0F',X'FF') → X'F0'
```

(3) Rules

1. The execution result of the scalar function `BITXOR` (value of the *n*th bit) is shown in the following table.

Table 8-42: Execution result of the scalar function BITXOR (value of the nth bit)

Value of the nth bit of target-data-1	Value of the nth bit of target-data-2	Execution result of the scalar function BITXOR (value of the nth bit)
0	0	0
	1	1
1	0	1
	1	0

2. The data type and data length of the execution result are determined by the data types and data lengths of *target-data-1* and *target-data-2*. The data type and data length of the execution result of the scalar function `BITXOR` are shown in the following table.

Table 8-43: Data type and data length of the execution result of the scalar function BITXOR

Data type and data length of target-data-1	Data type and data length of target-data-2	Data type and data length of the execution result
`BINARY(`*m*`)`	`BINARY(`*m*`)`	`BINARY(`*m*`)`
	`VARBINARY(`*Y*`)` Actual length of target data: *m*	`VARBINARY(`*Y*`)` Actual length of target data: *m*
`VARBINARY(`*X*`)` Actual length of target data: *m*	`BINARY(`*m*`)`	`VARBINARY(`*X*`)` Actual length of target data: *m*
	`VARBINARY(`*Y*`)` Actual length of target data: *m*	`VARBINARY(`MAX(*X*,*Y*)`)` Actual length of target data: *m*

Legend:

m: Data length or actual length

X: Data length (when $X \geq m$)

Y: Data length (when $Y \geq m$)

3. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
4. If either *target-data-1* or *target-data-2* has a null value, the execution result will be a null value.
5. If the actual length of *target-data-1* and *target-data-2* is 0 bytes, the execution result will be binary data whose actual length is 0 bytes.

8.12 Data conversion functions

This section describes the functions and specification formats of the data conversion functions.

8.12.1 ASCII

Returns the character code of the first character of the target data as an integer value.

(1) Specification format

```
scalar-function-ASCII ::= ASCII(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the target data.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify `CHAR` or `VARCHAR` type data for the target data.
- You cannot specify a dynamic parameter by itself for the target data.

The following example illustrates the result of executing the scalar function `ASCII`. The example assumes Unicode (UTF-8) as the character encoding.

Examples:

`ASCII('A')` → `65`

`ASCII('ABCD')` → `65`

`ASCII('Ⅱ')` → `14845345`

(3) Rules

1. The data type of the execution result is the `INTEGER` type.
2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. In either of the following cases, the execution result will be a null value:
    - If the target data is the null value
    - If the actual length of the target data is 0 bytes or 0 characters

(4) Example

Example:

From the character string data items in column `C1` of table `T1`, find the character string data items for which the first character falls in the range of ASCII codes.

```
SELECT "C1" FROM "T1" WHERE ASCII("C1")<128
```

Table T1

Col. C1 VARCHAR
A10101
Ⅱ3345
A3139922
Ⅲ84775

Retrieval results

A10101
A3139922

8.12.2 BIN

Converts binary data to a binary string representation (character string data consisting of `0` and `1`).

(1) Specification format

```
scalar-function-BIN ::= BIN(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the target binary data.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify `BINARY` or `VARBINARY` type data for the target data.
- You cannot specify a dynamic parameter by itself for the target data.
- You cannot specify binary data whose defined length is 4,001 bytes or greater for the target data.

The following example illustrates the result of executing the scalar function `BIN`.

Examples:

`BIN(B'10100100')` → `'10100100'`

`BIN(X'A4')` → `'10100100'`

(3) Rules

1. The data type and data length of the execution result are shown in the following table.

Table 8-44: Data type and data length of the execution result of the scalar function BIN

Data type and data length of target data			Data type and data length of the execution result		
Data type	**Defined length**	**Actual length**	**Data type**	**Defined length**	**Actual length**
`BINARY(`*n*`)`	$1 \leq n \leq 4{,}000$	Not applicable.	`VARCHAR`	$n \times 8$	$n \times 8$
`VARBINARY(`*n*`)`	$1 \leq n \leq 4{,}000$	*r*			$r \times 8$

Legend:

n: Defined length of target data

r: Actual length of target data

2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If the target data has a null value, the execution result will be a null value.
4. If the actual length of the target data is 0 bytes, the execution result will be data with an actual length of 0 bytes.

8.12.3 CAST

Converts the data type of the data.

(1) Specification format

```
scalar-function-CAST ::= CAST(data-to-convert AS post-conversion-data-type)

  data-to-convert ::= {value-expression|NULL}
  post-conversion-data-type ::= data-type
```

(2) Explanation of specification format

data-to-convert:

Specifies the data whose data type is to be converted.

Specify the data to be converted in the form of a value expression. Alternatively, specify `NULL`. For details about value expressions, see 7.20 Value expression.

post-conversion-data-type:

Specifies the data type after conversion. The following are examples:

- `INTEGER`

  Convert to `INTEGER` type data.

- `DECIMAL(5,2)`

  Convert to `DECIMAL` type data with a precision of 5 and a scaling of 2.

- `CHAR(8)`

  Convert to `CHAR` type data with a data length of 8 bytes.

For the specification formats of each data type, see 6.2.1 List of data types.

Note that you cannot specify a `VARCHAR` type whose data length exceeds 32,000 bytes for the post-conversion-data type.

The following example illustrates the result of executing the scalar function `CAST`.

Example:

Convert the `DECIMAL` type data `-12.37` to `INTEGER` type.

```
CAST(-12.37 AS INTEGER) → -12
```

(3) Rules

(a) Common rules

1. The data type of the execution result will be the data type specified in *post-conversion-data-type*.
2. If a dynamic parameter is specified by itself for *data-to-convert*, *post-conversion-data-type* will be assumed to be the data type of the dynamic parameter.
3. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
4. If the data to be converted has a null value, or you specify `NULL` for *data-to-convert*, the execution result will be a null value.
5. If the data to be converted is character string data with a length of 0 bytes or 0 characters, it is converted as follows:
   - When converting to `CHAR` type: It is converted to spaces. For example, in the case of `CHAR(3)`, it is converted to `'ΔΔΔ'`. Δ represents a half-width space.
   - When converting to `VARCHAR` type: It is converted to `VARCHAR` type data with a length of 0 bytes or 0 characters.
   - When converting to `BINARY` type: It is converted to `X'00'`. In the case of `BINARY(3)`, it is converted to `X'000000'`.
   - When converting to `VARBINARY` type: It is converted to `VARBINARY` type data with a length of 0 bytes or 0 characters.
   - In the case of other data types, it is converted to the null value.
6. The data types that can be converted are shown in the following table.

Table 8-45: Data types that can be converted

Data type of the data to be converted	Post-conversion data type					
	INTEGER, SMALLINT	DECIMAL, DOUBLE PRECISION	CHAR, VARCHAR	DATE, TIMESTAMP	TIME	BINARY, VARBINARY

`INTEGER,`
`SMALLINT`
	Y	Y	Y	Y	N	N

`DECIMAL,`
`DOUBLE PRECISION`
	Y	Y	Y	N	N	N

`CHAR,`
`VARCHAR`
	Y	Y	Y	Y	Y	Y

`DATE,`
`TIMESTAMP`
	Y	N	Y	Y	N	N

`TIME`	N	N	Y	N	Y	N

`BINARY,`
`VARBINARY`
	N	N	Y	N	N	Y

Legend:

Y: Can be converted.

N: Cannot be converted.

(b) Rules for converting numeric data

■ **To convert numeric data to numeric data:**

Conversion of numeric data to numeric data is governed by the rules described in Storage assignment of numeric data in (2) Storage assignments between data types in 6.2.2 Data types that can be converted, assigned, and compared.

■ **To convert character string data to numeric data:**

- Any character string data to be converted (after leading and trailing spaces are removed) must obey the rules for the description format of numeric literals. For the description format rules for numeric literals, see 6.3.2 Description format of literals.

  Examples of character string data that can be converted:

  `'219', '+56', '-3547', '-11.35', '887ΔΔ', 'Δ95Δ'`

  Examples of character string data that cannot be converted:

  `'a89', '77g9', '33Δ49'`

  Legend: Δ: Single-byte space
- If the character string data item is composed of only spaces, the null value is returned.
- Once the character string representation of the numeric literal has been converted to a numeric value, it is converted to the post-conversion data type. At that point, it is governed by the rules described in Storage assignment of numeric data in (2) Storage assignments between data types in 6.2.2 Data types that can be converted, assigned, and compared.

  Example:

  `CAST('11.35' AS INTEGER) → 11`

  Once the character string `'11.35'` has been converted to the `DECIMAL` type numeric value 11.35, it is converted to an `INTEGER` type numeric value. At that point, it is governed by the rules for storage assignment of numeric data, which in this case means that the decimal part is truncated.

■ **To convert datetime data to numeric data:**

Datetime data is converted to the cumulative number of days since January 1, year 1 (CE). In the case of January 1, year 1 (CE), the cumulative number of days is 1. In the case of January 2, year 1 (CE), the cumulative number of days is 2.

Examples:

`CAST(DATE'0001-01-03' AS INTEGER) → 3`

`CAST(TIMESTAMP'0001-01-05 11:03:58' AS INTEGER) → 5`

(c) Rules for converting to character string data

The rules for converting to character string data (rules about the length of data) are shown in the following table.

Table 8-46: Rules for converting to character string data (rules about the length of data)

Condition at the time of conversion	Rules for converting to character string data	
	If data of character string type or binary type is converted	If data of other types is converted
A < *B*	If the post-conversion data type is `CHAR`, it is left-aligned and padded with spaces on the right.	
A = *B*	The conversion is performed.	

Condition at the time of conversion	Rules for converting to character string data	
	If data of character string type or binary type is converted	If data of other types is converted
A > *B*	The data is left-aligned and the excess portion on the right is truncated.#1	The data cannot be converted. Conversion will result in an error.#2

Legend:

A: Length of the source data that is to be converted to character string data

B: Data length of the post-conversion data type

#1

If truncation occurs in the middle of a multi-byte character, part of the multi-byte character is returned as the value of the execution result.

#2

If the data type of the data to be converted is `DOUBLE PRECISION`, the number of decimal places of the mantissa is truncated to fit the data length specified in *post-conversion-data-type* (rounding to the nearest even number), so no error is generated. However, an error will be generated if the length of the data to be converted exceeds the data length specified in *post-conversion-data-type* even after all the decimal places of the mantissa have been truncated.

■ To convert `INTEGER`, `SMALLINT`, or `DECIMAL` type numeric data to character string data

- The result of converting numeric data to the format of a numeric literal is output as character string data. At that point, the results are output in the shortest format that can represent the numeric literal.

  However, conversion of `DECIMAL` type data is performed as follows:

  • The number of digits after the decimal point equals the scaling of the data type of the numeric data, and trailing zeros are not stripped.

  • If the precision of the data type of the numeric data is greater than the scaling, the number of digits in the integer part will not be 0.

  • The decimal point is always added.

  Example: `+0025.100` → `'25.100'`

  As shown in the example, the plus sign (+) is removed. In addition, any zeros are stripped from the beginning of the integer part.
- If the data to be converted is less than `0`, it is prefixed with a minus sign (`-`).

■ To convert `DOUBLE PRECISION` type numeric data to character string data

- The result of converting numeric data to the format of a floating-point numeric literal is output as character string data. At that point, the results are output in the shortest format that can represent the floating-point numeric literal.

  Examples:

  ```
  +1.0000000000000000E+010 → '1E10'
  +3.2000000000000000E+001 → '3.2E1'
  +0.1000000000000000E+001 → '1E0'
  +0.0000000000000000E+000 → '0E0'
  ```

  As shown in the examples, the sign is removed from the mantissa and any trailing zeros are removed from the decimal part. Also, the plus sign (+) and leading zeros are removed from the exponent.
- If the data to be converted is less than `0`, it is prefixed with a minus sign (`-`).
- Exponents that are less than `0` are prefixed with a minus sign (`-`).

■ To convert datetime data to character string data

- When datetime data is converted to character string data, it is converted to the format of the predefined output representation. When `DATE` type data is converted to character string data, it is converted to the format of the predefined output representation of a date. When `TIME` type data is converted to character string data, it is converted to the format of the predefined output representation of a time. When `TIMESTAMP` type data is converted to character string data, it is converted to the format of the predefined output representation of a time stamp. For details about the predefined output representations, see 6.3.3 Predefined character-string representations.

  Examples:

  ```
  CAST(DATE'2013-06-30' AS CHAR(10)) → '2013-06-30'
  CAST(DATE'0001-01-01' AS CHAR(10)) → '0001-01-01'
  CAST(TIME'05:33:48.123' AS CHAR(12)) → '05:33:48.123'
  CAST(TIMESTAMP'2013-06-30 11:03:58' AS CHAR(19)) → '2013-06-30 11:03:58'
  ```

- Conversions of datetime data to `CHAR(`*n*`)` or `VARCHAR(`*n*`)` must meet the following conditions:

Data type of the data to be converted		Condition on the post-conversion data length
`DATE`		$n \geq 10$
`TIME(`*p*`)`	when $p = 0$	$n \geq 8$
	when $p > 0$	$n \geq 9 + p$
`TIMESTAMP(`*p*`)`	when $p = 0$	$n \geq 19$
	when $p > 0$	$n \geq 20 + p$

  When *n* is less than the lengths indicated above, conversion is not possible.

- When converting `DATE` type data to `CHAR` type, if the data length of the post-conversion data is 11 bytes or greater, the results are left-aligned and padded with spaces on the right.

  Example:

  ```
  CAST(DATE'2013-06-30' AS CHAR(15)) → '2013-06-30ΔΔΔΔΔ'
  ```

  Legend: Δ: Single-byte space

- When converting `TIME` type data with fractional seconds precision *p* to `CHAR` type, if the data length of the post-conversion data is greater than or equal to 10 + *p* bytes (or greater than or equal to 9 bytes when *p* = 0), the results are left-aligned and padded with spaces on the right.

  Example:

  ```
  CAST(TIME'11:03:58.123' AS CHAR(13)) → '11:03:58.123Δ'
  ```

  Legend: Δ: Single-byte space

- When converting `TIMESTAMP` type data with fractional seconds precision *p* to `CHAR` type, if the data length of the post-conversion data is greater than or equal to 21 + *p* bytes (or greater than or equal to 20 bytes when *p* = 0), the results are left-aligned and padded with spaces on the right.

  Example:

  ```
  CAST(TIMESTAMP'2013-06-30 11:03:58' AS CHAR(20)) → '2013-06-30 11:03:58Δ'
  ```

  Legend: Δ: Single-byte space

▪ To convert binary data to character string data

- Only the data type is converted, and the data itself (character encoding itself) is not converted.

  Example:

  ```
  CAST(X'61626364' AS CHAR(4)) ==> 'abcd'
  ```

- If *length-of-data-before-type-conversion* > *length-of-data-after-type-conversion*, the excess portion on the right is truncated.

  Example:

  ```
  CAST(X'61626364' AS CHAR(3)) ==> 'abc'
  ```

  The underlined portion is truncated.

- If *length-of-data-before-type-conversion* < *length-of-data-after-type-conversion*, the results are padded with half-width spaces on the right.

  Example:

  ```
  CAST(X'61626364' AS CHAR(5)) ==> 'abcdΔ'
  ```

  Legend: Δ: Half-width space

(d) Rules for converting to datetime data

■ To convert `INTEGER` or `SMALLINT` type numeric data to datetime data

- The data is first converted to the cumulative number of days since January 1, year 1 (CE).
- The time portion of the `TIMESTAMP` type is converted to `00:00:00`, and fractional seconds are filled with zeros. The following shows examples.

  Example:

  ```
  CAST(2 AS DATE) → DATE'0001-01-02'
  CAST(2 AS TIMESTAMP(3)) → TIMESTAMP'0001-01-02 00:00:00.000'
  ```

- `INTEGER` and `SMALLINT` type data in the range 1 to 3,652,059 can be converted. Values outside this range generate an error.

■ To convert character string data to datetime data:

- The character string data to be converted (after leading and trailing spaces are removed) can be converted to `DATE` type data only when it adheres to the predefined input representation format of a date. For details about the predefined input representation of a date, see (a) Predefined input representation in (1) Predefined character-string representation of dates in 6.3.3 Predefined character-string representations.

  Example:

  ```
  CAST('2014-07-22ΔΔ' AS DATE) → DATE'2014-07-22'
  ```

  Examples of character string data that can be converted:

  ```
  '2014-06-30', '0001-01-02', 'ΔΔ2014-07-30', 'Δ2014/07/30ΔΔ'
  ```

  Examples of character string data that cannot be converted:

  ```
  '2013Δ06Δ30', '2013.06.30'
  ```

  Legend: Δ: Single-byte space

- The character string data to be converted (after leading and trailing spaces are removed) can be converted to `TIME` type data only when it adheres to the predefined input representation format of a time. For details about the predefined input representation of a time, see (a) Predefined input representation in (2) Predefined character-string representation of times in 6.3.3 Predefined character-string representations.

  Example:

  ```
  CAST('Δ19:46:23.123456' AS TIME(6)) → TIME'19:46:23.123456'
  ```

  Examples of character string data that can be converted:

  ```
  '18:05:22', '10:21:44.123', 'ΔΔ10:21:44.123456Δ'
  ```

  Examples of character string data that cannot be converted:

  ```
  '18Δ05Δ22', '10:21:44Δ123456'
  ```

  Legend: Δ: Single-byte space

- The character string data to be converted (after leading and trailing spaces are removed) can be converted to `TIMESTAMP` type data only when it adheres to the predefined input representation format of a time stamp. For details about the predefined input representation of a time stamp, see (a) Predefined input representation in (3) Predefined character-string representation of time stamps in 6.3.3 Predefined character-string representations.

  Example:

  `CAST('2014/08/02 11:03:58.123456Δ' AS TIMESTAMP(6)) → TIMESTAMP'2014-08-02 11:03:58.123456'`

  Examples of character string data that can be converted:

  `'2014-06-30 11:03:58', '2014/07/30 11:03:58.123', 'Δ2014/07/30 11:03:58.123456789ΔΔ'`

  Examples of character string data that cannot be converted:

  `'2014-06-30 11-03-58', '2014/07/30 11:03:58:123456'`

  Legend: Δ: Single-byte space

- If the number of digits in the fractional seconds of the character string data to be converted is greater than the number of digits in the fractional seconds of *post-conversion-data-type*, the fractional seconds beyond the number of digits in the fractional seconds of *post-conversion-data-type* are truncated.

  Example:

  `CAST('19:46:23.123456' AS TIME(3)) → TIME'19:46:23.123'`

- If the number of digits in the fractional seconds of the character string data to be converted is less than the number of digits in the fractional seconds of *post-conversion-data-type*, the fractional seconds are padded with zeros as necessary.

  Example:

  `CAST('2014-08-02 11:03:58.123' AS TIMESTAMP(9)) → TIMESTAMP'2014-08-02 11:03:58.123000000'`

- If the character string data item is composed of only spaces, the null value is returned.

■ **To convert datetime data to datetime data:**

The conversion rules for converting datetime data to datetime data are given in the following table.

Table 8-47: Conversion rules for converting datetime data to datetime data

Data type of the data to be converted	Specified post-conversion data type	Conversion rules
`DATE`	`DATE`	No conversion is performed.
	`TIMESTAMP(`*p2*`)`	• The time part is converted to `00:00:00`. • The fractional seconds are padded with zeros.
`TIME(`*p1*`)`	`TIME(`*p2*`)`	• When *p1* = *p2* No conversion is performed. • When *p1* > *p2* The fractional seconds beyond *p2* are truncated. • When *p1* < *p2* The missing fractional seconds are padded with zeros.
`TIMESTAMP(`*p1*`)`	`DATE`	Only the date part is converted.
	`TIMESTAMP(`*p2*`)`	• When *p1* = *p2* No conversion is performed. • When *p1* > *p2* The fractional seconds beyond *p2* are truncated. • When *p1* < *p2*

Data type of the data to be converted	Specified post-conversion data type	Conversion rules
		The missing fractional seconds are padded with zeros.

Legend:

p1, *p2*: Fractional seconds precision

(e) Rules for converting to binary data

■ To convert character string data to binary data

- Only the data type is converted, and the data itself (character encoding itself) is not converted.

  Example:

  `CAST('abcd' AS BINARY(4)) ==> X'61626364'`

- If *length-of-data-before-type-conversion* > *length-of-data-after-type-conversion*, the excess portion on the right is truncated.

  Example:

  `CAST('abcd' AS BINARY(3)) ==> X'616263'`

  The underlined portion is truncated.

  If truncation occurs in the middle of a multi-byte character, part of the multi-byte character is returned as the value of the execution result.

- If *length-of-data-before-type-conversion* < *length-of-data-after-type-conversion*, the results are padded with `X'00'` on the right.

  Example:

  `CAST('abcd' AS BINARY(5)) ==> X'6162636400'`

■ To convert binary data to binary data

- If *length-of-data-before-type-conversion* > *length-of-data-after-type-conversion*, the excess portion on the right is truncated.

  Example:

  `CAST(X'61626364' AS BINARY(3)) ==> X'616263'`

  The underlined portion is truncated.

  If truncation occurs in the middle of a multi-byte character, part of the multi-byte character is returned as the value of the execution result.

- If *length-of-data-before-type-conversion* < *length-of-data-after-type-conversion*, the results are padded with `X'00'` on the right.

  Example:

  `CAST(X'61626364' AS BINARY(5)) ==> X'6162636400'`

(4) Example

Example:

Convert the data in column `C2` in table `T1` from `TIMESTAMP` type to `DATE` type and retrieve the rows where column `C2` is July 21, 2013.

