

**uCosminexus Stream Data Platform -
Application Framework
Application Development Guide**

3020-3-V03(E)

■ Relevant program products

P-2464-9B17 uCosminexus Stream Data Platform - Application Framework 01-00 (for Windows Server 2008)

■ Trademarks

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

Java is a registered trademark of Oracle and/or its affiliates.

JDBC is either a registered trademark or a trademark of Oracle and/or its affiliates.

JDK is either a registered trademark or a trademark of Oracle and/or its affiliates.

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

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

Sun is either a registered trademark or a trademark of Oracle and/or its affiliates.

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

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

Portions of this software were developed at the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign.

Regular expression support is provided by the PCRE library package, which is open source software, written by Philip Hazel, and copyright by the University of Cambridge, England. The original software is available from <ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/>

This product includes software developed by Andy Clark.

This product includes software developed by Ben Laurie for use in the Apache-SSL HTTP server project.

This product includes software developed by Daisuke Okajima and Kohsuke Kawaguchi (<http://relaxngcc.sf.net/>).

This product includes software developed by IAIK of Graz University of Technology.

This product includes software developed by Ralf S. Engelschall <rse@engelschall.com> for use in the mod_ssl project (<http://www.modssl.org/>).

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>).

This product includes software developed by the Java Apache Project for use in the Apache JServ servlet engine project (<http://java.apache.org/>).

This product includes software developed by the University of California, Berkeley and its contributors.

This software contains code derived from the RSA Data Security Inc. MD5 Message-Digest Algorithm, including various modifications by Spyglass Inc., Carnegie Mellon University, and Bell Communications Research, Inc (Bellcore).

uCosminexus Stream Data Platform - Application Framework contains RSA(R) BSAFETM software from EMC Corporation.

Other product and company names mentioned in this document may be the trademarks of their respective owners. Throughout this document Hitachi has attempted to distinguish trademarks from descriptive terms by writing the name with the capitalization used by the manufacturer, or by writing the name with initial capital letters. Hitachi cannot attest to the accuracy of this information. Use of a trademark in this document should not be regarded as affecting the validity of the trademark.

■ Microsoft product name abbreviations

This manual uses the following abbreviations for Microsoft product names.

Full name or meaning	Abbreviation
Microsoft(R) Windows Server(R) 2008 Enterprise	Windows Server 2008 or Windows
Microsoft(R) Windows Server(R) 2008 Enterprise 32-bit	
Microsoft(R) Windows Server(R) 2008 R2 Enterprise	
Microsoft(R) Windows Server(R) 2008 R2 Standard	
Microsoft(R) Windows Server(R) 2008 Standard	

Full name or meaning	Abbreviation
Microsoft(R) Windows Server(R) 2008 Standard 32-bit	

■ 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.

Printed in Japan.

■ Edition history

Aug. 2011: 3020-3-V03(E)

■ Copyright

All Rights Reserved. Copyright (C) 2011, Hitachi, Ltd.

Preface

This manual explains how to code CQL for use in analyzing data with uCosminexus Stream Data Platform - Application Framework, and how to create custom adaptors.

It is intended to give you the ability to write CQL code for achieving analysis objectives and the ability to use APIs to create custom adaptors.

Intended readers

This manual is intended for programmers who use CQL to define queries, and who create application programs for use as adaptors.

Readers of this manual must have:

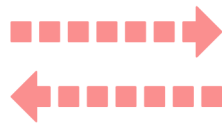
- A basic knowledge of operating systems
- A good grasp of Java programming

This manual also assumes that the reader is familiar with the manual *uCosminexus Stream Data Platform - Application Framework Description*, so we recommend that you first read this manual.

Conventions: Diagrams

This manual uses the following conventions in diagrams:

- Flow of stream data



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: <ul style="list-style-type: none">• From the File menu, choose Open.• Click the Cancel button.• In the Enter name entry box, type your name.

Font	Convention
<i>Italics</i>	<p><i>Italics</i> are used to indicate a placeholder for some actual text to be provided by the user or system. For example:</p> <ul style="list-style-type: none"> Write the command as follows: <code>copy <i>source-file</i> <i>target-file</i></code> The following message appears: A file was not found. (file = <i>file-name</i>) <p><i>Italics</i> are also used for emphasis. For example:</p> <ul style="list-style-type: none"> Do <i>not</i> delete the configuration file.
Code font	<p>A code font indicates text that the user enters without change, or text (such as messages) output by the system. For example:</p> <ul style="list-style-type: none"> At the prompt, enter <code>dir</code>. Use the <code>send</code> command to send mail. The following message is displayed: The password is incorrect.

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*.

Contents

Preface	i
Intended readers	i
Conventions: Diagrams	i
Conventions: Fonts and symbols.....	i
Conventions: Version numbers.....	ii

PART 1: Before Reading This Manual

1. Overview of Application Development in Stream Data Platform - AF	1
1.1 Flow from introduction to operation	2
1.2 Organization of this manual	5
1.3 Overview of application development.....	7
1.3.1 What you can define in a query	7
1.3.2 What you can do when creating a custom adaptor	8

PART 2: CQL Programming

2. Using CQL to Define Queries	9
2.1 CQL structure	10
2.2 Generating input relations using window operations	12
2.2.1 Types of window operations.....	12
2.2.2 Window operation examples	14
2.3 Data extraction using a relational operation.....	19
2.3.1 Relational operation types.....	19
2.3.2 Example of a linking process	20
2.3.3 Example of processing using an aggregate function.....	22
2.3.4 Notes on using the linking process and the ROWS window together	24
2.4 Conversion into output stream data using stream operation	26
2.4.1 Stream operation types	27
2.4.2 Stream operation examples	28
2.5 Limiting memory usage by specifying time division	32
2.6 Definition examples.....	35
2.6.1 Basic query definition example.....	35
2.6.2 Definition examples in which time division is specified	37
3. CQL Basic Items and Data Types	41
3.1 Defining queries using CQL.....	42

3.1.1	CQL format.....	42
3.1.2	Characters that can be used in CQL	44
3.1.3	Symbols used in the explanation of CQL syntax	44
3.2	Specifying basic items in CQL	46
3.2.1	Specifying keywords	46
3.2.2	Specifying numeric values.....	47
3.2.3	Delimiters	47
3.2.4	Specifying names.....	49
3.2.5	Name qualification	49
3.2.6	Specifying constants	52
3.3	CQL data types	58
3.3.1	Mapping between CQL data types and Java data types	58
3.3.2	Notes on the DECIMAL and NUMERIC types	61
3.4	Data comparison	63
3.4.1	Combinations of data types that can be compared	63
3.4.2	Notes on comparing data	63
3.5	Notes on query definitions and limit values	66
3.5.1	Notes on query definitions.....	66
3.5.2	Limit values applicable to query definitions	66
4.	CQL Reference	69
4.1	Format used for explaining the CQL syntax.....	70
4.2	CQL list.....	71
4.3	Definition CQL	73
4.3.1	REGISTER STREAM clause (stream definition)	73
4.3.2	REGISTER QUERY clause (query definition)	74
4.3.3	REGISTER QUERY_ATTRIBUTE clause (time division specification)....	75
4.4	Data manipulation CQL	80
4.4.1	Inquiry	80
4.4.2	Stream clause.....	80
4.4.3	Relation expression.....	81
4.4.4	SELECT clause.....	82
4.4.5	FROM clause.....	83
4.4.6	WHERE clause	84
4.4.7	GROUP BY clause	85
4.4.8	HAVING clause	85
4.4.9	UNION clause	86
4.4.10	Selection list	88
4.4.11	Select expression	88
4.4.12	Aggregate functions.....	90
4.4.13	Column specification list.....	93
4.4.14	Relation reference.....	93
4.4.15	Window specification	95
4.4.16	Time specification	97

4.4.17 Search condition	98
4.4.18 Comparison predicate	100
4.4.19 Value expression	103
4.4.20 Constant	106
5. Query Definition Samples	109
5.1 Types of query definition samples	110

PART 3: Creating Custom Adaptors

6. Creating Custom Adaptors	111
6.1 Types of custom adaptors that can be created	112
6.1.1 Data transmission applications and data reception applications	112
6.1.2 Classification based on how data is sent and received (RMI connection custom adaptor and in-process connection custom adaptor)	113
6.2 Creating an RMI connection custom adaptor	116
6.2.1 Sending stream data (RMI connection custom adaptor)	116
6.2.2 Receiving query result data (RMI connection custom adaptor)	118
6.3 Creating an in-process connection custom adaptor	120
6.3.1 Sending stream data (in-process connection custom adaptor)	121
6.3.2 Receiving query result data (in-process connection custom adaptor)	122
6.3.3 Notes	125
6.4 Other processes that may need to be implemented in custom adaptors	126
6.4.1 Detecting the trigger for terminating a data reception application (data processing termination notification)	126
6.4.2 Specifying time information in the data source mode	131
6.4.3 Preventing queue overflow	134
6.5 Compilation procedure	145
6.6 Notes on creating custom adaptors	146
7. APIs for Sending and Receiving Data	147
7.1 Syntax of APIs for sending and receiving data	148
7.2 List of APIs for sending and receiving data	149
7.3 SDPConnector interface (common API)	151
7.4 SDPConnectorFactory class (for RMI connection)	156
7.5 StreamEventListener interface (for in-process connection)	158
7.6 StreamInprocessUP interface (for in-process connection)	160
7.7 StreamInput interface (common API)	163
7.8 StreamOutput interface (common API)	170
7.9 StreamTime class (common API)	183
7.10 StreamTuple class (common API)	187
7.11 Exception class (common API)	192

8. Sample Programs Using APIs for Sending and Receiving Data	197
8.1 Sample program configuration	198
8.2 Sample program for an RMI connection custom adaptor.....	201
8.2.1 Procedure for executing the sample program for an RMI connection custom adaptor	201
8.2.2 Query group definition content (RMI connection custom adaptor)	204
8.2.3 Content of the RMI connection data transmission application.....	204
8.2.4 Content of the RMI connection data reception application	207
8.3 Sample program for an in-process connection custom adaptor.....	211
8.3.1 Procedure for executing the sample program for an in-process connection custom adaptor	211
8.3.2 Query group definition content (in-process connection custom adaptor) ..	213
8.3.3 Content of the in-process connection transmission/reception control application	213
8.3.4 Content of the in-process connection data transmission application.....	215
8.3.5 Content of the in-process connection data reception application (polling method).....	219
Appendix	223
A. Reference Material for This Manual	224
A.1 Related publications.....	224
A.2 Conventions: Abbreviations for product names	224
A.3 Conventions: Acronyms	225
A.4 Conventions: KB, MB, GB, and TB.....	225
Index	227

Chapter

1. Overview of Application Development in Stream Data Platform - AF

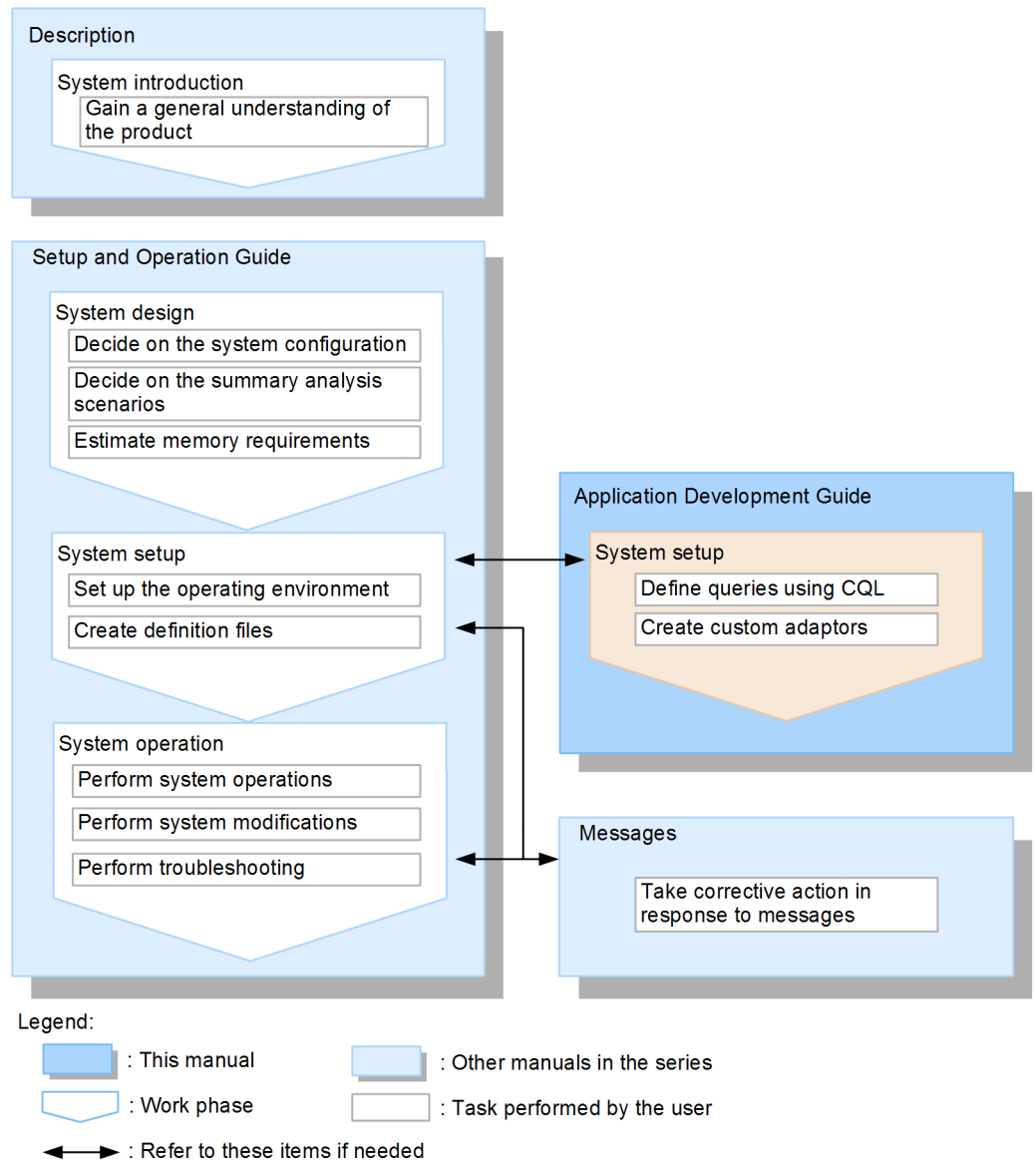
This chapter explains the process flow from the introduction of Stream Data Platform - AF to its operation, as well as the structure of this manual, and provides an overview of application development in Stream Data Platform - AF. We recommend that you familiarize yourself with these items before you read the rest of this manual.

- 1.1 Flow from introduction to operation
- 1.2 Organization of this manual
- 1.3 Overview of application development

1.1 Flow from introduction to operation

Figure 1-1 shows the process flow from the introduction of Stream Data Platform - AF to its operation and the relationship of each phase to the related manuals. Note that the manual titles shown omit *uCosminexus Stream Data Platform - Application Framework* to make it easier to read.

Figure 1-1: Processing flow from introduction to operation and relationship to related manuals



This manual explains the use of CQL for defining queries and for creating custom adaptors, which are part of the system setup phase.

Before reading this manual, read the manual *uCosminexus Stream Data Platform - Application Framework Description* to gain an overview of the Stream Data Platform

1. Overview of Application Development in Stream Data Platform - AF

- AF product.

If necessary, also refer to the related manuals for the various work phases.

1.2 Organization of this manual

This part explains the organization of this manual. This manual is organized into the following three parts and an appendix:

- **Part 1: Before Reading this Manual**

This is the part you are currently reading. It explains the basic information that we recommend you become familiar with before you read the rest of this manual (it includes the process flow from the introduction of Stream Data Platform - AF to its operation, the organization of this manual, and an overview of application development in Stream Data Platform - AF).

- **Part 2: CQL Programming**

Part 2 explains how to use CQL to define queries, the basic CQL items used in query definitions, and the CQL syntax.

- **Part 3: Creating Custom Adaptors**

Part 3 explains how to implement a custom adaptor, how to compile it, and the syntax of the APIs provided by Stream Data Platform - AF.

- **Appendix**

The appendix explains other information that you can reference while reading this manual.

The following table describes the content of the chapters included in each section, and the appendix.

Table 1-1: Content of each chapter and the appendix

Part	Chapter or appendix	Description
Part 1 Before Reading This Manual	<i>1. Overview of Application Development in Stream Data Platform - AF</i>	This is the part you are currently reading. It explains the basic information that we recommend you become familiar with before you read the rest of this manual.
Part 2 CQL Programming	<i>2. Using CQL to Define Queries</i>	Explains how to use CQL for defining queries.
	<i>3. CQL Basic Items and Data Types</i>	Explains the basic items and data types used in CQL.
	<i>4. CQL Reference</i>	Explains the CQL syntax.
	<i>5. Query Definition Samples</i>	Explains query definition samples.

Part	Chapter or appendix	Description
Part 3 Creating Custom Adaptors	<i>6. Creating Custom Adaptors</i>	Explains how to create custom adaptors.
	<i>7. APIs for Sending and Receiving Data</i>	Explains the syntax of the APIs for sending and receiving data used when creating custom adaptors.
	<i>8. Sample Programs Using APIs for Sending and Receiving Data</i>	Explains sample programs that use the APIs for sending and receiving data.
Appendix	<i>A. Reference Material for This Manual</i>	Explains other information that you can reference while reading this manual.

1.3 Overview of application development

This section provides an overview of application development in Stream Data Platform - AF.

The following tasks are performed in application development:

- Defining an analysis scenario (defining a query)
- Creating an application to exchange data with the stream data processing engine (creating a custom adaptor)

You must define an analysis scenario that matches the data you are analyzing and the analysis you want to perform.

A custom adaptor is an application that is only needed in a system that does not use a standard adaptor. If one of the standard adaptors will not work, you need to create a custom adaptor.

1.3.1 What you can define in a query

Stream Data Platform - AF uses predefined scenarios to analyze the stream data that has been input into the stream data processing engine and then outputs the results of analysis that you have set up to meet your objectives.

A scenario is defined as a query in a query definition file. The content to be defined in a query is described below.

- Data to be targeted for analysis

Stream data is time-series data that continues without interruption. To analyze stream data, you must specify the range of data to be included.

Using the concept of windows, Stream Data Platform - AF specifies the range of data targeted for analysis by treating stream data as finite data separated based on time stamps or number of tuples.

- Details to be analyzed

The intended analysis is carried out on stream data separated into windows. For example, you can track the change in values for a specific series of data, or combine multiple stream data values into new stream data and analyze it.

You define the details to be analyzed using relational operations such as selection and linkage, operations that use aggregate functions, and the like.

- Formatting the analysis results for output

The results of analysis are output as new stream data.

For example, you can output analyzed stream data when there is an increase or

decrease in the analysis results, or output all analysis results at specified time intervals.

A function called stream operation is used to output analyzed data.

You define queries using a query language called CQL, which is similar to SQL.

Details about how to define queries using CQL are explained in 2. *Using CQL to Define Queries*. Details about how to code CQL are also explained in 3. *CQL Basic Items and Data Types* and 4. *CQL Reference*. For details about the query definition file, see the *uCosminexus Stream Data Platform - Application Framework Setup and Operation Guide*.

You can also define a new query definition file based on a sample file. Details about the sample files are explained in 5. *Query Definition Samples*.

1.3.2 What you can do when creating a custom adaptor

Stream Data Platform - AF provides standard adaptors for inputting and outputting stream data. If you use these, you do not need to develop an application that uses the provided APIs.

However, the standard adaptors have the following limitations:

Limitations of the standard adaptors

- The only stream data formats that can be handled are files and HTTP packets.
- There is a limit on the number of scenarios (query groups) you can define.
- You cannot arbitrarily determine the process configuration of the adaptors (application) used for sending data to, or receiving data from, the stream data processing engine (for example, if a data-sending application is to be started in-process, the data-receiving application must also be started in-process).

For details about the standard adaptors, see the *uCosminexus Stream Data Platform - Application Framework Setup and Operation Guide*.

If you cannot use a standard adaptor, such as when you need to analyze stream data that is in an unusual format, you must create a custom adaptor.

A custom adaptor is created as a Java application that uses the APIs for sending and receiving data provided by Stream Data Platform - AF. Details about how to use the APIs for sending and receiving data, notes on application development, and how to compile adaptors are explained in 6. *Creating Custom Adaptors*. Additional details about the APIs for sending and receiving data are explained in 7. *APIs for Sending and Receiving Data*.

Stream Data Platform - AF also provides sample programs for creating custom adaptors. Details about these sample programs are explained in 8. *Sample Programs Using APIs for Sending and Receiving Data*.

Chapter

2. Using CQL to Define Queries

This chapter explains how to use CQL to define queries.

For details about the CQL coding methods described in this chapter, see 4. *CQL Reference*.

- 2.1 CQL structure
- 2.2 Generating input relations using window operations
- 2.3 Data extraction using a relational operation
- 2.4 Conversion into output stream data using stream operation
- 2.5 Limiting memory usage by specifying time division
- 2.6 Definition examples

2.1 CQL structure

Stream Data Platform - AF analyzes the stream data input into the stream data processing engine using predefined scenarios and outputs the analysis result that meets your objectives.

A scenario is defined using the CQL query language.

The following table shows the CQL structures used by Stream Data Platform - AF.

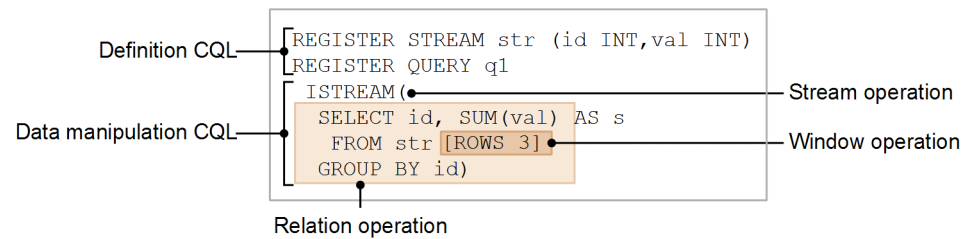
Table 2-1: CQL structures used by Stream Data Platform - AF

No.	CQL classification	Usage	Example of applicable CQL commands
1	Definition CQL	Defines streams and queries so they can be processed by the stream data processing engine.	REGISTER STREAM REGISTER QUERY, etc.
2	Data manipulation CQL	Defines the detailed stream data processing to be performed by a query. It is entered after the REGISTER QUERY clause. In data manipulation CQL, the following three types of operations can be defined in queries: <ul style="list-style-type: none"> Window operations Relation operations Stream operations 	SELECT FROM WHERE GROUP BY HAVING UNION, etc.

Details about CQL are explained in 4. *CQL Reference*.

The following figure shows an example of using CQL to specify a query.

Figure 2-1: Example of specifying a query using CQL



Definition CQL specifies the following information to the stream data processing engine:

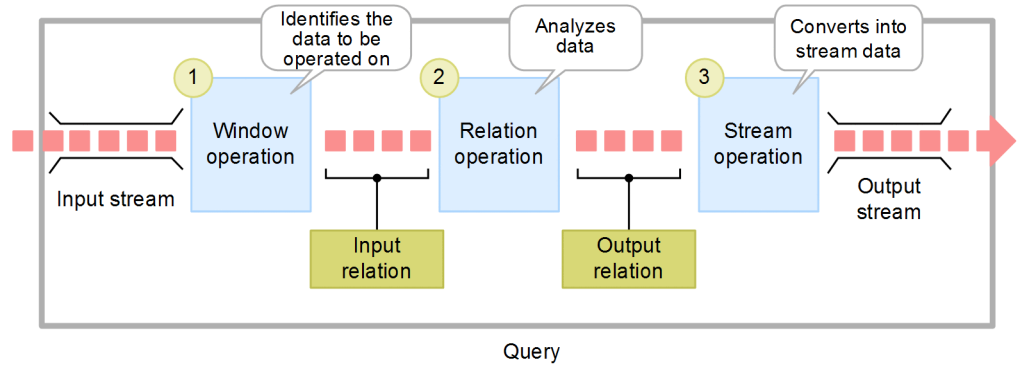
- Which stream is to be processed

- The query containing the data manipulation CQL that will perform the actual analysis

Data manipulation CQL defines the operations in a scenario that will analyze the data retrieved from the input stream queue.

The following figure shows the relationship between the window operations, relation operations, and stream operations defined in a query.

Figure 2-2: Relationship between operations



Each of these operations is explained below. The numbers in this explanation correspond to the numbers in the figure.

1. The data to be operated on is specified using a window operation. The stream data processing engine cannot process time-series input stream data or output stream data as is. You must use a window operation to isolate a tuple (a set of n -term pairs with a particular lifespan) (relation) from the stream data before the engine can process the data. A relation specifying the data to be operated on is called an *input relation*.

A window operation uses the `ROWS` window, `RANGE` window, `PARTITION BY` window, or the like to specify the data to be operated on based on the number of tuples or their time stamps.

2. A relation operation is used to perform the analysis that suits your purpose.

A relation operation performs operations such as selection, linkage, and aggregation to extract the resulting data as a relation. This relation is called an *output relation*.

3. A stream operation converts the data from a relation operation into stream data. Based on the changes specified in the output relation, data can be added, deleted, and grouped before it is output as stream data.

2.2 Generating input relations using window operations

This section explains how to use window operation to generate input relations.

A window operation isolates part of the stream data to specify the data to be processed. You can also specify the data to be processed by combining windows with different axes, such as the number of tuples and their time stamps.

2.2.1 Types of window operations

Window operations can use any of four types of windows to specify the data to be processed.

The input relation generated by each type of window is explained below.