```
SELECT * FROM "T1"
    WHERE CAST("C2" AS DATE)=DATE'2013-07-21'
```

Table T1

Column C1 CHAR	Column C2 TIMESTAMP
A10101	2013-06-30 14:55:03
A15014	2013-07-21 16:05:17
A31399	2013-07-21 03:24:33

Retrieval results

A15014	2013-07-21 16:05:17
A31399	2013-07-21 03:24:33

8.12.4 CHR

Return the character corresponding to the character code represented by the integer target data.

(1) Specification format

```
scalar-function-CHR ::= CHR(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the target data.

The value specified in *target-data* must be a character code corresponding to a single character, expressed as an integer greater than or equal to 0. For example, in the case of the multi-byte character expressed in hexadecimal as `0xE38182`, specify 14909826, which is the decimal equivalent of hexadecimal `0xE38182`.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify data of type `INTEGER` or `SMALLINT` for the target data.
- You cannot specify a dynamic parameter by itself for the target data.

The following example illustrates the result of executing the scalar function `CHR`. The example assumes Unicode (UTF-8) as the character encoding.

Examples:

`CHR(65)` → `'A'`

`CHR(97)` → `'a'`

`CHR(14845345)` → `'Ⅱ'`

(3) Rules

1. The data type of the execution result will be `VARCHAR(8)`.
2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If the target data is the null value or a negative value, the execution result will be a null value.

4. If the value of the target data exceeds 255, it is treated as a multi-byte character. For example, `14909826`, which is the decimal representation of hexadecimal `0xE38182`, is treated as a multi-byte character composed of the three bytes `0xE3`, `0x81`, and `0x82`.

(4) Example

Example:

Find the character string data items in column `C1` in table `T1` that end with the character `NL` (newline).

```
SELECT "C1" FROM "T1" WHERE SUBSTR("C1",-1)=CHR(10)
```

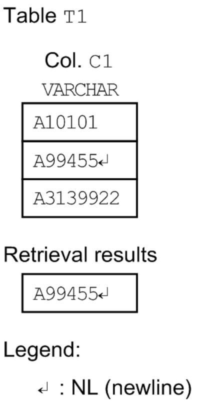


Table `T1`

Col. `C1` VARCHAR
A10101
A99455↲
A3139922

Retrieval results

A99455↲

Legend:

↲ : NL (newline)

8.12.5 CONVERT

Converts the data type of the data.

You can also specify a datetime format or number format to control the conversion.

- If you specify a datetime format
  - When converting datetime data to character string data, you can specify the output format of the character string data after conversion.
  - When converting character string data to datetime data, you can specify the input format of the character string data before conversion.
- If you specify a number format
  - When converting numeric data to character string data, you can specify the output format of the character string data after conversion.
  - When converting character string data to numeric data, you can specify the input format of the character string data before conversion.

The following examples illustrate the result of executing the scalar function `CONVERT`.

Example 1::

- Convert the `DECIMAL` type data -12.37 to `INTEGER` type.

  `CONVERT(-12.37,INTEGER)` → `-12`

Example 2: Specify a datetime format

- Convert the `TIMESTAMP` type data `TIMESTAMP'2013-07-30 11:03:58'` to `CHAR(10)` data.

CONVERT(TIMESTAMP'2013-07-30 11:03:58',CHAR(10),'YYYY/MM/DD') → '2013/07/30'

- Convert the CHAR type data 07/15/2013 12:34:56, which represents a datetime, to TIMESTAMP type.

  CONVERT('07/15/2013 12:34:56',TIMESTAMP,'MM/DD/YYYY HH:MI:SS') → TIMESTAMP'2013-07-15 12:34:56'

Example 3: Specify a number format

- Convert INTEGER type data to CHAR(7) data, and make it start with $ and have a comma between every 3 digits.

  CONVERT(1000,CHAR(7),'$9,999') → 'Δ$1,000'

  CONVERT(-1000,CHAR(7),'$9,999') → '-$1,000'

  Δ represents a single-byte space character.

- Convert INTEGER type data that starts with $ and has a comma between every 3 digits to CHAR type data.

  CONVERT('$1,000,000',INTEGER,'$9,999,999') → 1000000

  CONVERT('-$1,000',INTEGER,'$9,999,999') → -1000

(1) Specification format

```
scalar-function-CONVERT ::= CONVERT(data-to-convert,post-conversion-data-type[,format
-specification])

  data-to-convert ::= {value-expression|NULL}
  post-conversion-data-type ::= data-type
  format-specification ::= {datetime-format|number-format}
    datetime-format ::= literal
    number-format ::= literal
```

(2) Explanation of specification format

data-to-convert:

Specifies the data whose data type is to be converted.

Specify the data to be converted in the form of a value expression. Alternatively, specify NULL. For details about value expressions, see 7.20 Value expression.

post-conversion-data-type:

Specifies the data type after conversion. The following are examples:

- INTEGER

  Convert to INTEGER type data.

- DECIMAL(5,2)

  Convert to DECIMAL type data with a precision of 5 and a scaling of 2.

- CHAR(8)

  Convert to CHAR type data with a data length of 8 bytes.

- TIMESTAMP(3)

  Convert to TIMESTAMP type data with a fractional seconds precision of 3.

For the specification formats of each data type, see 6.2.1 List of data types.

Note that you cannot specify a VARCHAR type whose data length exceeds 32,000 bytes for the post-conversion-data type.

format-specification:

Specifies a datetime format or a number format.

datetime-format:

Specifies the datetime format in either of the following cases:

- When converting datetime data to character string data, specifies the output format of the character string data after conversion.
- When converting character string data to datetime data, specifies the input format of the character string data before conversion.

Specify a character string literal for the datetime format. For details about character string literals, see 6.3 Literals.

The following are examples of datetime formats:

Examples:

```
'YYYY-MM-DD HH:MI:SS'
'YYYY/MM/DD HH MI SS FF3'
'YYYY.MM.DD-HH:MI:SS.FF6'
'YYYY:MM'
'MM/DD-HH'
```

Items such as YYYY, MM, and DD in the examples above are called *datetime format elements*. For details about the elements that can be specified in the datetime format, see (3) Datetime format elements and rules.

The following examples illustrate the result of executing the scalar function CONVERT when a datetime format is specified.

- Examples of converting datetime data to character string data

Example of CONVERT specification	Execution result
CONVERT(DATE'2013-01-01',VARCHAR(20),'YYYY/MM/DD')	'2013/01/01'
CONVERT(DATE'2013-01-01',VARCHAR(20),'CC"st century"')	'21st century'
CONVERT(DATE'2013-01-01',VARCHAR(20),'EYYN/Q"Q"')	'H25/1Q'
CONVERT(DATE'2013-01-01',VARCHAR(20),'YY-WW')	'13-01'
CONVERT(DATE'2013-01-01',VARCHAR(20),'DD-Mon-YY')	'01-Jan-13'
CONVERT(DATE'2013-01-01',VARCHAR(20),'YYYY/MM/DD"("DY")"')	'2013/01/01(TUE)'
CONVERT(TIME'09:15:20.12',VARCHAR(20),'FMHH:MI:SS.FF6')	'9:15:20.120000'

- Examples of converting character string data to datetime data

Example of CONVERT specification	Execution result
CONVERT('01/02/2012 12:34:56',TIMESTAMP,'mm/dd/yyyy hh:mi:ss')	TIMESTAMP'2012-01-02 12:34:56'
CONVERT('平成25年1月1日　午前10時23分5秒',TIMESTAMP, 'fmeeyyn"年"mm"月"dd"日"pmnhh12"時"mi"分"ss"秒"')	TIMESTAMP'2013-01-01 10:23:05'
CONVERT('1 2 3 45',TIME(6),'FMHH MI SS FF2')	TIME'01:02:03.450000'

number-format:

Specifies the number format in either of the following cases:

- When converting numeric data to character string data, specifies the output format of the character string data after conversion.

- When converting character string data to numeric data, specifies the input format of the character string data before conversion.

Specify a character string literal for number format. For details about character string literals, see 6.3 Literals.

The following are examples of number format specifications.

Examples:

'$9,999,999'

'00,000.00'

Items such as $, 0, 9, . (period) and , (the three-digit comma separator) in the examples above are called the *elements* of the number format. For details about the elements that can be specified in a number format, see (4) Number format elements and rules.

The following examples illustrate the result of executing the scalar function CONVERT when a number format is specified.

- Examples of converting numeric data to character string data

Example of CONVERT specification	Execution result
CONVERT(1234567,CHAR(10),'9,999,999')	'Δ1,234,567'
CONVERT(1234,CHAR(10),'0,000,000')	'Δ0,001,234'
CONVERT(-1000,CHAR(7),'$9,999')	'-$1,000'
CONVERT(1000,VARCHAR(12),'LJ9,999"dollars"')	'1,000dollars'

Δ represents a single-byte space character.

- Examples of converting character string data to numeric data

Example of CONVERT specification	Execution result
CONVERT('1,234,567',INTEGER,'9,999,999')	1234567
CONVERT('12',INTEGER,'9,999,999')	12
CONVERT('$1,000,000',INTEGER,'$9,999,999')	1000000
CONVERT('1,000dollars',INTEGER,'9,999"dollars"')	1000
CONVERT('+1.23E+10*floating-point-character-string*',DOUBLE PRECISION,'9.99EEEE"*floating-point-character-string*"')	1.2300000000000000E10

(3) Datetime format elements and rules

(a) Datetime format elements

The table below shows the elements that can be specified in the datetime format.

Table 8-48: Elements that can be specified in the datetime format

No.	Meaning of datetime format	Element that can be specified in the datetime format	Description
1	Century	CC	Represents the century. The range of values is 00 to 99. Note that 00 represents the 100th century (the years 9901 to 9999 CE).
2	Year	YYYY	Represents the four-digit Western calendar year (CE). Values in the range 0001 to 9999 can be used.

No.	Meaning of datetime format	Element that can be specified in the datetime format	Description	
			Note that when converting character string data to datetime data, you cannot specify an era name if YYYY is specified.	
3		YY	Represents the lower two digits of the year. The range of values is 00 to 99.	
4	Era name	E	Represents the abbreviated form of the era name in Japanese. • 'M': Represents the Meiji era. • 'T': Represents the Taisho era. • 'S': Represents the Showa era. • 'H': Represents the Heisei era. • 'R': Represents the Reiwa era.	When converting character string data to datetime data, an era name must be specified together with a Japanese calendar year.
5		EE	Represents the era name in Japanese. • '明治': Represents the Meiji era. • '大正': Represents the Taisho era. • '昭和': Represents the Showa era. • '平成': Represents the Heisei era. • '令和': Represents the Reiwa era.	
6	Japanese calendar year	YYYYN	Represents a four-digit Japanese calendar year. The following shows the range of values that can be specified for each era: • Meiji: 0006 to 0045 • Taisho: 0001 to 0015 • Showa: 0001 to 0064 • Heisei: 0001 to 8011 • Reiwa: 0001 to 7981	When converting character string data to datetime data, a Japanese calendar year must be specified together with an era name.
7		YYN	Represents a two-digit Japanese calendar year. Values in the range 00 to 99 can be used. Note that if a year of three or more digits occurs when converting datetime data to character string data, only the lower two digits are converted.	
8	Quarter	Q	Represents the quarter. The range of values is 1 to 4. • 1: First quarter (January 1 to March 31) • 2: Second quarter (April 1 to June 30) • 3: Third quarter (July 1 to September 30) • 4: Fourth quarter (October 1 to December 31)	
9	Month	MM	Represents the month. Values in the range 01 to 12 can be used.	
10		MON	Represents the abbreviated form of the name of the month in English. • 'JAN': January • 'FEB': February • 'MAR': March • 'APR': April • 'MAY': May • 'JUN': June • 'JUL': July • 'AUG': August	

No.	Meaning of datetime format	Element that can be specified in the datetime format	Description
			• `'SEP'`: September • `'OCT'`: October • `'NOV'`: November • `'DEC'`: December
11		`MONTH`	Represents the name of the month in English. • `'JANUARYΔΔ'` • `'FEBRUARYΔ'` • `'MARCHΔΔΔΔ'` • `'APRILΔΔΔΔ'` • `'MAYΔΔΔΔΔΔ'` • `'JUNEΔΔΔΔΔ'` • `'JULYΔΔΔΔΔ'` • `'AUGUSTΔΔΔ'` • `'SEPTEMBER'` • `'OCTOBERΔΔ'` • `'NOVEMBERΔ'` • `'DECEMBERΔ'` Δ represents a single-byte space.
12	Week	`W`	Represents the week within the month. The range of values is `1` to `5`. • `1`: Represents days 1 to 7 of the month. • `2`: Represents days 8 to 14 of the month. • `3`: Represents days 15 to 21 of the month. • `4`: Represents days 22 to 28 of the month. • `5`: Represents day 29 until the end of the month. Note that the range of values will be `1` to `4` in February (except for leap years).
13		`WW`	Represents the week within the year. The range of values is `01` to `53`. For example, `01` represents January 1 to January 7, `02` represents January 8 to January 14, and so on.
14	Day	`DD`	Represents the ordinal date from the beginning of the month. The range of values is from `01` until the last day of the relevant month.
15		`DDD`	Represents the ordinal date from the beginning of the year. Values in the range `001` to `365` (`001` to `366` in leap years) can be used. For example, `001` represents January 1, and `002` represents January 2. `032` represents February 1.
16	Day of week	`D`	Represents the day of the week expressed as a number. Values in the range `1` to `7` can be used. • `1`: Sunday • `2`: Monday • `3`: Tuesday • `4`: Wednesday • `5`: Thursday • `6`: Friday • `7`: Saturday
17		`DAY`	Represents the day of the week in English. • `'SUNDAYΔΔΔ'`

No.	Meaning of datetime format	Element that can be specified in the datetime format	Description
			- 'MONDAYΔΔΔ' - 'TUESDAYΔΔ' - 'WEDNESDAY' - 'THURSDAYΔ' - 'FRIDAYΔΔΔ' - 'SATURDAYΔ' Δ represents a single-byte space.
18		`DY`	Represents the abbreviated form of the day of the week in English. - 'SUN': Sunday - 'MON': Monday - 'TUE': Tuesday - 'WED': Wednesday - 'THU': Thursday - 'FRI': Friday - 'SAT': Saturday
19		`DAYN`	Represents the day of the week in Japanese. Possible values are '日曜日': Sunday, '月曜日': Monday, '火曜日': Tuesday, '水曜日': Wednesday, '木曜日': Thursday, '金曜日': Friday, '土曜日': Saturday.
20		`DYN`	Represents the abbreviated form of the day of the week in Japanese. Possible values are '日': Sun, '月': Mon, '火': Tues, '水': Weds, '木': Thur, '金': Fri, '土': Sat.
21	Hour	`HH`	Represents the hour. Values in the range `00` to `23` can be used. When converting character string data to datetime data, `HH` and `HH24` cannot be specified with an AM/PM designation.
22		`HH24`	
23		`HH12`	Represents the hour. Values in the range `01` to `12` can be used. When converting character string data to datetime data, `HH12` must be specified together with an AM/PM designation.
24	AM/PM	`AM`	Represents the AM or PM designation in English.[#1] When converting character string data to datetime data, the AM/PM designation must be specified together with `HH12`.
25		`A.M.`	
26		`PM`	
27		`P.M.`	
28		`AMN`	Represents the AM or PM designation in Japanese.[#2]
29		`PMN`	
30	Minute	`MI`	Represents the minute. Values in the range `00` to `59` can be used.
31	Second	`SS`	Represents the second. Values in the range `00` to `59` can be used.
32		`SSSSS`	Represents seconds. Values in the range `00000` to `86399` can be used. The value represents the number of seconds that have elapsed since 00:00:00 (midnight). For example, `03600` denotes 1:00:00 AM.
33	Fractional seconds	`FF`	Represents the fractional seconds. When converting from character string data to datetime data, this is the number of digits of fractional seconds of *post-conversion-data-type*. When converting from datetime data to character string data, this is the number of digits of fractional seconds of *data-to-convert*.

No.	Meaning of datetime format	Element that can be specified in the datetime format	Description
			• If the number of digits of fractional seconds is 0, this specification is ignored. • If the number of digits of fractional seconds is 3, `FF` is equivalent to `FF3`. • If the number of digits of fractional seconds is 6, `FF` is equivalent to `FF6`. • If the number of digits of fractional seconds is 9, `FF` is equivalent to `FF9`. • If the number of digits of fractional seconds is 12, `FF` is equivalent to `FF12`.
34		`FF1`	Represents 1 digit of fractional seconds (`0` to `9`).
35		`FF2`	Represents 2 digits of fractional seconds (`00` to `99`).
36		`FF3`	Represents 3 digits of fractional seconds (`000` to `999`).
37		`FF4`	Represents 4 digits of fractional seconds (`0000` to `9999`).
38		`FF5`	Represents 5 digits of fractional seconds (`00000` to `99999`).
39		`FF6`	Represents 6 digits of fractional seconds (`000000` to `999999`).
40		`FF7`	Represents 7 digits of fractional seconds (`0000000` to `9999999`).
41		`FF8`	Represents 8 digits of fractional seconds (`00000000` to `99999999`).
42		`FF9`	Represents 9 digits of fractional seconds (`000000000` to `999999999`).
43		`FF10`	Represents 10 digits of fractional seconds (`0000000000` to `9999999999`).
44		`FF11`	Represents 11 digits of fractional seconds (`00000000000` to `99999999999`).
45		`FF12`	Represents 12 digits of fractional seconds (`000000000000` to `999999999999`).
46	Delimiting character	`-` (hyphen)	Characters used as delimiters between elements. **Example** `'YYYY-MM-DD HH:MI:SS'`
47		`/` (slash)	
48		`,` (comma)	
49		`.` (period)	
50		`:` (colon)	
51		`;` (semicolon)	
52		Space	

53	Other	`"`*character-string*`"`	You can specify an arbitrary string in double quotation marks (`"`).
			Example
			`CONVERT(DATE'2013-01-01',VARCHAR(20),'CC" 世紀 "')`
			`→ '21 世紀  ' ( 世紀 : Represents the century.)`
			The underlined portion indicates the relevant section.
			To specify a double quotation mark within the string itself, specify two consecutive double quotation marks (`""`).
			Example: Specify the character string `AB"CD`
			`"AB""CD"`

54		`FM`	Controls whether to delete the single-byte spaces at the end of the character strings denoted by `MONTH` and `DAY`, and whether to suppress zeros in numbers such as `YYYY`. For details, see (c) How to specify the datetime format element FM.

#1

- When converting character string data to datetime data, the conversion result will be the same regardless of whether you specify A.M., AM, P.M., or PM. If the target data uses AM, A.M., PM, or P.M., the conversion result will all be the same regardless of whether you specify AM, A.M., PM, or P.M. in the datetime format element. Uppercase and lowercase letters are treated the same.
- When converting datetime data to character string data, the conversion result will be the same regardless of whether you specify AM or PM. The conversion result will also be the same regardless of whether you specify A.M. or P.M.. The only difference between using AM vs. A.M., or PM vs. P.M., is whether the periods will appear in the character string after conversion.

#2

The execution results will be the same whether you specify AMN or PMN. The corresponding Japanese character strings are 午前 for AM and 午後 for PM.

Note

When datetime data is converted to character string data, it is converted based on the value of the datetime data to be converted, regardless of the specification of AM, PM, or other elements in the datetime format.

When character string data is converted to datetime data, it follows the specification of AM, PM, or other elements in the character string data to be converted, regardless of the specification of AM, PM, or other elements in the datetime format.

(b) Rules pertaining to datetime format

- The length of the datetime format cannot exceed 64 bytes.
- Characters specified in the datetime format that are not enclosed in double quotation marks (") must be single-byte.
- The letters specified in the datetime format are not case-sensitive. However, the following letters are case-sensitive:
  - The first letter in AM, A.M., PM, and P.M.
  - The first two letters in MON, MONTH, DAY, and DY
  - The letters in character strings that are enclosed in double quotation marks (")
- When converting character string data to datetime data, the following datetime format elements cannot be specified:
  - CC (century)
  - Q (quarter)
  - WW (week within the year)
  - W (week within the month)
  - YY (year expressed in two digits)
- When converting TIME type data to character string data, you cannot specify the following datetime formatting elements:
  - CC (century)
  - YYYY, YY (year)
  - E, EE (era name)
  - YYYYN, YYN (Japanese calendar year)
  - Q (quarter)

- MM, MON, MONTH (month)
- W, WW (week)
- DD, DDD (day)
- D, DAY, DAYN, DY, DYN (day of week)

- When converting character string data to datetime data, you cannot specify two or more datetime format elements with the same meaning. For example, the following are not allowed:

  Example 1: 'YYYY-MM-DD-YYYY'

  YYYY cannot be specified twice.

  Example 2: 'YYYY-MM-DD-EYYN'

  Because YYYY and YYN are datetime format elements with the same meaning, they cannot both be specified.

  The datetime format elements with the same meaning are shown in the following table:

Table 8-49: Datetime format elements with the same meaning

No.	Meaning of datetime format	Datetime format elements with the same meaning
1	Year	YYYY
2		YYYYN
3		YYN
4	Era name	E
5		EE
6	Month	MM
7		MON
8		MONTH
9		DDD
10	Day	DD
11		DDD
12	Hour	HH
13		HH24
14		HH12
15		SSSSS
16	AM/PM	AM
17		A.M.
18		PM
19		P.M.
20		AMN
21		PMN
22	Minute	MI
23		SSSSS
24	Second	SS

No.	Meaning of datetime format	Datetime format elements with the same meaning
25		SSSSS
26	Fractional seconds	FF
27		FF1
28		FF2
29		FF3
30		FF4
31		FF5
32		FF6
33		FF7
34		FF8
35		FF9
36		FF10
37		FF11
38		FF12

- When converting character string data to datetime data, if you specify the day of the week (D, DAY, DY, DAYN, or DYN) and the specified day of the week conflicts with the specified date, it does not result in an error.
- If you specify AM, A.M., PM, or P.M. in the datetime format when converting datetime data to character string data, if the first letter is uppercase, the entire element is output in uppercase, and if the first letter is lowercase, the entire element is output in lowercase.
- If an era name is used as a datetime format element, the range of the corresponding Western calendar years will be January 1, 1873 (January 1, Meiji 6) to December 31, 9999 (December 31, Reiwa 7981). The ranges of the corresponding Japanese calendar years are as follows.
    - Meiji: 06/01/ 01 to 45/07/29
    - Taisho: 01/07/30 to 15/12/ 24
    - Showa: 01/12/25 to 64/01/ 07
    - Heisei: 01/01/08 to 31/04/30
    - Reiwa: 01/05/01 to 7981/12/31

  However, if you specify Heisei as the era name when converting character string data into datetime data, you can specify December 31, Heisei 8011 or an earlier date.

  Example:

  ```
  CONVERT('05/01/0031/平成',DATE,'MM/DD/YYYYN/EE') → 2019-05-01
  CONVERT('12/31/8011/H',DATE,'MM/DD/YYYYN/E') → 9999-12-31
  ```
- The time represented by 0 hours in the HH24 representation is equivalent to the time 12:00 AM in the HH12 representation. The time represented by 12 hours in the HH24 representation is equivalent to the time 12:00 PM in the HH12 representation.
- Elements are extracted in order from the beginning (left) of the character string specified as the datetime format. In cases of overlapping element names, the longest possible element name is extracted. For example, if 'DDD' is specified, the first element extracted will be DDD, not D or DD.

- When converting character string data to datetime data, if you specify a character string enclosed in double quotation marks, make sure its case is consistent with the case of the letters in the target data. Note that uppercase and lowercase letters are distinguished inside a character string enclosed in double quotation marks when the character string is output.
- When converting character string data to datetime data, the conversion of `E` (Japanese era) is the same regardless of the case of the letters in the data being converted. When converting datetime data to character string data, the character string associated with `E` is always output in upper case.

(c) How to specify the datetime format element FM

■ When converting datetime data to character string data

If `FM` is not specified when `MONTH` or `DAY` is specified as an element of the datetime format, the character string after conversion is always nine characters long. If the resulting character string is shorter than nine characters, half-width spaces are added to produce a nine-character string.

Furthermore, zeros in the year, month, and date are not suppressed.

Example without FM:

```
CONVERT(DATE'2014-01-05',CHAR(17),'YYYY-MONTH-DD')
→ '2014-JANUARYΔΔ-05'
```

The half-width spaces following `JANUARY` are not deleted. Two half-width spaces are added to produce a nine-character string. Note that the zero in `05`, which is the day number, is not suppressed.

Example with FM:

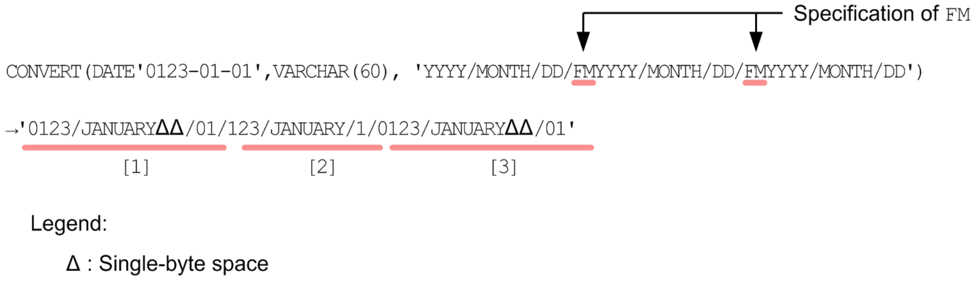
```
CONVERT(DATE'2014-01-05',CHAR(14),'FMYYYY-MONTH-DD')
→ 2014-JANUARY-5
```

The half-width spaces following `JANUARY` are deleted. Note that the zero in `05`, which is the day number, is suppressed.

Specify the datetime format element `FM` when you want to suppress zeros and remove single-byte spaces in the post-conversion character string data in this way.

In addition, by specifying `FM` in the middle of the datetime format, you can control the point at which this takes effect.

Example:



```
CONVERT(DATE'0123-01-01',VARCHAR(60), 'YYYY/MONTH/DD/FMYYYY/MONTH/DD/FMYYYY/MONTH/DD')
→'0123/JANUARYΔΔ/01/123/JANUARY/1/0123/JANUARYΔΔ/01'
```

Explanation

1. The half-width spaces in the character string that corresponds to `MONTH` (in this example, `JANUARYΔΔ`) are not deleted. Note that the zeros in the numbers that correspond to `YYYY` (in this example, `0123`) and `DD` (in this example, `01`) are not suppressed.
2. The character string corresponding to `MONTH` is `JANUARY`, with the spaces removed. Furthermore, the zeros are suppressed from the numbers corresponding to `YYYY` (`123`) and `DD` (`1`).

3. The half-width spaces in the character string that corresponds to MONTH (in this example, JANUARYΔΔ) are not deleted. Note that zeros in the numbers that correspond to YYYY (in this example, 0123) and DD (in this example, 01) are not suppressed.

■ When converting character string data to datetime data

- When the datetime format element MONTH or DAY is specified, spaces are required in the character string data to be converted (for example, JANUARY ΔΔ). If the character string data to be converted does not include the spaces (for example, JANUARY), it can still be converted if the datetime format element FM is specified.

  <Example that results in an error>

  ```
  CONVERT('2014-JANUARY-05',DATE,'YYYY-MONTH-DD') → Error
  ```

  This results in an error because the two spaces are missing from the end of JANUARY.

  <Example that does not result in an error>

  ```
  CONVERT('2014-JANUARY-05',DATE,'FMYYYY-MONTH-DD') → DATE'2014-01-05'
  ```

  Because FM is specified, the two spaces are not required at the end of JANUARY.

  Note that an error results if you express the month as JANUARY ΔΔ when FM is specified.

  Important

  When FM is specified, the conversion results might not always come out as intended. For example, if you attempt to convert the character string '2014111' to a DATE type value representing January 11, 2014, you will not obtain the intended result, as illustrated below.

  ```
  CONVERT('2014111',DATE,'FMYYYYMMDD') → DATE'2014-11-01'
  ```

  In the above example, the character string is converted to November 1, 2014.

- When you specify one of the datetime format elements listed in Table 8-50: Datetime format elements specifying numbers and their maximum number of characters including leading zeros as a numeric character, make sure that the number of numeric characters in the character string data to be converted is equal to the maximum number of characters indicated in Table 8-50: Datetime format elements specifying numbers and their maximum number of characters including leading zeros. For example, if the element MM is specified, the numeric character representing the months January through September must be expressed as 01 to 09 in the character string data (the leading zero is required). If there is no leading zero in the character string data to be converted, it can still be converted if the datetime format element FM is specified (no error results regardless of whether the leading zero is there).

  <Example that results in an error>

  ```
  CONVERT('2014:1:5',DATE,'YYYY:MM:DD') → Error
  ```

  <Examples that do not result in an error>

  ```
  CONVERT('2014:1:5',DATE,'FMYYYY:MM:DD') → DATE'2014-01-05'
  CONVERT('2014:01:05',DATE,'FMYYYY:MM:DD') → DATE'2014-01-05'
  ```

- Even when the value of an element listed in Table 8-50: Datetime format elements specifying numbers and their maximum number of characters including leading zeros that you specify with a numeric character is 0, you must specify at least one character for each element in the data to be converted. For example, when converting 0 hours, 0 minutes, and 0 seconds with the datetime format 'FMHH:MI:SS', the conversion works when the data to be converted is '0:0:0', but generates an error when the data to be converted is '0:0:'.

  However, in the case of elements FF and FF1 to FF12, which are unaffected by FM, you can omit the specification of 0. For example, when converting 0 hours, 0 minutes, and 0.000 seconds with the datetime format 'FMHH:MI:SS.FF3', the conversion works even when the data to be converted is '0:0:0.'.

- By specifying FM in the middle of the datetime format, you can control the point at which this takes effect.

- When FM is specified, HADB identifies the extent of a number corresponding to an element such as MM and DD based on either the first occurrence of a non-digit character or the point when the maximum number of characters in the specified datetime format has been reached. The following table shows the datetime format elements specifying numbers and their maximum number of characters, including leading zeros.

Table 8-50: Datetime format elements specifying numbers and their maximum number of characters including leading zeros

No.	Datetime format element specifying number	Maximum number of characters including leading zeros
1	YYYY	4
2	YYYYN	4
3	YYN	2
4	MM	2
5	DD	2
6	DDD	3
7	D	1
8	HH	2
9	HH24	2
10	HH12	2
11	MI	2
12	SS	2
13	SSSSS	5
14	FF	Not applicable
15	FF1	
16	FF2	
17	FF3	
18	FF4	
19	FF5	
20	FF6	
21	FF7	
22	FF8	
23	FF9	
24	FF10	
25	FF11	
26	FF12	

(4) Number format elements and rules

(a) Specification format for number format elements

This subsection describes the specification format for number format elements. Note that you must close up the spacing for any elements you omit. An error results if elements are specified in the wrong order, or if elements are specified where they are not permitted.

```
number-format ::= {fixed-point-representation | floating-point-representation | short
est-representation | hexadecimal-representation}


  fixed-point-representation ::=
       ["character-string"][[modifier-element][sign-element][B][currency-element]
       [numeric-element[[{delimiting-character-element | numeric-element}]...numeric-
element]][.] [numeric-element]...[sign-element]["character-string"]]

  floating-point-representation ::=
       ["character-string"][modifier-element][numeric-element]...[.][numeric-element]
... floating-point-element["character-string"]

  shortest-representation ::= ["character-string"]{TM | TM9 | TME}["character-string"
]

  hexadecimal-representation ::=
       ["character-string"][modifier-element][0]...hexadecimal-element[hexadecimal-el
ement]...["character-string"]

    modifier-element ::= {LJ | LS}
    sign-element ::= {MI | S | PR}
    currency-element ::= {$ | ¥}
    numeric-element ::= {0 | 9}
    delimiting-character-element ::= {, | Δ}
    floating-point-element ::= {EEEE | eeee}
    hexadecimal-element ::= {X | x}
```

Note:

- The delimiting character element Δ represents a single-byte space character.
- "*character-string*" represents an arbitrary character string enclosed in double quotation marks.

(b) Number format element

The table below shows the elements that can be specified in the number format.

In the table, Δ represents a single-byte space character.

Table 8-51: Elements that can be specified in the number format

No.	Type of element	Element that can be specified in the number format	Description
1	Delimiting character element	`,` (comma)	■ When converting numeric data to character string data • Specifies that a comma is to be inserted for the separation of numeric elements in the converted character string data. A comma is inserted at the position where the comma is specified. **Example** `CONVERT(1234567,CHAR(10),'9,999,999')`

No.	Type of element	Element that can be specified in the number format	Description

			→ 'Δ1,234,567'
			• If the number of digits in the integer part of the numeric data is less than the number of digits in the integer part specified in the number format, the extra commas are not inserted.
			Example
			`CONVERT(1234,CHAR(10),'9,999,999')`
			→ 'ΔΔΔΔ1,234'
			■ When converting character string data to numeric data
			• Specify this when there are commas in the character string data to be converted. The commas are removed from the specified positions during conversion to numeric data.
			Example
			`CONVERT('1,234,567',INTEGER,'9,999,999')`
			→ 1234567
			• An error results if the character string data to be converted does not have a comma at the position specified in the number format.
			Example
			`CONVERT('1234',INTEGER,'9,999')`
			→ Error
			• If the number of digits in the integer part of the character string data is less than the number of digits in the integer part specified in the number format, the extra commas are ignored.
			Example
			`CONVERT('1,234',INTEGER,'9,999,999')`
			→ 1234
			Specification rules
			You cannot specify a comma to the right of the period that represents the decimal point.
			Examples of number format specifications that result in an error:
			`'999,999.9,99'`
			`',999,999,999'`

2		Δ (single-byte space)	■ When converting numeric data to character string data
			• Specifies that a single-byte space is to be inserted for the separation of numeric elements in the converted character string data. A single-byte space is inserted at the position where the space is specified.
			Example
			`CONVERT(1234567,CHAR(10),'9 999 999')`
			→ 'Δ1Δ234Δ567'
			■ When converting character string data to numeric data
			• Specify this when there are spaces in the character string data to be converted. The spaces are removed from the specified positions during conversion to numeric data.
			Example
			`CONVERT('1 234 567',INTEGER,'9 999 999')`
			→ 1234567
			• An error results if the character string data to be converted does not have a space at the position specified in the number format.
			Example
			`CONVERT('1234',INTEGER,'9 999')`
			→ Error

No.	Type of element	Element that can be specified in the number format	Description

			• If the number of digits in the integer part of the character string data is less than the number of digits in the integer part specified in the number format, the extra spaces are ignored.
			Example
			`CONVERT('1 234',INTEGER,'9 999 999')`
			→ `1234`
			Specification rules
			You cannot specify a space to the right of the period that represents the decimal point.
			Example of a number format specification that results in an error:
			`'9.99 9'`

3	Decimal point character	`.` (period)	Specifies the position of the decimal point using a period. In the number format, the numeric element before the period represents the integer part, and the numeric element after the period represents the decimal part.
			■ When converting numeric data to character string data
			• The numeric data is converted into the integer and decimal parts specified in the number format.
			Example
			`CONVERT(1234.56,CHAR(9),'9,999.99')`
			→ `'Δ1,234.56'`
			• If the number of decimal places in the numeric data is greater than the number of decimal places specified in the number format, the numeric value is rounded during conversion. The rounding method is the same as for the scalar function `ROUND`. For details about the scalar function `ROUND`, see 8.4.9 ROUND.
			Example
			`CONVERT(1.56,CHAR(4),'9.9')`
			→ `'Δ1.6'`
			■ When converting character string data to numeric data
			• The left side of the period in the character string data is used as the integer part of the numeric data, and the right side of the period is used as the decimal part.
			Example
			`CONVERT('1,234.56',DECIMAL(6,2),'9,999.99')`
			→ `1234.56`
			Specification rules
			Only one period can be specified. It must be specified before or after a numeric element, or between two numeric elements.
			Example of a number format specification that results in an error:
			`'.$999'`

4	Currency element	`$`	■ When converting numeric data to character string data
			• Specifies that `$` (a dollar sign) is to be added in the converted character string data.
			Example
			`CONVERT(1000,CHAR(7),'$9,999')`
			→ `'Δ$1,000'`
			`CONVERT(-1000,CHAR(7),'$9,999')`
			→ `'-$1,000'`
			■ When converting character string data to numeric data
			• Specifies this when there is a `$` (dollar sign) in the character string data to be converted. The dollar sign is removed during conversion to numeric data.
			Example

No.	Type of element	Element that can be specified in the number format	Description

			`CONVERT('$1,000',INTEGER,'$9,999')`
			→ `1000`
			• An error results if there is no dollar sign in front of the number.
			Example
			`CONVERT('1,000',INTEGER,'$9,999')`
			→ `Error`
			• There must be no space between the dollar sign and the beginning of the number.
			Example
			`CONVERT('$ 1,000',INTEGER,'$9,999')`
			→ `Error`

5		¥	■ When converting numeric data to character string data
			• Specifies that ¥ (a yen sign) is to be added in the converted character string data.
			Example
			`CONVERT(1000,CHAR(7),'¥9,999')`
			→ `'Δ¥1,000'`
			`CONVERT(-1000,CHAR(7),'¥9,999')`
			→ `'-¥1,000'`
			■ When converting character string data to numeric data
			• Specifies this when there is a ¥ (yen sign) in the character string data to be converted. The yen sign is removed during conversion to numeric data.
			Example
			`CONVERT('¥1,000',INTEGER,'¥9,999')`
			→ `1000`
			• An error results if there is no yen sign in front of the number.
			Example
			`CONVERT('1,000',INTEGER,'¥9,999')`
			→ `Error`
			• There must be no space between the yen sign and the beginning of the number.
			Example
			`CONVERT('¥ 1,000',INTEGER,'¥9,999')`
			→ `Error`

6	Numeric element	0	Represents a single numeric digit when converting the digits corresponding to a numeric value in the data to be converted. The following description pertains to the numeric element `0` specified in a fixed-point representation. For details about the numeric element `0` in a floating-point or hexadecimal representation, see the descriptions of the floating-point and hexadecimal elements.
			■ When converting numeric data to character string data
			• The total number of `0` and `9` elements indicates the maximum number of digits after conversion.
			Example
			`CONVERT(1234.5,CHAR(10),'00,000.00')`
			→ `'Δ01,234.50'`
			• If the number of decimal places in the numeric data is greater than the number of decimal places specified in the number format, the numeric value is rounded during conversion. The rounding method is the same as for the scalar function `ROUND`. For details about the scalar function `ROUND`, see 8.4.9 ROUND.
			Example
			`CONVERT(2.34567,CHAR(5),'0.00')`
			→ `'Δ2.35'`

No.	Type of element	Element that can be specified in the number format	Description

			• A sign is prefixed to the character string data that is converted according to the specifications of the period, delimiters, and numeric elements in the number format. In the case of 0 or a positive value, a single-byte space is used. In the case of a negative value, a minus sign (-) is used. However, when a sign element is specified, it appends a character string indicating a sign as specified in the sign element.
			Examples
			`CONVERT(-1234.5,CHAR(8),'0,000.0')`
			→ `'-1,234.5'`
			`CONVERT(0,CHAR(8),'0,000.0')`
			→ `'Δ0,000.0'`
			• If the number of digits in the integer or decimal part of the numeric data is less than the number of digits in the integer or decimal part specified in the number format, the extra digits are converted to 0 in the character string.
			Example
			`CONVERT(1.1,CHAR(10),'0,000.000')`
			→ `'Δ0,001.100'`
			• If the number of digits in the integer part of the numeric data is greater than the number of digits in the integer part specified in the number format, the numeric data is converted to a hash-mark (#) filled character string.
			Example
			`CONVERT(1234,CHAR(3),'00')`
			→ `'###'`
			■ When converting character string data to numeric data
			• The conversion is the same whether the numeric element `0` or `9` is specified. You can convert with numeric element `0` even if there are single-byte spaces in the digits beyond the number of significant digits of the integer part of the character string data. Similarly, you can convert with numeric element `9` even if there are zeros in the digits beyond the number of significant digits of the integer part of the character string data.
			Example
			`CONVERT('0,123',INTEGER,'9,999')`
			→ `123`
			• The mapping between the digits of the character string data and the number format begins at the decimal point, with the integer digits moving towards the left, in the order ones, tens, and so on, and the decimal digits moving toward the right, in the order first decimal place, second decimal place, and so on. Note that even if the number of digits differs between the character string data and the number format, conversion is possible if the number of digits in the integer part of the character string data is less than the number of digits in the integer part specified in the number format, and the number of decimal places in the character string data is less than the number of decimal places specified in the number format.
			Example
			`CONVERT('1,234.56',DECIMAL(8,3),'00,000.000')`
			→ `1234.560`
			An error is generated in the following cases.
			• If the number of digits in the integer part of the character string data is greater than the number of digits in the integer part specified in the number format
			Example
			`CONVERT('1234',INTEGER,'00')`
			→ `Error`
			• If the number of decimal places in the character string data is greater than the number of decimal places specified in the number format

No.	Type of element	Element that can be specified in the number format	Description

			Example
			`CONVERT('1.234',DECIMAL(2,1),'0.0')`
			`→Error`
			- If there is a single-byte space between the sign and the most significant digit in the character string data
			Example
			`CONVERT('- 1,234',INTEGER,'0,000')`
			`→Error`
			Specification rules
			The total number of `0` and `9` numeric elements cannot exceed 38.