(1) *Specification based on the number of data items (ROWS window)*

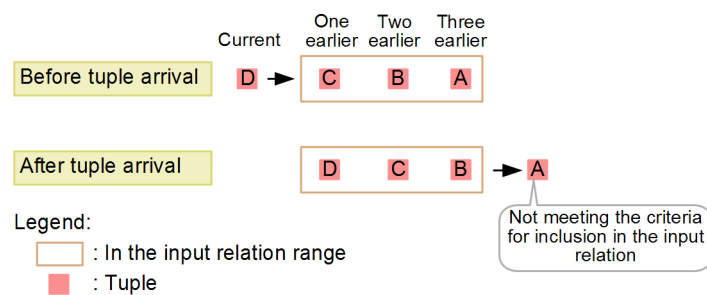
The ROWS window specifies an input relation range based on the number of tuples.

For example, if you want the three most recent tuples from the input stream data *s1* to be in the input relation, enter the CQL as follows:

```
SELECT ... FROM s1 [ROWS 3]...
```

The following figure shows the content of the input relation in this case.

Figure 2-3: ROWS window example



If tuple D arrives when there are already three tuples (A, B, and C) in the input relation, the most recent three tuples, including the newly arrived tuple, are now in the input relation. In this case, the oldest tuple (A) is removed from the input relation.

(2) *Specification based on time interval (RANGE window)*

The RANGE window specifies an input relation range based on a time interval.

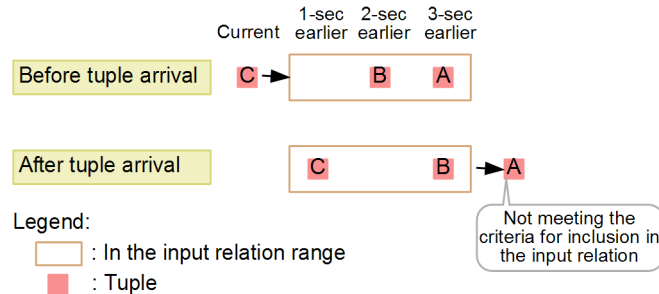
For example, if you want the last three seconds of input stream data *s2* to be in the

input relation, enter the CQL as follows:

```
SELECT ... FROM s2 [RANGE 3 SECOND]...
```

The following figure shows the content of the input relation in this case.

Figure 2-4: RANGE window example



When tuple C arrives, the tuples from the last three seconds are in the input relation. After tuple C is added to the input relation, tuple B becomes the oldest tuple that is less than 3 seconds old, and as a result tuple A is removed from the input relation.

(3) Specification based on the time stamp of the arriving tuple (NOW window)

The NOW window uses only the time stamp of the arriving tuple to determine the data to be operated on.

Whereas the input relations specified by the other windows are *line* relations having a count or time range, the NOW window makes a *point* relation using the time stamp of the arriving tuple to determine the target of operation.

For example, for the input stream data *s3*, to make the tuple having the same time stamp as the arriving tuple the operation target, the CQL is defined as follows:

```
SELECT ... FROM s3 [NOW]
```

(4) Specification based on data group (PARTITION BY window)

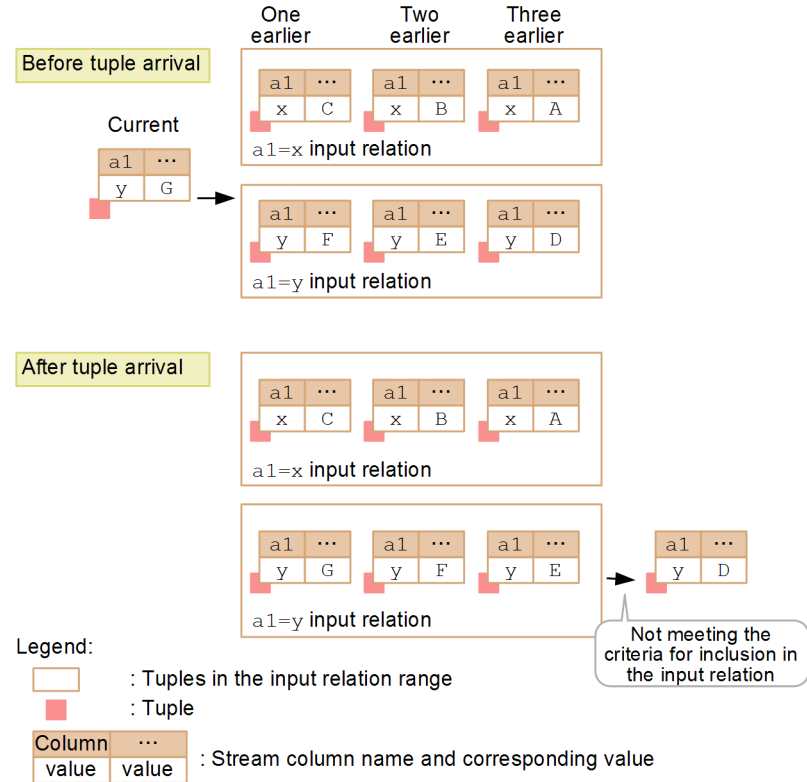
The PARTITION BY window specifies an input relation range based on the number of tuples for each tuple type.

For example, for the input stream data *s4*, to keep the last three tuples for each value in the column specification list *a1* in the input relations, enter the CQL as follows:

```
SELECT ... FROM s4 [PARTITION BY a1 ROWS 3]
```

The following figure shows the content of the input relation in this case.

Figure 2-5: PARTITION BY window example



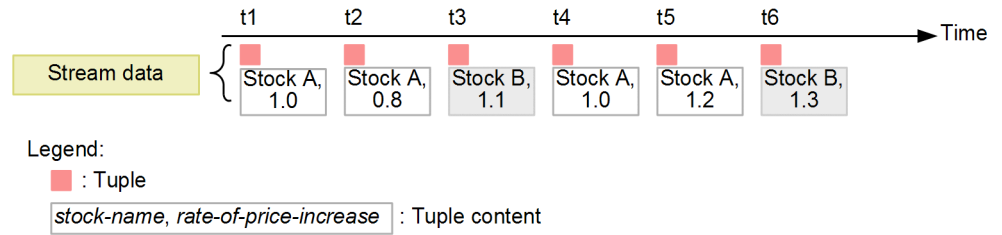
Column *a1* has a value of either *x* or *y*, and for each of these the last three tuples are kept in the input relation. When a new tuple having the value *y* for column *a1* arrives, the oldest tuple (tuple D) having the value *y* for column *a1* is removed from the input relation.

2.2.2 Window operation examples

This subsection provides examples of individual window operations.

Assuming stream data with the configuration shown in the following figure arrives, the input relation created by the specification of each window is explained.

Figure 2-6: Configuration of the tuples used in the window operation examples



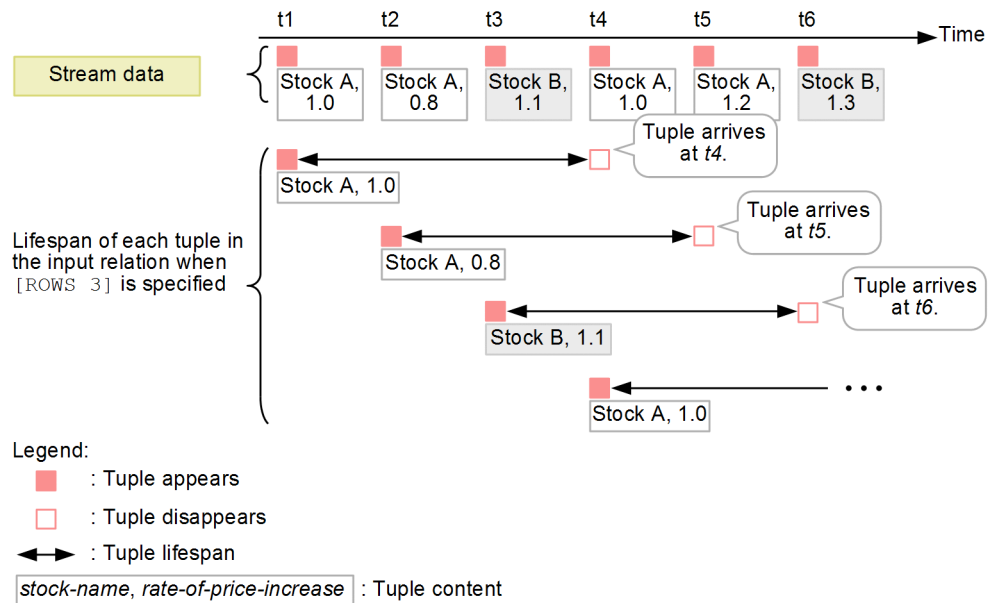
The horizontal axis in the figure shows time, increasing from left to right. Times $t1$ to $t6$ indicate the time at which tuples arrived. Each tuple consists of a stock name and a rate of price increase.

(1) Example of specification based on the number of data items ([ROWS 3] example)

This subsection shows an example of a window operation that specifies [ROWS 3]. This specification means that the number of tuples in the input relation is three.

The following figure shows lifespan of each tuple in the input relation when [ROWS 3] is specified.

Figure 2-7: Lifespan of each tuple in the input relation when [ROWS 3] is specified



As the time passes from $t1$ to $t2$ to $t3$, tuples (Stock A, 1.0), (Stock A, 0.8), and (Stock

B, 1.1) arrive sequentially and are included in the input relation.

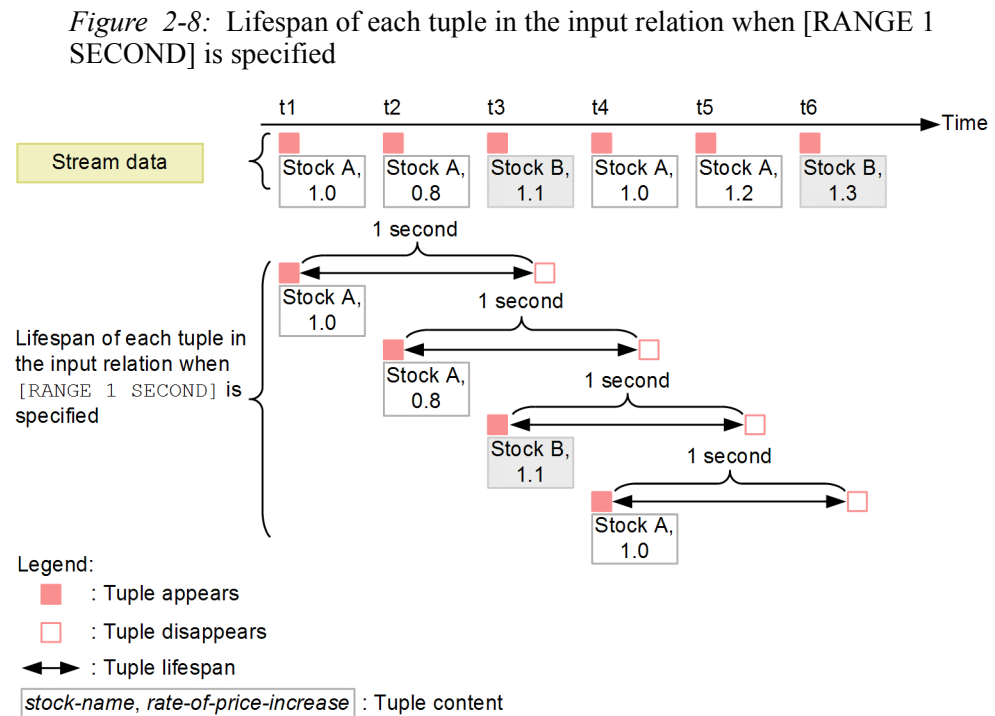
When the tuple (Stock A, 1.0) arrives at time t_4 , the number of tuples in the input relation exceeds three, and the oldest tuple in the input relation expires. In this example, the tuple (Stock A, 1.0) generated at time t_1 is deleted from the input relation and the tuple (Stock A, 1.0) that arrived at time t_4 is newly added to the input relation.

(2) Example of specification based on time ([RANGE 1 SECOND] example)

This subsection shows an example of a window operation that specifies [RANGE 1 SECOND]. This specification means that lifespan of any given tuple will only be one second in the input relation.

Each tuple included in the input relation disappears after one second regardless of the number of tuples that arrive subsequently.

The following figure shows the lifespan of each tuple in the input relation when [RANGE 1 SECOND] is specified.



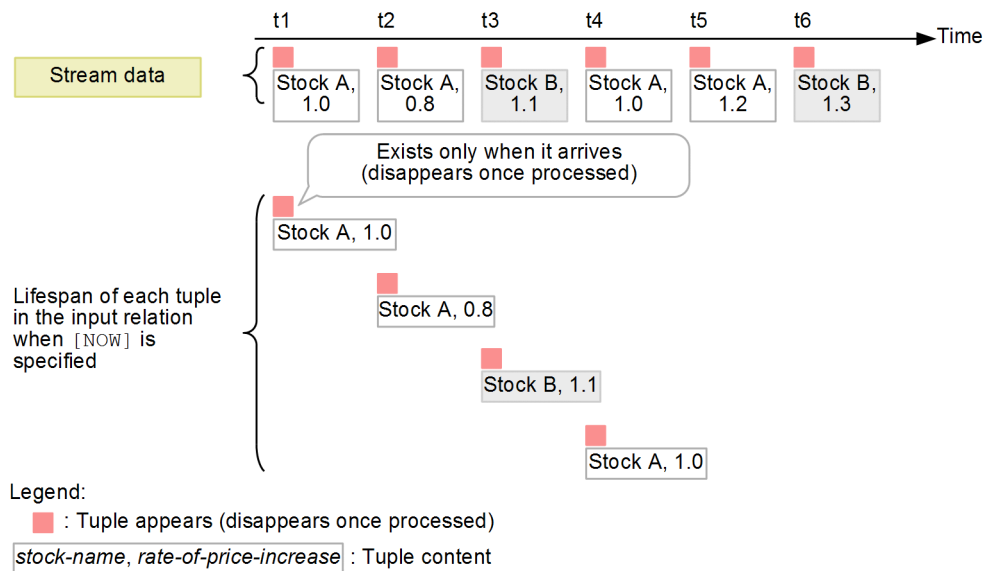
Tuples that arrive at time t_1 to t_4 disappear after one second regardless of when other tuples arrive.

(3) Example of specification based on the time stamp of the arriving tuple (NOW example)

This subsection shows an example of a window operation that specifies `[NOW]`. This specification means that the input relation only contains the time stamp of the current tuple.

The following figure shows the lifespan of each tuple in the input relation when `[NOW]` is specified.

Figure 2-9: Lifespan of each tuple in the input relation when `[NOW]` is specified



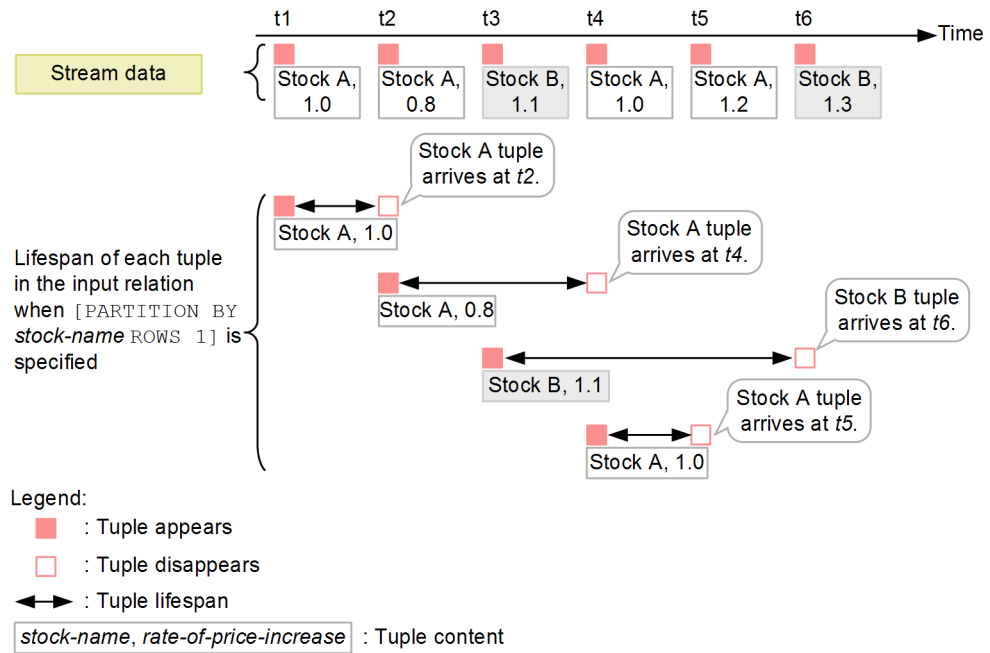
When `[NOW]` is specified, the tuple only exists in the input relation when it arrives. Once an operation is performed on the tuple, it disappears from the input relation.

(4) Example of specification based on data group ([PARTITION BY stock-name ROWS 1] example)

This subsection shows an example of a window operation that specifies `[PARTITION BY stock-name ROWS 1]`. This specification means that the number of tuples specified by the `ROWS` operand are included in the input relation for each stock name.

The following figure shows the lifespan of each tuple in the input relation when `[PARTITION BY stock-name ROWS 1]` is specified.

Figure 2-10: Lifespan of each tuple in the input relation when ([PARTITION BY stock-name ROWS 1] is specified



When this is specified, one tuple is included for Stock names A and B.

When the new tuple (Stock A, 0.8) arrives at time t_2 while the tuple (Stock A, 1.0), which arrived at time t_1 , is still in the input relation, the older tuple (Stock A, 1.0) disappears and is replaced with the tuple (Stock A, 0.8). If the tuple (Stock B, 1.1) subsequently arrives at time t_3 , the tuple (Stock A, 0.8) does not disappear since the new arrival is for a different stock, and the new tuple (Stock B, 1.1) is included in the input relation. The tuple (Stock A, 0.8), which arrived at time t_2 , stays in the input relation until the newer tuple (Stock A, 1.0) arrives at time t_4 , and the tuple (Stock B, 1.1), which arrived at time t_3 , stays in the input relation until the newer tuple (Stock B, 1.3) arrives at time t_6 .

2.3 Data extraction using a relational operation

This section explains data extraction using a relational operation.

A relational operation manipulates the stream data in the input relation using selection, linkage, and aggregate functions, and extracts the resulting data as an output relation.

2.3.1 Relational operation types

A relational operation extracts results using the following three types of operations:

- **Select**

Extracts the tuples that satisfy the specified condition from an input relation containing n -tuples.

An example follows:

```
REGISTER QUERY q1 SELECT s1.a FROM s1 [ROWS 10] WHERE s1.a > 10;
```

In this example, the tuples that satisfy the condition specified beginning with the WHERE clause are extracted from the input relation $s1$.

- **Link**

Extracts from multiple input relations containing n -tuples the results in which data items are combined where a specified condition is satisfied.

An example follows:

```
REGISTER QUERY q2 SELECT s1.a, s1.b, s2.b FROM s1 [ROWS 10], s2 [ROWS 10]
WHERE s1.a = s2.a;
```

In this example, the tuples that satisfy the condition specified beginning with the WHERE clause are extracted from the input relations $s1$ and $s2$.

- **Aggregate function**

Extracts the result obtained from executing an aggregate function on an input relation that contains n -tuples.

An example follows:

```
REGISTER QUERY q3 SELECT SUM(s1.a) AS c1 FROM s1 [ROWS 10];
```

In this example, the result obtained from executing the aggregate function SUM on the input relation `s1` is extracted.

For details about processing a relation operation, see 4. *CQL Reference*.

Of the processes performed in relation operations, linking and computational operations involving aggregate functions are described as examples below.

2.3.2 Example of a linking process

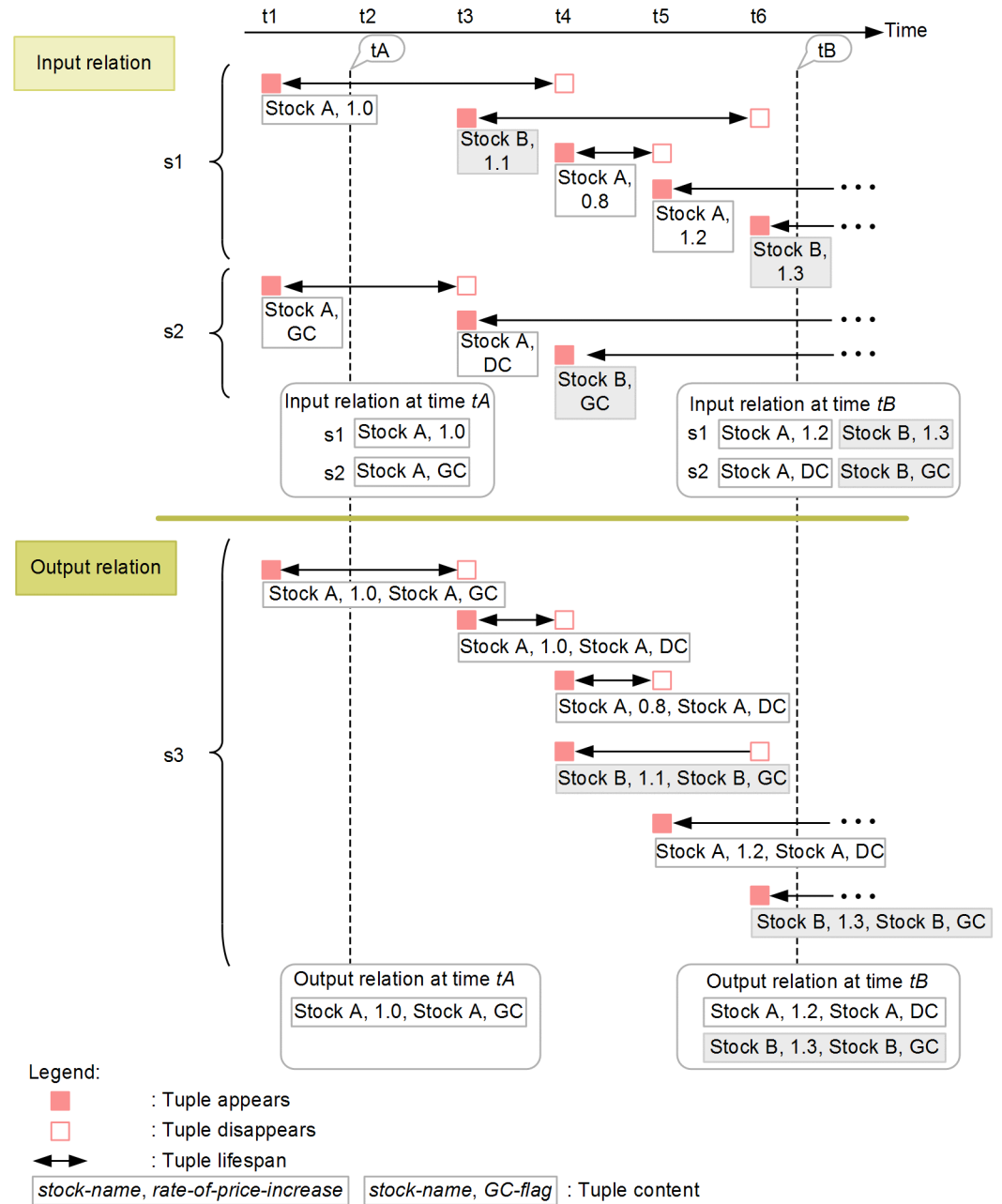
This subsection shows an example of extracting the result of linking two types (`s1` and `s2`) of input relations for each stock name.

An example of a CQL definition is described below.

```
REGISTER QUERY q
  SELECT s1.stock-name, s1.rate-of-price-increase, s2.stock-name, s2.GC-flag
  FROM s1 [PARTITION BY s1.stock-name ROWS 1],
        s2 [PARTITION BY s2.stock-name ROWS 1]
  WHERE s1.stock-name=s2.stock-name;
```

The following figure shows examples of input and output relations when this CQL is executed.

Figure 2-11: Example of a linking process



The horizontal axis indicates time, which advances from left to right. Times $t1$ to $t6$

indicate the time at which tuples arrived. The tuples in input relation s_1 consist of a stock name and a rate of price increase, while the tuple in s_2 consists of a stock name and a GC flag. The value of the GC flag is either DC (Dead Cross) or GC (Golden Cross).

In this example, tuples are linked based on the stock name and the result is output.

At time t_A , a tuple (Stock A, 1.0) and a tuple (Stock A, GC) are in input relations s_1 and s_2 , respectively. As a result of linking these tuples based on the stock name, a tuple (Stock A, 1.0, Stock A, GC) is put in the output relation.

Likewise, at time t_B , the tuples (Stock A, 1.2) and (Stock B, 1.3) are in input relation s_1 , while the tuples (Stock A, DC) and (Stock B, GC) are in input relation s_2 . As a result of linking these tuples based on the stock name, the tuples (Stock A, 1.2, Stock A, DC) and (Stock B, 1.3, Stock B, GC) are put in the output relation.

In this way, when a linking process is performed at a specified time, the results extracted to the output relation as tuples depend on the lifespan of the tuples in the input relation.

2.3.3 Example of processing using an aggregate function

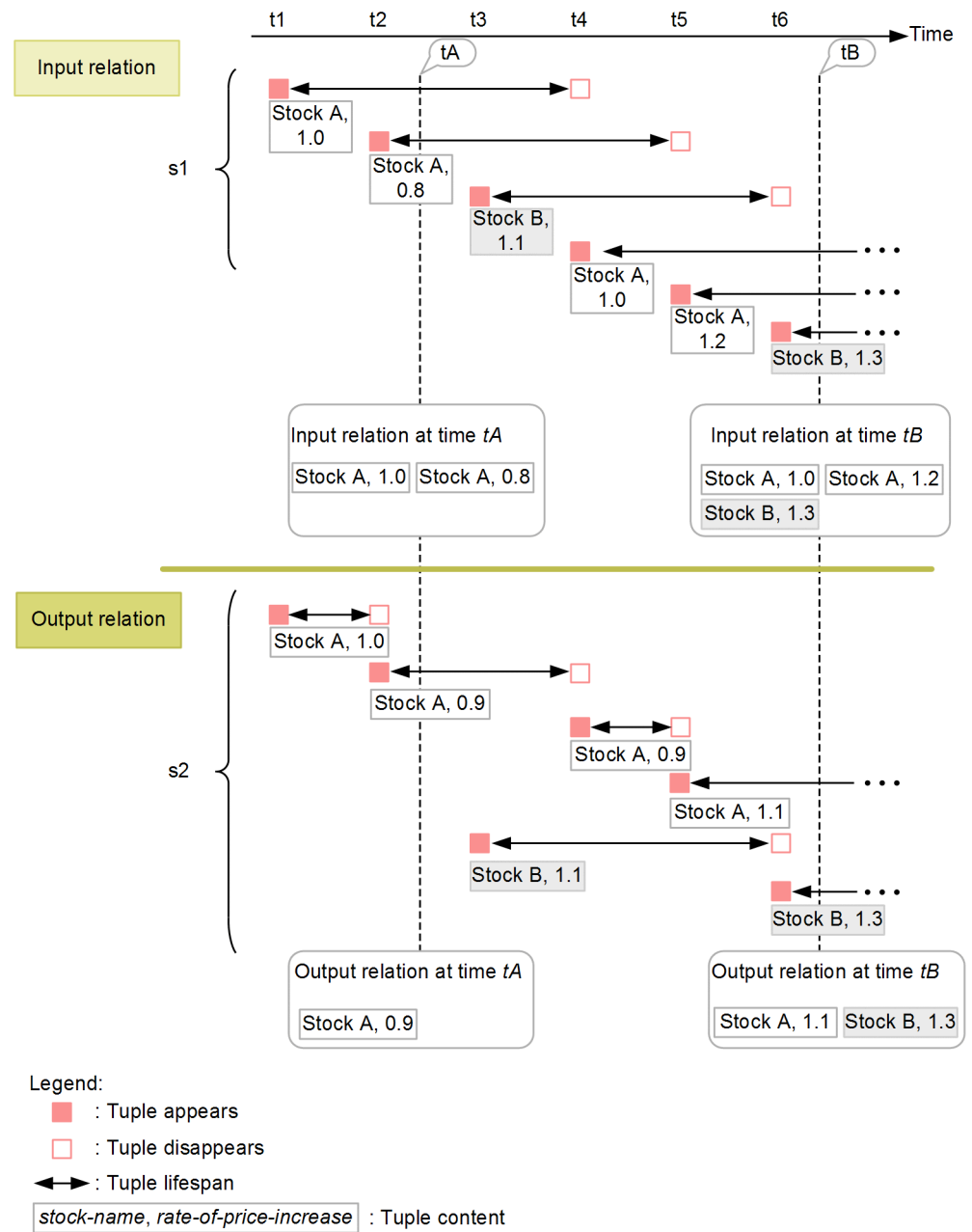
This subsection shows an example of calculating the average rate of price increase for each stock, using the GROUP BY clause and the aggregate function AVG for tuples that contain a stock name and a rate of price increase.

An example of a CQL definition is described below. In the actual CQL definition, use single-byte alphanumeric characters to specify the column names. For AVG, specify an alias.

```
REGISTER QUERY q
  SELECT AVG(rate-of-price-increase)
  FROM s1[ROWS 3] GROUP BY stock-name;
```

The following figure shows the input and output relations when this CQL is executed.

Figure 2-12: Example of processing using an aggregate function



The horizontal axis indicates time, which advances from left to right. Times t_1 to t_6 indicate the time at which tuples arrived.

In this example, for each stock name identified using the `GROUP BY` clause, the aggregate function `AVG` (for calculating the average) is used to specify the rate of price increase. Note the input relation here has a duration specified by `[ROWS 3]`.

The set in the input relation at time tA is (Stock A, 1.0)(Stock A, 0.8) and the average by stock name is (Stock A, 0.9).

Likewise, the set at time tB is (Stock A, 1.0)(Stock A, 1.2)(Stock B, 1.3), and the averages by stock name are (Stock A, 1.1)(Stock B, 1.3).

In this way, the average rate of price increase by stock name at the specified time is extracted to the output relation as tuples that depend on the lifespan of the tuples in the input relation.

2.3.4 Notes on using the linking process and the ROWS window together

This subsection gives more details about using the linking process and the `ROWS` window together.

If the number of tuples that are output as a result of the linking process exceeds the number of tuples specified in the `ROWS` window, only the number of tuples specified in the `ROWS` window are output; not all of the processing results are output.

This problem occurs if you specify queries that satisfy the following conditions:

1. A linking operation in a query generates multiple data items at the same time.
2. In a separate query following the first query, processing is performed that uses the results of the query in 1 above; in this query, the specified number of tuples in the `ROWS` window is less than the number of data items generated in the first query.

The following example shows queries that satisfy these conditions.

```
REGISTER STREAM s1 (c1 INT);
REGISTER STREAM s2 (c2 INT);
REGISTER QUERY q1 ISTREAM(          query-in-condition-1
  SELECT * FROM s1[ROWS 3], s2[ROWS 3]);
REGISTER QUERY q2 ISTREAM(          query-in-condition-2
  SELECT * FROM q1[ROWS 1]);
```

For these queries, let us assume that a tuple arrives at stream `s1` or `s2` at time t . In query `q1`, two stream data items are input and linked. Since `[ROWS 3]` is specified for the input stream data, the linking operation at time t generates three tuples having the same time stamp.

Since the number of tuples (1) specified in the `ROWS` window in query `q2` is less than the number of data items (3) generated in query `q1` and still in the window, only the last tuple arriving in the window at time t survives. Of the processing results output

from query q_1 , the remaining two tuples are not output when q_2 is processed.

In this example, you would need to change the CQL by specifying a value greater than 3 in the `ROWS` window in query q_2 .

2.4 Conversion into output stream data using stream operation

This section explains the use of stream operations to convert the results of analysis into an output stream data.

The stream operation specifies how the analysis results are to be output as stream data based on changes to the data in the output relation. A stream operation can output the following three types of tuples as stream data:

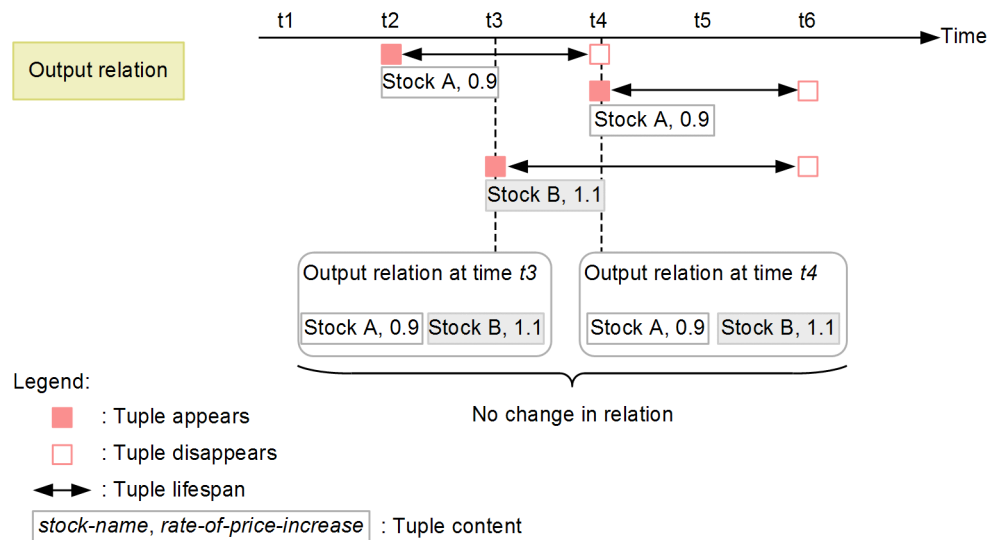
- Tuples that have been added to the output relation
- Tuples that have been deleted from the output relation
- Sets of tuples inside the output relation

Hint:

A change to the data in the output relation means a change when the output relation is compared at two points in time. Even if a specific tuple in the output relation changes, it is not considered a change if the overall result remains the same.

An example follows:

Figure 2-13: Example in which the relation has not changed



In the figure, as a tuple (Stock B, 1.1) is added to the output relation (Stock A, 0.9) at time t_2 , the output relation changes to (Stock A, 0.9)(Stock B, 1.1) at time t_3 . Then, at time t_4 , (Stock A, 0.9) disappears, but (Stock A, 0.9) is generated as another tuple at the same time, and therefore the content of the output relation remains the same as (Stock A, 0.9)(Stock B, 1.1). In this case, the output relation is considered not to have changed at time t_4 .

2.4.1 Stream operation types

This subsection explains the types of stream operations and how to specify them.

(1) Specification for outputting the tuples that were added to the output relation (ISTREAM)

ISTREAM is an operation that outputs the tuples that were added to the output relation. An example of how to specify this follows:

```
ISTREAM( SELECT ... FROM s1 [ROWS 10] ... )
```

(2) Specification for outputting the tuples that were deleted from the output relation (DSTREAM)

DSTREAM is an operation that outputs the tuples that were deleted from the output relation. An example of how to specify this follows:

```
DSTREAM( SELECT ... FROM s2 [ROWS 10] ... )
```

(3) Specification for outputting all tuples in the output relation at a specified time interval (*RSTREAM*)

RSTREAM is an operation that outputs all tuples in the output relation at a specified time interval. An example of how to specify this follows:

```
RSTREAM[1 SECOND] ( SELECT ... FROM s3 [ROWS 10] ... )
```

Note:

If you use the data source mode as the timestamp mode, there is a risk that the intended content may not be output when you specify *RSTREAM*.

In the data source mode, the relation time is advanced as tuples arrive. For example, even if *RSTREAM* sets the output interval to one minute, if a tuple arrives at 09:01:00 and the next tuple arrives at 10:00:00, no stream data is output between 09:02 and 09:59. All of the tuples for one hour will end up being output all at once when the tuple arrives at 10:00:00.

Therefore, be sure to understand how this works when using *RSTREAM* in the data source mode.

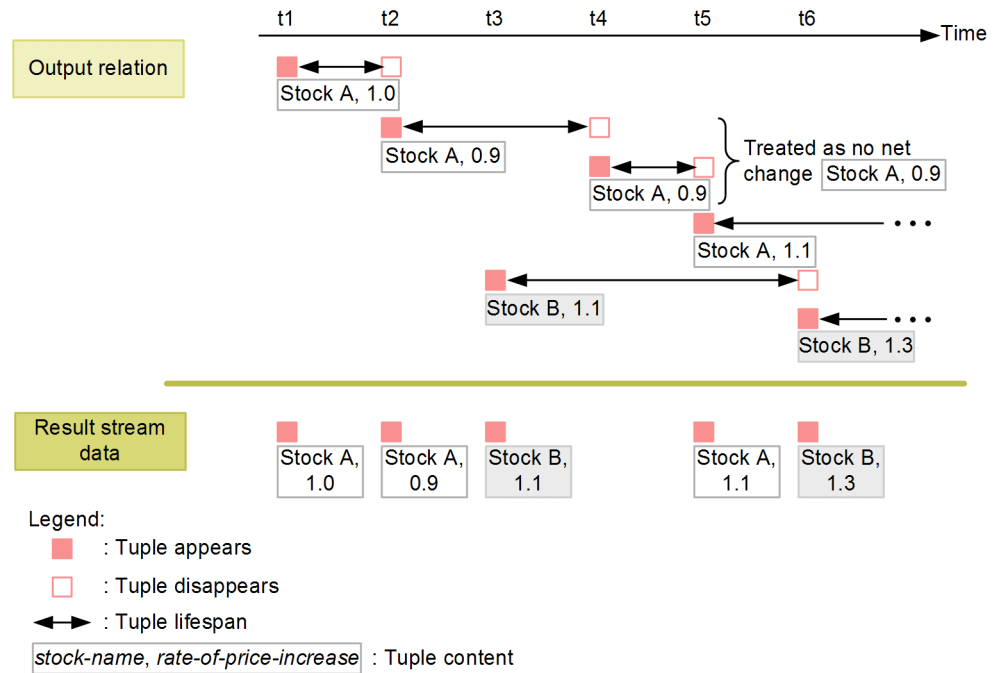
2.4.2 Stream operation examples

This subsection shows examples of the stream data that is output when each type of stream operation is used. Note that the examples here show cases in which *PARTITION BY stock-name ROWS 1* is specified in the window operation.

(1) Example of outputting the tuples that were added to the output relation (*ISTREAM*)

The following figure shows the output relation and an example of the result stream data when *ISTREAM* is specified.

Figure 2-14: Output relation and resulting stream data when ISTREAM is specified



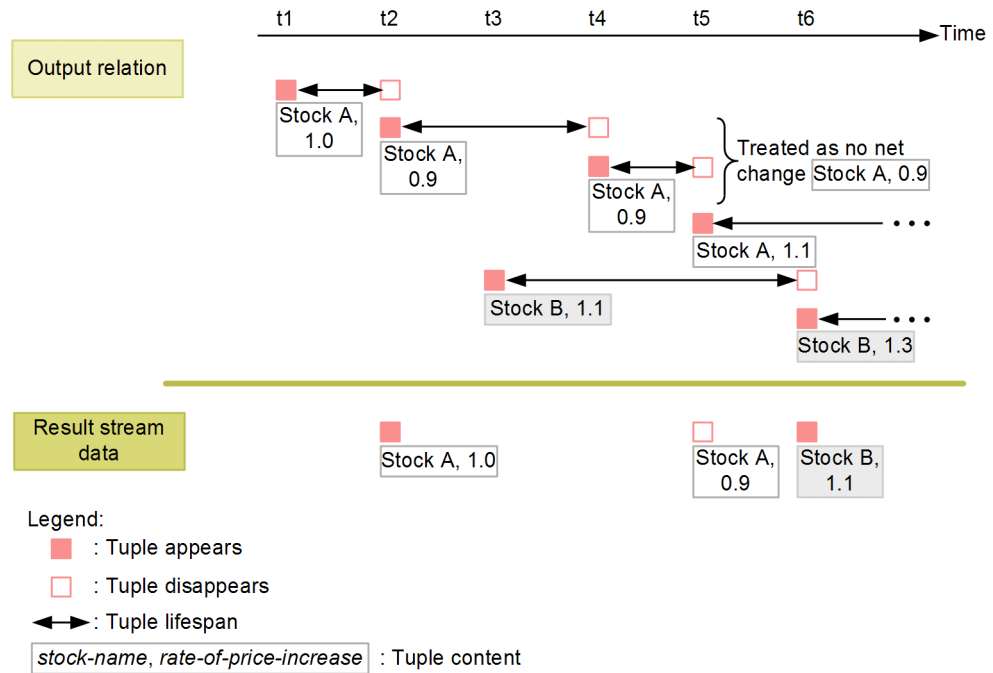
When a tuple is added to the output relation, the corresponding tuple is output to the result stream data. In this example, when the tuple (Stock A, 1.0) is added to the output relation at time $t1$, the tuple (Stock A, 1.0) is also output to the result stream data at time $t1$. Then, when tuples are added at time $t2$ or $t3$, they are also output to the result stream data.

When the tuple (Stock A, 0.9) is added at time $t4$, the tuple (Stock A, 0.9) which was added at time $t2$ is deleted, so there is no change in the relation. Consequently, nothing is output to the result stream data.

(2) Example of outputting the tuples that were deleted from the output relation (DSTREAM)

The following figure shows the output relation and an example of the result stream data when DSTREAM is specified.

Figure 2-15: Output relation and result stream data when DSTREAM is specified



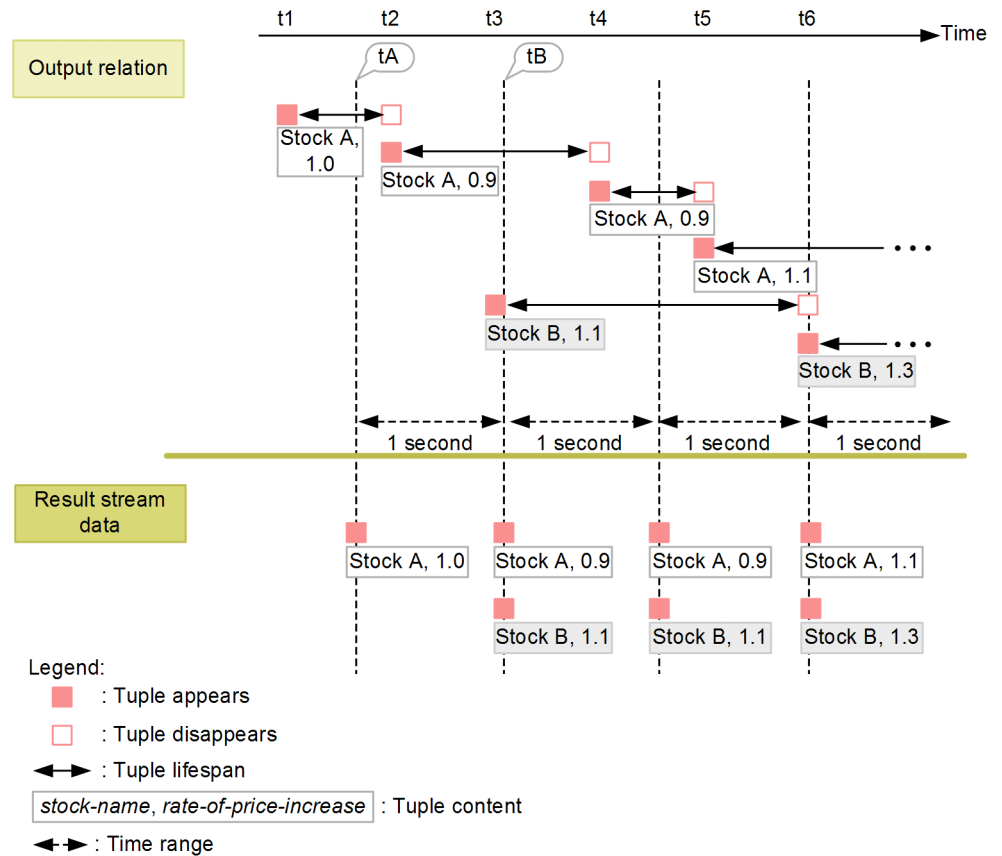
When a tuple is deleted from the output relation, the tuple is output to the result stream data. In this example, when the tuple (Stock A, 1.0) is deleted from the output relation at time t_2 , it is also output to the result stream data.

When the tuple (Stock A, 0.9) is deleted at time t_4 , the tuple (Stock A, 0.9) is added at the same time, so there is no change in the relation. Consequently, nothing is output to the result stream data just like when ISTREAM was specified.

(3) Example of outputting all tuples in the output relation at a specified time interval (RSTREAM)

The following figure shows the output relation and an example of the result stream data when RSTREAM is specified.

Figure 2-16: Output relation and result stream data when RSTREAM is specified



Tuples in the output relation are output to the result stream data at 1-second intervals. At time t_A , only the tuple (Stock A, 1.0) is in the output relation, and therefore only it is output to the result stream data. A second later at time t_B , the tuples in the output relation consist of (Stock A, 0.9) and (Stock B, 1.1) and therefore they are output to the result stream data.

Thereafter, the content of the output relation is output to the result stream data every second.

2.5 Limiting memory usage by specifying time division

This section explains how to use time division to limit memory usage when processing a large volume of received data by specifying the `RANGE` window.

When you specify the `RANGE` window in a query, all data received within the specified timeframe is processed. Therefore, depending on the specified time interval and the volume of data, memory usage may increase, adversely affecting system performance.

In contrast, you can keep the volume of data to be processed by the `RANGE` window at a constant level by specifying time division.

Time division is a facility that divides the total range of a relation specified in the `RANGE` window into arbitrary units of time and then processes each of these units. The divided range of the relation is called the mesh. The dividing interval (in milliseconds) is called the mesh interval.

If the amount of received data to be processed in the time range specified for the `RANGE` window is large, each mesh unit can be preprocessed and the results stored in the `RANGE` window as pseudo tuples. By making these pseudo tuples the target of processing for the `RANGE` window, the number of tuples in the `RANGE` window is kept constant. In this way, you can prevent excessive memory usage.

For example, when there are 100,000 tuples in the `RANGE` window, by specifying time division and preprocessing these 100,000 tuples into eight pseudo tuples, you can limit the number of tuples in memory to 8.

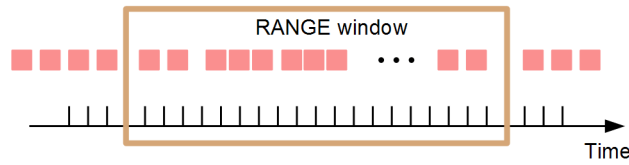
You can specify time division for queries that satisfy the following two conditions:

- The `RANGE` window is used.
- One of the following aggregate functions is used as the relation operation:
 - `SUM` (for computing the sum total)
 - `MAX` (for computing the maximum value)
 - `MIN` (for computing the minimum value)

The following figure shows the differences in operations when time division is and is not specified.

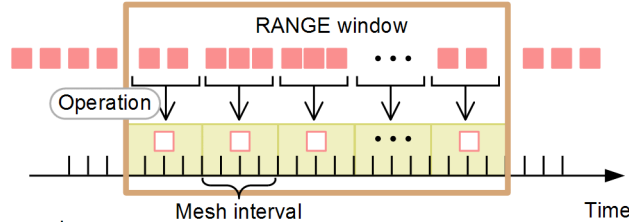
Figure 2-17: Differences in operations when time division is and is not specified

- Operation in the **RANGE** window when time division is not specified



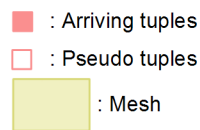
All tuples in the **RANGE** window will be processed.

- Operation in the **RANGE** window when time division is specified



Operations are first performed for mesh units, creating pseudo tuples. These are then processed in the **RANGE** window.

Legend:



When time division is not specified, all tuples in the **RANGE** window will be processed. For example, if the interval in the **RANGE** window is set to one minute and the total number of tuples that arrive in one minute is 100,000, the number of tuples to be processed is 100,000. For an operation that computes the maximum value, all 100,000 tuples must be compared in order to determine the maximum value.

On the other hand, when time division is specified, the pseudo tuples become the data being processed. Pseudo tuples are the results of the requested operation executed at the specified mesh interval. For example, if the **RANGE** window has been divided into eight segments by a mesh interval, eight pseudo tuples are created. For an operation that computes the maximum value, the maximum value can be determined by comparing the content of the eight pseudo tuples. In this way, specifying time division can reduce the number of tuples in the **RANGE** window and keep memory usage from increasing.

Hint:

Using time division is effective when the stream data transmission rate is high and the number of arriving tuples staying in the `RANGE` window is large.

Since specifying time division can keep the memory usage in the Java Heap Area constant, it can prevent unnecessary increases in memory usage.

Time division can be applied to queries that have processing logic that determines the maximum value (`MAX`), minimum value (`MIN`), or sum total (`SUM`) per unit time.

For examples of specifying time division, see 2.6.2 *Definition examples in which time division is specified*.

2.6 Definition examples

This section provides examples of query definition using CQL.

2.6.1 Basic query definition example

This subsection describes a basic query definition example involving stock prices, in which the window, relation, and stream operations are used.

The following figure gives an example of the stock price information stream data handled in this example.

Figure 2-18: Example of stock price information stream data

Stock price information stream data (*stock*)

Time	Stock code (stockID)	Stock name (stockName)	Stock price (currentPrice)	Trading volume (tradingVolume)
10:00:00	6501	A	780	12442000
10:00:05	1468	B	1650	3318000
10:00:10	2282	C	1518	1347000
...

Timestamp
Stock price information

The stream data handled in this example is data that contains stock price information related to the stock code (*stockID*), the stock name (*stockName*), the stock price (*currentPrice*), and the trading volume (*tradingVolume*), and is time stamped by Stream Data Platform - AF.

This stream data is used to calculate the rate of increase in the stock price. The rate of price increase is calculated by comparing the current data with the data from one minute ago.

To calculate the rate of price increase, the following two types of queries are used:

1-minute data calculation query

```
REGISTER QUERY stockDStream
DSTREAM ( SELECT * FROM stock[RANGE 1 MINUTE] );
```

- The RANGE window with the interval set to one minute keeps tuples in the input relation for one minute.
- DSTREAM is specified to output tuples once their one-minute lifespan has expired. In this way, data that is one minute old is output as stream data.

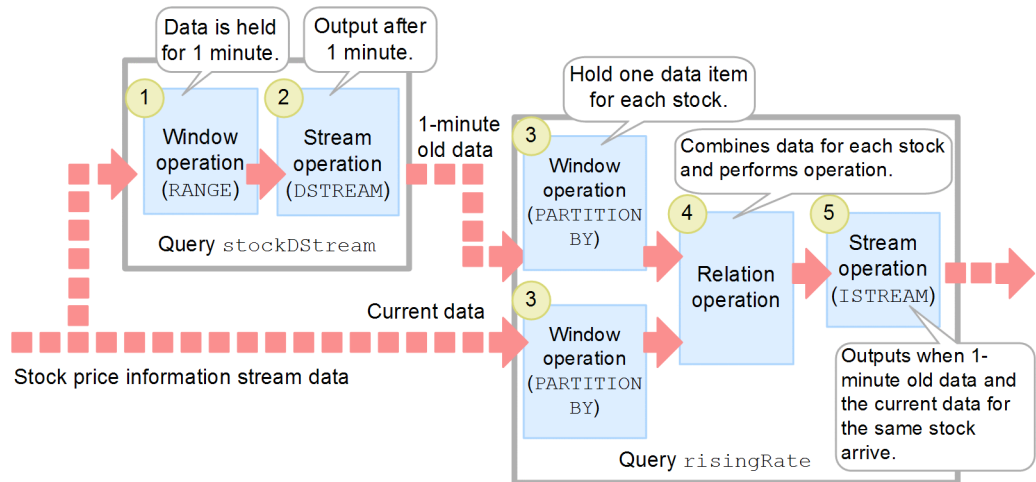
Price increase rate calculation query

```
REGISTER QUERY risingRate
ISTREAM ( SELECT stock.stockID, stock.stockName,
stock.currentPrice / stockDStream.currentPrice AS stockRate
FROM stock[PARTITION BY stock.stockID ROWS 1],
stockDStream[PARTITION BY stockDStream.stockID ROWS 1]
WHERE stock.stockID = stockDStream.stockID );
```

- The PARTITION BY window is used to hold the latest tuple for each stock.
- The current stock price information stream data is merged with the stream data from one minute ago to calculate the price increase rate.