7		`9`	Represents a single numeric digit when converting the digits corresponding to a numeric value in the data to be converted. The following description pertains to the numeric element `9` specified in a fixed-point representation. For details about the numeric element `9` in a floating-point or hexadecimal representation, see the descriptions of the floating-point and hexadecimal elements.
			■ When converting numeric data to character string data
			- The total number of 0 and 9 elements indicates the maximum number of digits after conversion.
			Example
			`CONVERT(1234.5,CHAR(10),'99,999.99')`
			`→' ΔΔ1,234.50'`
			- If the number of decimal places in the numeric data is greater than the number of decimal places specified in the number format, the numeric value is rounded during conversion. The rounding method is the same as for the scalar function `ROUND`. For details about the scalar function `ROUND`, see 8.4.9 ROUND.
			Example
			`CONVERT(2.34567,CHAR(5),'9.99')`
			`→'Δ2.35'`
			- A sign is prefixed to the character string data that is converted according to the specifications of the period, delimiters, and numeric elements in the number format. In the case of 0 or a positive value, a single-byte space is used. In the case of a negative value, a minus sign (-) is used. However, when a sign element is specified, it appends a character string indicating a sign as specified in the sign element.
			Examples
			`CONVERT(1234.5,CHAR(8),'9,999.9')`
			`→'Δ1,234.5'`
			`CONVERT(-1234.5,CHAR(8),'9,999.9')`
			`→'-1,234.5'`
			- If the number of digits in the integer part of the numeric data is less than the number of digits in the integer part specified in the number format, the extra digits are converted to single-byte spaces. In addition, if the number of decimal places in the numeric data is less than the number of decimal places specified in the number format, the extra digits are converted to 0 in the character string.
			Example
			`CONVERT(1.1,CHAR(10),'9,999.999')`
			`→' ΔΔΔΔ1.100'`
			- If no numeric element is specified for the decimal part, when the result of rounding the numeric data to the number of digits in the number format is 0, it is converted to the character string `0`.
			Example
			`CONVERT(0.1,CHAR(2),'9')`

No.	Type of element	Element that can be specified in the number format	Description
			→ 'Δ0' If a numeric element is specified for the decimal part, the integer part in the converted character string data is converted to a single-byte space, not 0. **Example** CONVERT(0.1,CHAR(5),'9.99') → 'ΔΔ.10' CONVERT(0,CHAR(5),'9.99') → 'ΔΔ.00' Finally, in the case of a negative value, it will be converted as follows. **Example** CONVERT(-0.1,CHAR(5),'9.99') → 'Δ-.10' • If numeric element 0 is specified before a numeric element 9, any numeric element 9s that follows the specified numeric element 0 are treated as numeric element 0s. **Example** CONVERT(1,CHAR(5),'0999') → 'Δ0001' • If the number of digits in the integer part of the numeric data is greater than the number of digits in the integer part specified in the number format, the numeric data is converted to a hash-mark (#) filled character string. **Example** CONVERT(1234,CHAR(3),'99') → '###' ■ When converting character string data to numeric data • The conversion is the same whether the numeric element 0 or 9 is specified. You can convert with numeric element 0 even if there are single-byte spaces in the digits beyond the number of significant digits of the integer part of the character string data. Similarly, you can convert with numeric element 9 even if there are zeros in the digits beyond the number of significant digits of the integer part of the character string data. **Example** CONVERT('0,123',INTEGER,'9,999') → 123 • The mapping between the digits of the character string data and the number format begins at the decimal point, with the integer digits moving towards the left, in the order ones, tens, and so on, and the decimal digits moving toward the right, in the order first decimal place, second decimal place, and so on. Note that even if the number of digits differs between the character string data and the number format, conversion is possible if the number of digits in the integer part of the character string data is less than the number of digits in the integer part specified in the number format, and the number of decimal places in the character string data is less than the number of decimal places specified in the number format. **Example** CONVERT('1,234.56',DECIMAL(8,3),'99,999.999') → 1234.560 An error is generated in the following cases. • If the number of digits in the integer part of the character string data is greater than the number of digits in the integer part specified in the number format **Example** CONVERT('1234',INTEGER,'99')

No.	Type of element	Element that can be specified in the number format	Description

			→`Error`
			- If the number of decimal places in the character string data is greater than the number of decimal places specified in the number format
			Example
			`CONVERT('1.234',DECIMAL(4,3),'9.9')`
			→`Error`
			- If there is a single-byte space between the sign and the most significant digit in the character string data
			Example
			`CONVERT('- 1,234',INTEGER,'9,999')`
			→`Error`
			Specification rules
			The total number of `0` and `9` numeric elements cannot exceed 38.

8	Floating-point element	`EEEE`
		`eeee`
			Specify this element to indicate a floating-point numeric literal. The conversion is the same whether used with `0` or `9` as the numeric element for the digits. The following are examples of number format specifications.
			Examples
			- `'9.999EEEE'`
			- `'9.999eeee'`
			- `'9.EEEE'`
			- `'9EEEE'`
			- `'.9EEEE'` #
			- `'99.9EEEE'`#
			#: Can be specified only when converting character string data to numeric data.
			■ When converting numeric data to character string data
			- The numeric data is converted to the format of a floating-point numeric literal in accordance with the specification in the number format.
			Example
			`CONVERT(12.3,CHAR(9),'9.99EEEE')`
			→ `'Δ1.23E+01'`
			`CONVERT(0.01,CHAR(9),'9.99EEEE')`
			→ `'Δ1.00E-02'`
			If the exponent of the converted character string data is 0 or a positive value, a plus sign (+) is prefixed to the exponent.
			The exponent of the converted character string data will be either 2 or 3 digits. If the value is `0`, the exponent will be `00`.
			- If the number of decimal places in the numeric data is greater than the number of decimal places specified in the number format, the numeric value is rounded during conversion. The rounding method is the same as for the scalar function `ROUND`. For details about the scalar function `ROUND`, see 8.4.9 ROUND.
			Example
			`CONVERT(34.56,CHAR(9),'9.99EEEE')`
			→ `'Δ3.46E+01'`
			- A sign is prefixed to the character string data that is converted according to the specifications of the period, delimiters, numeric elements, and `EEEE` element in the number format. In the case of 0 or a positive value, a single-byte space is used. In the case of a negative value, a minus sign (-) is used.
			Example
			`CONVERT(0,CHAR(9),'9.99EEEE')`
			→ `'Δ0.00E+00'`
			`CONVERT(-1,CHAR(9),'9.99EEEE')`

No.	Type of element	Element that can be specified in the number format	Description

			→ `'-1.00E+00'`
			• If this element is specified as lowercase `eeee`, the `E` indicating a floating-point numeric literal is converted to lowercase `e`.
			Example
			`CONVERT(1,CHAR(9),'9.99eeee')`
			→ `'Δ1.00e+00'`
			• For the integer part, exactly one numeric element must be specified, or else an error results.
			Example
			`CONVERT(1,CHAR(9),'99.9EEEE')`
			→ `Error`
			■ When converting character string data to numeric data
			• Character string data that is expressed in the notation of a floating-point numeric literal is converted to floating-point numeric data.
			Examples
			`CONVERT('1.23E+10`*floating-point-character-string*`',DOUBLE PRECISION,'9.99EEEE"`*floating-point-character-string*`"')`
			→ `1.2300000000000000E10`
			`CONVERT('-1.23E+10`*floating-point-character-string*`',DOUBLE PRECISION,'9.99EEEE"`*floating-point-character-string*`"')`
			→ `-1.2300000000000000E10`
			• The element `EEEE` and the `E` character in the data to be converted are not case-sensitive.
			Example
			`CONVERT('1.23e+10',DOUBLE PRECISION,'9.99EEEE')`
			→ `1.2300000000000000E10`
			An error is generated in the following cases.
			• If the number of digits in the integer part of the character string data is greater than the number of digits in the integer part specified in the number format
			Example
			`CONVERT('12.3E+1',DOUBLE PRECISION,'9.9EEEE')`
			→ `Error`
			• If the number of decimal places in the character string data is greater than the number of decimal places specified in the number format
			Example
			`CONVERT('1.234E+1',DOUBLE PRECISION,'9.9EEEE')`
			→ `Error`
			• If there is a single-byte space between the sign of the mantissa and the most significant digit in the character string data
			Example
			`CONVERT('- 1.23E+1',DOUBLE PRECISION,'9.99EEEE')`
			→ `Error`
			Specification rules
			The sum of the number of numeric elements specified in the integer part and the decimal part cannot exceed 17.
			You must specify either `EEEE` or `eeee`. Mixing uppercase and lowercase letters is not permitted.

9	Sign element	`MI`	■ When converting numeric data to character string data • If the numeric data is a negative value, a minus sign (-) is appended to the end of character string data that has been converted according to the specifications

No.	Type of element	Element that can be specified in the number format	Description

			of the period, delimiters, and numeric elements in the number format. In the case of 0 or a positive value, a single-byte space is used.
			Examples
			`CONVERT(-123,CHAR(4),'999MI')`
			→ `'123-'`
			`CONVERT(123,CHAR(4),'999MI')`
			→ `'123Δ'`
			■ When converting character string data to numeric data
			• When the numeric data is converted, the position where the `MI` element is specified is interpreted as the sign. In the case of a minus sign (−), it is converted to a negative value; in the case of plus sign (+) or a single-byte space, it is converted to a value greater than or equal to 0.
			Example
			`CONVERT('123-',INTEGER,'999MI')`
			→ `-123`
			The conversion also results in a 0 or a positive value if the end of the data to be converted, excluding the part specified in "*character-string*", is a number or period.
			Example
			`CONVERT('123$',INTEGER,'999MI"$"')`
			→ `123`

10		`S` (if `S` is specified at the beginning of the element)	■ When converting numeric data to character string data
			• A sign is prefixed before the character string data that has been converted according to the specifications of the period, delimiters, and numeric elements in the number format. In the case of a negative value a minus sign (-) is used, whereas in the case of 0 or a positive value, a plus sign (+) is used.
			Example
			`CONVERT(123,CHAR(4),'S999')`
			→ `'+123'`
			• If a currency element is specified in the number format, the sign is added in front of the currency symbol.
			Example
			`CONVERT(123,CHAR(5),'S$999')`
			→ `'+$123'`
			■ When converting character string data to numeric data
			• Converts the character string data to numeric data according to the sign at the beginning of the character string. An error results if there is no sign.
			Example
			`CONVERT('+$123',INTEGER,'S$999')`
			→ `123`
			`CONVERT('123',INTEGER,'S999')`
			→ `Error`
			• An error results if there is a single-byte space between the sign and the most significant digit in the character string data.
			Example
			`CONVERT('+ 123',INTEGER,'S999')`
			→ `Error`
			• When a currency element is specified in the number format, an error results if there is no sign in front of the currency symbol in the character string data.
			Example
			`CONVERT('+$123',INTEGER,'S$999')`

No.	Type of element	Element that can be specified in the number format	Description
			→ 123 `CONVERT('$123',INTEGER,'S$999')` → Error

11		`S` (if `S` is specified at the end of the element)	■ When converting numeric data to character string data
			• A sign is affixed at the end of the character string data that has been converted according to the specifications of the period, delimiters, and numeric elements in the number format. In the case of a negative value, a minus sign (-) is used, whereas in the case of 0 or a positive value a plus sign (+) is used.
			Example
			`CONVERT(123,CHAR(4),'999S')`
			→ `'123+'`
			■ When converting character string data to numeric data
			• Converts the character string data to numeric data according to the sign in the position of the element `S` specified in the number format. An error results if there is no sign.
			Examples
			`CONVERT('123+',INTEGER,'999S')`
			→ 123
			`CONVERT('123',INTEGER,'999S')`
			→ Error

12		`PR`	■ When converting numeric data to character string data
			• If the numeric data is a negative value, the character string data that was converted according to the number format is enclosed in <>.
			Example
			`CONVERT(-123,CHAR(5),'999PR')`
			→ `'<123>'`
			• If the numeric data is 0 or a positive value, single-byte spaces are inserted instead of <>.
			Example
			`CONVERT(123,CHAR(5),'999PR')`
			→ `'Δ123Δ'`
			• If a currency element or `"`*character-string*`"` is specified in the number format, the corresponding characters are set inside <> when the numeric data has a negative value.
			Examples
			`CONVERT(-123,CHAR(6),'$999PR')`
			→ `'<$123>'`
			`CONVERT(-123,CHAR(12),'999PR"dollars"')`
			→ `'<123dollars>'`
			■ When converting character string data to numeric data
			• If a number is enclosed in <> in the character string data, it is converted to a negative value and <> is removed. If the number is not enclosed in <>, it is converted to 0 or a positive value.
			Example
			`CONVERT('<123>',INTEGER,'999PR')`
			→ -123
			`CONVERT('123',INTEGER,'999PR')`
			→ 123

No.	Type of element	Element that can be specified in the number format	Description

13	Character string	"*character- string*"
		(character string enclosed in double quotation marks)
			A character string enclosed in double quotation marks (") can be specified at the beginning or end of the number format. Double-byte characters are also permitted.
			■ When converting numeric data to character string data
			• The character string enclosed in double quotation marks is inserted at the beginning or end of the converted character string data.
			Example
			`CONVERT(123,CHAR(11),'999"dollars"')`
			→ `'Δ123dollars'`
			• Uppercase and lowercase letters are distinguished in the character string enclosed in double quotation marks.
			■ When converting character string data to numeric data
			• The character string enclosed in double quotation marks is removed from the character string data when it is converted to numeric data.
			Example
			`CONVERT('123dollars',INTEGER,'999"dollars"')`
			→ `123`
			• An error results if there is no character string enclosed in double quotation marks in the character string data.
			Example
			`CONVERT('123',INTEGER,'999"dollars"')`
			→ `Error`
			However, if there are one or more contiguous single-byte spaces at the beginning of a character string enclosed in double quotation marks specified at the beginning of the number format, or at the end of a character string enclosed in double quotation marks specified at the end of the number format, it does not generate an error if the spaces do not occur in the character string data.
			Example
			`CONVERT('dollars123',INTEGER,'" dollars"999')`
			→ `123`
			• Make sure the case of the letters in the character string enclosed in double quotation marks is consistent with the case of the letters in the data to be converted.
			Specification rules
			You can specify character strings enclosed in double quotation marks at both the beginning and end of the number format.
			You cannot specify a character string enclosed in double quotation marks between other elements.
			To specify a double quotation mark in the string itself, specify two consecutive double quotation marks.

14	Hexadecimal element	`X`
		`x`
			■ When converting numeric data to character string data
			• Converts numeric data to the hexadecimal digits representing the specified number.
			Example
			`CONVERT(10,CHAR(5),'XXXX')`
			→ `'ΔΔΔΔA'`
			`CONVERT(10,CHAR(5),'0XXX')`
			→ `'Δ000A'`
			• One single-byte space is inserted before the hexadecimal digits in the converted character string data.
			Example
			`CONVERT(10,CHAR(2),'X')`
			→ `'ΔA'`

No.	Type of element	Element that can be specified in the number format	Description
			• The elements 0, X, and x correspond to one converted hexadecimal digit. • The maximum integer value that can be converted is the maximum positive value that can be represented in the DECIMAL type (fixed-point). • If the number of hexadecimal digits converted to character string data is less than the specified number of digits (the total number of X and x elements), the converted character string data is right-aligned and padded with single-byte spaces. **Example** CONVERT(10,CHAR(5),'XXXX') → 'ΔΔΔΔA' If you want to pad with zeros instead of single-byte spaces, specify the element 0 at the beginning. Multiple consecutive 0 elements can be specified, but they must be specified in front of an X or x element. **Example** CONVERT(10,CHAR(5),'0XXX') → 'Δ000A' • If the specified numeric value is not an integer, it is rounded to an integer. The rounding method is the same as for the scalar function ROUND. **Example** CONVERT(10.5,CHAR(6),'0XXXX') → 'Δ0000B' • If the numeric data is a negative value, the numeric data is converted to a hash-mark (#) filled character string. **Example** CONVERT(-20,CHAR(6),'0XXXX') → '######' • If the first-specified element X is uppercase, the converted hexadecimal digits will also be uppercase (A to F). If it is lowercase (x), the converted hexadecimal digits will also be lowercase (a to f). **Example** CONVERT(10,CHAR(5),'xXXX') → 'ΔΔΔΔa' • If the number of characters converted to hexadecimal digits is greater than the number of digits specified in the number format (the total number of elements 0, X, and x), they are converted to hash marks (#). **Example** CONVERT(1024,CHAR(5),'XX') → '###ΔΔ' ■ When converting character string data to numeric data • The hexadecimal digits (0 to 9, A to F, a to f) in the character string data are converted to numeric data. **Example** CONVERT('AB',INTEGER,'XXXX') → 171 • Elements 0, X, and x are treated the same when converting from hexadecimal digits to hexadecimal numeric data. However, element 0 can only be specified before element X or x. • The execution results will be the same whether you specify element X or x. In addition, conversion is possible even when the hexadecimal digits in the character string data include a mixture of uppercase and lowercase. **Example** CONVERT('Ab',INTEGER,'xXXx')

No.	Type of element	Element that can be specified in the number format	Description

			→ `171`
			• The element `0` can be specified even when there is a single-byte space in front of the hexadecimal digits in the character string data.
			Example
			`CONVERT('`*hexadecimal-character-string* `A',INTEGER,'"`*hexadecimal-character-string*`"0XXX')`
			→ `10`
			• Conversion is possible when there is a leading zero, even when element `0` is not specified in the number format. However, conversion is only possible when the number of hexadecimal digits, including leading zeros, is less than or equal to the number of digits specified in the number format (the total number of `X` and `x` elements).
			Example
			`CONVERT('00A',INTEGER,'XXXXX')`
			→ `10`
			• Conversion is possible even when the number of digits in the character string data is less than the number of digits specified in the number format (the total number of elements `0`, `X`, or `x`).
			Example
			`CONVERT('A',INTEGER,'XXX')`
			→ `10`
			• The hexadecimal digits in the character string data are treated as 0 or positive integer values.
			An error is generated in the following cases.
			• There are characters in the character string data other than hexadecimal digits (`0` to `9`, `A` to `F`, `a` to `f`)
			• If the number of digits in the character string data is greater than the number of digits specified in the number format (the total number of elements `0`, `X`, or `x`)
			Example
			`CONVERT('0001',INTEGER,'XX')`
			→ `Error`
			Specification rules
			The total number of elements `0`, `X`, and `x` cannot exceed 32.

15	Modifier element	`LS`	■ When converting numeric data to character string data
			• Removes contiguous single-byte spaces from the beginning of the converted character string data. The removed spaces are inserted at the end of the character string.
			Example
			`CONVERT(1,CHAR(4),'LS000')`
			→ `'001Δ'`
			`CONVERT(1,CHAR(4),'LS999')`
			→ `'1ΔΔΔ'`
			• Single-byte spaces inside a character string enclosed in double quotation marks are not affected.
			■ When converting character string data to numeric data
			• The element `LS` is ignored. The character string data is converted to numeric data according to the rest of the number format.
			Specification rules
			`LS` can only be specified once in the number format.

16		`LJ`	■ When converting numeric data to character string data

No.	Type of element	Element that can be specified in the number format	Description
			- Removes single-byte spaces from the beginning and end of the converted character string data. **Example** `CONVERT(123,VARCHAR(3),'LJ999')` → `'123'` - Single-byte spaces inside a character string enclosed in double quotation marks (") are not affected. ■ When converting character string data to numeric data - The element `LJ` is ignored. The character string data is converted to numeric data according to the rest of the number format. Specification rules `LJ` can only be specified once in the number format.

17	Other	`B`	■ When converting numeric data to character string data
			- This element denotes a single-byte space when the value of the data to be converted (as a result of rounding to the number of digits in the number format) is 0. Sign elements and currency elements are also set to spaces.
			Examples
			`CONVERT(0,CHAR(4),'B999')`
			→ `'ΔΔΔΔ'`
			`CONVERT(0,VARCHAR(4),'LJB999')`
			→ `''`
			■ When converting character string data to numeric data
			- The element `B` is ignored. The character string data is converted to numeric data according to the rest of the number format.