The following figure shows the flow of operations for these queries.

Figure 2-19: Flow of operations for calculating stock price increase rate



An explanation of the processing flow is provided below. The numbers in the explanation correspond to the numbers in the figure.

1. Window operation for the query *stockDStream*

A window operation is executed.

The stream *stock* entered by specifying FROM *stock* is held for one minute because of the specification of [RANGE 1 MINUTE].

2. Stream operation for the query *stockDStream*

Data that is older than one minute is output by the stream operation DSTREAM as stream data (1-minute old data).

3. Window operation for the query `risingRate`

Two types of data, i.e., the stream `stock` and the result stream (1-minute old data) that was output in operation 2, are input because of the specification of `FROM stock..., stockDStream...`

Based on the specification of `[PARTITION BY... ROWS 1]`, the input data is grouped for each stock code (`stockID`) and the latest data item is held for each stock code. As a result, the following data is held for each stock code:

- The latest data item out of the current stock price data in the input stream `stock`
- The latest data item out of the stock price from one minute ago in the input stream `stockDStream` (result stream of the query `stockDStream`)

4. Relation operation for the query `stockDStream`

For each stock code, the stock code, the stock name, and the rate of price increase (*current stock price \div price from one minute ago*) are linked.

5. Stream operation for the query `stockDStream`

When the data from one minute ago and the current data for the same stock arrive, the linked data in 4 is output as a result stream.

2.6.2 Definition examples in which time division is specified

This subsection gives examples of the following two types of query definitions in which time division is specified:

- Example in which a single aggregate function is used
- Example in which multiple aggregate functions are used

In this example, a relation operation using an aggregate function is performed on the arriving tuples that are in the 1-minute `RANGE` window. The following stream is handled in this example:

Stream handled in the time division specification example

- The stream name is `stock`.
- Contains a column named `price`.

(1) Example in which a single aggregate function is defined

In the following definition example, the sum total of the column `price` in the stream `stock` is determined from the tuples that are in the `RANGE` window.

```

REGISTER STREAM stock(price INTEGER, name VARCHAR(10)); ...(1)
REGISTER QUERY_ATTRIBUTE q1 STREAM_NAME=stock PERIOD=300MS
TARGETS=SUM(price); ...(2)
REGISTER QUERY q1 ...(3)
ISTREAM(
    SELECT name, SUM (price) AS s1
    FROM stock[RANGE 1 MINUTE]
    GROUP BY name
);

```

Note:

Numbers (1)-(3) correspond to the explanations shown below. They are not coded in the actual definition.

The name of the query in which time division is used is q1 specified in (3). Therefore, the query name q1 is also specified in the QUERY_ATTRIBUTE statement in (2).

In this example, the sum total of the column price is determined using time division for the stream stock specified in (1). Therefore, stock is specified for the STREAM_NAME= data identifier in (2) and the aggregate function SUM (price) is specified for TARGETS=.

If you want to determine the maximum or minimum value instead of the sum total, specify the aggregate function MAX or MIN.

(2) Example in which multiple aggregate functions are defined

In the following definition example, the sum total, maximum value, and minimum value of the column price in the stream stock are determined from the tuples in the RANGE window.

```

REGISTER STREAM stock(price INTEGER, name VARCHAR(10));
REGISTER QUERY q0
ISTREAM(
    SELECT name, price AS price0, price AS price1, price AS price2
    FROM stock[NOW]
);
REGISTER QUERY_ATTRIBUTE q1 STREAM_NAME=q0 PERIOD=300MS
TARGETS=SUM(price0),MAX(price1),MIN(price2);
REGISTER QUERY q1
ISTREAM(
    SELECT q0.name, SUM(q0.price0) AS sum_price,
           MAX(q0.price1) AS max_price, MIN(q0.price2) AS min_price
    FROM q0[RANGE 1 MINUTE]
    GROUP BY q0.name
);

```

When using time division, you cannot execute multiple aggregate functions for the

same column name of the same data identifier at the same time. For example, you cannot specify `TARGETS=SUM(price), MAX(price), MIN(price)` for a query on column name `price` of the data identifier `stock`.

When specifying multiple aggregate functions, you must provide separate column names as the targets of the aggregate functions. In this example, the individual aggregate functions `SUM`, `MAX`, and `MIN` are executed after the column name is changed from `price` to `price0`, `price1`, and `price2`, respectively.

Chapter

3. CQL Basic Items and Data Types

This chapter explains CQLs basic items and data types.

- 3.1 Defining queries using CQL
- 3.2 Specifying basic items in CQL
- 3.3 CQL data types
- 3.4 Data comparison
- 3.5 Notes on query definitions and limit values

3.1 Defining queries using CQL

You use CQL to define queries. CQL uses the following items (basic items) to define processing:

- Keywords
- Numeric values
- Delimiters
- Names
- Constants

A keyword is a character string that specifies a data-processing function, such as `SELECT` and `WHERE`. The numeric values, delimiters, names, and constants that are used to define processing targets and processing ranges for a keyword are called operands.

When specifying these basic items, you must pay attention to the characters and data types that can be used.

This section explains the format of the basic items and the characters that can be used when using CQL for defining queries. It also explains the symbols used in the explanation of CQL syntax provided later.

3.1.1 CQL format

Enter CQL using the following format:

- CQL uses a free format. Operands are specified in the order described in each CQL format. For the CQL formats, see *4. CQL Reference*.
- CQL statements are separated using a semicolon (;).

A statement may span multiple lines, but multiple statements cannot be entered on a single line.

Correct and incorrect coding examples follow:

Correct coding example

```
REGISTER STREAM
  s1(id INT,name VARCHAR(10));
REGISTER QUERY q1
  SELECT s1.name FROM s1[ROWS 10];
```

Incorrect coding example

```
REGISTER STREAM s1(id INT); REGISTER QUERY q1 SELECT * FROM s1[ROWS 10];
```

An error occurs because two REGISTER statements are entered on a single line.

- To enter comments, use two forward slashes (//). After the two forward slashes are entered, the rest of the text on the line is treated as a comment. You cannot insert comments in the middle of a statement.

Correct and incorrect coding examples follow:

Correct coding example

```
// comments
REGISTER STREAM
  s1(id INT,name VARCHAR(10)); // comment
```

Incorrect coding example

```
REGISTER QUERY q1 // comment
  SELECT s1.name FROM s1[ROWS 10];
```

An error occurs because a comment is inserted in the middle of the REGISTER QUERY statement.

If you need to enter two forward slashes (//) in the middle of a character string for a purpose other than entering comments, make sure the forward slashes do not occur at the beginning of a line. Correct and incorrect coding examples follow:

Correct coding example

```
REGISTER QUERY q1 SELECT * FROM s1[NOW] WHERE s1.c1='ab//cd';
```

Incorrect coding example

```
REGISTER QUERY q1 SELECT * FROM s1[NOW] WHERE s1.c1='ab
//cd';
```

An error occurs because a line begins with two forward slashes (the line is treated as a comment line).

3.1.2 Characters that can be used in CQL

This subsection explains the characters that can be used in CQL.

(1) Character code that can be used

The only character code that can be used in CQL is MS932.

(2) Characters that can be used

The following table shows the characters that can be used in CQL.

Table 3-1: Characters that can be used in CQL

No.	Type	Characters that can be used
1	Name	Single-byte alphanumeric characters and the underscore (<code>_</code>) can be used. However, a name must start with a single-byte alphanumeric character. For details about name specification, see 3.2.4 <i>Specifying names</i> .
2	Character string constant	Single- and double-byte characters can be used.
3	Other types	<p>Single-byte characters and the special symbols listed below can be used.</p> <ul style="list-style-type: none"> Single-byte characters Uppercase letters (A to Z), lowercase letters (a to z), numbers (0 to 9), spaces, and underscore (<code>_</code>) Special symbols Comma (<code>,</code>): Selection lists, value expressions, and the like Period (<code>.</code>): Floating-point constants, column specifications, date/time data, and the like Colon (<code>:</code>): Time, timestamp data Hyphen (<code>-</code>): Date, timestamp data Single quotation mark (<code>'</code>): To enclose a character string Left parenthesis (<code>(</code>) and right parenthesis (<code>)</code>): Relation expressions, to enclose cast specifications, and so on Less than operator (<code><</code>), greater than operator (<code>></code>), equal to operator (<code>=</code>), and exclamation mark (<code>!</code>): Comparative operators Plus sign (<code>+</code>), minus sign (<code>-</code>), and forward slash (<code>/</code>): Arithmetic operators Asterisks (<code>*</code>): Arithmetic operator, whole column output, etc. Semicolon (<code>;</code>): Statement delimiter Square brackets (<code>[]</code>): Enclosing tab code for time specifications, line feed code, carriage return code Two forward slashes (<code>//</code>): Comments

3.1.3 Symbols used in the explanation of CQL syntax

The following table shows the symbols used in the explanation of CQL syntax. From here on, Chapters 3 and 4 uses these symbols when explaining CQL syntax.

Table 3-2: Symbols used in the explanation of CQL syntax

Symbol	Meaning	Example
{ }	One of the multiple items enclosed in these symbols must be selected.	{ <i>character-string-constant</i> <i>numeric-constant</i> } Either a character string constant or numeric constant must be entered.
[]	Items enclosed in these symbols are optional. If multiple items are listed, all may be omitted or one of them must be selected as in the symbols { }.	[<i>SECOND</i> <i>MILLISECOND</i>] Both may be omitted, or <i>SECOND</i> or <i>MILLISECOND</i> can be entered.
...	The item preceding this symbol can be repeated as needed.	<i>relation-reference</i> [, <i>relation-reference</i>] ... Relation reference may be entered repeatedly.
' (' ') ' '	The item enclosed in these symbols must be entered as is including the parentheses (()).	' (' <i>TINYINT</i> ') ' ' <i>TINYINT</i> must be entered enclosed in parentheses (()).
' [' '] ' '	The item enclosed in these symbols must be entered as is including the square brackets ([]).	' [' <i>window-specification</i> '] ' ' Window specification must be entered enclosed in square brackets ([]).
::=	The item on the left side of this symbol is defined as the item on the right.	<i>selection-list</i> ::= <i>select-expression</i> [, <i>selection-list</i>]
Δn	Indicates <i>n</i> or more delimiters. If <i>n</i> is omitted, <i>n</i> = 1 is assumed.	NOT { Δ 0 ' (<i>search-condition</i> ') ' Δ <i>comparison-predicate</i> } Enter zero or more delimiters between NOT and the (search condition), or enter one or more delimiters between NOT and the comparison predicate.

3.2 Specifying basic items in CQL

This section explains how to specify basic items in CQL.

3.2.1 Specifying keywords

A word, such as the name of a CQL statement (for example, `SELECT` or `UNION`) that must be specified in order to use a function is called a *keyword*.

For details about the meanings of keywords, see *4. CQL Reference*.

The following figure shows examples of specifying keywords.

Figure 3-1: Keyword specification examples

REGISTER	STREAM	(ID	C1	CHAR	,	C2	INTEGER)	;
Keyword	Keyword		Name	Name	Keyword		Name	Keyword		
				(reserved word)			(reserved word)			

Since most keywords are defined as system reserved words, they must be specified in their predetermined positions.

A CQL keyword that is not defined as a reserved word, however, can also be used as a name. In this case, the names are not case-sensitive.

The following table shows the character strings defined as system reserved words, in alphabetic order grouped by their leading character.

Table 3-3: Character strings defined as system reserved words

Leading character	Reserved words
A	ALL, AND, ANY, AS, ASC, AST, AVG
B	BETWEEN, BIGINT, BINARY, BOOLEAN, BY
C	CASE, CHAR, CHARACTER, CON, COUNT
D	DATE, DAY, DEC, DECIMAL, DEFAULT, DELETABLE, DELETE, DESC, DISTINCT, DOUBLE, DSTREAM
E	ELSE, END, EOF, EQ, EXISTS
F	FALSE, FLOAT, FROM
G	GE, GT, GROUP
H	HAVING, HOUR

Leading character	Reserved words
I	IDSTREAM, IN, INSERT, INSTANT, INT, INTEGER, IS, ISTREAM
J	No reserved words
K	No reserved words
L	LATEST, LE, LIKE, LIMIT, LLP, LP, LRP, LT
M	MAX, MILLISECOND, MIN, MINUTE, MIS
N	NE, NOT, NOW, NULL, NUMERIC
O	OR, ORDER, OTHER
P	P_INT, PARTITION, PLS, POM, PR, PREV
Q	No reserved words.
R	RANGE, RANKING, REAL, RECENT, RELATIONAL, ROWS, RSTREAM
S	SECOND, SELECT, SMALLINT, SOME, SQU, SUB, SUM
T	THEN, TINYINT, TIME, TIMESTAMP, TRUE
U	UNBOUNDED, UNION, UNKNOWN, UPDATE
V	VARBINARY, VARCHAR
W	WHEN, WHERE, WITH, WQU
X	No reserved words
Y	No reserved words
Z	No reserved words

3.2.2 Specifying numeric values

Numeric values (when specifying data type inside parentheses, time, number of lines, and so on) other than value expressions to be entered in CQL are entered using the notation for unsigned integers.

3.2.3 Delimiters

(1) Delimiter types

The following characters can be entered as delimiters:

- Single-byte space
- Carriage return code

- line feed code
- Tab code

When using a space as a delimiter, use a single-byte space. Double-byte spaces are not treated as delimiters.

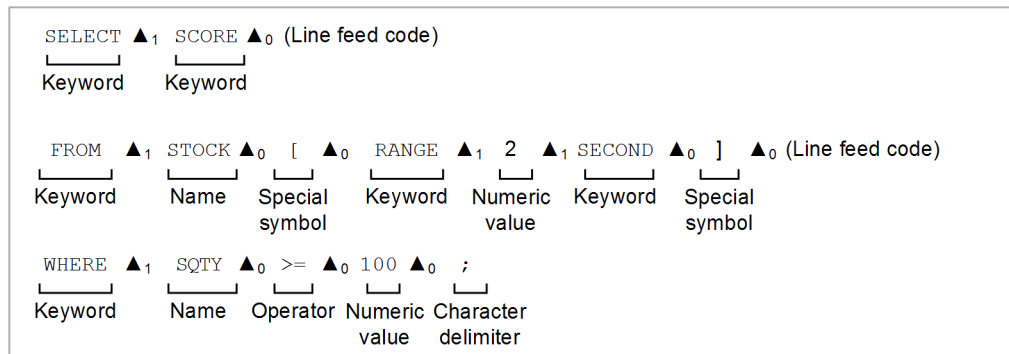
(2) Where delimiters are positioned

Delimiters are entered in the following positions:

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

The following figure shows examples of using delimiters.

Figure 3-2: Delimiter usage examples



Legend:

- ▲₀ : Zero or more single-byte spaces
- ▲₁ : One or more single-byte spaces

(3) Where delimiters can be entered

Delimiters can be entered where entering delimiters is not prohibited in (4) *Where delimiters cannot be entered*, and either before or after the following special characters:

, , ., -, +, *, ' , (,) , < , > , = , ^ , ! , # , \$, / , [,] , carriage return code, line feed code, tab code, and ;

However, for a character string constant that uses single quotation mark (') or a comparative operator such as >=, delimiters cannot be inserted between two special characters.

Examples where delimiters can be entered follow:

Example 1: Entering a space before the special character (.)

SCORE Δ .NAME

Example 2: Entering a space after the special character (=)

SCORE= Δ 1

(4) *Where delimiters cannot be entered*

Delimiters cannot be entered in the following positions:

- In the middle of a keyword

Example: S Δ ELECT

- In the middle of a name

Example: ST Δ OCK

- In the middle of a numeric constant

Example: 678 Δ 9

- In the middle of an operator

Example: < Δ =

3.2.4 Specifying names

The following rules apply when specifying names:

- Characters that can be used in names are single-byte alphanumeric characters and the underscore (_).
- A name must begin with a single-byte alphabetic character, and must not begin with a number or an underscore (_).
- Names are not case-sensitive. All letters are treated as single-byte uppercase letters.
- A name must not contain a space.
- A name cannot be the same as a CQL reserved word (a name can be the same as a keyword that is not a reserved word).
- A name must not be enclosed in double quotation marks (").
- The number of characters that can be specified for a name is between 1 and 100.

3.2.5 Name qualification

Name qualification is used to make a name unique by concatenating a data identifier and a column name using a period (.), as in *data-identifier.column-name*. This form of name qualification is called *column specification*.

(1) Data identifier specification

When two or more data identifiers are specified in a single CQL statement, a qualifier is used to uniquely associate each specified column name to a stream, relation or alias. This is called data identifier specification.

As shown in the following specification format, a stream name, relation name, or alias is specified as the qualifier.

Data identifier specification format

data-identifier : = { *stream-name* | *relation-name* | *alias* }

For details about stream names and relation names, see 4.4.14 *Relation reference*. For aliases, see 4.4.11 *Select expression* and 4.4.14 *Relation reference*.

(2) Column specification

A column name qualified by a data identifier is called *column specification*.

When executing a search that uses multiple data identifiers (a search in which two or more data identifiers are linked), more than one search-target data identifier may contain columns having the same name. In such a case, column specification clearly identifies the data identifier that corresponds to the specified column.

The specification format follows:

Format for column specification

column-specification : = *data-identifier* . *column-name*

When using a data identifier to qualify a column name, the specified data identifier must specify a column name that is within its valid range. A *valid range of data identifiers* means the column names defined by reference to relations, streams, or aliases in the FROM clause. If an undefined column name is specified, an error occurs.

For the name of the stream or relation returned as a result of an inquiry, the name of the query in which the inquiry is defined is used.

A column name is included with either of the following:

- A stream or relation indicated by a data identifier that specifies a column name that is within the valid range
- A stream or relation acquired as a result of a query

The column name for the stream or relation is returned in the query result according to the rules described below.

- For a relation that is returned by a relation expression specifying the UNION clause, the column name of the relation that is returned as the result of the first relation expression (excluding the UNION clause) becomes the column name of the relation that is returned as the result of the query.

In the example shown below, the only column name that is output from q_1 , which is specified in the relation reference in q_2 , is c_1 . Neither d_1 nor e_1 is acquired by q_2 .

```
REGISTER QUERY q1
  SELECT s1.c1 FROM s1[NOW] ...
  UNION
  SELECT s2.d1 FROM s2[NOW] ...
  UNION
  SELECT s3.e1 FROM s3[NOW] ...;

REGISTER QUERY q2
  SELECT q1.c1 FROM q1 ...;
```

- For a relation that is returned by a relation expression that does not specify the UNION clause, the column name returned by the i -th select expression becomes the column name of the i -th relation acquired.

In the example shown below, the column names that are output from q_1 , which is specified in the relation reference in q_2 , are c_1 , d_1 , and e_1 in that order, determined by the order specified in the select expression in q_1 .

```
REGISTER QUERY q1
  SELECT s1.c1, s2.d1, s3.e1 FROM s1[NOW], s2[NOW], s3[NOW];

REGISTER QUERY q2
  SELECT * FROM q1;
```

- For a stream that is returned by the stream clause, the column name of the relation that is returned as the result of the relation expression in the stream clause becomes the column name of the stream that is returned as the result of the query.

For details about the stream clause, see 4.4.2 *Stream clause*. For details about the relation expression, see 4.4.3 *Relation expression*.

(3) Range variable

The identifier used as the qualifier for column specification is called a *range variable*. A range variable has a valid range and a name. A data identifier specified using a range variable is also called a *table ID*.

An alias is a range variable. When an alias is specified, the source data identifier specified for the alias is no longer a valid range for the column name.

Examples of the valid range for column names when an alias is specified in a relation reference or select expression are described below.

- When an alias is specified in a relation reference

Correct coding example

```
REGISTER QUERY q1 SELECT a1.a FROM s1[NOW] AS a1 WHERE a1.a > 1;
```

Incorrect coding example

```
REGISTER QUERY q2 SELECT s1.a FROM s1[NOW] AS a1 WHERE s1.a > 1;
```

Since `a1` is defined as an alias of `s1`, `s1` is no longer a valid range for the column name `a`. Consequently, `s1` cannot be used to specify a column in the `SELECT` clause or `WHERE` clause.

- When an alias is specified in a select expression

Correct coding example

```
REGISTER QUERY q1 SELECT s1.a AS a1 FROM s1[NOW];
REGISTER QUERY q2 SELECT a1 FROM q1 WHERE a1 > 1;
```

Incorrect coding example

```
REGISTER QUERY q1 SELECT s1.a AS a1 FROM s1[NOW];
REGISTER QUERY q3 SELECT q1.a FROM q1 WHERE q1.a > 1;
REGISTER QUERY q4 SELECT a FROM q1 WHERE a > 1;
```

Since `a1` is defined as an alias of `s1.a`, `s1.a` is no longer a valid range for the column name `a`. Consequently, `q1.a` and `a` cannot be used for column specification in the `SELECT` clause or `WHERE` clause.

3.2.6 Specifying constants

A constant is a data item whose value cannot be changed in a program. This subsection explains how to specify constants. For details about the types of data values that can be specified by constants, see 3.3 *CQL data types*.

(1) Constant types and notation

Constants include *numeric constants* for expressing numeric values and *character string constants* for expressing character strings.

The following types of numeric constants are available:

- Integer constant

- Floating point constant
- Decimal constant

The following types of character string constants are available:

- Character string
- Date constant
- Time constant
- Timestamp constant

When specifying date and time constants, the specified values must be entered according to the time zone of the JavaVM on which Stream Data Platform - AF is running.

The following table shows the notations for constants.

Table 3-4: Notations for constants

No.	Constant		Notation		Data type in the stream data processing engine
1	Numeric constant	Integer constant	<p>For INTEGER: <i>[sign]integer</i> For BIGINT: <i>[sign]integer</i> [L l]</p> <p>Examples: -123 45 6789L</p>	<p>For <i>integer</i>, specify a series of numbers. Specify + or - for the sign. The data type of a constant that ends with either character L or character l is BIGINT.</p>	INTEGER, BIGINT
		Floating point constant	<p>Decimal format: <i>[sign]integer-portion . decimal-portion</i></p> <p>Examples: -12.3 456.0 0.789</p>	<p>Integers are used to enter both the integer and decimal portions, separated by the decimal point (.). The integer portion, the decimal portion, and the decimal point (.) must always be entered. Specify + or - for the sign.</p>	DOUBLE

No.	Constant		Notation		Data type in the stream data processing engine
		Decimal constant	<p>Decimal format: <i>[sign]integer-portion [. decimal-portion]</i> {D d}</p> <p>Examples: -12.3D 456.0d 0.789d -123D 45d</p>	<p>Integers are used to enter both the integer and decimal portions, separated by the decimal point (.). Specify + or - for the sign. The integer portion and either the character D or the character d at the end must always be entered. The decimal portion and the decimal point (.) are optional.</p>	DECIMAL NUMERIC
2	Character string constant		<p><i>'character-string'</i></p> <p>Examples: 'HITACHI ' '98 ' 'DB0002 '</p>	<p>Specify a character string of single-byte characters or double-byte characters, enclosed in single quotation marks ('), for <i>character-string</i>[#]. To include a single quotation mark (') in the character string, use two consecutive single quotation marks (' ') to enter a single quotation mark. For example, to specify one single quotation mark in a character string, enter ' ' ' ' .</p> <p>For details about the specification formats for date data, time data, and timestamp data, see (2) <i>Character string showing dates</i>, (3) <i>Character string showing time</i>, and (4) <i>Character string showing timestamp data</i>.</p>	VARCHAR

#

Double quotation marks (") cannot be used to enclose a character string. Double quotation marks cannot be used in a character string either (this is because it is not possible to escape double quotation marks in a character string).

(2) Character string showing dates

Specify dates using the following format:

'YYYY-MM-DD'

YYYY

0001 to 9999 (Year)

MM

01 to 12 (Month)

DD

01 to the last day of the month specified by MM (Day)

To enter a date constant in a character string expression, enter the year (*YYYY*), month (*MM*), day (*DD*) separated by hyphens (-). If there are less digits than required for the year (*YYYY*), month (*MM*), or day (*DD*), enter leading zeros. Do not insert spaces between the numbers and the hyphen.

An example is shown below in which a character string showing a date is entered as a constant.

Specification example of March 25, 2010

```
'2010-03-25'
```

Note the following points when specifying dates:

- A character string that does not follow the format described above is not recognized as date data. It is treated as a character string constant.
- If an out-of-range value is specified for the year, month, or day, the incorrect value is carried forward and recognized as shown in the following example:

Example of a date when an out-of-range value is specified

Data entered: '2010-02-30'

Date is recognized as: March 2, 2010

If out-of-range values are specified, they may result in unexpected dates. Therefore, always specify values that are within the correct date range.

(3) Character string showing time

Specify time using the following format:

```
'hh:mm:ss'
```

hh

00 to 23 (hours)

mm

00 to 59 (minutes)

ss

00 to 59 (seconds)[#]

#

00 to 61 (seconds) if there is a leap second.