18		`TM`
		`TM9`
		`TME`
			■ When converting numeric data to character string data
			- Converts the numeric data according to the specified element (`TM`, `TM9`, or `TME`).
			`TM` or `TM9`: Convert according to integer literal or decimal literal notation.
			`TME`: Convert according to the notation for floating-point numeric literals.
			Examples
			`CONVERT(1.28E2,CHAR(3),'TM')`
			→ `'128'`
			`CONVERT(128,CHAR(6),'TME')`
			→ `'1.28E2'`
			The result of converting the numeric data to the format of a numeric literal is output as character string data. At that time, it is converted to the shortest format able to represent the numeric literal.
			- When the numeric data cannot be represented in integer literal or decimal literal notation, it is converted to the notation for floating-point numeric literals even if `TM` or `TM9` is specified.
			■ When converting character string data to numeric data
			- The conversion result is the same whether `TM`, `TM9`, or `TME` is specified.
			- The character string data must conform to the following notations:
			• Integer literal notation
			• Decimal literal notation
			• Floating-point numeric literal notation
			Separators are permitted in the character string data but are not subject to conversion.
			Example
			`CONVERT('Result /* Comment */ 12345',INTEGER,'"Result"TM')`

No.	Type of element	Element that can be specified in the number format	Description
			→ 12345

(c) Rules pertaining to number format

- The length of the number format cannot exceed 64 bytes.
- Characters specified in the number format that are not enclosed in double quotation marks (`"`) must be single-byte.
- Uppercase and lowercase letters are treated the same in number format elements, except in the case of the elements `EEEE`, `X`, and character strings enclosed in double quotation marks.
- When converting character string data to numeric data, numeric elements are typically required in the number format. The exceptions are the hexadecimal element `X` and the shortest representation elements `TM`, `TM9`, `TME`, which do not require numeric elements.
- If no numeric element is specified immediately before the decimal point character (`.`), a numeric element must be specified immediately after the decimal point character (`.`).
- If you specify a currency element, decimal point character (`.`), or `B` element in the number format, a numeric element must be specified.
- If you specify a modifier element in the number format, you must specify a numeric element or hexadecimal element `X`.
- The sign element `S` can be specified either before or after a numeric element.
- The sign elements `MI` and `PR` can be specified only after a numeric element.
- If a sign element is specified, it must be one of the elements `S`, `MI`, or `PR`.
- The elements listed below can be specified two or more times in the character string specified in the number format. Elements other than these cannot be specified more than once.
  - Comma as a delimiting character element
  - Single-byte space as a delimiting character element
  - Numeric element (`0` or `9`)
  - `"`*character-string*`"` (character string enclosed in double quotation marks)
  - Hexadecimal element (`X` or `x`)

(5) Rules

(a) Common rules

1. The data type of the execution result will be the data type specified in *post-conversion-data-type*.
2. If a dynamic parameter is specified by itself for *data-to-convert*, *post-conversion-data-type* will be assumed to be the data type of the dynamic parameter.
3. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
4. If the data to be converted has a null value, or you specify `NULL` for *data-to-convert*, the execution result will be a null value.
5. If the data to be converted is character string data with a length of 0 bytes or 0 characters, it is converted as follows:

- When converting to CHAR type: it is converted to spaces. In the case of CHAR(3), it is converted to 'ΔΔΔ'. Δ represents a half-width space.
- When converting to VARCHAR type: it is converted to VARCHAR type data with a length of 0 bytes or 0 characters.
- When converting to BINARY type: It is converted to X'00'. In the case of BINARY(3), it is converted to X'000000'.
- When converting to VARBINARY type: It is converted to VARBINARY type data with a length of 0 bytes or 0 characters.
- In the case of other data types, it is converted to the null value.

6. The data types that can be converted (with no format specified) are shown in the following table:

Table 8-52: Data types that can be converted (with no format specified)

Data type of the data to be converted	Post-conversion data type					
	INTEGER, SMALLINT	DECIMAL, DOUBLE PRECISION	CHAR, VARCHAR	DATE, TIMESTAMP	TIME	BINARY, VARBINARY
INTEGER, SMALLINT	Y	Y	Y	Y	N	N
DECIMAL, DOUBLE PRECISION	Y	Y	Y	N	N	N
CHAR, VARCHAR	Y	Y	Y	Y	Y	Y
DATE, TIMESTAMP	Y	N	Y	Y	N	N
TIME	N	N	Y	N	Y	N
BINARY, VARBINARY	N	N	Y	N	N	Y

Legend:

Y: Can be converted.

N: Cannot be converted.

7. The data types that can be converted (with a format specified) are shown in the following table.

Table 8-53: Data types that can be converted (with a format specified)

Data type of the data to be converted	Post-conversion data type					
	INTEGER, SMALLINT	DECIMAL, DOUBLE PRECISION	CHAR, VARCHAR	DATE, TIMESTAMP	TIME	BINARY, VARBINARY
INTEGER, SMALLINT	N	N	Y#1	N	N	N
DECIMAL, DOUBLE PRECISION	N	N	Y#1	N	N	N
CHAR, VARCHAR	Y#1	Y#1	N	Y#2	Y#2	N

Data type of the data to be converted	Post-conversion data type					
	INTEGER, SMALLINT	DECIMAL, DOUBLE PRECISION	CHAR, VARCHAR	DATE, TIMESTAMP	TIME	BINARY, VARBINARY

`DATE,`
`TIMESTAMP`
	N	N	Y#2	N	N	N

`TIME`	N	N	Y#2	N	N	N

`BINARY,`
`VARBINARY`
	N	N	N	N	N	N

Legend:

Y: Can be converted.

N: Cannot be converted.

#1:

Can be converted when a number format is specified.

#2:

Can be converted when a datetime format is specified.

8. If a format specification is used, the data is first converted according to the specification, and then converted to the post-conversion data type according to the storage assignment rules.

   For details about the format specification in the case of the datetime format, see (3) Datetime format elements and rules. For details about the format specification in the case of the number format, see (4) Number format elements and rules.

   For details about the storage assignment rules, see (2) Storage assignments between data types in 6.2.2 Data types that can be converted, assigned, and compared.

(b) Rules for converting numeric data

■ **To convert numeric data to numeric data:**

Conversion of numeric data to numeric data is governed by the rules described in Storage assignment of numeric data in (2) Storage assignments between data types in 6.2.2 Data types that can be converted, assigned, and compared.

■ **To convert character string data to numeric data (with no number format specified):**

- Any character string data to be converted (after leading and trailing spaces are removed) must obey the rules for the description format of numeric literals. For the description format rules for numeric literals, see 6.3.2 Description format of literals.

  Examples of character string data that can be converted:

  `'219', '+56', '-3547', '-11.35', '887ΔΔ', 'Δ95Δ'`

  Examples of character string data that cannot be converted:

  `'a89', '77g9', '33Δ49'`

  Legend: Δ: Single-byte space

- If the character string data item is composed of only spaces, the null value is returned.

- Once the character string representation of the numeric literal has been converted to a numeric value, it is converted to the post-conversion data type. At that point, it is governed by the rules described in Storage assignment of numeric data in (2) Storage assignments between data types in 6.2.2 Data types that can be converted, assigned, and compared.

  Example:

```
CONVERT('11.35',INTEGER) → 11
```

Once the character string `'11.35'` has been converted to the `DECIMAL` type numeric value 11.35, it is converted to an `INTEGER` type numeric value. At that point, it is governed by the rules for storage assignment of numeric data, which in this case means that the decimal part is truncated.

■ When converting character string data to numeric data (with a number format specified)

- The format of the character string data to be converted must match the number format specification. However, conversion is possible even when there are single-byte spaces surrounding the number format or the character string data to be converted.

  Example:

  ```
  CONVERT('Δ1,234Δ',INTEGER,'9,999') → 1234
  CONVERT('ΔΔ1,234',INTEGER,'Δ9,999Δ') → 1234
  ```

  Legend: Δ: Single-byte space

- If the character string data is composed of only a single-byte space, the null value is returned.

- If there is a character string enclosed in double quotation marks in the number format, that character string, along with any surrounding spaces, is excluded from the character string data that is converted to numeric data according to the number format.

- Once the character string data has been converted to a numeric value according to number format, it is converted to the post-conversion data type. At that point, the rules described in Storage assignment of numeric data in (2) Storage assignments between data types in 6.2.2 Data types that can be converted, assigned, and compared apply.

  Example:

  ```
  CONVERT('1,000.22',INTEGER,'9,999.99') → 1000
  ```

  After the character string `'1,000.22'` is converted to the `DECIMAL` type numeric value `1000.22`, it is converted to an `INTEGER` type numeric value. At that point, it is governed by the rules for storage assignment of numeric data, which in this case means that the decimal places are truncated.

■ To convert datetime data to numeric data:

Datetime data is converted to the cumulative number of days since January 1, year 1 (CE). In the case of January 1, year 1 (CE), the cumulative number of days is 1. In the case of January 2, year 1 (CE), the cumulative number of days is 2.

Examples:

```
CONVERT(DATE'0001-01-03',INTEGER) → 3
CONVERT(TIMESTAMP'0001-01-05 11:03:58',INTEGER) → 5
```

(c) Rules for converting to character string data

The rules for converting to character string data (rules about the length of data) are shown in the following table.

Table 8-54: Rules for converting to character string data (rules about the length of data)

Condition at the time of conversion	Rules for converting to character string data	
	If data of character string type or binary type is converted	If data of other types is converted
A < *B*	If the post-conversion data type is `CHAR`, it is left-aligned and padded with spaces on the right.	
A = *B*	The conversion is performed	
A > *B*	The data is left-aligned and the excess portion on the right is truncated.#1	The data cannot be converted. Conversion will result in an error.#2

Legend:

A: Length of the source data that is to be converted to character string data

B: Data length of the post-conversion data type

#1

If truncation occurs in the middle of a multi-byte character, part of the multi-byte character is returned as the value of the execution result.

#2

If the data type of the data to be converted is `DOUBLE PRECISION` and no number format is specified, the number of decimal places of the mantissa is truncated to fit the data length specified in *post-conversion-data-type* (rounding to the nearest even number), so no error is generated. However, an error will be generated if the length of the data to be converted exceeds the data length specified in *post-conversion-data-type* even after all the decimal places of the mantissa have been truncated.

■ To convert `INTEGER`, `SMALLINT`, or `DECIMAL` type numeric data to character string data (with no number format specified)

- The result of converting numeric data to the format of a numeric literal is output as character string data. At that point, the results are output in the shortest format that can represent the numeric literal.

  However, conversion of `DECIMAL` type data is performed as follows:

  ■ The number of digits after the decimal point equals the scaling of the data type of the numeric data, and trailing zeros are not stripped.

  ■ If the precision of the data type of the numeric data is greater than the scaling, the number of digits in the integer part will not be 0.

  ■ The decimal point is always added.

  Example: `+0025.100` → `'25.100'`

  As shown in the example, the plus sign (+) is removed. Any zeros are also stripped from the beginning of the integer part.

- If the data to be converted is less than `0`, it is prefixed with a minus sign (`-`).

■ To convert `DOUBLE PRECISION` type numeric data to character string data (with no number format specified)

- The result of converting numeric data to the format of a floating-point numeric literal is output as character string data. At that point, the results are output in the shortest format that can represent the floating-point numeric literal.

  Examples:

  ```
  +1.0000000000000000E+010 → '1E10'
  +3.2000000000000000E+001 → '3.2E1'
  +0.1000000000000000E+001 → '1E0'
  +0.0000000000000000E+000 → '0E0'
  ```

  As shown in the examples, the sign is removed from the mantissa and any trailing zeros are removed from the decimal part. Also, the plus sign (+) and leading zeros are removed from the exponent.

- If the data to be converted is less than `0`, it is prefixed with a minus sign (`-`).

- Exponents that are less than `0` are prefixed with a minus sign (`-`).

■ To convert numeric data to character string data (with a number format specified)

- The numeric data is converted to the format of a numeric literal, and then converted to character string data according to the specified number format.

  ```
  CONVERT(1000,VARCHAR(6),'LJ$9,999') → '$1,000'
  ```

- If the numeric data cannot be converted according to the number format, it returns the character string padded with hash marks (#). Following the number format, delimiters, currency elements, decimal points, signs, numeric elements, and character strings enclosed in double quotation marks are replaced with hash marks (#). If double-byte characters are specified in the character string, they are replaced with hash marks (#) in proportion to their character size (in bytes).

  Example:

  ```
  CONVERT(1000,CHAR(3),'99') → '###'
  ```

■ To convert datetime data to character string data (with no datetime format specified):

- When datetime data is converted to character string data, it is converted to the format of the predefined output representation. When DATE type data is converted to character string data, it is converted to the format of the predefined output representation of a date. When TIME type data is converted to character string data, it is converted to the format of the predefined output representation of a time. When TIMESTAMP type data is converted to character string data, it is converted to the format of the predefined output representation of a time stamp. For details about the predefined output representations, see 6.3.3 Predefined character-string representations.

  Examples:

  ```
  CONVERT(DATE'2013-06-30',CHAR(10)) → '2013-06-30'
  CONVERT(DATE'0001-01-01',CHAR(10)) → '0001-01-01'
  CONVERT(TIME'05:33:48.123',CHAR(12)) → '05:33:48.123'
  CONVERT(TIMESTAMP'2013-06-30 11:03:58',CHAR(19)) → '2013-06-30 11:03:58'
  ```

- Conversions of datetime data to CHAR(*n*) or VARCHAR(*n*) must meet the following conditions:

Data type of the data to be converted		Condition on the post-conversion data length
DATE		$n \geq 10$
TIME(*p*)	When $p = 0$	$n \geq 8$
	When $p > 0$	$n \geq 9 + p$
TIMESTAMP(*p*)	When $p = 0$	$n \geq 19$
	When $p > 0$	$n \geq 20 + p$

  When *n* is less than the lengths indicated above, conversion is not possible.

- When converting DATE type data to CHAR type, if the data length of the post-conversion data is 11 bytes or greater, it is left-aligned and padded with spaces on the right.

  Example:

  ```
  CONVERT(DATE'2013-06-30',CHAR(15)) → '2013-06-30ΔΔΔΔΔ'
  ```

  Legend: Δ: Single-byte space

- When converting TIME type data with fractional seconds precision *p* to CHAR type, if the data length of the post-conversion data is greater than or equal to 10 + *p* bytes (or greater than or equal to 9 bytes when *p* = 0), it is left-aligned and padded with spaces on the right.

  Example:

  ```
  CONVERT(TIME'11:03:58.123',CHAR(13)) → '11:03:58.123Δ'
  ```

  Legend: Δ: Single-byte space

- When converting TIMESTAMP type data with fractional seconds precision *p* to CHAR type, if the data length of the post-conversion data is greater than or equal to 21 + *p* bytes (or greater than or equal to 20 bytes when *p* = 0), it is left-aligned and padded with spaces on the right.

  Example:

`CONVERT(TIMESTAMP'2013-06-30 11:03:58',CHAR(20)) → '2013-06-30 11:03:58Δ'`

Legend: Δ: Single-byte space

■ To convert datetime data to character string data (with a datetime format specified):

- Datetime data is converted to character string data according to the specified datetime format.
- When you specify a datetime format element that is not in the datetime data to be converted, that element is set to a default character string.

  Example:

  `CONVERT(DATE'2013-07-30',CHAR(16),'YYYY/MM/DD HH:MI') → '2013/07/30 00:00'`

  The datetime data to be converted in the above example is `DATE` type, which has no time elements, but because time elements (`HH` and `MI`) are specified in the datetime format, those portions are set to `'00'` by default.

  The default character strings are shown in the following table:

Table 8-55: Default character strings for datetime format elements

No.	Datetime format element		Default character string
1	Time	`HH`	`'00'`
2		`HH24`	
3		`HH12`	`'12'`
4	AM/PM	`AM`	`'AM'`
5		`A.M.`	`'A.M.'`
6		`PM`	`'AM'`
7		`P.M.`	`'A.M.'`
8		`AMN`	`'午前'`
9		`PMN`	
10	Minute	`MI`	`'00'`
11	Second	`SS`	`'00'`
12		`SSSSS`	`'00000'`
13	Fractional seconds	`FF1`	Any digits to the right of the fractional seconds precision of the target data are padded with zeros.
14		`FF2`	
15		`FF3`	
16		`FF4`	
17		`FF5`	
18		`FF6`	
19		`FF7`	
20		`FF8`	
21		`FF9`	
22		`FF10`	
23		`FF11`	
24		`FF12`	

- When converting datetime data to CHAR type, if the data length after conversion is less than the data length specified for the *post-conversion-data-type*, it is left-aligned and padded with spaces on the right.

  Example:

  `CONVERT(DATE'2013-07-30',CHAR(12),'YYYY/MM/DD') → '2013/07/30ΔΔ'`

  Legend: Δ: Single-byte space

- If the datetime format element MON, MONTH, DAY, or DY is specified when converting datetime data to character string data, depending on whether the first and second letters of the elements are uppercase or lowercase, the post-conversion character string will vary as follows:

  • If the first letter is lowercase, the post-conversion character string will be entirely lowercase.

  • If the first letter is uppercase and the second letter is lowercase, the first letter of the post-conversion character string will be uppercase, and the second and subsequent letters will be lowercase.

  • If the first and second letters are uppercase, the post-conversion character string will be entirely uppercase.

  This is illustrated in the following examples:

Specified datetime format element	Post-conversion character string
mon	'jan'
Mon	'Jan'
MON or MOn	'JAN'

  The above examples illustrate the case for January.

- If you specify FF1 to FF11 in the datetime format and the number of digits in the fractional seconds of the data to be converted exceeds the number of digits specified in the datetime format, the excess digits in the fractional seconds of the datetime format are truncated.

  Example:

  `CONVERT(TIME'15:16:17.123456',CHAR(9),'HHMISS.FF2') → '151617.12'`

■ To convert binary data to character string data

- Only the data type is converted, and the data itself (character encoding itself) is not converted.

  Example:

  `CONVERT(X'61626364',CHAR(4)) ==> 'abcd'`

- If length-of-data-before-type-conversion > length-of-data-after-type-conversion, the excess portion on the right is truncated.

  Example:

  CONVERT(X'61626<u>364</u>',CHAR(3)) ==> 'abc'

  The underlined portion is truncated.

- If *length-of-data-before-type-conversion* < *length-of-data-after-type-conversion*, the results are padded with half-width spaces on the right.

  Example:

  `CONVERT(X'61626364',CHAR(5)) ==> 'abcdΔ'`

  Legend: Δ: Half-width space

(d) Rules for converting to datetime data

■ To convert INTEGER or SMALLINT type numeric data to datetime data

- The numeric data is converted to DATE or TIMESTAMP type data based on a starting point of January 1, 0001.

  Example:

`CONVERT(2,DATE) → DATE'0001-01-02'`

- The time portion of the `TIMESTAMP` type is converted to `00:00:00`, and the fractional seconds are filled with zeros.

  Example:

  `CONVERT(2,TIMESTAMP(3)) → TIMESTAMP'0001-01-02 00:00:00.000'`

- `INTEGER` and `SMALLINT` type data in the range `1` to `3652059` can be converted. Values outside this range generate an error.

■ To convert character string data to datetime data (with no datetime format specified):

- The character string data to be converted (after leading and trailing spaces are removed) can be converted to `DATE` type data only when it adheres to the predefined input representation format of a date. For details about the predefined input representation of a date, see (a) Predefined input representation in (1) Predefined character-string representation of dates in 6.3.3 Predefined character-string representations.

  Example:

  `CONVERT('2014-07-22ΔΔ',DATE) → DATE'2014-07-22'`

  Examples of character string data that can be converted:

  `'2014-06-30', '0001-01-02', 'ΔΔ2014-07-30', 'Δ2014/07/30ΔΔ'`

  Examples of character string data that cannot be converted:

  `'2013Δ06Δ30', '2013.06.30'`

  Legend: Δ: Single-byte space

- The character string data to be converted (after leading and trailing spaces are removed) can be converted to `TIME` type data only when it adheres to the predefined input representation format of a time. For details about the predefined input representation of a time, see (a) Predefined input representation in (2) Predefined character-string representation of times in 6.3.3 Predefined character-string representations.

  Example:

  `CONVERT('Δ19:46:23.123456',TIME(6)) → TIME'19:46:23.123456'`

  Examples of character string data that can be converted:

  `'18:05:22', '10:21:44.123', 'ΔΔ10:21:44.123456Δ'`

  Examples of character string data that cannot be converted:

  `'18Δ05Δ22', '10:21:44Δ123456'`

  Legend: Δ: Single-byte space

- The character string data to be converted (after leading and trailing spaces are removed) can be converted to `TIMESTAMP` type data only when it adheres to the predefined input representation format of a time stamp. For details about the predefined input representation of a time stamp, see (a) Predefined input representation in (3) Predefined character-string representation of time stamps in 6.3.3 Predefined character-string representations.

  Example:

  ```
  CONVERT('2014/08/02 11:03:58.123456Δ',TIMESTAMP(6)) →
  TIMESTAMP'2014-08-02 11:03:58.123456'
  ```

  Examples of character string data that can be converted:

  ```
  '2014-06-30 11:03:58', '2014/07/30 11:03:58.123', 'Δ2014/07/30
  11:03:58.123456789ΔΔ'
  ```

  Examples of character string data that cannot be converted:

  `'2014-06-30 11-03-58','2014/07/30 11:03:58:123456'`

  Legend: Δ: Single-byte space

- If the number of digits in the fractional seconds of the character string data to be converted is greater than the number of digits in the fractional seconds of *post-conversion-data-type*, the fractional seconds beyond the number of digits in the fractional seconds of *post-conversion-data-type* are truncated.