To enter a time constant in a character string expression, enter the hour (*hh*), minutes (*mm*), and seconds (*ss*) separated by colons (:). If there are fewer digits than required for the hour (*hh*), minutes (*mm*), or seconds (*ss*), enter leading zeros. Do not insert spaces between the numbers and the colon.

An example is shown below in which a character string showing time is entered as a constant.

Specification example of 10 pm, 8 minutes, 26 seconds

'22:08:26'

Note the following points when specifying time data:

- A character string that does not follow the format described above is not recognized as time data. It is treated as a character string constant.
- If an out-of-range value is specified for the hour, minutes, or seconds, the incorrect value is carried forward and recognized as shown in the following example:

Example when an out-of-range value is specified

Data entered: '25:08:26'

Time is recognized as: 1 am, 8 minutes, 26 seconds

(4) Character string showing timestamp data

Specify timestamp data in the following format:

'YYYY-MM-DD Δ hh:mm:ss [.SSSSSSSS]'

YYYY

0001 to 9999 (Year)

MM

01 to 12 (Month)

DD

01 to the last day of the month specified by MM (Day)

hh

00 to 23 (hours)

mm

00 to 59 (minutes)

ss

00 to 59 (seconds)[#]

#

00 to 61 (seconds) if there is a leap second.

SSSSSSSS

Fractional seconds of from 0 to 9 digits (*S*: 0 to 9)

To enter a timestamp constant in a character string expression, enter the year (*YYYY*), month (*MM*), day (*DD*) separated by hyphens (-), enter a single-byte space, and then enter the hour (*hh*), minutes (*mm*), and seconds (*ss*) separated by colons (:). If there are fewer digits than required for the year (*YYYY*), month (*MM*), day (*DD*), hour (*hh*), minutes (*mm*), or seconds (*ss*), enter leading zeros. Do not insert spaces between the numbers and the hyphen or between the numbers and the colon. Be sure to enter a single-byte space between the day (*DD*) and the hour (*hh*).

An example is shown below in which a character string showing a timestamp is entered as a constant.

Specification example of 10 pm, 8 minutes, 26 seconds, August 1, 2008

'2008-08-01 22:08:26'

Note the following points when specifying timestamp data:

- A character string that does not follow the format described above is not recognized as timestamp data. It is treated as a character string constant.
- To enter fractional-second data, specify the seconds (*ss*) and fractional seconds (*SSSSSSSS*) separated by a period. If the fractional-second data is omitted, it is assumed to be 0. If the fractional seconds have three, six, or nine digits, this means that time is specified in 1-millisecond, 1-microsecond, or 1-nanosecond units, respectively.
- If an out-of-range value is specified, the incorrect value is carried forward and recognized as shown in the following example:

Example when an out-of-range value is specified

Data entered: '2008-02-30 25:08:26'

Time is recognized as: 1 am, 8 minutes, 26 seconds, March 2, 2008

3.3 CQL data types

This section explains CQL data types.

3.3.1 Mapping between CQL data types and Java data types

CQL data types are mapped to Java data types.

The following table shows the mapping between CQL data types and Java data types.

Table 3-5: Mapping between CQL data types and Java data types

Class	CQL data type	Data format	Java data type	Data range	Remarks
Numeric data	INT [EGER]	Integer, 4 bytes	Integer class	-2,147,483,648 through 2,147,483,647	--
	SMALLINT	Integer, 2 bytes	Short class	-32,768 through 32,767	--
	TINYINT	Integer, 1 byte	Byte class	-128 through 127	--
	BIGINT	Integer, 8 bytes	Long class	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807	--
	DEC [IMAL] [' ('m') '] #1	Decimal format	java.math.BigDecimal class	$-10^{38}+1$ through $10^{38}-1$	Decimal number whose precision (the total number of digits) is m digits (not including the + or - sign). m is a positive integer in the range of $1 \leq m \leq 38$. If m is omitted, 15 is assumed.
	NUMERIC [' ('m') '] #1				
	REAL	Real number, 4 bytes	Float class	<ul style="list-style-type: none"> -3.402823466E+38 through -1.175494351E-38 0 1.175494351E-38 through 3.402823466E+38 	Data cannot be input using an exponential expression

Class	CQL data type	Data format	Java data type	Data range	Remarks
	FLOAT DOUBLE	Real number, 8 bytes	Double class	<ul style="list-style-type: none"> -1.7976931348623157E+308 through -2.2250738585072014E-308 0 2.2250738585072014E+308 through 1.7976931348623157E+308 	Data cannot be input using an exponential expression
Character data	CHAR [ACTE R] [' (' <i>n</i> ') ']	Fixed-length character string (with <i>n</i> characters)	java.lang .String class	1 to 255 characters	<i>n</i> is a positive integer in the range of $1 \leq n \leq 255$. If <i>n</i> is omitted, 1 is assumed. The necessary number of spaces are added to fill the string. If the data length exceeds the specified number of characters <i>n</i> , an error occurs.
	VARCHAR ' (' <i>n</i> ') ' '	Variable-length character string (with the maximum of <i>n</i> characters)		1 to 32,767 characters	<i>n</i> is a positive integer in the range of $1 \leq n \leq 32,767$. Spaces for filling the string are not added. If the data length exceeds the specified number of characters <i>n</i> , an error occurs.
Date data	DATE ^{#2}	Date (year-month-day)	java.sql .Date class	YYYYMMDD YYYY: 0001 to 9999 (Year) MM: 01 to 12 (Month) DD: 01 to the last day of the applicable month (Day)	--

Class	CQL data type	Data format	Java data type	Data range	Remarks
Time data	TIME ^{#2}	Time (hour-minutes-seconds)	java.sql. Time class	<i>hhmmss</i> <i>hh</i> : 00 to 23 (hour) <i>mm</i> : 00 to 59 (minutes) <i>ss</i> : 00 to 59 (seconds)	--
Timestamp data	TIMESTAMP ['(' <i>p</i> ')'] ^{#2}	Date and time (year-month-day + hour-minutes-seconds + nanoseconds)	java.sql. Timestamp class	<i>YYYYMMDDhhmmss</i> [<i>nn....n</i>] <i>YYYY</i> : 0001 to 9999 (Year) <i>MM</i> : 01 to 12 (Month) <i>DD</i> : 01 to the last day of the applicable month (Day) <i>hh</i> : 00 to 23 (hour) <i>mm</i> : 00 to 59 (minutes) <i>ss</i> : 00 to 59 (seconds) <i>nn....n</i> : <i>p</i> -digit fractional seconds (<i>n</i> : 0 to 9)	<i>p</i> is a positive integer. <i>n</i> is in the range of $0 \leq n \leq 9$. If <i>p</i> is omitted, 3 is assumed. If the number of digits in <i>p</i> is 3, 6, or 9, time is being specified in 1-millisecond, 1-microsecond, or 1-nanosecond units, respectively. If the specified data exceeds <i>p</i> , an error occurs.

Legend:

Class: Classification

--: Not applicable

#1

For notes related to the DECIMAL and NUMERIC types, see 3.3.2 *Notes on the DECIMAL and NUMERIC types*.

#2

For data of the DATE, TIME, and TIMESTAMP types, be careful not to specify out-of-range values. If out-of-range values are specified for these data types, no error occurs, but there is a risk that an unintended date might execute an undesired

process.

Reference note:

Mapping between CQL data types and Java data types follows the JDBC API, which defines the mapping between SQL data types and Java data types.

3.3.2 Notes on the DECIMAL and NUMERIC types

If the data for the DECIMAL or NUMERIC types exceeds the specified number of digits, an error occurs.

Also, if the processes described below are performed and the number of digits in the result exceeds the number of digits specified by the `query.decimalMaxPrecision` property, the result is rounded to the specified number of digits (the values are not rounded if these processes are not executed).

- If an operation is performed that includes the DECIMAL type (NUMERIC type) in the operand of the four arithmetic operations (binomial operations) or the AVG argument of the aggregate function SUM
- If data that is not in the DECIMAL type (NUMERIC type) is cast (including implicit casting) to the DECIMAL type (NUMERIC type)

How the value will be rounded and expressed is specified by the `query.decimalRoundingMode` property. For details about properties, see the *uCosminexus Stream Data Platform - Application Framework Setup and Operation Guide*.

The following table shows examples of how values are rounded. In these examples, the `query.decimalMaxPrecision` property is set to 1 (rounding the number of digits to 1).

Table 3-6: Value rounding examples (when `query.decimalMaxPrecision=1`)

Operation result (number of digits: 2)	Value specified in the <code>query.decimalRoundingMode</code> property						
	UP	DOWN	CEILING	FLOOR	HALF_UP	HALF_DOWN	HALF_EVEN
5.5	6	5	6	5	6	5	6
2.5	3	2	3	2	3	2	2
1.6	2	1	2	1	2	2	2
1.1	2	1	2	1	1	1	1
1.0	1	1	1	1	1	1	1
-1.0	-1	-1	-1	-1	-1	-1	-1

Operation result (number of digits: 2)	Value specified in the <code>query.decimalRoundingMode</code> property						
	UP	DOWN	CEILING	FLOOR	HALF_UP	HALF_DOWN	HALF_EVEN
-1.1	-2	-1	-1	-2	-1	-1	-1
-1.6	-2	-1	-1	-2	-2	-2	-2
-2.5	-3	-2	-2	-3	-3	-2	-2
-5.5	-6	-5	-5	-6	-6	-5	-6

For a numeric constant (decimal constant) in the `DECIMAL` type (`NUMERIC` type), the number of digits is not checked. If the value is out of range, it is shown as a value described in CQL, regardless of how the `query.decimalMaxPrecision` or `query.decimalRoundingMode` properties are specified.

The following table shows how to specify a digit count for transmitted data, operation results, and numeric constants in the `DECIMAL` type (`NUMERIC` type), and what happens if the specified number of digits is exceeded.

Table 3-7: How to specify a digit count in the `DECIMAL` type (`NUMERIC` type) and the action taken if the specified number of digits is exceeded

Type		Digit count specification method	Action if the specified number of digits is exceeded
Transmitted data		Type name of schema specification character string	If the digit count is exceeded during data transmission, an error occurs.
Operation result	<ul style="list-style-type: none"> Four arithmetic operations Aggregate function <code>SUM</code>/<code>AVG</code> Casting 	<code>query.decimalMaxPrecision</code> property	The method specified by the <code>query.decimalRoundingMode</code> property is used to round the digit count so that it matches the digit count specified by the <code>query.decimalMaxPrecision</code> property.
	Others		The operation result is displayed as is (the value is not rounded).
Numeric constant (decimal constant)		No specification	Shown using the value as it is in CQL.

3.4 Data comparison

This section explains how data is compared in a query.

3.4.1 Combinations of data types that can be compared

The following table shows the combinations of data types that can be compared in the search condition specified in the `WHERE` or `HAVING` clause. For details about the `WHERE` clause, see 4.4.6 *WHERE clause*. For details about the `HAVING` clause, see 4.4.8 *HAVING clause*.

The comparison rules explained here also apply to the `GROUP BY` clause and aggregate functions.

Table 3-8: Combinations of data types that can be compared

No.	Data type	Numeric data	Character data	Date data	Time data	Timestamp data
1	Numeric data	Y	D	N	N	N
2	Character data	D	Y	D	D	D
3	Date data	N	D	Y	N	N
4	Time data	N	D	N	Y	N
5	Timestamp data	N	D	N	N	Y

Legend:

Y: Can be compared.

D: Can be compared depending on the condition. For details about the cases that can be compared, see 3.4.2(2) *Comparison of character data*.

N: Cannot be compared.

3.4.2 Notes on comparing data

This subsection provides more information on comparing data.

(1) Data type general comparison

In a search specified by the `GROUP BY` clause, lines in which all strings are the same are treated as in the same group.

(2) Comparison of character data

Comparison of character data follows the rules for comparing `String` type data in Java.

Character data can only be compared with date, time, or timestamp data if the character string constant conforms to the date, time, or timestamp data format.

Numeric data from a WHERE clause can only be compared with character data if the character data is explicitly cast to the numeric data type and if the character data format conforms to the comparison-target numeric data type.

Hint:

About casting of character data

Besides character data that appears in comparisons, character data that appears in a value expression can also be cast.

The following table shows the formats of character data that can be compared with numeric data after the character data is cast.

Table 3-9: Formats of character data that can be compared with numeric data after being cast

Numeric data type	Cast specification	Character data format	
		Rules	Example
Integer type	(TINYINT) (SMALLINT) (INT [EGER]) (BIGINT)	<ul style="list-style-type: none"> Specifies only the numbers 0 to 9. A period must not be included. The value must fit in the numeric data range. For a positive value, the plus sign (+) cannot be specified. 	<p>Examples in which comparison can be made</p> <ul style="list-style-type: none"> '12' '-1' (The minus sign is coded.) <p>Examples in which comparison causes an error</p> <ul style="list-style-type: none"> '12.0' (A period is included.) '1b3' (A non-numeric value is included.) '+1' (The plus sign is coded.)
Real number and decimal number types	(REAL) (FLOAT) (DOUBLE) (DEC [IMAL])	<ul style="list-style-type: none"> Specifies only the numbers 0 to 9. A period may be included. The value must fit in the numeric data range. The plus sign (+) or minus sign (-) can be specified for the value. 	<p>Examples in which comparison can be made</p> <ul style="list-style-type: none"> '12' '12.0' (A period is included.) '-1' (The minus sign is coded.) '+1' (The plus sign is coded.) <p>Example in which comparison causes an error</p> <ul style="list-style-type: none"> '1b3.0' (A non-numeric value is included.)

Note: The addition of the plus or minus sign follows the implementation method for the `java.lang.Number` class.

For details about the formats with casting specified, see *4.4.19 Value expression*.

(3) Comparison of numeric data

If the data types to be compared are different, the data type with the smaller range is implicitly cast to the data type with the larger range, and the two are then compared.

The hierarchy of range sizes is as follows:

DECIMAL = NUMERIC > FLOAT = DOUBLE > REAL > BIGINT > INTEGER > SMALLINT > TINYINT

(4) Comparison of date, time, and timestamp data

Comparison of date, time, and timestamp data follows the rules for comparing the Java `java.sql.Date` class, `java.sql.Time` class, and `java.sql.Timestamp` classes.

3.5 Notes on query definitions and limit values

This section provides more information on defining queries, and the limit values applicable to query definitions.

3.5.1 Notes on query definitions

- Define the query names and stream names so that they are unique for the specific SDP server. If you try to define a name that is already defined, an error occurs when you try to enter the query group.
- A stream or query defined in a query definition file cannot be referenced from a query defined in another query definition file.

3.5.2 Limit values applicable to query definitions

The following table shows the limit values applicable to query definitions.

Table 3-10: Limit values applicable to query definitions

No.	Item		Value range
1	Number of CQL commands that can be specified in a single input file		1 to 1,024
2	Length of a CQL command		1 to 300,000 characters
3	Number of strings inside a stream		1 to 3,000
4	Length of each stream data item (size of a line)		1 to 2,147,483,647 bytes
5	Name size (stream name, relation name, query name, column name, and alias)		1 to 100 characters
6	Number of search items (number of select expressions in the <code>SELECT</code> clause)		1 to 3,000
7	Column specification list		1 to 255
8	Number of comparison predicates and parentheses inside a search condition		1 to 255
9	The total number of streams and relations that can be specified in a CQL query		1 to 64
10	Number of lines that can exist in the <code>ROWS</code> window		1 to 100,000
11	Time specification	SECOND	1 to 2,678,400
12		MILLISECOND	1 to 86,400,000
13		MINUTE	1 to 44,640

No.	Item		Value range
14		HOUR	1 to 744
15		DAY	1 to 31

Chapter

4. CQL Reference

This chapter explains the CQL syntax.

- 4.1 Format used for explaining the CQL syntax
- 4.2 CQL list
- 4.3 Definition CQL
- 4.4 Data manipulation CQL

4.1 Format used for explaining the CQL syntax

This section describes the format used for explaining the CQL syntax.

The items explained for each CQL are listed below. Note that not all of these items are used in every explanation, depending on the CQL statement. For details about the symbols used in the CQL syntax, see *3.1.3 Symbols used in the explanation of CQL syntax*.

(1) Format

Explains the format.

(2) Function

Explains the function.

(3) Operands

Explains the item that can be specified for each operand and the cases in which the operand needs to be specified.

(4) Syntax rules

Explains the syntax rules.

(5) Notes

Explains other facts you need to know.

(6) Usage example

Provides usage examples.

4.2 CQL list

There are two types of CQL: definition CQL for defining streams and queries, and data manipulation CQL that is used in the `REGISTER QUERY` and `REGISTER QUERY_ATTRIBUTE` clauses of definition CQL.

The following table lists the definition CQL.

Table 4-1: Definition CQL list

No.	CQL	Explanation
1	<i>REGISTER STREAM clause</i>	Defines a stream
2	<i>REGISTER QUERY clause</i>	Defines a query.
3	<i>REGISTER QUERY_ATTRIBUTE clause</i>	Specifies time division for a query. The query for which time division is to be used must be defined following this clause.

The following table lists the data manipulation CQL.

Table 4-2: Data manipulation CQL list

No.	Type	Explanation
1	<i>Inquiry</i>	Searches data in a relation or the stream defined in the <code>REGISTER STREAM</code> clause.
2	<i>Stream clause</i>	Converts data to be output into a stream.
3	<i>Relation expression</i>	Searches data in one or more relations and filters the results of the search.
4	<i>SELECT clause</i>	Specifies how the searched result (select expression) is to be output and acquires the search result as a relation.
5	<i>FROM clause</i>	Specifies one or more relations (relation references). Relations acquired by the <code>FROM</code> clause becomes the target of the <code>WHERE</code> or <code>HAVING</code> clauses.
6	<i>WHERE clause</i>	Specifies a search condition for a relation acquired by the <code>FROM</code> clause.
7	<i>GROUP BY clause</i>	Specifies the columns (grouped columns) in the relation acquired by the preceding clause. Grouping is performed on the specified grouped columns only.
8	<i>HAVING clause</i>	Specifies a search condition for a relation acquired by the <code>FROM</code> clause, <code>WHERE</code> clause, or <code>GROUP BY</code> clause. A search condition specified in the <code>HAVING</code> clause executes a logical operation, and only the true results are acquired as a relation.

4. CQL Reference

No.	Type	Explanation
9	<i>UNION clause</i>	Links together multiple <code>SELECT</code> clauses and executes them as a single CQL statement.
10	<i>Selection list</i>	Specifies one or more select expressions.
11	<i>Select expression</i>	Specifies the item to be output from a search result.
12	<i>Aggregate functions</i>	Calculates a value from multiple rows.
13	<i>Column specification list</i>	Specifies one or more columns.
14	<i>Relation reference</i>	Specifies the relation to be searched. Relation reference is specified using the <code>FROM</code> clause.
15	<i>Window specification</i>	Specifies how long stream data stays in a relation. Window specification is specified using a relation reference.
16	<i>Time specification</i>	Specifies a time unit.
17	<i>Search condition</i>	Executes a logical operation using the specified condition and acquires only true results as a new relation. A search condition is specified using the <code>WHERE</code> or <code>HAVING</code> clauses.
18	<i>Comparison predicate</i>	Specifies the condition for determining a true or false logical value.
19	<i>Value expression</i>	Specifies a value.
20	<i>Constant</i>	Specifies a constant.

4.3 Definition CQL

This section explains definition CQL.

4.3.1 REGISTER STREAM clause (stream definition)

(1) Format

```
REGISTER STREAM ::= REGISTER ▲ STREAM ▲ stream-name
                  ▲ schema-specification-character-strings
```

```
schema-specification-character-strings ::= ' ( ' column-name ▲ type-name [ , column-
name ▲ type-name ] . . . ' ) '
```

(2) Function

Defines a stream.

This clause registers a stream to the system catalog areas that store stream management tables, and causes the system to begin accepting the stream.

(3) Operands

stream-name

For a stream (input stream), specify any name that is unique within the stream data processing system. For details about specifying names, see 3.2.4 *Specifying names*.

schema-specification-character-strings

For the stream defined by the stream name, specify column names and their type names.

column-name

Specify any column name that is unique within the stream data processing system. For details about specifying names, see 3.2.4 *Specifying names*.

The number of column names that can be specified in the range from 1 to 3,000.

type-name

For details about type names, see 3.3 *CQL data types*.

(4) Syntax rules

Specify all schema specification character strings in a single definition. For example, when defining a stream named `s1`, specify all of the necessary schema specification character strings as shown in Example 1.

Example 1 (correct specification example):

```
REGISTER STREAM s1(id INT, name VARCHAR(10));
```

When defining a stream named `s1`, if schema specification character strings are specified separately as shown in Example 2, two streams having the same name end up being specified, resulting in an error.

Example 2 (incorrect specification example):

```
REGISTER STREAM s1 (id INT);
REGISTER STREAM s1 (name VARCHAR(10));
```

(5) Notes

None.

(6) Usage example

As a schema, a stream named `s1`, having a column named `id` of type `INT` and a column named `name` consisting of character data of one to ten characters in length, is defined in the system catalog.

```
REGISTER STREAM s1(id INT, name VARCHAR(10));
```

4.3.2 REGISTER QUERY clause (query definition)

(1) Format

```
REGISTER QUERY ::= REGISTER ▲ QUERY ▲ query-name ▲ inquiry
```

(2) Function

Defines a query.

This clause registers a query to the query management table (part of the query repository), and begins query processing.

(3) Operands

query-name

Specify any query name that is unique within the stream data processing system. For details about specifying names, see 3.2.4 *Specifying names*.

The name specified here becomes the name of the stream returned by an inquiry (output stream) or a relation.

inquiry

Defines an inquiry. For details about definitions, see 4.4.1 *Inquiry*.

(4) Syntax rules

Specify all inquiries for a single query in a single definition. For example, when defining a query `q1`, specify all of the necessary inquiries as shown in Example 1.

Example 1 (correct specification example):

```
REGISTER QUERY q1 SELECT s1.a, s1.b FROM s1[ROWS 10];
```

If the inquiries are divided and specified for the query `q1`, as shown in Example 2, two queries with the same name end up being specified, resulting in an error.

Example 2 (incorrect specification examples)

```
REGISTER QUERY q1 SELECT s1.a FROM s1[ROWS 10];
REGISTER QUERY q1 SELECT s1.b FROM s1[ROWS 10];
```

(5) Notes

None.

(6) Usage example

Defines a query `q1` that outputs data from column `a` of stream `s1`.

```
REGISTER QUERY q1 SELECT s1.a FROM s1[ROWS 10];
```

4.3.3 REGISTER QUERY_ATTRIBUTE clause (time division specification)

(1) Format

```
REGISTER QUERY_ATTRIBUTE ::= REGISTER ▲ QUERY_ATTRIBUTE ▲ query-name
                                ▲ STREAM_NAME=data-identifier
                                ▲ PERIOD=mesh-interval
                                ▲ TARGETS=aggregate-function-1 '(' column-name-1 ') ' ,
                                . . . , aggregate-function-N '(' column-name-N ') ' ;
```

(2) Function

Specifies time division for a query.

(3) Operands

query-name

Specify a name that is unique in the stream data processing system for the query that is the target of the time division specification immediately following this clause. For details about specifying names, see 3.2.4 *Specifying names*.

The query name must not be the same as the name specified in another `REGISTER QUERY_ATTRIBUTE` clause. Also, you must specify the `RANGE` window in the `REGISTER QUERY` clause that defines the query that is the target of time division.

data-identifier

Specify the stream name or query name where the column name that is the target of time division is defined. The stream name or query name must be defined before the `REGISTER QUERY_ATTRIBUTE` clause.

mesh-interval

Specify a mesh interval in milliseconds. The unit `ms` must be entered.

You cannot specify a value of less than 100 milliseconds. If the mesh interval exceeds the `RANGE` window interval specified in the query name that is the target for time division, the mesh interval is set to the `RANGE` window interval.

For example, if the mesh interval is 1,000 milliseconds and the `RANGE` window interval is 500 milliseconds, the mesh interval is set to 500 milliseconds. If the mesh interval is smaller than the `RANGE` window interval but is greater than 86,400,000 milliseconds, the mesh interval is set to 86,400,000 milliseconds.

If you specify a value of less than 100 milliseconds, the mesh interval is set to 100 milliseconds.

aggregate-function

Specify an aggregate function. The following table shows the aggregate functions that can be specified here.

Table 4-3: Aggregate functions that can be specified

No.	Aggregate function	Meaning
1	SUM	Used for calculating the sum total.
2	MAX	Used for calculating the maximum value.
3	MIN	Used for calculating the minimum value.

The number of aggregate functions that can be specified is in the range from 1 to 256.

column-name

Specify the column name that the data identifier specified in `STREAM_NAME` has.

Each column name must be unique.

The number of column names that can be specified is in the range from 1 to 256.

(4) Syntax rules

None.

(5) Notes

- In the `REGISTER QUERY_ATTRIBUTE` clause, the query name, the data identifier, and the column name must each be unique.
- The stream name or query name specified in `STREAM_NAME` must be defined before the `REGISTER QUERY_ATTRIBUTE` clause or must be pre-registered.
- The column names specified as the arguments of the aggregate functions specified in `TARGETS` must be column names that are included in the data identifier.
- Special characters (such as a semicolon and period) must not be included in the

query name, data identifier, mesh interval, aggregate function, or column name.

- REGISTER, QUERY_ATTRIBUTE, STREAM_NAME, PERIOD, and TARGETS must not be specified as the query name, data identifier, or column name.
- The definition of a query (the REGISTER QUERY clause) that is the target of time division must satisfy the following conditions:
 - Multiple streams are not specified in the FROM clause.
 - The WHERE clause is not specified.
 - The GROUP BY clause is specified.
 - It must be defined immediately after the REGISTER QUERY_ATTRIBUTE clause.
- The column names specified after TARGETS in the REGISTER QUERY_ATTRIBUTE clause must have a one-to-one correspondence to the column names in the data identifier specified in STREAM_NAME. The column names in a query that is the target of time division must also have a one-to-one correspondence to the column names in the data identifier specified in STREAM_NAME.

An example is shown below in which time division is specified for query q1. The underline indicates the areas referred to in the above note.

Correct specification example:

```
REGISTER STREAM stock(price INTEGER, name VARCHAR(10));
REGISTER QUERY q0
ISTREAM(
SELECT name, price AS price0, price AS price1
FROM stock[NOW]
);
REGISTER QUERY_ATTRIBUTE q1
STREAM_NAME=q0 PERIOD=300ms
TARGETS=SUM(price0),MAX(price1);
REGISTER QUERY q1
ISTREAM(
SELECT q0.name, SUM(q0.price0) AS sum_price, MAX(q0.price1)
AS max_price
FROM q0[RANGE 1 MINUTE]
GROUP BY q0.name
);
```

The column names (price0 and price1) specified in REGISTER QUERY q0, REGISTER QUERY_ATTRIBUTE q1, and REGISTER QUERY q1 all correspond to one another. When specifying time division, the column names must correspond to one another as shown in the above example.

Incorrect specification example:

```
REGISTER STREAM stock(price INTEGER, name VARCHAR(10));
REGISTER QUERY q0
ISTREAM(
SELECT name, price AS price0, price AS price1, price AS
price2, price AS price3
FROM stock[NOW]
);
REGISTER QUERY ATTRIBUTE q1
STREAM_NAME=q0 PERIOD=300ms
TARGETS=SUM(price0),MAX(price1),MIN(price2);
REGISTER QUERY q1
ISTREAM(
SELECT q0.name, SUM(q0.price0) AS sum_price, MAX(q0.price1)
AS max_price
FROM q0[RANGE 1 MINUTE]
GROUP BY q0.name
);
```

The column names (price0, price1, price2, and price3) specified in REGISTER QUERY q0, the column names (price0, price1, and price2) specified in REGISTER QUERY_ATTRIBUTE q1, and the column names (price0 and price1) specified in REGISTER QUERY q1 do not correspond to one another, resulting in an error.

- In a query for which time division is specified, tuples are grouped into pseudo tuples for each mesh interval. This changes the frequency at which summary results are output. For example, if SUM is specified to calculate a sum total, a result is generated for each group of pseudo tuples, and so the frequency at which results are output is different from that when a result is generated for each input tuple at each moment in time.

(6) Usage examples

The underline indicates the key parts of the examples.

Example 1:

Determining the sum total of the column price in the stream stock, which is found in a tuple in the RANGE window:

```
REGISTER STREAM stock(price INTEGER, name VARCHAR(10));
REGISTER QUERY ATTRIBUTE q1 STREAM_NAME=stock PERIOD=300ms
TARGETS=SUM(price);
REGISTER QUERY q1
ISTREAM(
SELECT name, SUM (price) AS s1
FROM stock[RANGE 1 MINUTE]
GROUP BY name
);
```


This specifies time division for the query `q1` specified in the `REGISTER QUERY_ATTRIBUTE` clause. To obtain the sum total of the column `price` in the stream `stock` defined in the `REGISTER STREAM` clause, `stock` is specified for the data identifier and `SUM(price)` is specified for `TARGETS`. To obtain the maximum or minimum value instead of the total value of `price`, specify `MAX` or `MIN`.

Example 2:

Determining the sum total, the maximum value, and the minimum value of the column `price` in the stream `stock`, which is found in a tuple in the `RANGE` window:

```
REGISTER STREAM stock(price INTEGER, name VARCHAR(10));
REGISTER QUERY q0
ISTREAM(
SELECT name, price AS price0, price AS price1, price AS price2
FROM stock[NOW]
);
REGISTER QUERY_ATTRIBUTE q1 STREAM NAME=q0 PERIOD=300ms
TARGETS=SUM(price0),MAX(price1),MIN(price2);
REGISTER QUERY q1
ISTREAM(
SELECT q0.name, SUM(q0.price0) AS sum_price,
MAX(q0.price1) AS max_price, MIN(q0.price2) AS min_price
FROM q0[RANGE 1 MINUTE]
GROUP BY q0.name
);
```

Note that, to perform more than one aggregate function on the column `price` of the data identifier `stock` in the query, you cannot specify the same column name `price` more than once in `TARGETS`. That is, you cannot specify `TARGETS=SUM(price), MAX(price), MIN(price)`.

In this case, the column `price` must be redefined separately for the sum total, the maximum value, and the minimum value. In this example, a data identifier (`q0` here) is defined that has columns `price0`, `price1`, and `price2`, which are obtained by cloning the column `price` using a CQL statement. Using the column names defined in `q0` when specifying the operations in `q1` allows you to perform multiple aggregate functions on the same data.

4.4 Data manipulation CQL

This section explains data manipulation CQL.

4.4.1 Inquiry

(1) Format

inquiry ::= { *relation-expression* | *stream-clause* ' (' *relation-expression* ') ' }

(2) Function

Searches the data in a relation or the stream defined in the REGISTER STREAM clause.

(3) Operands

relation-expression

For details about specifying a relation expression, see 4.4.3 *Relation expression*.

If you specify a relation expression without specifying a stream clause, the result of the inquiry is acquired as a relation.

stream-clause

To specify a stream clause, enter a relation expression enclosed in parentheses (()) following the stream clause. For details about specifying a stream clause, see 4.4.2 *Stream clause*.

When a stream clause is specified, the result of the inquiry is acquired as a stream.

(4) Syntax rules

None.

(5) Notes

None.

(6) Usage example

Specifies a relation expression that generates a relation from the data in column *a* of stream *s1*. The generated relation is output as the relation *q1*. The underlined part indicates the inquiry.

```
REGISTER QUERY q1 SELECT s1.a FROM s1[ROWS 10];
```

4.4.2 Stream clause

(1) Format

stream-clause ::= { ISTREAM | DSTREAM
| RSTREAM [' [' *integer-constant* [▲ *time-specification*] '] '] }

(2) Function

Converts the data to be output into a stream.

(3) Operands**ISTREAM**

Outputs only the items added to the output relation.

DSTREAM

Outputs only the items deleted from the output relation.

RSTREAM

Outputs all of the items in the output relation at regular time intervals.

integer-constant

Specifies the time interval. For the value range of an integer constant, see *4.4.16 Time specification*.

If you do not specify a constant, 1 second is assumed.

time-specification

Specifies a unit for the time interval specified in *integer-constant*. For details about specifying time, see *4.4.16 Time specification*.

If you do not specify this, the value specified in *integer-constant* is assumed to be in seconds.

(4) Syntax rules

Enter the details of the stream clause to be defined in the inquiry. Specify a relation expression, enclosed in parentheses (()) following the stream clause.

(5) Notes

None.

(6) Usage example

Generates a stream that has the time stamps of the newly added data in relation `s1`. The generated stream is output as the stream `q1`. The underlined part indicates the stream clause.

```
REGISTER QUERY q1 ISTREAM (SELECT * FROM s1[ROWS 100]);
```

4.4.3 Relation expression**(1) Format**

relation-expression : := *SELECT-clause* **▲** *FROM-clause* [**▲** *WHERE-clause*]
[**▲** *GROUP-BY-clause* [**▲** *HAVING-clause*]] [**▲** *UNION-clause*]

(2) *Function*

Searches the data from one or more relations and filters the search results.

(3) *Operands*

SELECT-clause

For details about specifying the `SELECT` clause, see 4.4.4 *SELECT clause*.

FROM-clause

For details about specifying the `FROM` clause, see 4.4.5 *FROM clause*.

WHERE-clause

For details about specifying the `WHERE` clause, see 4.4.6 *WHERE clause*.

GROUP-BY-clause

For details about specifying the `GROUP BY` clause, see 4.4.7 *GROUP BY clause*.

HAVING-clause

For details about specifying the `HAVING` clause, see 4.4.8 *HAVING clause*.

UNION-clause

For details about specifying the `UNION` clause, see 4.4.9 *UNION clause*.

(4) *Syntax rules*

Enter the details of the relation expression in an inquiry. Specify a search method and a filtering method after the `SELECT` clause and the `FROM` clause, respectively.

(5) *Notes*

None.

(6) *Usage example*

Outputs the data in column `a` where the value of column `b` in relation `s1` is less than 20. The underlined part indicates the relation expression.

```
REGISTER QUERY q1 SELECT s1.a FROM s1[ROWS 100] WHERE s1.b < 20;
```

4.4.4 **SELECT clause**

(1) *Format*

SELECT-clause ::= `SELECT` ▲ { *selection-list* | * }

(2) *Function*

Specifies the items to be output for the searched result (select expression) and retrieves the select expression as a relation.

(3) Operands**selection-list**

For details about specifying a selection list, see *4.4.10 Selection list*.

Outputs all columns of the relation.

This returns all columns in the order specified in the `FROM` clause of the relation expression. The column order in each relation is the order specified in the schema specification character strings when the stream was defined, or in the selection list when the query was defined.

(4) Syntax rules

None.

(5) Notes

None.

(6) Usage example

Outputs all data of relation `s1`. The underlined part indicates the `SELECT` clause.

```
REGISTER QUERY q1 SELECT * FROM s1 [ROWS 10] ;
```

4.4.5 FROM clause**(1) Format**

FROM-clause : := `FROM` **▲** *relation-reference* [, *relation-reference*] . . .

(2) Function

Specifies one or more relations (relation references). The relation that is retrieved by the `FROM` clause becomes the target of the `WHERE` or `HAVING` clauses.

If no `WHERE` or `HAVING` clauses are specified, the relation acquired by the `FROM` clause becomes the target of the `SELECT` clause.

If multiple relations are specified, the relation returned by the `FROM` clause is formed by connecting the rows from each relation in the order in which they are defined. The total number of rows is the product of the number of rows in the individual relations.

(3) Operands**relation-reference**

For details about specifying a relation reference, see *4.4.14 Relation reference*.

The number of relation references that can be specified is in the range from 1 to 64.

(4) Syntax rules

None.

(5) Notes

The relation returned by the `FROM` clause is a single group without grouped columns.

(6) Usage example

Specifies relation `s1` is to be returned. The underlined part indicates the `FROM` clause.

```
REGISTER QUERY q1 SELECT * FROM s1 [ROWS 100];
```

4.4.6 WHERE clause**(1) Format**

WHERE-clause : := `WHERE` **▲** *search-condition*

(2) Function

Specifies a search condition for the relation returned by the `FROM` clause.

If the `WHERE` clause is specified without the `HAVING` clause, the relation returned by the `WHERE` clause becomes the target of the `SELECT` clause.

If there is no `WHERE` clause, all rows in the relation returned by the `FROM` clause become the targets of the `SELECT` clause.

(3) Operands**search-condition**

For details about specifying a search condition, see *4.4.17 Search condition*.

If the preceding `FROM` clause contains only one relation reference, only a column name needs to be specified for the search condition. If the `FROM` clause contains multiple relation references, you must specify a data identifier and a column name for each column specified in the search condition to make the column names unique.

(4) Syntax rules

If τ is used to denote the relation returned by the preceding `FROM` clause, the search condition applies to each row in τ . The relation returned by the `WHERE` clause consists of a subset of the rows in τ for which the search condition is true.

(5) Notes

The relation returned by the `WHERE` clause is a single group without grouped columns.

(6) Usage example

Outputs the data in columns `a` and `b` if the value of column `b` in relation `s1` is greater than 10. The underlined part indicates the `WHERE` clause.

```
REGISTER QUERY q1 SELECT s1.a, s1.b FROM s1 [ROWS 100] WHERE s1.b
```

> 10;

4.4.7 GROUP BY clause

(1) Format

GROUP-BY-clause ::= GROUP ▲ BY ▲ *column-specification-list*

(2) Function

Specifies the columns (grouped columns) in the relation returned by the previously specified clause. Grouping is performed on the specified grouped columns only.

In grouping, rows with the same value in the column specified by the GROUP BY clause are grouped together and output as a row.

(3) Operands

column-specification-list

For details about specifying a column specification list, see *4.4.13 Column specification list*.

When specifying columns in the column specification list, you cannot specify the same column name more than once. If the preceding FROM clause references only one relation, you only need to specify column names in the column specification list. If the FROM clause references multiple relations, you must specify a data identifier and a column name for each column in the column specification list to make the column names unique.

(4) Syntax rules

None.

(5) Notes

If the GROUP BY clause is specified in a relation expression, the only columns you can specify in the select expression in the SELECT clause are elements (grouped column names) specified by the GROUP BY clause, or a value expression that starts with one of these column names.

(6) Usage example

Groups columns b and c of relation s1, sums the values in column a, and outputs the data in columns b and c. The underlined part indicates the GROUP BY clause.

```
REGISTER QUERY q1 SELECT SUM(s1.a) AS sum_a, s1.b, s1.c
                        FROM s1[ROWS 100] GROUP BY s1.b,s1.c;
```

4.4.8 HAVING clause

(1) Format

HAVING-clause ::= HAVING ▲ *search-condition*

(2) Function

Specifies a search condition for a relation to be returned by the `FROM` clause, `WHERE` clause, or `GROUP BY` clause. The `HAVING` clause causes a logical operation to be performed based on the specified search condition and only returns the data that meets the condition as a relation.

When the `HAVING` clause is specified, the relation returned by the `HAVING` clause becomes the target of the `SELECT` clause.

(3) Operands**search-condition**

For details about specifying a search condition, see *4.4.17 Search condition*.

To use a column specification in a search condition, specify either grouped columns or specify a column as an argument of an aggregate function. For details about grouped columns, see *4.4.7 GROUP BY clause*, and for details about aggregate functions, see *4.4.12 Aggregate functions*.

(4) Syntax rules

Selects groups for which the search condition specified in the `HAVING` clause is true.

(5) Notes

If you omit the `HAVING` clause, all groups in the relation resulting from the preceding `GROUP BY` or `FROM` clause are selected.

(6) Usage example

Groups columns `b` and `c` of relation `s1` and outputs the data in columns `a`, `b`, and `c` when the average value of column `a` is greater than 10. The underlined part indicates the `HAVING` clause.

```
REGISTER QUERY q1 SELECT AVG(s1.a) AS a1,s1.b,s1.c FROM s1 [ROWS
100]
                        GROUP BY s1.b,s1.c HAVING AVG(s1.a) > 10;
```

4.4.9 UNION clause**(1) Format**

UNION-clause ::= `UNION` ▲ [`ALL` ▲] *relation-expression*

(2) Function

Links together multiple `SELECT` clauses and executes them as a single CQL statement.

(3) Operands**ALL**

If `ALL` is specified, duplicate rows are allowed.

If UNION is specified without the ALL operand, duplicate rows are excluded.

Excluding duplicate rows means that, if duplicate rows (identical rows consisting of the items specified in the select expression) exist in the search result, they are deleted and only a single row is output.

relation-expression

For details about specifying a relation expression, see *4.4.3 Relation expression*.

(4) Syntax rules

A query specifying the following two conditions at the same time causes a syntax error:

- UNION ALL operand is used.
- In multiple SELECT clauses, only a single stream or relation is specified in the FROM clause.

The following example causes a syntax error. The underlined part indicates the location that causes the error.

```
REGISTER QUERY q1 ISTREAM(
  SELECT * FROM s1 [RANGE 3 SECOND]
  UNION ALL SELECT * FROM s1 [RANGE 5 SECOND] );
```

(5) Notes

- When using the UNION clause to link multiple SELECT clauses, use the same number, type, and number of character data characters in the selection lists of all SELECT clauses.
- For a relation returned by a relation expression that uses the UNION clause, the column names of the relation that is returned by the first relation expression specified (excluding the UNION clause) become the column names in the relation returned as the result of the inquiry.

For details, see *3.2.5(2) Column specification*.

- If the stream data specified in the FROM clauses is identical, you cannot use UNION ALL to link SELECT clauses that simply output all columns that are input as is.

The following example is incorrect. The underlined part indicates the applicable SELECT clause.

```
SELECT * FROM s1 [RANGE 3 SECOND] UNION ALL SELECT * FROM
s1 [RANGE 5 SECOND] ;
```

Since the same stream s1 is input into both SELECT clauses and * is specified for the operand, all columns that are input are output as is, resulting in an error.

(6) Usage example

Outputs all data in relations `s1` and `s2`. The underlined part indicates the UNION clause.

```
REGISTER QUERY s1 SELECT * FROM s1 [ROWS 100] UNION SELECT * FROM  
s2 [ROWS 100] ;
```

4.4.10 Selection list**(1) Format**

selection-list ::= *select-expression* [, *selection-list*]

(2) Function

Specifies one or more select expressions.

(3) Operand

select-expression

For details about specifying a select expression, see 4.4.11 *Select expression*.

The number of select expressions that can be specified is in the range from 1 to 3,000.

(4) Syntax rules

Specify a selection list following the `SELECT` clause.

(5) Notes

Only relation data acquired by the `FROM` clause can be specified in a selection list.

(6) Usage example

Outputs the data in columns `a` and `b` of relation `s1`. The underlined part indicates a selection list.

```
REGISTER QUERY q1 SELECT s1.a, s1.b FROM s1 [ROWS 100] ;
```

4.4.11 Select expression**(1) Format**

select-expression ::= { { *column-specification* | *column-name* } [▲ AS ▲ *alias*]
| { *value-expression* | *aggregate-function* } ▲ AS ▲ *alias* }

(2) Function

Specifies the item to be output as the result of a search.

(3) Operands

column-specification

For details about specifying columns, see 3.2.5(2) *Column specification*.

column-name

If there is only one relation reference in the FROM clause that follows, you can specify just a column name.

If the succeeding FROM clause references multiple relations, you must use a column specification.

alias

Specify a unique name in the following cases:

- If column names are duplicated
- If the name of column specification is long and complex
- If a value expression is neither a column name nor a column specification

The alias must follow the standard naming rules. For details about the naming rules, see *3.2.4 Specifying names*.

All alias names must be unique in any given SELECT clause. Also, you cannot specify a name that is the same as the table ID of another range variable. For details about range variables and table IDs, see *3.2.5 Name qualification*.

value-expression

For details about specifying a value expression, see *4.4.19 Value expression*.

If you specify a column name for the *i*-th select expression, that column name is used to reference the *i*-th column of the relation returned by the relation expression. If you do not specify a column name for the *i*-th select expression, the following occurs:

- If the value expression references a column specification, the column name specified in that column specification for the *i*-th column is used.
- If the value expression references another type of specification, you must specify an alias.

aggregate-function

For details about specifying an aggregate function, see *4.4.12 Aggregate functions*.

A column specification or column name must be included in the argument of an aggregate function.

(4) Syntax rules

- An alias specified for a select expression can only be used with the returned relation. It cannot be used by a clause that follows the select expression.

For example, alias *a1* and *a2*, defined in *q1* as shown below, can be used by *q2*.

Correct specification example:

```
REGISTER QUERY q1 SELECT s1.c1 AS a1, s2.c1 AS a2 FROM
```

```
s1[NOW], s2[NOW];
REGISTER QUERY q2 SELECT a1, a2 FROM q1 WHERE a1 > 1 GROUP
BY a1, a2
   HAVING a2 > 1;
```

Aliases a1 and a2, specified in the select expression as shown below, cannot be used by the succeeding clause.

Incorrect specification example:

```
REGISTER QUERY q1 SELECT s1.c1 AS a1, s2.c1 AS a2 FROM
s1[NOW], s2[NOW]
   WHERE a1 > 1 GROUP BY a1, a2 HAVING a2 > 1;
```

- When specifying an alias in a select expression, use a name that is different from the column names used by other select expressions.

Correct specification example:

```
REGISTER QUERY q1 SELECT s1.c1, s1.c2+1 AS a1 ... ;
```

In the following example, an error occurs because c1, which was already used in another select expression, is used:

Incorrect specification example:

```
REGISTER QUERY q1 SELECT s1.c1, s1.c2+1 AS c1 ... ;
```

(5) Notes

Only defined stream data can be used for specifying a select expression.

(6) Usage example

Outputs the data in columns a and b of relation s1. The underlined parts indicate select expressions.

```
REGISTER QUERY q1 SELECT s1.a, s1.b FROM s1[ROWS 100];
```

4.4.12 Aggregate functions

(1) Format

aggregate-function ::= { COUNT ' (' *value-expression* | * ') ' | *general-aggregate-function* }

general-aggregate-function ::= { MAX | MIN | SUM | AVG } ' (' *value-expression* ') '

(2) Function

Calculates a value using data from multiple rows.

(3) Operands**COUNT**

Counts the number rows that were input.

value-expression

Specifies an argument for the aggregate function. For details about value expressions, see *4.4.19 Value expression*.

*

Uses all of the rows in the relation.

general-aggregate-function

Specify MAX, MIN, SUM, or AVG. Specify a value expression that includes a column specification in the argument.

MAX

Aggregate function for determining the maximum value.

MIN

Aggregate function for determining the minimum value.

SUM

Aggregate function for determining the sum total.

AVG

Aggregate function for determining the average.

(4) Syntax rules

- For general aggregate functions, there is no argument modifier that specifies ALL (acquire all). All aggregate functions work as if ALL was specified.
- Specify an aggregate function in either the select expression or the HAVING clause of the relation expression that contains the aggregate function.
- The relation returned by the last FROM, WHERE, and GROUP BY clause that is processed is used as the input for the aggregate function. If a GROUP BY clause is not specified however, the result from the FROM or WHERE clause is treated as a group with no grouped columns.
- If you enter an aggregate function, column specification, or column name in the select expression of a SELECT clause of a relation expression, the GROUP BY clause is assumed.
- The following table shows the argument types (column data types) for aggregate functions and the corresponding result types.

Table 4-4: Aggregate function argument types and corresponding result types

No.	Argument type	Aggregate function				
		COUNT (argument)	MAX (argument)	MIN (argument)	SUM (argument)	AVG (argument)
1	INTEGER	INTEGER	INTEGER	INTEGER	INTEGER	INTEGER
2	SMALLINT	INTEGER	SMALLINT	SMALLINT	SMALLINT	SMALLINT
3	TINYINT	INTEGER	TINYINT	TINYINT	TINYINT	TINYINT
4	BIGINT	INTEGER	BIGINT	BIGINT	BIGINT	BIGINT
5	DECIMAL	INTEGER	DECIMAL	DECIMAL	DECIMAL	DECIMAL
6	NUMERIC	INTEGER	NUMERIC	NUMERIC	NUMERIC	NUMERIC
7	REAL	INTEGER	REAL	REAL	REAL	REAL
8	FLOAT	INTEGER	FLOAT	FLOAT	FLOAT	FLOAT
9	DOUBLE	INTEGER	DOUBLE	DOUBLE	DOUBLE	DOUBLE
10	CHAR	INTEGER	CHAR	CHAR	--	--
11	VARCHAR	INTEGER	VARCHAR	VARCHAR	--	--
12	DATE	INTEGER	DATE	DATE	--	--
13	TIME	INTEGER	TIME	TIME	--	--
14	TIMESTAMP	INTEGER	TIMESTAMP	TIMESTAMP	--	--

Legend:

--: Cannot be used.

In this table, if the argument type for an aggregate function is TINYINT, do not use the AVG function if there are more than 128 rows of input data. If the argument type is SMALLINT, do not use the AVG function if there are more than 32,768 rows of input data.

(5) Notes

If an overflow occurs while executing an aggregate function, processing continues. In this case, an integer will remain in the overflowed state (the higher bytes that do not fit are lost and the lower bytes are used as is) and a floating-point value will become infinity. If an overflow occurs in a floating-point value during an operation involving both a DECIMAL or NUMERIC type value and a floating-point value, however, an error (query group lockup) occurs.

(6) Usage example

Outputs the number of rows for column `a` of relation `s1`. The underlined part indicates the aggregate function.

```
REGISTER QUERY q1 SELECT COUNT(s1.a) AS a1 FROM s1 [ROWS 100];
```

4.4.13 Column specification list**(1) Format**

column-specification-list ::= *column-specification* [, *column-specification-list*]

(2) Function

Specifies one or more columns.

(3) Operands**column-specification**

For details about column specification, see 3.2.5(2) *Column specification*.

The number of columns that can be specified is in the range from 1 to 255.

(4) Syntax rules

For details about the syntax rules for the column specification list, see 3.2.5(2) *Column specification*.

(5) Notes

For notes on the column specification list, see 3.2.5(2) *Column specification*.

(6) Usage example

Groups columns `b` and `c` of relation `s1` and outputs the data in columns `a`, `b`, and `c`. The underlined part indicates the column specification list.

```
REGISTER QUERY q1 SELECT SUM(s1.a) AS a1, s1.b, s1.c FROM s1 [ROWS 100]
                        GROUP BY s1.b, s1.c;
```

4.4.14 Relation reference**(1) Format**

relation-reference ::= { *relation-name*
 | *stream-name* ' [' *window-specification* '] ' }
 [▲ AS ▲ *alias*]

(2) Function

Specifies the relation to be searched. The relation to be referenced is specified using the `FROM` clause.

(3) Operands**relation-name**

Specifies the name of the relation to be searched.

For the relation name, you can specify a query name that was returned without converting the inquiry result into a stream using a stream clause.

stream-name

Specifies the name of the stream to be searched.

For the stream name, you can specify a query name that was returned by turning the inquiry result into a stream using a stream clause.

window-specification

For details about window specification, see *4.4.15 Window specification*.

alias

Specify an alias if the name of the relation being referenced is long and complex. The alias name must follow the standard naming rules. For details about the naming rules, see *3.2.4 Specifying names*.

An alias name must be unique in any given `SELECT` clause. Also, you cannot specify a name that is the same as another table ID. For details about table IDs, see *3.2.5 Name qualification*.