  Example:

  ```
  CONVERT('19:46:23.123456',TIME(3)) → TIME'19:46:23.123'
  ```

- If the number of digits in the fractional seconds of the character string data to be converted is less than the number of digits in the fractional seconds of *post-conversion-data-type*, the fractional seconds are padded with zeros as necessary.

  Example:

  ```
  CONVERT('2014-08-02 11:03:58.123',TIMESTAMP(9)) → TIMESTAMP'2014-08-02
  11:03:58.123000000'
  ```

- If the character string data item is composed of only spaces, the null value is returned.

■ To convert character string data to datetime data (with a datetime format specified):

- To convert character string data to `DATE` type, specify the year, month, and day elements in the datetime format. If you specify other elements (for example, time), they will not affect the results. For details about the elements of the datetime format, see Table 8-49: Datetime format elements with the same meaning.

- To convert character string data to `TIME` type, specify the hour, minute, and second elements in the datetime format. If you specify other elements (for example, the day), they will not affect the results. For details about the elements of the datetime format, see Table 8-49: Datetime format elements with the same meaning.

- To convert character string data to `TIMESTAMP` type, specify the year, month, day, hour, minute, and second elements in the datetime format. For details about the elements of the datetime format, see Table 8-49: Datetime format elements with the same meaning.

- Consecutive single-byte spaces are stripped from the beginning and end of the character string data to be converted, and then the data is converted to datetime data according to the datetime format. In addition, parts inside the datetime format that correspond to consecutive single-byte spaces at the beginning or end of the character string data are ignored. Therefore, the following example does not generate an error.

  Example:

  ```
  CONVERT('Δ19Δ46Δ23ΔΔΔ',TIME(12),'"ΔΔΔ"FMΔHHΔMIΔSSΔFFΔ')
  → TIME'19:46:23.000000000000'
  ```

  Legend: Δ: Single-byte space

  > **Note**
  >
  > In the above example, the single-byte spaces are handled as follows:
  >
  > 1. Consecutive single-byte spaces at the beginning and end of the character string data to be converted are ignored.
  >
  >   `'Δ19Δ46Δ23ΔΔΔ' → '19Δ46Δ23'`
  >
  > 2. Consecutive single-byte spaces at the beginning and end of the datetime format are ignored.
  >
  >   `'"ΔΔΔ"FMΔHHΔMIΔSSΔFFΔ' → 'FMΔHHΔMIΔSSΔFF'`
  >
  >   The "ΔΔΔ" part is ignored because it corresponds to consecutive single-byte spaces at the beginning of the character string data. The final single-byte space is ignored because it corresponds to consecutive single-byte spaces at the end of the character string data.
  >
  > 3. Because there are no characters corresponding to `FM`, the single-byte space after `FM` corresponds to consecutive single-byte spaces at the beginning of the character string data and is ignored.
  >
  >   `'FMΔHHΔMIΔSSΔFF' → 'FMHHΔMIΔSSΔFF'`

4. Because there are no fractional seconds in the character string data to be converted, the single-byte space before `FF` corresponds to consecutive single-byte spaces at the end of the character string data and is ignored.

   `'FMHHΔMIΔSSΔFF' → 'FMHHΔMIΔSSFF'`

   If fractional seconds were specified, the single-byte space before `FF` would not correspond to consecutive single-byte spaces at the end of the character string data.

- If the character string data item is composed of only spaces, the null value is returned.
- If no fractional second elements are specified in the datetime format, and the data is converted to `TIME` or `TIMESTAMP` type data with a fractional seconds precision of `3` or more, the values of the fractional seconds after the conversion will be 0.

  Example:

  `CONVERT('151617',TIME(3),'HHMISS') → TIME'15:16:17.000'`
- The conversion is the same regardless of whether uppercase or lowercase is used for the datetime format elements. Similarly, the conversion is the same regardless of whether uppercase or lowercase is used in the data to be converted. However, uppercase and lowercase are distinguished inside character strings enclosed in double quotation marks (`"`).
- If `FF` or one of `FF1` to `FF12` is specified in the datetime format, the numeric characters in the character string corresponding to the datetime format are extracted during the conversion. At this time, numeric characters are extracted until a non-numeric character is encountered, or until the length associated with the element in the datetime format is reached. If the length of the numeric characters in a character string is shorter than the length associated with the corresponding element in the datetime format, the missing part is converted to `0`.

  Example:

  `CONVERT('151617.12',TIME(3),'HHMISS.FF3') → TIME'15:16:17.120'`
- If `FF` or one of `FF1` to `FF12` is specified in the datetime format, and the number of digits of fractional seconds in the character string data is less than the fractional seconds precision of the datetime data, the missing fractional seconds are converted to 0.

  Example:

  `CONVERT('151617.123',TIME(6),'HHMISS.FF3') → TIME'15:16:17.123000'`
- If `FF` or one of `FF1` to `FF12` is specified in the datetime format, and the number of digits of fractional seconds in the character string data is greater than the fractional seconds precision of the datetime data, the excess fractional seconds in the datetime data are not converted.

  Example:

  `CONVERT('151617.123456',TIME(3),'HHMISS.FF6') → TIME'15:16:17.123'`

■ To convert datetime data to datetime data:

The conversion rules for converting datetime data to datetime data are given in the following table.

Table 8-56: Conversion rules for converting datetime data to datetime data

Data type of the data to be converted	Specified post-conversion data type	Conversion rules
`DATE`	`DATE`	No conversion is performed.
	`TIMESTAMP(`*p2*`)`	• The time part is converted to `00:00:00`. • The fractional seconds are filled with zeros.
`TIME(`*p1*`)`	`TIME(`*p2*`)`	• When *p1* = *p2* No conversion is performed. • When *p1* > *p2*

Data type of the data to be converted	Specified post-conversion data type	Conversion rules
		The fractional seconds beyond *p2* are truncated. • When *p1* < *p2* The missing fractional seconds are padded with zeros.
`TIMESTAMP(`*p1*`)`	`DATE`	Only the date part is converted.
	`TIMESTAMP(`*p2*`)`	• When *p1* = *p2* No conversion is performed. • When *p1* > *p2* The fractional seconds beyond *p2* are truncated. • When *p1* < *p2* The missing fractional seconds are padded with zeros.

Legend:

p1, *p2*: Fractional seconds precision

(e) Rules for converting to binary data

■ To convert character string data to binary data

- Only the data type is converted, and the data itself (character encoding itself) is not converted.

  Example:

  ```
  CONVERT('abcd',BINARY(4)) ==> X'61626364'
  ```

- If *length-of-data-before-type-conversion* > *length-of-data-after-type-conversion*, the excess portion on the right is truncated.

  Example:

  ```
  CONVERT('abcd',BINARY(3)) ==> X'616263'
  ```

  The underlined portion is truncated.

  If truncation occurs in the middle of a multi-byte character, part of the multi-byte character is returned as the value of the execution result.

- If *length-of-data-before-type-conversion* < *length-of-data-after-type-conversion*, the results are padded with `X'00'` on the right.

  Example:

  ```
  CONVERT('abcd',BINARY(5)) ==> X'6162636400'
  ```

■ To convert binary data to binary data

- If *length-of-data-before-type-conversion* > *length-of-data-after-type-conversion*, the excess portion on the right is truncated.

  Example:

  ```
  CONVERT(X'61626364',BINARY(3)) ==> X'616263'
  ```

  The underlined portion is truncated.

  If truncation occurs in the middle of a multi-byte character, part of the multi-byte character is returned as the value of the execution result.

- If *length-of-data-before-type-conversion* < *length-of-data-after-type-conversion*, the results are padded with `X'00'` on the right.

  Example:

  ```
  CONVERT(X'61626364',BINARY(5)) ==> X'6162636400'
  ```

(6) Examples

Example 1:

Convert the data in column C2 in table T1 from CHAR type to DATE type and retrieve the rows where column C2 is July 20, 2013.

In column C2, the CHAR type data representing the date is stored in the format MM/DD/YYYY.

```
SELECT * FROM "T1"
    WHERE CONVERT("C2",DATE,'MM/DD/YYYY')=DATE'2013-07-20'
```

Table T1

Column C1	Column C2
CHAR	CHAR
A10101	06/14/2013
A15014	07/20/2013
A31399	07/11/2013

Retrieval results

A15014	07/20/2013

Example 2:

Retrieve the rows from table T1 where column C1 is A10101, and convert the data in the corresponding column C2 from INTEGER type to CHAR type. During conversion, prefix the character string with the currency symbol $ and separate every three digits with a comma.

```
SELECT "C1",CONVERT("C2",CHAR(13),'$999,999,999') FROM "T1"
    WHERE "C1"='A10101'
```

Table T1

Col. C1	Col. C2
CHAR	INTEGER
A10101	123000000
A15014	555550000
A31399	277965400

Retrieval results

A10101	Δ$123,000,000

Note: Δ represents a single-byte space

Example 3:

In this example, column C2 in table T1 holds CHAR type data representing the price, including the currency symbol $ and commas between every three digits. Convert column C2 from CHAR type to INTEGER type and retrieve the rows for which the discounted price is greater than or equal to $1,000.

```
SELECT * FROM "T1"
    WHERE CONVERT("C2",INTEGER,'$9,999')*0.7>=1000
```

Table T1

Col. C1 CHAR	Col. C2 CHAR
A10101	$1,000
A15014	$2,000
A31399	$3,000

Retrieval results

A15014	$2,000
A31399	$3,000

8.12.6 HEX

Converts binary data to a hexadecimal string representation (character string data consisting of 0 to 9, and A to F).

(1) Specification format

```
scalar-function-HEX ::= HEX(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the target binary data.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- Specify BINARY or VARBINARY type data for the target data.
- You cannot specify a dynamic parameter by itself for the target data.
- You cannot specify binary data whose defined length is 16,001 bytes or greater for the target data.

The following example illustrates the result of executing the scalar function HEX.

Examples:

HEX(B'10100100') → 'A4'

HEX(X'1234') → '1234'

(3) Rules

1. The data type and data length of the execution result are shown in the following table.

Table 8-57: Data type and data length of the execution result of the scalar function HEX

Data type and data length of target data			Data type and data length of the execution result		
Data type	**Defined length**	**Actual length**	**Data type**	**Defined length**	**Actual length**
BINARY(*n*)	$1 \leq n \leq 16{,}000$	Not applicable.	VARCHAR	$n \times 2$	$n \times 2$

Data type and data length of target data			Data type and data length of the execution result		
Data type	**Defined length**	**Actual length**	**Data type**	**Defined length**	**Actual length**
`VARBINARY(`*n*`)`	1 ≤ *n* ≤ 16,000	*r*			*r* × 2

Legend:

n: Defined length of target data

r: Actual length of target data

2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If the target data has a null value, the execution result will be a null value.
4. If the actual length of the target data is 0 bytes, the execution result will be data with an actual length of 0 bytes.

8.13 NULL evaluation functions

This section describes the functions and specification formats of the NULL evaluation functions.

8.13.1 COALESCE

Evaluates the specified target data in the order *target-data-1*, *target-data-2*, ..., and then returns the first non-null value.

(1) Specification format

```
scalar-function-COALESCE ::= COALESCE(target-data-1[,target-data-2]...)

  target-data-1 ::= value-expression
  target-data-2 ::= value-expression
```

(2) Explanation of specification format

target-data-1, *target-data-2*...:

Specifies the target data.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- In the target data, specify data whose data types can be compared. For details about data types that can be compared, see (1) Data types that can be compared in 6.2.2 Data types that can be converted, assigned, and compared. However, note the following exceptions:

  • DATE type data cannot be compared to character string data (even to the predefined input representation of a date).

  • TIME type data cannot be compared to character string data (even to the predefined input representation of a time).

  • TIMESTAMP type data cannot be compared to character string data (even to the predefined input representation of a time stamp).
- You cannot specify a dynamic parameter by itself for *target-data-1*.
- If you specify a dynamic parameter for *target-data-2*, or later, the data type of the dynamic parameter is assumed to be the data type of *target-data-1*.
- A maximum of 255 target data items can be specified.

(3) Rules

1. The data type and data length of the execution result are determined according to the rules described in 7.20.2 Data types of the results of value expressions.
2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
3. If all the target data has a null value, the execution result will be a null value.
4. COALESCE(*target-data-1*,*target-data-2*) is equivalent to the following CASE expression:

```
CASE
   WHEN target-data-1 IS NOT NULL THEN target-data-1
   ELSE target-data-2
END
```

5. COALESCE(*target-data-1*,*target-data-2*,...,*target-data-n*) is equivalent to the following CASE expression (where *n* is greater than or equal to 3).

```
CASE
   WHEN target-data-1 IS NOT NULL THEN target-data-1
   ELSE COALESCE(target-data-2,...,target-data-n)
END
```

(4) Example

Example:

Execute the scalar function COALESCE on the values of columns C1 to C3 in table T1.

```
SELECT COALESCE("C1","C2","C3") FROM "T1"
```

Table T1

Column C1	Column C2	Column C3
INTEGER	INTEGER	INTEGER
10	20	30
10	NULL	30
NULL	20	30
NULL	NULL	30
NULL	NULL	NULL

Retrieval results

10
10
20
30
NULL

8.13.2 ISNULL

Evaluates the specified target data in the order *target-data-1*, *target-data-2*, and then returns the first non-null value.

Note

The scalar functions ISNULL and NVL are functionally equivalent.

(1) Specification format

```
scalar-function-ISNULL ::= ISNULL(target-data-1,target-data-2)

  target-data-1 ::= value-expression
  target-data-2 ::= value-expression
```

(2) Explanation of specification format

target-data-1, *target-data-2*:

Specifies the target data.

The following rules apply:

- Specify *target-data-1* and *target-data-2* in the form of value expressions. For details about value expressions, see 7.20 Value expression.
- In *target-data-1* and *target-data-2*, specify data whose data types can be compared. For details about data types that can be compared, see (1) Data types that can be compared in 6.2.2 Data types that can be converted, assigned, and compared.
- If the data type of *target-data-1* is `DATE`, `TIME`, or `TIMESTAMP`, you can specify a character string literal that adheres to the format of the predefined input representation for *target-data-2*. For details about the predefined input representations, see 6.3.3 Predefined character-string representations.
- You cannot specify a dynamic parameter by itself for *target-data-1*.
- If you specify a dynamic parameter for *target-data-2*, the data type of the dynamic parameter is assumed to be the data type of *target-data-1*.
- You must specify a value for *target-data-2* that is capable of being assigned to the data type of *target-data-1*. For details about storage assignments, see (2) Storage assignments between data types in 6.2.2 Data types that can be converted, assigned, and compared.

(3) Rules

1. The data type and data length of the execution result will be the data type and data length of *target-data-1*.
2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If *target-data-1* and *target-data-2* have null values, the execution result will be a null value.
4. If *target-data-1* has a null value, the value of *target-data-2* is converted to the data type and data length of *target-data-1*.

(4) Example

Example:

Execute the scalar function `ISNULL` on the values in column `C1` and column `C2` in table `T1`.

```
SELECT ISNULL("C1","C2") FROM "T1"
```

Table `T1`

Column `C1`	Column `C2`
`DECIMAL(3,1)`	`INTEGER`
10.5	20
10.5	NULL
NULL	20
NULL	NULL

Retrieval results

10.5
10.5
20.0
NULL

8.13.3 NULLIF

Compares *target-data-1* to *target-data-2* and return NULL if they are equal, or *target-data-1* if they are not equal.

(1) Specification format

```
scalar-function-NULLIF ::= NULLIF(target-data-1,target-data-2)

  target-data-1 ::= value-expression
  target-data-2 ::= value-expression
```

(2) Explanation of specification format

target-data-1, *target-data-2*:

Specifies the target data to be compared.

The following rules apply:

- Specify *target-data-1* and *target-data-2* in the form of value expressions. For details about value expressions, see 7.20 Value expression.
- In *target-data-1* and *target-data-2*, specify data whose data types can be compared. For details about data types that can be compared, see (1) Data types that can be compared in 6.2.2 Data types that can be converted, assigned, and compared. However, note the following exceptions:

  • `DATE` type data cannot be compared to character string data (even to the predefined input representation of a date).

  • `TIME` type data cannot be compared to character string data (even to the predefined input representation of a time).

  • `TIMESTAMP` type data cannot be compared to character string data (even to the predefined input representation of a time stamp).
- You cannot specify dynamic parameters by themselves for both *target-data-1* and *target-data-2*.
- If you specify a dynamic parameter for either *target-data-1* or *target-data-2*, the data type of the other one will be assumed to be the data type of the dynamic parameter.

(3) Rules

1. The data type and data length of the execution result are determined according to the rules described in 7.20.2 Data types of the results of value expressions.
2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If *target-data-1* has a null value, the execution result will be a null value.
4. `NULLIF(`*target-data-1*`,`*target-data-2*`)` is equivalent to the following `CASE` expression.

```
CASE
   WHEN target-data-1 = target-data-2 THEN NULL
   ELSE target-data-1
END
```

(4) Example

Example:

Compare the values of columns `C1` and `C2` in table `T1`.

```
SELECT NULLIF("C1","C2") FROM "T1"
```

Table T1

Column C1 INTEGER	Column C2 INTEGER
10	10
10	20

Retrieval results

NULL
10

8.13.4 NVL

Evaluates the specified target data in the order *target-data-1*, *target-data-2*, and then returns the first non-null value.

> **Note**
>
> The scalar functions NVL and ISNULL are functionally equivalent.

(1) Specification format

```
scalar-function-NVL ::= NVL(target-data-1,target-data-2)

  target-data-1 ::= value-expression
  target-data-2 ::= value-expression
```

(2) Explanation of specification format

target-data-1, *target-data-2*:

Specifies the target data.

The following rules apply:

- Specify *target-data-1* and *target-data-2* in the form of value expressions. For details about value expressions, see 7.20 Value expression.
- In *target-data-1* and *target-data-2*, specify data whose data types can be compared. For details about data types that can be compared, see (1) Data types that can be compared in 6.2.2 Data types that can be converted, assigned, and compared.
- If the data type of *target-data-1* is DATE, TIME, or TIMESTAMP, you can specify a character string literal that adheres to the format of the predefined input representation for *target-data-2*. For details about the predefined input representations, see 6.3.3 Predefined character-string representations.
- You cannot specify a dynamic parameter by itself for *target-data-1*.
- If you specify a dynamic parameter for *target-data-2*, the data type of the dynamic parameter is assumed to be the data type of *target-data-1*.
- You must specify a value for *target-data-2* that is capable of being assigned to the data type of *target-data-1*. For details about storage assignments, see (2) Storage assignments between data types in 6.2.2 Data types that can be converted, assigned, and compared.

(3) Rules

1. The data type and data length of the execution result will be the data type and data length of *target-data-1*.
2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If *target-data-1* and *target-data-2* have null values, the execution result will be a null value.
4. If *target-data-1* has a null value, the value of *target-data-2* is converted to the data type and data length of *target-data-1*.

(4) Example

Example:

Execute the scalar function `NVL` on the values in column `C1` and column `C2` in table `T1`.

```
SELECT NVL("C1","C2") FROM "T1"
```

Table `T1`

Column `C1`	Column `C2`
`DECIMAL(3,1)`	`INTEGER`
10.5	20
10.5	NULL
NULL	20
NULL	NULL

Retrieval results

10.5
10.5
20.0
NULL

8.14 Information acquisition functions

This section describes the functions and specification formats of the information acquisition functions.

8.14.1 LENGTHB

Returns the length of the target data in bytes.

(1) Specification format

```
scalar-function-LENGTHB ::= LENGTHB(target-data)

  target-data ::= value-expression
```

(2) Explanation of specification format

target-data:

Specifies the target data whose length is to be determined.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- For the target data, specify numeric data, character string data, datetime data, or binary data.
- You cannot specify a dynamic parameter by itself for the target data.

(3) Rules

1. The data type of the execution result is the `INTEGER` type.
2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If the target data has a null value, the execution result will be a null value.
4. The following table shows the value of the execution result for each target data type.

Table 8-58: Value of execution result for each target data type

No.	Data type of target data		Value of execution result (bytes)
1	`INTEGER`		`8`
2	`SMALLINT`		`4`
3	`DECIMAL`	When the precision is 1 to 4	`2`
4		When the precision is 5 to 8	`4`
5		When the precision is 9 to 16	`8`
6		When the precision is 17 to 38	`16`
7	`DOUBLE PRECISION`		`8`
8	`CHAR(`*n*`)`		*n*
9	`VARCHAR`		Actual length

No.	Data type of target data	Value of execution result (bytes)
10	`DATE`	4
11	`TIME(`*p*`)`	3 + ↑*p* ÷ 2↑
12	`TIMESTAMP(`*p*`)`	7 + ↑*p* ÷ 2↑
13	`BINARY(`*n*`)`	*n*
14	`VARBINARY`	Actual length

(4) Example

Example 1 (in a case where the target data is character string data)

Determine the actual lengths of the `VARCHAR` type data in column `C1` from table `T1`.

The assumed character encoding is Unicode (UTF-8).