(4) Syntax rules

- Specify a relation reference in the `FROM` clause.
- An alias specified in a relation reference is only valid in a single query statement, and cannot be used by multiple query statements.

For example, aliases `a1` and `a2`, specified in the relation reference as shown below, can only be used in a single query statement.

Correct specification example:

```
REGISTER QUERY q1 SELECT a1.c1, a2.c1 FROM s1[NOW] AS a1,
s2[NOW] AS a2
                        WHERE a1.c1 > 1 GROUP BY a1.c1, a2.c1
                        HAVING a2.c1 > 1;
```

In the following example, aliases `a1` and `a2` specified in the relation reference of `q2` are not acquired from `q1`, so they cannot be used.

Incorrect specification example:

```
REGISTER QUERY q1 SELECT a1.c1, a2.c1 FROM s1[NOW] AS a1,
s2[NOW] AS a2;
REGISTER QUERY q2 SELECT a1.c1, a2.c1 FROM q1 WHERE a1.c1 > 1
                        GROUP BY a1.c1, a2.c1 HAVING a2.c1 > 1;
```


- When specifying an alias in a relation reference, use a name that is different from the stream name and the query name.

In the following example, trying to specify `s2` as an alias fails because it is already used as a stream name.

Incorrect specification example:

```
REGISTER STREAM s1 (id INT, name VARCHAR(10));
REGISTER STREAM s2 (id INT, name VARCHAR(10));
REGISTER QUERY q1 SELECT * FROM s1[NOW] AS s2;
```

In the following example, trying to specify `q1` as an alias fails because it is already defined as a query name.

Incorrect specification example:

```
REGISTER QUERY q1 SELECT * FROM s1[NOW];
REGISTER QUERY q2 SELECT * FROM s2[NOW] AS q1;
```

- When specifying streams having the same name in a relation reference, specify aliases as shown in the following example so that each stream can be uniquely identified.

Correct specification example

```
REGISTER QUERY q1 ISTREAM(
    SELECT ... FROM s1[RANGE 5] AS old, s1[NOW] AS new ...);
```

In the following example, an error occurs because the streams cannot be uniquely identified.

Incorrect specification example

```
REGISTER QUERY q1 ISTREAM(
    SELECT ... FROM s1[RANGE 5], s1[NOW] ...);
```

(5) Notes

None.

(6) Usage example

Outputs the data in columns `a` and `b` of relation `s1`. The underlined part indicates a relation reference.

```
REGISTER QUERY q1 SELECT s1.a, s1.b FROM s1[ROWS 100];
```

4.4.15 Window specification

(1) Format

```
window-specification ::= {ROWS ▲ integer-constant
                        | RANGE ▲ integer-constant [ ▲ time-specification]
                        | NOW}
```

| PARTITION ▲ BY ▲ *column-specification-list* ▲ ROWS ▲ *integer-constant* }

(2) Function

Specifies how long data is retained in a stream. Window specification is entered in a relation reference.

(3) Operands

ROWS

Specify a value in the range from 1 to 100,000 for the number of rows that are retained in the stream.

RANGE

Specifies how long data stays in the stream. If you specify time division for a query, you must specify RANGE.

integer-constant

Specifies an amount of time. For the range of integer constant values, see *4.4.16 Time specification*.

time-specification

Specifies a unit for the time specified in *integer-constant*. For details about time specification, see *4.4.16 Time specification*.

If you do not specify this, the value specified in *integer-constant* is assumed to be in seconds.

NOW

Only items currently in the stream will be processed.

PARTITION BY

Applies ROWS processing to each column name value specified in the column specification list.

column-specification-list

For details about specifying a column specification list, see *4.4.13 Column specification list*.

For the column specification list, you can specify a column name.

(4) Syntax rules

None.

(5) Notes

- If ROWS specifies a value that exceeds the limit of the row count range, the maximum value of the range is assumed and processing continues.

- If ROWS is specified and the number of rows in the stream is 0, the minimum value is assumed and processing continues. If a negative value or a value other than an integer constant is specified, an error occurs.

(6) Usage examples

Usage example 1

Specifies that rows remain in the stream in relation `s1` for 2 seconds. The underlined part indicates the window specification.

```
REGISTER QUERY q1 SELECT * FROM s1 [RANGE 2 SECOND];
```

Usage example 2

Specifies that two rows are to remain in the stream for each value of column `a` in relation `s1`. The underlined part indicates the window specification.

```
REGISTER QUERY q1 SELECT * FROM s1 [PARTITION BY s1.a ROWS 2];
```

4.4.16 Time specification

(1) Format

time-specification ::= { SECOND | MILLISECOND | MINUTE | HOUR | DAY }

(2) Function

Specifies a unit of time.

(3) Operands

SECOND

Specifies seconds as the unit of time.

MILLISECOND

Specifies milliseconds as the unit of time.

MINUTE

Specifies minutes as the unit of time.

HOUR

Specifies hours as the unit of time.

DAY

Specifies days as the unit of time.

(4) Syntax rules

Separately from the time specification, the time value itself is specified as an integer constant. The following table shows the relationship between time specification and integer constants.

Table 4-5: Relationship between time specification and integer constants

No.	Time specification (for specifying a time unit)		Integer constant (for specifying a time value)	
	Operand	Meaning	Minimum value	Maximum value
1	SECOND	Second	1	2678400
2	MILLISECOND	Millisecond	1	86400000
3	MINUTE	Minute	1	44640
4	HOUR	Hour	1	744
5	DAY	Day	1	31

(5) Notes

- If a value exceeding the time value specification range is entered for the integer constant, the maximum value of the range is assumed and processing continues.
- If 0 is specified as the integer constant for the time value, the minimum value 1 is assumed and processing continues. If a negative value or a value other than an integer constant is specified, an error occurs.

(6) Usage example

Specifies that rows remain in the stream in relation `s1` for 2 seconds. The underlined part indicates the time specification.

```
REGISTER QUERY q1 SELECT * FROM s1[RANGE 2 SECOND];
```

4.4.17 Search condition**(1) Format**

For the HAVING clause

search-condition ::= *comparison-predicate* [**AND** *search-condition*]

For the WHERE clause

search-condition ::= { ' (' *search-condition* ') ' | *comparison-predicate* }
 | NOT { **AND** ' (' *search-condition* ') ' | *comparison-predicate* }

| *search-condition* **OR** *search-condition* { ' (' *search-condition* ') ' | *comparison-predicate* }

| *search-condition* **AND** *search-condition* { ' (' *search-condition* ') ' | *comparison-predicate* } }

(2) Function

Performs the logical operation specified in the search condition and returns only those results for which the condition is true in a relation.

A search condition is specified using the `WHERE` or `HAVING` clauses.

(3) Operands

comparison-predicate

For details about specifying a comparison predicate, see *4.4.18 Comparison predicate*.

(4) Syntax rules

Specify a search condition in a `WHERE` or `HAVING` clause. The logical operations that can be specified differ depending on the clause. The following table shows the logical operations that can be specified.

Table 4-6: Logical operations that can be specified

No.	Logical operation	WHERE clause	HAVING clause
1	AND	Can be specified.	Can be specified.
2	NOT	Can be specified.	Cannot be specified.
3	OR	Can be specified.	Cannot be specified.

(5) Notes

- The following limitations apply when specifying a `WHERE` clause:
 - The logical operator `OR` can only be specified if all of the comparison predicates for two terms are in the same relation.
 - The logical operator `NOT` can only be specified if all of the comparison predicates in the right monomial are in the same relation.

Multiple relations cannot be specified. The following shows examples of a relation in which `s1` and `s2` both use the schemas (`c1 INT`, `c2 INT`, and `c3 INT`).

Correct specification example: `OR` using the same relation is specified

```
s1.c1 < 1 OR s1.c2 < 1
```

Incorrect specification example: `OR` using multiple relations is specified

```
s1.c1 < 1 OR s2.c1 < 1
```

Correct specification example: `NOT` using the same relation is specified

```
NOT(s1.c1 < 1) AND NOT(s2.c1 < 1)
```

Incorrect specification example: `NOT` using multiple relations is specified

```
NOT((s1.c1 < 1) AND (s2.c1 < 1))
```

Correct specification example: OR and NOT using the same relation are specified

```
(s1.c1 < 1 OR s1.c2 < 1) AND NOT(s2.c1 < 1)
```

Incorrect specification example: OR and NOT using multiple relations are specified

```
s1.c1 < 1 OR s1.c2 < 1 AND NOT(s2.c1 < 1) ...
```

- Logical operations are evaluated according to the following hierarchy: what is in parentheses, NOT, AND, and OR. For example, the following two operations have the same meaning:

```
s1.c1 < 1 OR (s1.c2 < 1 AND NOT (s1.c3 < 3))
s1.c1 < 1 OR (s1.c2 < 1 AND s1.c3 >= 3)
```

- The parentheses () for search conditions may be omitted.
- A range from 1 to 255 comparison predicates and sets of parentheses can be specified in a search condition. Examples of how this is counted follow:

```
NOT c1>1
```

Since `c1>1` is the only comparison predicate, the count is 1.

```
NOT(c1>1)
```

Since there are a comparison predicate (`c1>1`) and a set of parentheses, the count is 2.

```
c1>1 AND c2>1
```

Since there are two comparison predicates (`c1>1` and `c2>1`), the count is 2.

```
(c1>1) AND (c2>1)
```

Since there are two comparison predicates (`c1>1` and `c2>1`) and two sets of parentheses, the count is 4.

(6) Usage example

Outputs data in which the value of column a in relation s1 is greater than 1 but less than 5. The underlined part indicates a search condition.

```
REGISTER QUERY q1 SELECT * FROM s1[ROWS 100] WHERE s1.a < 5 AND  
s1.a > 1;
```

4.4.18 Comparison predicate

(1) Format

For the WHERE clause

comparison-predicate ::= *value-expression comparison-operator value-expression*

comparison-operator ::= { < | <= | > | >= | = | != }

For the HAVING clause

comparison-predicate ::= *comparison-operand* *comparison-operator*
comparison-operand

comparison-operand ::= { *column-specification* | *column-name* | *aggregate-function* | *constant* }

comparison-operator ::= { < | <= | > | >= | = | != }

(2) Function

Specifies the condition for determining a true or false logical value.

(3) Operands**value-expression**

For details about specifying a value expression, see 4.4.19 *Value expression*.

comparison-operator

Specify <, <=, >, >=, =, or != as the comparison operator.

comparison-operand

Specify a column specification, an aggregate function, or a constant as the targets of the comparison.

If the preceding FROM clause only references a single relation, you can specify just a column name for that comparison operand.

column-specification

For details about column specification, see 3.2.4 *Specifying names*.

column-name

If the succeeding FROM clause only references a single relation, you can specify just a column name.

If the succeeding FROM clause references multiple relations, you must use a column specification.

aggregate-function

For details about aggregate functions, see 4.4.12 *Aggregate functions*.

constant

For details about constants, see 4.4.20 *Constant*.

(4) Syntax rules

The following table shows the meanings of comparison operators, using comparison operand X, the comparison operator, and comparison operand Y as comparison predicates.

Table 4-7: Comparison operator types and functions

No.	Comparison operator specification	Meaning
1	<i>comparison-operand</i> $x = \text{comparison-operand } y$	True if comparison operand X is equal to comparison operand Y.
2	<i>comparison-operand</i> $x \neq \text{comparison-operand } y$	True if comparison operand X is not equal to comparison operand Y.
3	<i>comparison-operand</i> $x < \text{comparison-operand } y$	True if comparison operand X is less than comparison operand Y.
4	<i>comparison-operand</i> $x \leq \text{comparison-operand } y$	True if comparison operand X is less than or equal to comparison operand Y.
5	<i>comparison-operand</i> $x > \text{comparison-operand } y$	True if comparison operand X is greater than comparison operand Y.
6	<i>comparison-operand</i> $x \geq \text{comparison-operand } y$	True if comparison operand X is greater than or equal to comparison operand Y.

- For the left and right comparison operands, only specify data types that can be compared. For details about data types that can be compared, see 3.4 *Data comparison*.
- If the data types being compared differ in how numeric data is compared, the data type with the wider range of values is used for comparison. The range hierarchy is as follows:
DECIMAL = NUMERIC > FLOAT = DOUBLE > REAL > BIGINT > INTEGER > SMALLINT > TINYINT

(5) Notes

- A constant cannot be compared with another constant. If an attempt is made to do this, an error occurs.
- A column specification or a column name must be included in either a comparison operand or a value expression.
- In a value expression specified for a comparison predicate, the four arithmetic operations cannot be executed on different data identifiers. For example, the following specification is invalid:
WHERE s1.a + s2.a < 5

(6) Usage example

Outputs data in which the value of column a in relation s1 is less than 5. The underlined part indicates a comparison predicate.

```
REGISTER QUERY q1 SELECT * FROM s1[RANGE 10 SECOND] WHERE s1.a
```


< 5;

4.4.19 Value expression

(1) Format

value-expression ::= [*sign*] *term* [*operator* [*sign*] *term*] . . .
term ::= [*cast-specification*] *value-expression-primary*
operator ::= { + | - | * | / }
sign ::= { + | - }

cast-specification ::= { ' (' TINYINT ') ' | ' (' SMALLINT ') ' | ' (' INT [EGER] ') ' | ' (' BIGINT ') ' | ' (' REAL ') ' | ' (' FLOAT ') ' | ' (' DOUBLE ') ' | ' (' DEC [IMA L] ') ' }

value-expression-primary ::= { *column-specification* | *column-name* | *constant* | ' (' *value-expression* ') ' }

(2) Function

Specifies a value.

(3) Operands

sign

Specify + or -.

term

Specify a value expression primary and, optionally, a cast specification.

cast specification

For details about the data types that can be specified for cast specification, see 3.3 *CQL data types*.

value expression primary

Specify a column specification, a column name, a constant, or a value expression.

column-specification

For details about column specifications, see 3.2.5(2) *Column specification*.

column-name

If the succeeding FROM clause only references a single relation, you can specify just a column name.

If the succeeding FROM clause references multiple relation, you must specify a column specification.

constant

For details about constants, see *4.4.20 Constant*.

operator

Specify +, -, *, or /. When specifying a signed term after an operator, enclose the term in parentheses. Signs and operator examples follow:

-a+ (-b)
a* (-b+1)
a/ (-b) +1

(4) Syntax rules

- You can perform the four arithmetic operations by specifying them in a value expression.
- The data type of the result of a value expression is the same as the data type of the result of the four arithmetic operations or of a value expression primary. The data structure is also the same as the arithmetic operations or a value expression primary.
- The four arithmetic operations (binomial operations) only use numeric data. Therefore, non-numeric data (such as character data and time data) cannot be included in an operation. If it is included, an error occurs.

Operations can be executed between numeric data of different types. The following table shows the relationship between the data types used in binomial operations and the resulting data type.

Table 4-8: Relationship between the data types used in the four arithmetic operations (binomial operations) and the resulting data type

No.	Data type of first operand	Data type of second operand								
		TNY	SML	INT	BIG	RL	FLT	DBL	DEC	NUM
1	TNY	TNY	SML	INT	BIG	RL	FLT	DBL	DEC	NUM
2	SML	SML	SML	INT	BIG	RL	FLT	DBL	DEC	NUM
3	INT	INT	INT	INT	BIG	RL	FLT	DBL	DEC	NUM
4	BIG	BIG	BIG	BIG	BIG	RL	FLT	DBL	DEC	NUM
5	RL	RL	RL	RL	RL	RL	FLT	DBL	DEC	NUM
6	FLT	FLT	FLT	FLT	FLT	FLT	FLT	FLT	DEC	NUM
7	DBL	DBL	DBL	DBL	DBL	DBL	DBL	DBL	DEC	NUM
8	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC	DEC
9	NUM	NUM	NUM	NUM	NUM	NUM	NUM	NUM	NUM	NUM

Legend:

TNY: TINYINT

SML: SMALLINT

INT: INTEGER

BIG: BIGINT

RL: REAL

FLT: FLOAT

DBL: DOUBLE

DEC: DECIMAL

NUM: NUMERIC

- For an operation that includes the DECIMAL or NUMERIC type, values in other data types are rounded according to the value of the property listed below and converted into values in the DECIMAL type (NUMERIC type) before they are used in the operation.
 - The `query.decimalMaxPrecision` property of `system_config.properties`
This property specifies precision.
 - The `query.decimalRoundingMode` property of `system_config.properties`
This property specifies a rounding method.

The operation result is also rounded in the same way.

(5) Notes

- If an overflow occurs when performing a binomial operation, processing continues. In this case, an integer will remain in the overflowed state (the higher bytes that do not fit are lost and the lower bytes are used as is) and a floating-point value will become infinity. If an overflow occurs in a floating-point value during an operation involving both a DECIMAL or NUMERIC type value and a floating-point value, however, an error (query group lockup) occurs.
- If 0 is specified for the second operand of a division operation, an error occurs.
- An binomial operation on character data or character string constant results in an error.
- Casting involving date/time data (including character string constant) results in an error.
- For character data, only character data whose format matches numeric data can

be cast. For details about character data formats, see 3.4 *Data comparison*.

- When a value is cast into the `DECIMAL` type, it is rounded according to the value of the property listed below before being converted.

- The `query.decimalMaxPrecision` property of `system_config.properties`

This property specifies precision.

- The `query.decimalRoundingMode` property of `system_config.properties`

This property specifies a rounding method.

(6) Usage example

Outputs the values obtained by casting column `a` in relation `s1` into `DECIMAL` data and column `b` into `INT` data. The underlined parts indicate value expressions.

```
REGISTER QUERY q1 SELECT (DECIMAL)s1.a AS xxx, (INT)s1.b AS yyy
FROM s1 [ROWS 100];
```

4.4.20 Constant

(1) Format

constant ::= { *character-string-constant* | *numeric-constant* }

character-string-constant ::= { *date-data* | *time-data* |
timestamp-data | *character-string* }

numeric-constant ::= { *integer-constant* | *floating-point-constant* | *decimal-constant* }

(2) Function

Specifies a constant. You can specify one or more values.

(3) Operands

character-string-constant

Specifies date data, time data, timestamp data, or a character string. The data type is `VARCHAR`.

date-data

For details about specifying date data, see 3.2.6(2) *Character string showing dates*.

time-data

For details about specifying time data, see 3.2.6(3) *Character string showing time*.

timestamp-data

For details about specifying timestamp data, see 3.2.6(4) *Character string showing timestamp data*.

character-string

For details about specifying character strings, see 3.2.6(1) *Constant types and notation*.

numeric-constant

Specifies an integer constant, floating-point constant, or decimal constant.

integer-constant

If an integer constant ends with the character `L` or `l`, the data type becomes `BIGINT`. If only a number is specified, the data type becomes `INTEGER`.

For details about specifying an integer constant, see 3.2.6(1) *Constant types and notation*.

floating-point-constant

The data type of a floating-point constant is `DOUBLE`.

For details about specifying a floating-point constant, see 3.2.6(1) *Constant types and notation*.

decimal-constant

The data type of a decimal constant is `DECIMAL` or `NUMERIC`.

For details about specifying a decimal constant, see 3.2.6(1) *Constant types and notation*.

(4) Syntax rules

None.

(5) Notes

- There are no escape characters for character strings. If a tab is inserted in a character string or if a new line is entered, that character code is embedded into the character string constant.
- A floating-point constant that overflows does not cause an error. The value becomes infinity.
- A decimal constant that is outside the allowable range for the `DECIMAL` type (`NUMERIC` type) does not cause an error. The value is expressed as a value described in CQL.

(6) Usage example

Outputs the result obtained by adding `-5.3` to the value of column `a` in relation `s1` and multiplying column `b` by `4`. The underlined part indicates a constant.

4. CQL Reference

```
REGISTER QUERY q1 SELECT -5.3+s1.a AS xxx, 4*s1.b AS xxy FROM  
s1 [ROWS 100];
```

Chapter

5. Query Definition Samples

This chapter explains the types of query definition samples provided by Stream Data Platform - AF.

5.1 Types of query definition samples

5.1 Types of query definition samples

Stream Data Platform - AF provides four types of query definition samples, as described in the table below. These samples are stored in the `api`, `file`, and `httppacket` directories under the *installation-directory*\samples\.

Table 5-1: Types of query definition samples

No.	Storage destination	File name	Explanation
1	api\query\	Inprocess_QueryTest	Query definition sample for in-process connection custom adaptor
2		RMI_QueryTest	Query definition sample for RMI connection custom adaptor
3	file\query\	Inprocess_QueryTest	Query definition sample using the standard adaptor to access input data in files
4	httppacket\query\	Inprocess_QueryTest	Query definition sample using the standard adaptor to access HTTP packets as input data

Chapter

6. Creating Custom Adaptors

This chapter explains how to create custom adaptors.

You create a custom adaptor as needed when handling a data type that a standard adaptor cannot handle.

For details about the APIs used when creating a custom adaptor, see 7. *APIs for Sending and Receiving Data*.

- 6.1 Types of custom adaptors that can be created
- 6.2 Creating an RMI connection custom adaptor
- 6.3 Creating an in-process connection custom adaptor
- 6.4 Other processes that may need to be implemented in custom adaptors
- 6.5 Compilation procedure
- 6.6 Notes on creating custom adaptors

6.1 Types of custom adaptors that can be created

This section explains the types of custom adaptors that can be created.

A custom adaptor is an application whose function is to exchange data between input and output sources and an SDP server. It is created using an API provided with Stream Data Platform - AF.

Depending on what they process, custom adaptors can be classified into data transmission applications and data reception applications. Data transmission and data reception applications can be further classified into RMI connection custom adaptors and in-process connection custom adaptors, depending on how they exchange data with the SDP server.

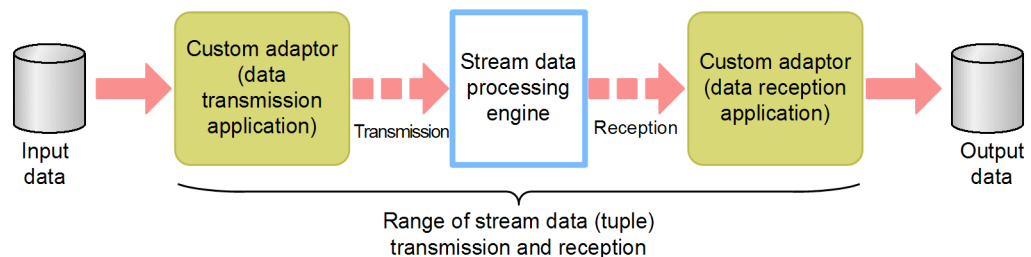
6.1.1 Data transmission applications and data reception applications

Depending on what they process, custom adaptors can be classified into the following two types:

- Data transmission application
Receives input data and sends stream data to the SDP server.
- Data reception application
Receives stream data for the result (query result) of an analysis by the SDP server.

The following figure shows where custom adaptors are positioned.

Figure 6-1: Positioning of custom adaptors



A data transmission application receives data in an arbitrary format and sends it as stream data to the stream data processing engine. A data reception application receives the stream data processed by the stream data processing engine and outputs it as data in an arbitrary format.

6.1.2 Classification based on how data is sent and received (RMI connection custom adaptor and in-process connection custom adaptor)

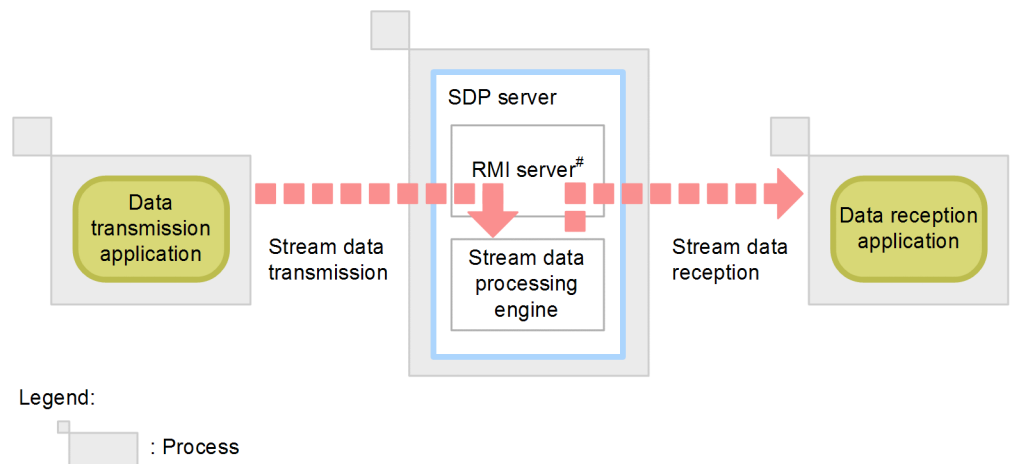
Depending on the technique used for exchanging data with the SDP server (application connection method), custom adaptors can be classified into the following two types:

■ RMI connection custom adaptor

Uses the Java RMI framework for exchanging data with the SDP server. The custom adaptor is started as a separate process from the SDP server.

The following figure shows the flow of stream data when an RMI connection custom adaptor is used.

Figure 6-2: RMI connection custom adaptor

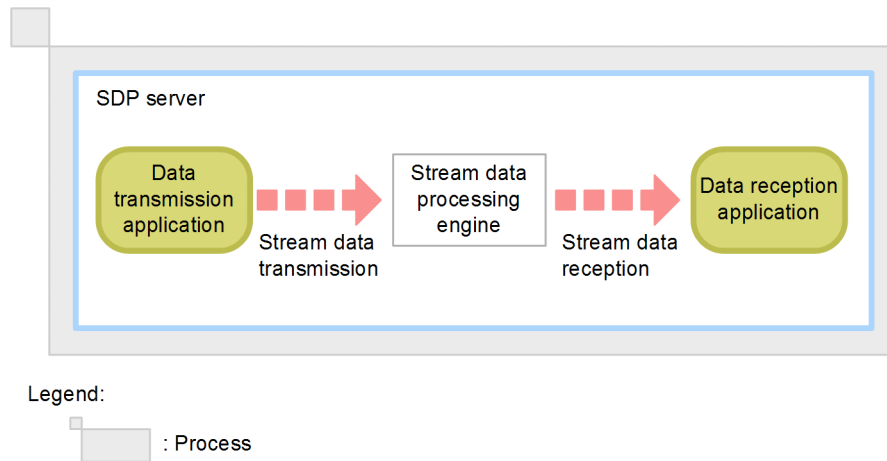


#: The RMI server performs all RMI communications. It runs on the SDP server.