```
SELECT LENGTHB("C1") FROM "T1"
```

Table `T1`

Col. `C1` VARCHAR
ABC
ⅠⅡⅢ
ABCⅠⅡⅢ

Retrieval results

3
9
12

Example 2 (in a case where the target data is binary data)

Determine the actual data length for each row of columns `C1` (`VARBINARY(5)`) and `C2` (`BINARY(5)`) in table `T1`.

```
SELECT LENGTHB("C1"), LENGTHB("C2") FROM "T1"
```

Table `T1`

Col. `C1` `VARBINARY(5)`	Col. `C2` `BINARY(5)`
`X'ABCD'`	`X'ABCD000000'`
`X'ABCDEF'`	`X'ABCDEF0000'`
`X'ABCDEF10'`	`X'ABCDEF1000'`

Retrieval results

2	5
3	5
4	5

8.15 Comparison functions

This section describes the functions and specification formats of the comparison functions.

8.15.1 DECODE

Compares the values in the target data and the comparison data one at a time, and if there is a match, returns the corresponding value as the return value. If no match is found between the target data and comparison data, returns the predefined return value.

When multiple comparison data items are specified, it returns the return value corresponding to the first comparison data item that is matched.

(1) Specification format

```
scalar-function-DECODE ::= DECODE(target-data,comparison-data,return-value
                             [,comparison-data,return-value]...
                             [,predefined-return-value])

  target-data ::= {value-expression | NULL}
  comparison-data ::= {value-expression | NULL}
  return-value ::= {value-expression | NULL}
  predefined-return-value ::= {value-expression | NULL}
```

(2) Explanation of specification format

target-data:

Specifies the target data. Specify the target data in the form of a value expression, or as `NULL`. For details about value expressions, see 7.20 Value expression.

comparison-data:

Specifies the comparison data.

The following rules apply:

- Specify the comparison data in the form of a value expression, or as `NULL`. For details about value expressions, see 7.20 Value expression.
- The first specification of *comparison-data* cannot be `NULL`.
- The first specification of *comparison-data* cannot be a dynamic parameter by itself.
- If you specify a dynamic parameter by itself for the second or subsequent specification of *comparison-data*, the data type of the dynamic parameter is assumed to be the data type of the first *comparison-data*.

return-value:

Specifies the value to return when the target data matches a comparison data item.

The following rules apply:

- Specify the return value in the form of a value expression, or as `NULL`. For details about value expressions, see 7.20 Value expression.
- The first-specified return value cannot be `NULL`.
- The first-specified return value cannot be a dynamic parameter by itself.

- If you specify a dynamic parameter by itself for the second or subsequent return value, the data type of the dynamic parameter is assumed to be the data type of the first return value.

predefined-return-value:

Specifies a predefined value to return when the target data does not match any of the comparison data. If *predefined-return-value* is omitted, `NULL` is assumed.

The following rules apply:

- Specify the predefined return value in the form of a value expression, or as `NULL`. For details about value expressions, see 7.20 Value expression.
- If you specify a dynamic parameter by itself for *predefined-return-value*, the data type of the dynamic parameter is assumed to be the data type of the first return value.

(3) Rules

1. You must specify numeric data, character string data, or datetime data for *target-data*, *comparison-data*, *return-value*, and *predefined-return-value*.
2. When `NULL` is specified for *target-data*, *comparison-data*, *return-value*, or *predefined-return-value*, it denotes the null value.
3. You must specify data types that can be compared (except when specifying `NULL`) for *target-data* and *comparison-data*. For details about data types that can be compared, see (1) Data types that can be compared in 6.2.2 Data types that can be converted, assigned, and compared.

   Note, however, that if *target-data* and *comparison-data* are character string data or datetime data, specify a combination of data types based on the following table.

Table 8-59: Combinations of data types that can be specified for the target data and the comparison data for the scalar function DECODE

Target data		**Comparison data**						
		Character string data				**Datetime data**		
		Character string literal that is the predefined input representation for date data	**Character string literal that is the predefined input representation for time data**	**Character string literal that is the predefined input representation for time stamp data**	**Other data**	**Date data**	**Time data**	**Time stamp data**
Character string data	Character string literal that is the predefined input representation for date data	Y	Y	Y	Y	N	N	N
	Character string literal that is the predefined input representatio	Y	Y	Y	Y	N	N	N

Target data		Comparison data						
		Character string data				Datetime data		
		Character string literal that is the predefined input representation for date data	Character string literal that is the predefined input representation for time data	Character string literal that is the predefined input representation for time stamp data	Other data	Date data	Time data	Time stamp data
	n for time data							
	Character string literal that is the predefined input representation for time stamp data	Y	Y	Y	Y	N	N	N
	Other data	Y	Y	Y	Y	N	N	N
Datetime data	Date data	Y	N	Y	N	Y	N	Y
	Time data	N	Y	N	N	N	Y	N
	Time stamp data	Y	N	Y	N	Y	N	Y

Legend:

Y: Can be specified.

N: Cannot be specified.

4. You must specify data types that can be compared (except when specifying `NULL`) for *return-value* and *predefined-return-value*. For details about data types that can be compared, see (1) Data types that can be compared in 6.2.2 Data types that can be converted, assigned, and compared.

   However, if *return-value* and *predefined-return-value* are character string data or datetime data, specify a combination of data types based on the following table.

Table 8-60: Combinations of data types that can be specified for the return value and the predefined return value for the scalar function DECODE

Return value		Predefined return value, or return value#						
		Character string data				Datetime data		
		Character string literal that is the predefined input representation for date data	Character string literal that is the predefined input representation for time data	Character string literal that is the predefined input representation for time stamp data	Other data	Date data	Time data	Time stamp data
Character string data	Character string literal that is the predefined input representation for date data	Y	Y	Y	Y	N	N	N
	Character string literal that is the predefined input representation for time data	Y	Y	Y	Y	N	N	N
	Character string literal that is the predefined input representation for time stamp data	Y	Y	Y	Y	N	N	N
	Other data	Y	Y	Y	Y	N	N	N
Datetime data	Date data	N	N	N	N	Y	N	Y
	Time data	N	N	N	N	N	Y	N
	Time stamp data	N	N	N	N	Y	N	Y

Legend:

Y: Can be specified.

N: Cannot be specified.

#

If multiple *return-value* items are specified and all *return-value* items are character string data or datetime data, the combinations of data types that can be specified for each *return-value* item are as shown in Table 8-60: Combinations of data types that can be specified for the return value and the predefined return value for the scalar function DECODE.

5. You can specify a maximum of 127 *comparison-data* and *return-value* pairs.

6. The data type and data length of the execution result depends on the data types of the results of *return-value* and *predefined-return-value*, and is determined according to the rules described in 7.20.2 Data types of the results of value expressions.

   Note that the specification of NULL for *return-value* and *predefined-return-value* does not affect the data type and data length of the execution result.

7. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).

8. If the target data is the null value and NULL is specified for one of the comparison data items, the return value associated with that item is returned.

(4) Examples

Example 1:

Convert the abbreviations of country names in column C2 from table T1 as follows:

- JPN → Japan
- IND → India
- Null value → NODATA
- Other → Other

```
SELECT "C1",DECODE("C2",'JPN','Japan','IND','India',NULL,'NODATA','Other')
    FROM "T1"
```

Table T1

Column C1	Column C2
N001	JPN
N003	JPN
N010	IND
N050	CHN
N085	IND
N100	NULL

Retrieval results

N001	Japan
N003	Japan
N010	India
N050	Other
N085	India
N100	NODATA

Example 2:

Search the employee table (EMPLIST) as follows:

- Determine the number of males and females in each section (SCODE)

```
SELECT "SCODE",SUM(DECODE("SEX",'M',1,0)) AS "Men",
               SUM(DECODE("SEX",'F',1,0)) AS "Women"
    FROM "EMPLIST"
    GROUP BY "SCODE"
```

Retrieval results

SCODE	Men	Women
S001	12	5
S002	21	18
S003	19	33

8.15.2 GREATEST

Returns the greatest value among the specified target data.

In addition to comparing numeric data items, you can also compare character string data items and datetime data items.

(1) Specification format

```
scalar-function-GREATEST ::= GREATEST(target-data-1[,target-data-2]...)

  target-data-1 ::= value-expression
  target-data-2 ::= value-expression
```

(2) Explanation of specification format

target-data-1, *target-data-2*, ...:

Specifies the numeric data whose greatest value is to be determined.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- A maximum of 255 target data items can be specified.
- The data types that can be specified for the target data are numeric data, character string data, and datetime data.
- You must specify data types that can be compared in *target-data-1*, *target-data-2*, ... For details about data types that can be compared, see (1) Data types that can be compared in 6.2.2 Data types that can be converted, assigned, and compared. However, note the following exceptions:

  • `DATE` type data cannot be compared to character string data (even to the predefined input representation of a date).

  • `TIME` type data cannot be compared to character string data (even to the predefined input representation of a time).

  • `TIMESTAMP` type data cannot be compared to character string data (even to the predefined input representation of a time stamp).

  For details about predefined input representations, see 6.3.3 Predefined character-string representations.
- You cannot specify a dynamic parameter by itself for *target-data-1*.
- If a dynamic parameter is specified by itself for *target-data-2*, or later, the data type of *target-data-1* is assumed for the data type of the dynamic parameter.

(3) Rules

1. The data type and data length of the execution result are determined according to the rules described in 7.20.2 Data types of the results of value expressions.

2. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).

3. If any of the specified target data has a null value, the execution result will be a null value.

(4) Example

Example:

Determine the greatest value among the values of columns C1 to C4 in table T1.

```
SELECT GREATEST("C1","C2","C3","C4") FROM "T1"
```

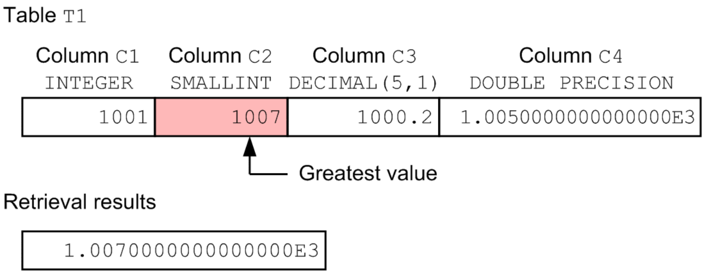


Table T1

Column C1	Column C2	Column C3	Column C4
INTEGER	SMALLINT	DECIMAL(5,1)	DOUBLE PRECISION
1001	1007	1000.2	1.0050000000000000E3

Greatest value

Retrieval results

1.0070000000000000E3

8.15.3 LEAST

Returns the smallest value among the specified target data items.

In addition to comparing numeric data items, you can also compare character string data items and datetime data items.

(1) Specification format

```
scalar-function-LEAST ::= LEAST(target-data-1[,target-data-2]...)

  target-data-1 ::= value-expression
  target-data-2 ::= value-expression
```

(2) Explanation of specification format

target-data-1, *target-data-2*, ...:

Specifies the numeric data whose smallest value is to be determined.

The following rules apply:

- Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.
- A maximum of 255 target data items can be specified.
- The data types that can be specified for the target data are numeric data, character string data, and datetime data.
- You must specify data types that can be compared in *target-data-1*, *target-data-2*, ... For details about data types that can be compared, see (1) Data types that can be compared in 6.2.2 Data types that can be converted, assigned, and compared. However, note the following exceptions:

  • DATE type data cannot be compared to character string data (even to the predefined input representation of a date).

  • TIME type data cannot be compared to character string data (even to the predefined input representation of a time).

- `TIMESTAMP` type data cannot be compared to character string data (even to the predefined input representation of a time stamp).

  For details about predefined input representations, see 6.3.3 Predefined character-string representations.
- You cannot specify a dynamic parameter by itself for *target-data-1*.
- If a dynamic parameter is specified by itself for *target-data-2*, or later, the data type of *target-data-1* is assumed for the data type of the dynamic parameter.

(3) Rules

1. The data type and data length of the execution result are determined according to the rules described in 7.20.2 Data types of the results of value expressions.
2. The `NOT NULL` constraint does not apply to the value of the execution result (the null value is allowed).
3. If any of the specified target data has a null value, the execution result will be a null value.

(4) Example

Example:

Determine the smallest value among the values of columns `C1` to `C4` in table `T1`.

```
SELECT LEAST("C1","C2","C3","C4") FROM "T1"
```

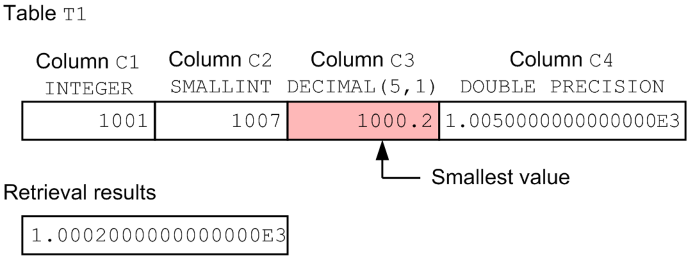


Table `T1`

Column `C1` `INTEGER`	Column `C2` `SMALLINT`	Column `C3` `DECIMAL(5,1)`	Column `C4` `DOUBLE PRECISION`
1001	1007	1000.2	1.0050000000000000E3

Smallest value

Retrieval results

1.0002000000000000E3

8.15.4 LTDECODE

Compares the values in the target data and in the comparison data one at a time, and, if any value in the target data is less than the value in the comparison data, returns the corresponding return value. If no value in the target data is less than any of the values in the comparison data, this function returns the predefined return value.

If multiple comparison data items are specified, the function returns the return value that corresponds to the first comparison data item whose value is greater than the value in the target data.

(1) Specification format

```
scalar-function-LTDECODE ::= LTDECODE(target-data,comparison-data,return-value
                         [,comparison-data,return-value]...
                         [,predefined-return-value])

  target-data ::= value-expression
  comparison-data ::= value-expression
  return-value ::= {value-expression | NULL}
  predefined-return-value ::= {value-expression | NULL}
```

(2) Explanation of specification format

target-data:

Specifies the target data. Specify the target data in the form of a value expression. For details about value expressions, see 7.20 Value expression.

comparison-data:

Specifies the comparison data. Specify the comparison data in the form of a value expression. For details about value expressions, see 7.20 Value expression.

return-value:

Specifies the value to be returned if the value of the target data is less than the value of the comparison data. Specify the return value in the form of a value expression, or as `NULL`. For details about value expressions, see 7.20 Value expression.

predefined-return-value:

Specifies the predefined value to be returned if the value of the target data is equal to or greater than any of the values of the comparison data. Specify the predefined return value in the form of a value expression, or as `NULL`. For details about value expressions, see 7.20 Value expression.

Note that if *predefined-return-value* is omitted, `NULL` is assumed.

(3) Rules

1. You must specify numeric data, character string data, or datetime data for *target-data*, *comparison-data*, *return-value*, and *predefined-return-value*.
2. When `NULL` is specified for *return-value* or *predefined-return-value*, it denotes the null value.
3. You must specify data types that can be compared for *target-data* and *comparison-data*. For details about data types that can be compared, see (1) Data types that can be compared in 6.2.2 Data types that can be converted, assigned, and compared.

   Note, however, that if *target-data* and *comparison-data* are character string data or datetime data, specify a combination of data types based on the following table.

Table 8-61: Combinations of data types that can be specified for the target data and the comparison data for the scalar function LTDECODE

Target data		**Comparison data**						
		Character string data				**Datetime data**		
		Character string literal that is the predefined input representation for date data	**Character string literal that is the predefined input representation for time data**	**Character string literal that is the predefined input representation for time stamp data**	**Other data**	**Date data**	**Time data**	**Time stamp data**
Character string data	Character string literal that is the predefined input representatio	Y	Y	Y	Y	N	N	N

Target data		Comparison data						
		Character string data				Datetime data		
		Character string literal that is the predefined input representation for date data	Character string literal that is the predefined input representation for time data	Character string literal that is the predefined input representation for time stamp data	Other data	Date data	Time data	Time stamp data
	n for date data							
	Character string literal that is the predefined input representation for time data	Y	Y	Y	Y	N	N	N
	Character string literal that is the predefined input representation for time stamp data	Y	Y	Y	Y	N	N	N
	Other data	Y	Y	Y	Y	N	N	N
Datetime data	Date data	Y	N	Y	N	Y	N	Y
	Time data	N	Y	N	N	N	Y	N
	Time stamp data	Y	N	Y	N	Y	N	Y

Legend:

Y: Can be specified.

N: Cannot be specified.

4. You must specify data types that can be compared (except when specifying NULL) for *return-value* and *predefined-return-value*. For details about data types that can be compared, see (1) Data types that can be compared in 6.2.2 Data types that can be converted, assigned, and compared.

However, if *return-value* and *predefined-return-value* are character string data or datetime data, specify a combination of data types based on the following table.

Table 8-62: Combinations of data types that can be specified for the return value and the predefined return value for the scalar function LTDECODE

Return value		**Predefined return value, or return value#**						
		Character string data				**Datetime data**		
		Character string literal that is the predefined input representation for date data	**Character string literal that is the predefined input representation for time data**	**Character string literal that is the predefined input representation for time stamp data**	**Other data**	**Date data**	**Time data**	**Time stamp data**
Character string data	Character string literal that is the predefined input representation for date data	Y	Y	Y	Y	N	N	N
	Character string literal that is the predefined input representation for time data	Y	Y	Y	Y	N	N	N
	Character string literal that is the predefined input representation for time stamp data	Y	Y	Y	Y	N	N	N
	Other data	Y	Y	Y	Y	N	N	N
Datetime data	Date data	N	N	N	N	Y	N	Y
	Time data	N	N	N	N	N	Y	N
	Time stamp data	N	N	N	N	Y	N	Y

Legend:

Y: Can be specified.

N: Cannot be specified.

#

If multiple *return-value* items are specified and all *return-value* items are character string data or datetime data, the combinations of data types that can be specified for each *return-value* item are as shown in Table 8-62: Combinations of data types that can be specified for the return value and the predefined return value for the scalar function LTDECODE.

5. You can specify a maximum of 256 *target-data*, *comparison-data*, *return-value*, and *predefined-return-value* items in total.
6. The data type and data length of the execution result depends on the data types of the results of *return-value* and *predefined-return-value*, and is determined according to the rules described in 7.20.2 Data types of the results of value expressions.

   Note that the following specification for *return-value* and *predefined-return-value* does not affect the data type and data length of the execution result.

   - The specification of only a single dynamic parameter
   - `NULL`
7. The NOT NULL constraint does not apply to the value of the execution result (the null value is allowed).
8. You must specify *comparison-data* and *return-value* as a set. The *return-value* that corresponds to *comparison-data* is the value to be specified following the *comparison-data*.
9. For *target-data* or *comparison-data*, you must specify at least one value expression other than the single dynamic parameter that is specified.
10. For *return-value* or *predefined-return-value*, you must specify at least one value expression other than the following:
    - The specification of only a single dynamic parameter
    - `NULL`
11. If you specify only a single dynamic parameter for *target-data*, the data type of the dynamic parameter of *target-data* is assumed to be the data type of the first *comparison-data* item. Note, however, that if you specify only a single dynamic parameter for the first *comparison-data* item, the data type of the second or subsequent *comparison-data* item (which is not an item for which only a single dynamic parameter is specified) is assumed.
12. If only a single dymanic parameter is specified for *comparison-data*, the data type of *target-data* is assumed as the data type of the dynamic parameter. Note, however, that if you specify only a single dynamic parameter for *target-data*, the data type of the first *comparison-data* item is assumed. In addition, if you specify only a single dynamic parameter for the first *comparison-data* item, the data type of the second or subsequent *comparison-data* item (which is not an item for which only a single dynamic parameter is specified) is assumed.

    The data types to be assumed are described based on the following specification examples:

    - *Specification example 1*

      ```
      LTDECODE(?, 10, 'A', 20, 'C')
      ```

      Because the first *comparison-data* item (`10`) is of the `INTEGER` type, the data type of the dynamic parameter of *target-data* is assumed to be `INTEGER`.
    - *Specification example 2*

      ```
      LTDECODE(CURRENT_DATE, ?, 'A', DATE'2017/01/01', 'C')
      ```

      Because *target-data* (`CURRENT_DATE`) is of the `DATE` type, the data type of the dynamic parameter of *comparison-data* is assumed to be `DATE`.
    - *Specification example 3*

      ```
      LTDECODE(?, ?, 'A', 'B', 'C')
      ```

      For the first *comparison-data* item, only a single dynamic parameter is specified. The data type of the second *comparison-data* item (`'B'`) is `CHAR(1)`. For this reason, the data type of the dynamic parameter of *target-data* and the data type of the dynamic parameter of the first *comparison-data* item are both assumed to be `CHAR(1)`.

13. If you specify only a single dynamic parameter for *return-value* or *predefined-return-value*, the data type of the dynamic parameter is assumed to be the data type of the scalar function's execution result.

    The data types to be assumed are described based on the following specification examples:

    - *Specification example 1*

      ```
      LTDECODE("C1", 10, -1, 20, 0, 30, ?)
      ```

      Because the data type of the execution result of `LTDECODE` is `INTEGER`, the data type of the dynamic parameter of *return-value* is assumed to be `INTEGER`.

    - *Specification example 2*

      ```
      LTDECODE("C1", DATE'2017/01/01', 'A', DATE'2017/02/01',
               ?, DATE'2017/02/01', 'BB')
      ```

      Because the data type of the execution result of `LTDECODE` is `VARCHAR(2)`, the data type of the dynamic parameter of *return-value* is assumed to be `VARCHAR(2)`.

    - *Specification example 3*

      ```
      LTDECODE("C1", 10, 'A', 20, ?, ?)
      ```

      Because the data type of the execution result of `LTDECODE` is `VARCHAR(1)`, the data types of the dynamic parameters of *return-value* and *predefined-return-value* are both assumed to be `VARCHAR(1)`.

14. The return value that corresponds to the comparison data item whose value is greater than the value in the target data (the following comparison predicate is true) is returned.

    ```
    Target data < Comparison data
    ```

15. If there are multiple comparison data items whose values are greater than the value in the target data, the return value that corresponds to the first comparison data item is returned.

(4) Examples

Example 1:

Convert the values in column `C1` of table `T1` as follows. Then, store the converted values in column `C2`.

- Value less than 0 → Null value
- Value greater than or equal to 1 → 2

```
SELECT "C1", LTDECODE("C1", 0, NULL, 1, "C1", 2) "C2"
    FROM "T1"
```

Table T1

Column C1
-1.6
-0.5
0.2
0.5
0.6
1.0
1.2

Search result

Column C1	Column C2
-1.6	NULL
-0.5	NULL
0.2	0.2
0.5	0.5
0.6	0.6
1.0	2.0
1.2	2.0

Example 2:

Convert the values in the height column (HEIGHT) of the employee table (EMPLIST) as follows. Then, store the converted values in column HEIGHT2.

- Value less than 150 → 150
- Value greater than or equal to 190 → 190

```
SELECT "USERID", LTDECODE("HEIGHT", 150, 150, 190, "HEIGHT", 190) "HEIGHT2"
    FROM "EMPLIST"
```

Employee table (EMPLIST)

USERID	HEIGHT
U00001	148
U00002	158
U00003	166
U00004	171
U00005	178
U00006	182
U00007	195

Search result

USERID	HEIGHT2
U00001	150
U00002	158
U00003	166
U00004	171
U00005	178
U00006	182
U00007	190

Example 3:

Search the employee table (EMPLIST), and determine the following values:

- Based on the ages (AGE) of the employees, determine the number of employees in each age group.

```
SELECT "GEN", COUNT("GEN") "GEN-NUM"
   FROM "EMPLIST"
     GROUP BY LTDECODE("AGE", 20, 'Under  20'
                            , 30, '20s'
                            , 40, '30s', '40 and older')
   "GEN"
```

Employee table (EMPLIST)

USERID	AGE
U00001	18
U00002	22
U00003	26
U00004	30
U00005	35
U00006	38
U00007	45

Search result

GEN	GEN-NUM
Under 20	1
20s	2
30s	3
40 and older	1

Appendixes

A. SQL Reverse Lookup Reference

The following table lists the relevant SQL syntax organized according to purpose.

Table A-1: Relevant SQL syntax organized by purpose

No.	Category	Purpose	Relevant SQL syntax
1	Data retrieval	Retrieve data by specifying a range.	BETWEEN predicate
2		Retrieve data that matches any of multiple values.	IN predicate
3		Retrieve data that contains a specific character string.	LIKE predicate
4		Retrieve data by using a regular expression	LIKE_REGEX predicate
5		Retrieve null-valued data.	NULL predicate
6		Eliminate duplication in the retrieval results.	SELECT DISTINCT
7		Sort retrieval results in ascending or descending order.	ORDER BY clause
8		Specify the maximum number of rows in the retrieval results.	LIMIT clause
9		Re-use the same derived table within a SELECT statement.	WITH clause
10		Change a column name in the retrieval results.	AS clause
11		Retrieve by specifying multiple branch conditions.	CASE expression
12		Retrieve by joining multiple tables.	Joined tables
13		Perform a subquery.	Subquery
14			EXISTS predicate
15			IN predicate
16			Comparison predicate
17			Quantified predicate
18		Create the union of query results from multiple tables.	UNION ALL UNION DISTINCT
19	Data deletion	Delete all the rows in a base table.	TRUNCATE TABLE statement
20		Delete all the rows in a chunk in a base table.	PURGE CHUNK statement
21	Data aggregation	Determine the sum of retrieved values.	General set function SUM
22		Determine the maximum value.	General set function MAX
23		Determine the minimum value.	General set function MIN
24		Determine the average of retrieved values.	General set function AVG
25		Determine the row count (number of results)	General set function COUNT
26			Set function COUNT(*)
27		Determine the standard deviation of a population.	General set function STDDEV_POP
28		Determine the standard deviation of a sample.	General set function STDDEV_SAMP
29		Determine the variance of a population.	General set function VAR_POP
30		Determine the variance of a sample.	General set function VAR_SAMP

No.	Category	Purpose	Relevant SQL syntax
31		Determine the median of an ordered set of values.	Inverse distribution function MEDIAN
32		Determine the percentile of an ordered set of values.	Inverse distribution function PERCENTILE_CONT
33			Inverse distribution function PERCENTILE_DISC
34		Set a range in which to aggregate data.	Window functions
35		Aggregate the data into groups.	GROUP BY clause
36			HAVING clause
37	Character strings	Check whether the target data contains character strings that meet the search condition expression.	Scalar function CONTAINS
38		Concatenate two character string data items.	Scalar function CONCAT
39			Concatenation operations
40		Remove specific characters from character string data.	Scalar function TRIM
41			Scalar function LTRIM
42			Scalar function RTRIM
43		Extract a substring from character string data.	Scalar function SUBSTR
44			Scalar function LEFT
45			Scalar function RIGHT
46		Pad the beginning or end of character string data with any specified character string.	Scalar function LPAD
47			Scalar function RPAD
48		Replace any character string in the target data.	Scalar function REPLACE
49		Replace any character in character string data.	Scalar function TRANSLATE
50		Determine the number of characters in character string data.	Scalar function LENGTH
51		Search the target data for a character string and return the starting position of the string.	Scalar function INSTR
52		Convert uppercase letters to lowercase.	Scalar function LOWER
53		Convert lowercase letters to uppercase.	Scalar function UPPER
54	Binary data	Concatenate two binary data items.	Scalar function CONCAT
55		Extract a substring from binary data.	Scalar function SUBSTRB
56		Determine the value resulting from shifting the bits of a binary data value to the left.	Scalar function BITLSHIFT
57		Determine the value resulting from shifting the bits of a binary data value to the right.	Scalar function BITRSHIFT
58		Determine the bitwise logical AND of two binary data items.	Scalar function BITAND
59		Determine the bitwise inclusive OR of two binary data items.	Scalar function BITOR
60		Determine the bitwise logical NOT of a binary data item.	Scalar function BITNOT
61		Determine the bitwise exclusive OR of two binary data items.	Scalar function BITXOR

No.	Category	Purpose	Relevant SQL syntax
62		Convert binary data to a binary string representation (character string data consisting of 0 and 1).	Scalar function BIN
63		Convert binary data to a hex string representation (character string data consisting of 0 to 9, and A to F).	Scalar function HEX
64	Numerical calculations	Determine the remainder after a division.	Scalar function MOD
65		Determine the absolute value.	Scalar function ABS
66		Determine the square root.	Scalar function SQRT
67		Determine the sign of the data (positive, negative, or 0).	Scalar function SIGN
68		Determine pseudorandom numbers that follow a uniform distribution and are greater than or equal to the value specified for the minimum value and less than the value specified for the maximum value.#	Scalar function RANDOM
69			Scalar function RANDOMCURSOR
70			Scalar function RANDOMROW
71		Determine pseudorandom numbers that follow a normal distribution with an average μ, and a standard deviation σ.	Scalar function RANDOM_NORMAL
72	Rounding	Round off a numeric value.	Scalar function ROUND
73		Truncate a numeric value.	Scalar function TRUNC
74		Determine the greatest integer that is equal to or less than the specified numeric value.	Scalar function FLOOR
75		Determine the smallest integer that is equal to or greater than the specified numeric value.	Scalar function CEIL
76	Exponent and logarithm	Determine a power of the specified data.	Scalar function POWER
77		Determine the logarithm of the specified antilogarithm and base.	Scalar function LOG
78		Determine the natural logarithm.	Scalar function LN
79		Determine a power of the base of the natural logarithm.	Scalar function EXP
80	Trigonometric functions	Determine the sine (SIN trigonometric function).	Scalar function SIN
81		Determine the cosine (COS trigonometric function).	Scalar function COS
82		Determine the tangent (TAN trigonometric function).	Scalar function TAN
83		Determine the inverse sine (inverse trigonometric function).	Scalar function ASIN
84		Determine the inverse cosine (inverse trigonometric function).	Scalar function ACOS
85		Determine the inverse tangent (inverse trigonometric function).	Scalar function ATAN
86			Scalar function ATAN2
87		Determine the hyperbolic sine.	Scalar function SINH
88		Determine the hyperbolic cosine.	Scalar function COSH
89		Determine the hyperbolic tangent.	Scalar function TANH
90		Convert an angle from radians to degrees.	Scalar function DEGREES
91		Convert an angle from degrees to radians.	Scalar function RADIANS

No.	Category	Purpose	Relevant SQL syntax
92		Determine the value of π .	Scalar function PI
93	Date and time	Extract a portion of a date or time (for example, extract only the month).	Scalar function EXTRACT
94		Given a date, determine the ordinal number of the date in the year.	Scalar function DAYOFYEAR
95		Given a date, determine what day of week it falls on as an ordinal number from the first day in the week.	Scalar function DAYOFWEEK
96		Determine the date of the last day of the specified month.	Scalar function LASTDAY
97		Determine the difference between the start date and time and the end date and time.	Scalar function DATEDIFF
98		Determine a person's age on a reference date given their birth date.	Scalar function GETAGE
99		Round a date by the year, month, day, hour, or second.	Scalar function ROUND
100		Truncate a date by the year, month, day, hour, or second.	Scalar function TRUNC
101		Determine the current date.	Datetime information acquisition function CURRENT_DATE
102		Determine the current time.	Datetime information acquisition function CURRENT_TIME
103		Determine the current data and time stamp.	Datetime information acquisition function CURRENT_TIMESTAMP
104		Perform operations on datetime data.	Datetime operations
105			Labeled duration
106	Null value	Determine the first non-null value among the specified data.	Scalar function COALESCE
107			Scalar function ISNULL
108			Scalar function NVL
109	Data comparison	Determine whether two data items are equal.	Scalar function NULLIF
110		Compare the values in the target data and the comparison data one at a time, and if there is a match, return the corresponding return value.	Scalar function DECODE
111		Compare the values in the target data and in the comparison data one at a time, and, if any value in the target data is less than the value in the comparison data, return the corresponding return value.	Scalar function LTDECODE
112		Determine the greatest value.	Scalar function GREATEST
113		Determine the smallest value.	Scalar function LEAST
114	Data types	Convert the data type.	Scalar function CAST
115			Scalar function CONVERT
116	Data information acquisition	Determine the number of bytes in the target data.	Scalar function LENGTHB
117		Determine the character code of the first character of character string data.	Scalar function ASCII

No.	Category	Purpose	Relevant SQL syntax
118		Determine the character corresponding to numeric value character code in the target data.	Scalar function CHR
119	User information	Determine the authorization identifier of the currently executing HADB user.	User information acquisition function CURRENT_USER

#

There are differences in specifications among the scalar functions RANDOM, RANDOMCURSOR, and RANDOMROW. For details about the differences in specifications, see (6) List of scalar functions that return pseudorandom numbers in 8.4.5 RANDOM.

B. List of Functions

The following table provides a list of functions.

Table B-1: List of functions

No.	Function			Use
1	Set functions	MAX		Determine the maximum value.
2		MIN		Determine the minimum value.
3		SUM		Determine the sum of the retrieved values.
4		AVG		Determine the average of the retrieved values.
5		COUNT		Determine the row count (number of results).
6		COUNT(*)		Determine the row count (number of results).
7		STDDEV_POP		Determine the standard deviation of a population.
8		STDDEV_SAMP		Determine the standard deviation of a sample.
9		VAR_POP		Determine the variance of a population.
10		VAR_SAMP		Determine the variance of a sample.
11		MEDIAN		Determine the median of an ordered set of values.
12		PERCENTILE_CONT		Determine the percentile of an ordered set of values.
13		PERCENTILE_DISC		
14	Mathematical functions	Trigonometric functions	SIN	Return the sine (SIN trigonometric function) of the target data, which must be specified in radians.
15			COS	Return the cosine (COS trigonometric function) of the target data, which must be specified in radians.
16			TAN	Return the tangent (TAN trigonometric function) of the target data, which must be specified in radians.
17			ASIN	Return the angle (in radians) that is the inverse sine of the target data, in the range - π /2 to π /2.
18			ACOS	Return the angle (in radians) that is the inverse cosine of the target data, in the range 0 to π .
19			ATAN	Return the angle (in radians) that is the inverse tangent of the target data, in the range - π /2 to π /2.
20			ATAN2	Return the angle (in radians) that is the inverse tangent of *y*/*x*, in the range - π to π .
21			SINH	Return the hyperbolic sine of the target data.
22			COSH	Return the hyperbolic cosine of the target data.
23			TANH	Return the hyperbolic tangent of the target data.
24			DEGREES	Return the result of converting an angle from radians to degrees.
25			RADIANS	Return the result of converting an angle from degrees to radians.
26			PI	Return the value of π .

No.	Function			Use
27		Exponent and logarithm	POWER	Return the result of raising the target data to a specified power.
28			LOG	Return the logarithm of the target data (antilogarithm) to the specified base.
29			LN	Return the natural logarithm of the target data.
30			EXP	Return the result of raising the base of the natural logarithm to a power.
31		Numerical calculation	MOD	Return the remainder after dividing the dividend by the divisor.
32			ABS	Return the absolute value of the target data.
33			SQRT	Return the square root of the target data.
34			SIGN	Return the sign of the target data (+1 for positive, -1 for negative, 0 for 0).
35			RANDOM	Return pseudorandom numbers that follow a uniform distribution and are greater than or equal to the minimum specified value and less than the maximum specified value.
36			RANDOMCURSOR	Return pseudorandom numbers that follow a uniform distribution and are greater than or equal to the value specified for the minimum value and less than the value specified for the maximum value. If an SQL statement contains multiple RANDOMCURSOR functions for which the same identification number is specified, those functions always return the same values.
37			RANDOMROW	Return pseudorandom numbers that follow a uniform distribution and are greater than or equal to the value specified for the minimum value and less than the value specified for the maximum value. If a query specification contains multiple RANDOMROW functions for which the same identification number is specified, those functions return the same values for each result row of the query specification.
38			RANDOM_NORMAL	Return pseudorandom numbers that follow a normal distribution with an average μ and a standard deviation σ.
39		Rounding	ROUND	Return the value of the target data rounded to the *n*th digit after the decimal point.
40			TRUNC	Return a value that has been truncated to the specified number of decimal places.
41			FLOOR	Return the greatest integer that is equal to or less than the value of the target data.
42			CEIL	Return the smallest integer that is equal to or greater than the target data.
43	Character string functions	Character string retrieval	CONTAINS	Return whether character strings that meet the search condition expression are included in the target data.
44		Concatenating character string data	CONCAT	Concatenate two character string data items.
45		Extracting a substring from character string data	SUBSTR	Extract a substring from a character string starting from any position in the character string data.

No.	Function			Use
46			LEFT	Extract a substring from a character string starting from the beginning (leftmost position) of the character string data.
47			RIGHT	Extract a substring from a character string starting from the end (rightmost position) of the character string data.
48		Removing characters from character string data	TRIM	Remove instances of the specified characters from the target character string. The characters can be removed in any of the following ways: • Remove the specified characters starting from the beginning of the character string. • Remove the specified characters starting from the end of the character string. • Remove characters starting from both the beginning and the end of the character string.
49			LTRIM	Remove instances of the specified characters, starting from the beginning of the target character string.
50			RTRIM	Remove instances of the specified characters, starting from the end of the target character string.
51		Padding character strings	LPAD	Pad the beginning of the target data (from the left) with the padding character string up to the specified number of characters.
52			RPAD	Pad the end of the target data (from the right) with the padding character string up to the specified number of characters.
53		Replacement of character strings in character string data	REPLACE	Replace any character string in the target data. All instances of the character string to be replaced in the target data are replaced with the replacement character string.
54		Replacement of characters in character string data	TRANSLATE	Replace any character in the target data.
55		Number of characters in character string data	LENGTH	Return the number of characters in the target character string.
56		Starting position of a character string in character string data	INSTR	Search the target data for a character string and return the starting position of the string.
57		Conversion between uppercase and lowercase letters	LOWER	Convert uppercase letters (A to Z) to lowercase letters (a to z) in character string data.
58			UPPER	Convert lowercase letters (a to z) to uppercase letters (A to Z) in character string data.
59	Datetime functions	DATEDIFF		Return the difference between the start date and time and the end date and time.
60		DAYOFWEEK		Return the day of the week that the specified date falls on.
61		DAYOFYEAR		Return the specified date as the number of days elapsed since January 1 of that year.
62		EXTRACT		Extract a part (year, month, day, hour, minute, or second) from data representing the date and time.
63		GETAGE		Determine a person's age on a reference date given their birth date.

No.	Function			Use
64		LASTDAY		Return the date or datetime of the last day of the month specified in the datetime data.
65		ROUND		Return the datetime data rounded to the unit specified in the datetime format.
66		TRUNC		Return the datetime data truncated to the unit specified in the datetime format.
67	Binary column functions	Concatenating binary data	CONCAT	Concatenate two binary data items.
68		Extracting a substring from binary data	SUBSTRB	Extract a substring from binary data starting from any position in the binary data.
69		Bit operations on binary data	BITAND	Return the bitwise logical AND of two binary data items.
70			BITOR	Return the bitwise inclusive OR of two binary data items.
71			BITNOT	Return the bitwise logical NOT of a binary data item.
72			BITXOR	Return the bitwise exclusive OR of two binary data items.
73			BITLSHIFT	Return the value resulting from shifting the bits of a binary data value to the left.
74			BITRSHIFT	Return the value resulting from shifting the bits of a binary data value to the right.
75	Data conversion functions	CAST		Convert the data type of the data.
76		CONVERT		Convert the data type of the data. In addition, by specifying a datetime format or number format, you can control the conversion as follows. By specifying a datetime format: • When converting datetime data to character string data, you can specify the output format of the character string data after conversion. • When converting character string data to datetime data, you can specify the input format of the character string data before conversion. By specifying a number format: • When converting numeric data to character string data, you can specify the output format of the character string data after conversion. • When converting character string data to numeric data, you can specify the input format of the character string data before conversion.
77		ASCII		Return the character code of the first character of the target data as an integer value.
78		CHR		Return the character corresponding to a character code represented by the integer target data.
79		BIN		Convert binary data to a binary string representation (character string data consisting of 0 and 1).
80		HEX		Convert binary data to a hexadecimal string representation (character string data consisting of 0 to 9, and A to F).
81	NULL evaluation functions	COALESCE		Evaluate the specified target data items in the order in which they are specified, and then return the first non-null value.
82		ISNULL		

No.	Function		Use
83		NULLIF	Compare target data 1 to target data 2 and return NULL if they are equal, or target data 1 if they are not equal.
84		NVL	Evaluate the specified target data items in the order in which they are specified, and then return the first non-null value.
85	Information acquisition functions	LENGTHB	Return the length of the target data in bytes.
86	Comparison functions	DECODE	Compare the values in the target data and the comparison data one at a time, and if there is a match, return the corresponding return value. If no match is found between the target data and comparison data, this function returns the predefined return value.
87		LTDECODE	Compare the values in the target data and in the comparison data one at a time, and, if any value in the target data is less than the value in the comparison data, return the corresponding return value. If no value in the target data is less than any of the values in the comparison data, this function returns the predefined return value.
88		GREATEST	Return the greatest value among the specified target data.
89		LEAST	Return the smallest value among specified target data.
90	Datetime information acquisition functions	CURRENT_DATE	Return the current date.
91		CURRENT_TIME	Return the current time.
92		CURRENT_TIMESTAMP	Return the current time stamp (date and time).
93	User information acquisition function	CURRENT_USER	Return the authorization identifier of the currently executing HADB user.

Index

Hitachi, Ltd.
6-6, Marunouchi 1-chome, Chiyoda-ku, Tokyo, 100-8280 Japan