■ In-process connection custom adaptor

Exchanges data with the SDP server from inside the SDP server process. The custom adaptor runs as part of the SDP server.

The following figure shows the flow of stream data when an in-process connection custom adaptor is used.

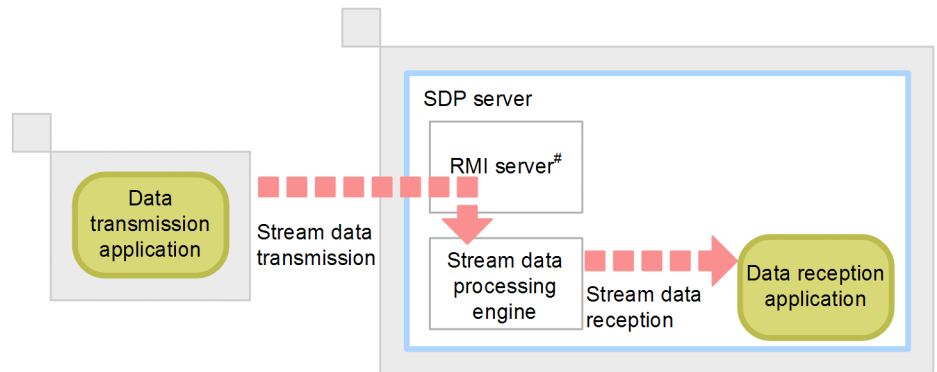
Figure 6-3: In-process connection custom adaptor

When using custom adaptors, it is possible for a data transmission application and a data reception application to use different data exchange methods.

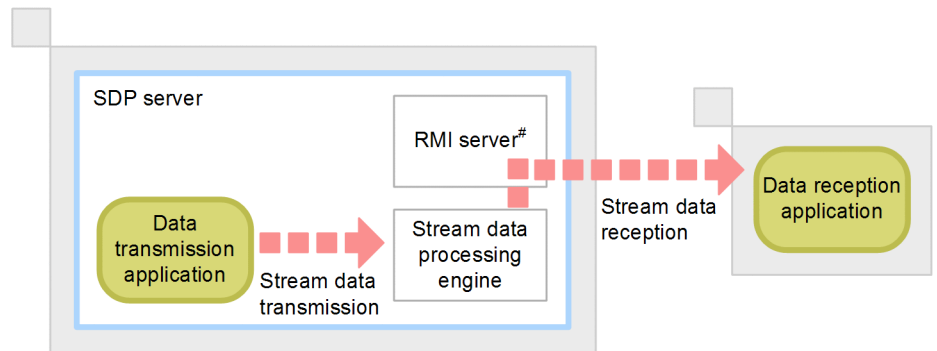
For example, as shown in the figure below, you can create the data transmission application as an RMI connection custom adaptor and the data reception application as an in-process connection custom adaptor, or create the data transmission application as an in-process connection custom adaptor and the data reception application as an RMI connection custom adaptor.

Figure 6-4: Using different data exchange methods for the data transmission application and the data reception application


- Case in which the data transmission application is an RMI connection custom adaptor and the data reception application is an in-process connection custom adaptor



- Case in which the data transmission application is an in-process connection custom adaptor and the data reception application is an RMI connection custom adaptor



Legend:

 : Process

#: The RMI server performs all RMI communications. It runs on the SDP server.

6.2 Creating an RMI connection custom adaptor

This section explains how to create an RMI connection custom adaptor.

An RMI connection custom adaptor, which runs as a separate process, connects to the SDP server using RMI communication and acquires an `SDPConnector` type object. To connect to the SDP server, the `connect` method in the `SDPConnectorFactory` class is used.

Stream data is sent and received using methods in the `StreamInput` interface and the `StreamOutput` interface. A `StreamInput` type object and a `StreamOutput` type object are created using the `SDPConnector` interface method. You can also create both a `StreamInput` type object and a `StreamOutput` type object from a single `SDPConnector` type object.

An overview of the interfaces used for sending and receiving stream data is provided below.

For sending stream data

For sending stream data, the `StreamInput` interface is used.

A custom adaptor uses an `SDPConnector` type object acquired from the `connect` method in the `SDPConnectorFactory` class to generate a `StreamInput` type object. The generated `StreamInput` type object uses the `put` method to send data.

For receiving stream data

For receiving stream data, the `StreamOutput` interface is used.

A custom adaptor uses an `SDPConnector` type object acquired from the `connect` method in the `SDPConnectorFactory` class to generate a `StreamOutput` type object. The generated `StreamOutput` type object uses the `get` or `getAll` method to receive data.

The `get` or `getAll` method dynamically acquires the result data from the SDP server. This data acquisition method is called the *polling method*.

6.2.1 Sending stream data (RMI connection custom adaptor)

This subsection explains the basic processing flow for using an RMI connection custom adaptor to send stream data, based on an implementation example.

Implementation example

```
public class RMI_SendSample {
    public static void main(String[] args) {
        try {
            // 1. Connect to the SDP server.
```

```

SDPConnector con = SDPConnectorFactory.connect();

// 2. Connect to the input stream to be sent.
StreamInput in = con.openStreamInput("GROUP","STREAM1");

// 3. Send tuples.
Object[] data=new Object[]{new Integer(1)};
StreamTuple tuple=new StreamTuple(data);
try {
    in.put(tuple);

    // 4. Send data transmission completion notification.
    in.putEnd();
} catch (SDPClientQueryGroupStateException e) {
    System.out.println("Query group stopped");
}

// 5. Disconnect from the input stream.
in.close();

// 6. Disconnect from the SDP server.
con.close();
} catch (SDPClientException e) {
    System.err.println(e.getMessage());
}
}
}

```

Explanation of the implementation details

The meaning of each process is explained below. The numbers correspond to the comment numbers in the implementation example.

1. Connects to the SDP server and acquires the SDPConnector type object (con).
`SDPConnector con = SDPConnectorFactory.connect();`
2. Uses the SDPConnector type object (con) to connect to an input stream that has "GROUP" as its group name and "STREAM1" as its stream name, and acquires the StreamInput type object (in).
`StreamInput in = con.openStreamInput("GROUP","STREAM1");`
3. Uses the StreamInput type object (in) to send tuples (stream data).
`in.put(tuple);`
4. Uses the StreamInput type object (in) to send a data transmission completion notification.
`in.putEnd();`

5. Uses the `StreamInput` type object (`in`) to disconnect from the input stream.
`in.close();`
6. Uses the `SDPConnector` type object (`con`) to disconnect from the SDP server.
`con.close();`

6.2.2 Receiving query result data (RMI connection custom adaptor)

This subsection explains the basic processing flow for using an RMI connection custom adaptor to receive query result data, based on an implementation example.

Implementation example

```
public class RMI_ReceiveSample {
    public static void main(String[] args) {
        try {
            // 1. Connect to the SDP server.
            SDPConnector con = SDPConnectorFactory.connect();

            // 2. Connect to the output stream.
            StreamOutput o = con.openStreamOutput("GROUP", "QUERY1");

            // 3. Receive tuples.
            try {
                while(true) {
                    ArrayList tupleList = o.getAll();
                }
            } catch (SDPClientEndOfStreamException e) {
                System.out.println("Data reception completed");
            } catch (SDPClientQueryGroupStateException e) {
                System.out.println("Query group stopped");
            }

            // 4. Disconnect from the output stream.
            o.close();

            // 5. Disconnect from the SDP server.
            con.close();
        } catch (SDPClientException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Explanation of the implementation details

The meaning of each process is explained below. The numbers correspond to the comment numbers in the implementation example.

1. Connects to the SDP server and acquires the `SDPConnector` type object (`con`).
`SDPConnector con = SDPConnectorFactory.connect();`
2. Uses the `SDPConnector` type object (`con`) to connect to an output stream that has "GROUP" as its group name and "QUERY1" as its query name, and acquires the `StreamOutput` type object (`o`).
`StreamOutput o = con.openStreamOutput("GROUP", "QUERY1");`
3. Uses the `StreamOutput` type object (`o`) to acquire the query result tuples (`tupleList`).
`ArrayList tupleList = o.getAll();`
4. When receiving is completed, the `StreamOutput` type object (`o`) is used to disconnect from the output stream.
`o.close();`
5. Uses the `SDPConnector` type object (`con`) to disconnect from the SDP server.
`con.close();`

6.3 Creating an in-process connection custom adaptor

This section explains how to create an in-process connection custom adaptor.

To implement an in-process connection custom adaptor, you need to create a class containing an instance of the `StreamInprocessUP` interface.

An in-process connection custom adaptor receives an `SDPConnector` type object generated by the SDP server as a parameter of the `execute` method in the implemented `StreamInprocessUP` interface class. For the `execute` method, define a process that is equivalent to the `main` method of the custom adaptor.

Both the `StreamInput` interface and the `StreamOutput` interface are used for sending and receiving stream data. Both a `StreamInput` type object and a `StreamOutput` type object are generated by the method of the `SDPConnector` type object passed as the parameter of the `execute` method.

An overview of the interfaces used for sending and receiving stream data is provided below.

For sending stream data

For sending stream data, the `StreamInput` interface is used.

A custom adaptor uses an `SDPConnector` type object acquired from the `connect` method in the `SDPConnectorFactory` class to generate a `StreamInput` type object. The generated `StreamInput` type object uses the `put` method to send data.

For receiving stream data

For receiving stream data, the `StreamOutput` interface is used.

A custom adaptor uses an `SDPConnector` type object acquired from the `connect` method in the `SDPConnectorFactory` class to generate a `StreamOutput` type object. The generated `StreamOutput` type object uses one of the methods described below to acquire the stream data (tuples) from the query result on the SDP server.

- Polling method

A method that gives a high priority to reducing latency and which dynamically acquires tuples from the SDP server. The `get` or `getAll` method is used to receive tuples.

- Callback method

A method in which the actual method for acquiring tuples is initiated by the SDP server when tuples are generated on the SDP server. In this case, the actual method to be called by the SDP server must be predefined. Although

the callback method increases latency compared to the polling method, CPU efficiency is higher since it is executed only when results are generated.

To use an in-process connection custom adaptor, you need to define the following in `user_app.in-process-connection-application-name.properties`.

```
user_app.classname=custom-adaptor-main-class-name
user_app.classpath_dir=custom-adaptor-main-class-path-name
```

For details about how to specify

`user_app.in-process-connection-application-name.properties`, see the manual *uCosminexus Stream Data Platform - Application Framework Setup and Operation Guide*.

6.3.1 Sending stream data (in-process connection custom adaptor)

This subsection explains the basic processing flow for using an in-process connection custom adaptor to send stream data, based on an implementation example.

Implementation example

```
public class Inpro_SendSample implements StreamInprocessUP {
    // Implement StreamInprocessUP.
    public void execute(SDPConnector con) {
        // 1. Connect to the input stream to be sent.
        StreamInput in = con.openStreamOutput("GROUP", "STREAM1");

        // 2. Send tuples.
        Object[] data=new Object[]{new Integer(1)};
        StreamTuple tuple=new StreamTuple(data);
        in.put(tuple);

        // 3. Send data transmission completion notification.
        in.putEnd();

        // 4. Disconnect from the input stream.
        in.close();

        // 5. Disconnect from the SDP server.
        con.close();
    }
}
```

Explanation of the implementation details

The meaning of each process is explained below. The numbers correspond to the comment numbers in the implementation example.

1. Uses the `SDPConnector` type object (`con`), which was passed as a parameter of the `execute` method, to connect to an input stream that has "GROUP" as its query group name and "STREAM1" as its stream name, and acquires the `StreamInput` type object (`in`).
`StreamInput in = con.openStreamInput("GROUP", "STREAM1");`
2. Uses the `StreamInput` type object (`in`) to send tuples.
`in.put(tuple);`
3. Uses the `StreamInput` type object (`in`) to send a data transmission completion notification.
`in.putEnd();`
4. Uses the `StreamInput` type object (`in`) to disconnect from the input stream.
`in.close();`
5. Uses the `SDPConnector` type object to disconnect from the SDP server.
`con.close();`

6.3.2 Receiving query result data (in-process connection custom adaptor)

This subsection explains the basic flow for receiving the query result data of an in-process connection custom adaptor, based on an implementation example.

Either the polling method or the callback method can be used by the in-process connection adaptor for receiving query result data. Each of these methods is explained below.

(1) Polling method

This method dynamically receives stream query result data from the SDP server.

An implementation example of receiving query result data using the polling method is described below.

Implementation example

```
public class Inpro_ReceiveSample1 implements StreamInprocessUP
{
    // Implement StreamInprocessUP.
    public void execute(SDPConnector con) {
        try {
            // 1. Connect to the output stream.
            StreamOutput o = con.openStreamOutput("GROUP", "QUERY1");

            // 2. Receive tuples.
            try {
                while(true) {
                    ArrayList tupleList = o.getAll();
```

```

    }
    } catch (SDPClientEndOfStreamException e) {
        System.out.println("Data reception completed");
    }

    // 3. Disconnect from the output stream.
    o.close();

    // 4. Disconnect from the SDP server.
    con.close();
    } catch (SDPClientException e) {
        System.err.println(e.getMessage());
    }
    }
}

```

Explanation of the implementation details

The meaning of each process is explained below. The numbers correspond to the comment numbers in the implementation example.

1. Uses the `SDPConnector` type object (`con`), which has been passed as a parameter of the `execute` method, to connect to an output stream that has "GROUP" as its query group name and "QUERY1" as its query name, and acquires the `StreamOutput` type object (`o`).
`StreamOutput o = con.openStreamOutput("GROUP", "QUERY1");`
2. Uses the `StreamOutput` type object (`o`) to acquire the query result data (tuples).
`ArrayList tupleList = o.getAll();`
3. When receiving is completed, the `StreamOutput` type object (`o`) is used to disconnect from the output stream.
`o.close();`
4. Uses the `SDPConnector` type object (`con`) to disconnect from the SDP server.
`con.close();`

(2) *Callback method*

In this method, the actual method for receiving query result data is initiated by the SDP server.

Implementation examples of receiving query result data using the callback method are described below.

To use the callback method for receiving the query result data, you need to implement the following two methods:

- The `execute` method for the class in which the `StreamInprocessUP` interface is implemented
- The `onEvent` method for the callback class in which the `StreamEventListener` interface is implemented

An implementation example of each is explained below.

Implementation example (using the `execute` method)

```
public class Inpro_ReceiveSample2 implements StreamInprocessUP
{
    // Implement StreamInprocessUP.
    public void execute(SDPConnector con) { // Process that
        // is executed when the application starts.
        // 1. Connect to the output stream.
        StreamOutput o = con.openStreamOutput("GROUP","QUERY2");

        // 2. Generate a listener object for callback.
        CallBack notifiable = new CallBack();

        // 3. Register the listener object for callback.
        o.registerForNotification(notifiable);
    }
    public void stop() { // Process that is executed when
        // the application stops.
        // 4. Cancel the listener object for callback.
        o.unregisterForNotification(notifiable);

        // 5. Disconnect from the SDP server.
        con.close();
    }
}
```

Implementation example (using the `onEvent` method)

```
public class CallBack implements StreamEventListener {
    // Implement StreamEventListener.
    public void onEvent(StreamTuple tuple) {
        // 6. Processing when tuples are received
        System.out.println("Tuple received:" + tuple);
    }
}
```

Explanation of the implementation details

The meaning of each process is explained below. The numbers correspond to the comment numbers in the implementation examples.

1. Uses the `SDPConnector` type object (`con`), which is specified in the parameter

of the `execute` method, to connect to an output stream that has "GROUP" as its query group name and "QUERY2" as its query name, and acquires the `StreamOutput` type object (o).

```
StreamOutput o = con.openStreamOutput("GROUP", "QUERY2");
```

2. Generates an object (notifiable) in the implemented class (CallBack) of the `StreamEventListener` interface.

```
CallBack notifiable = new CallBack();
```
3. Uses the `StreamOutput` type object (o) to register the listener object for the callback created in step 2.

```
o.registerForNotification(notifiable);
```
4. When stopping the custom adaptor, the `StreamOutput` type object (o) is used to cancel the listener object for callback (notifiable).

```
o.unregisterForNotification(notifiable);
```
5. When stopping the custom adaptor, the `SDPConnector` type object (con) is used to disconnect from the SDP server.

```
con.close();
```
6. When the tuples of the query result arrive at the output stream, the content defined in the `onEvent` method of the `StreamEventListener` interface is executed.

```
public void onEvent(StreamTuple tuple) {
// 6. Processing when tuples are received
}
```

6.3.3 Notes

When implementing an in-process connection custom adaptor, do not use the `System.exit` method. Because the in-process connection custom adaptor is executed in the same process as the SDP server, if the `System.exit` method is called in the in-process connection custom adaptor, the SDP server will also be terminated.

6.4 Other processes that may need to be implemented in custom adaptors

This section explains other processes that may need to be implemented in custom adaptors. Implement the necessary processes by combining them with the basic processes described in *6.2 Creating an RMI connection custom adaptor* or *6.3 Creating an in-process connection custom adaptor*.

- Detecting the trigger for terminating a data reception application (data processing termination notification)
- Specifying time information in the data source mode
- Preventing queue overflow

Some of the functions explained in this section require you to specify parameters in definition files. For details about each definition file, see the manual *uCosminexus Stream Data Platform - Application Framework Setup and Operation Guide*.

For details about the APIs used by processes, see *7. APIs for Sending and Receiving Data*.

6.4.1 Detecting the trigger for terminating a data reception application (data processing termination notification)

Since a data reception application cannot be terminated by a command in Stream Data Platform - AF, a termination process must be implemented in the data reception application.

This section explains an implementation that enables a data reception application to detect the trigger for terminating itself. The method shown here can be used in the following cases, in which the trigger for terminating stream data transmission can be detected by the stream data processing engine beforehand:

- Case in which file data is processed as stream data
- Case in which the time for ending data transmission from the data-sending source is constant and known

Reference note:

A data processing termination notification is not required to terminate a data transmission application. The data transmission application will be terminated when the transmission of input data stops and the transmission of data to the stream data processing engine is completed.

(1) Method of detecting the trigger for terminating a data reception application

A data reception application keeps receiving result data as long as queries are being executed by the stream data processing engine. For the data reception application to end, it must detect the following states:

- Execution of queries by the stream data processing engine is completely finished.
- No query result data remains in the output stream.

The data reception application can detect these states by catching the `SDPClientEndOfStreamException` or `SDPClientQueryGroupStopException` exceptions that occur when the data reception API (`get` or `getAll` method) is called. Catching these exceptions allows you to terminate the data reception application.

To generate these exceptions, a data processing termination notification must be sent to the output stream. This is done by having the data transmission application call a method or execute a command.

These notification methods are explained in (2) *Data transmission termination notification based on the data transmission application calling a method (putEnd method)* and (3) *Query group stop notification based on the execution of the `sdpcqlstop` command*.

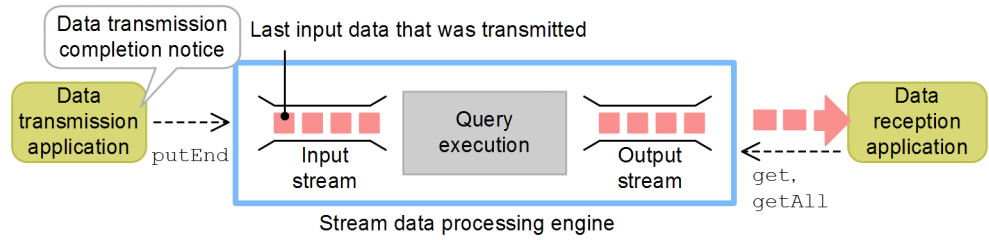
(2) Data transmission termination notification based on the data transmission application calling a method (putEnd method)

When the data transmission application is finished transmitting data, the `putEnd` method should be called. When the output stream runs out of query result data, the `SDPClientEndOfStreamException` exception is thrown to the data reception application.

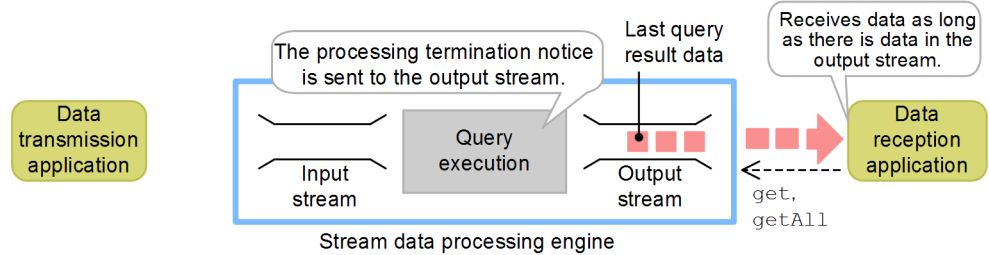
The following figure shows the flow for sending a data transmission termination notification by calling the `putEnd` method.

Figure 6-5: Flow for sending a data transmission termination notification by the data transmission application

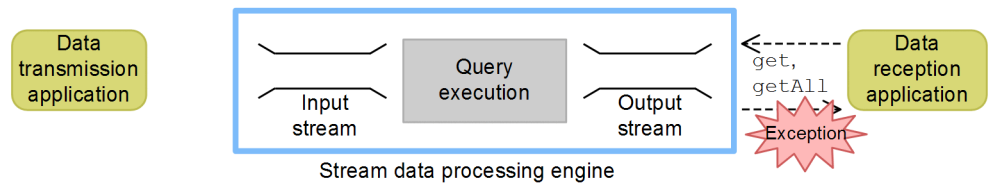
1. The `putEnd` method is used to send a data transmission completion notice from the data transmission application.



2. The stream data processing engine sends a processing termination notice to the output stream. The data reception application continues to receive data as long as there is data in the output stream.



3. When the output stream runs out of data, an exception is thrown in response to the data reception application's calling of the `get` or `getAll` method. In this way, data processing completion can be recognized.



Legend:

----> : Flow of method calls and the exception that occurs when a method is called

The details of the flow shown in the figure are explained below. The numbers correspond to the numbers in the figure.

1. When the transmission of stream data to the stream data processing engine is completely finished, the data transmission application calls the `putEnd` method to send a data transmission termination notification to the applicable input stream.

2. Once a transmission termination notification is received for all input streams in the query group, the stream data processing engine executes a query on all tuples in the input streams, outputs the query result data to the appropriate output streams, and then sends a processing termination notification to all output streams.

Even after the processing termination notification is issued, the data reception application continues to receive data as long as there is query result data in the output stream. To receive data, the `get` or `getAll` method is used.

3. When all query result data in the output streams has been received and the output stream runs out of data, the exception `SDPClientEndOfStreamException` is thrown the next time the data reception application calls the `get` or `getAll` method.

Using this exception as the trigger, the data reception application can be stopped.

If the query group contains an input stream for which the `putEnd` method has not been called, the exception `SDPClientEndOfStreamException` does not occur when the data reception application calls the `get` or `getAll` method.

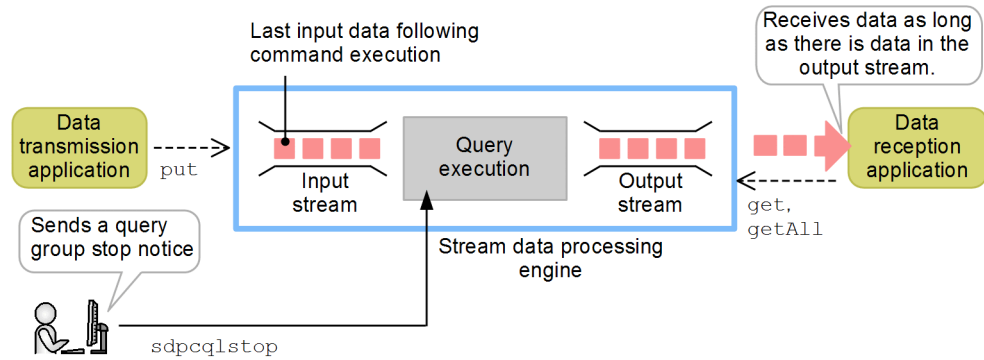
(3) Query group stop notification based on the execution of the `sdpcqlstop` command

The `sdpcqlstop` command is executed to stop the query group. When the output stream runs out of query result data, the exception `SDPClientQueryGroupStopException` is thrown to the data reception application.

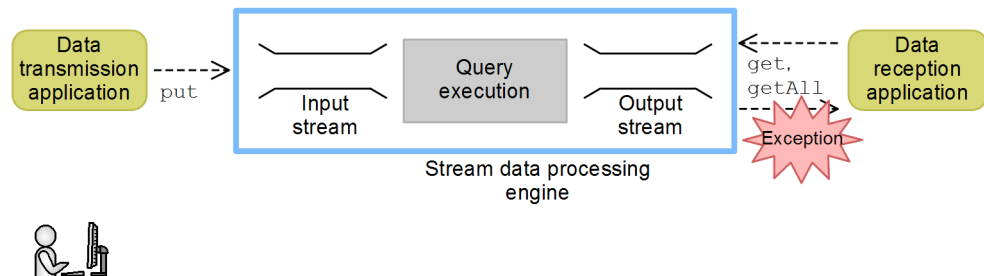
The following figure shows the flow for using the `sdpcqlstop` command to stop a query group.

Figure 6-6: Flow for using the sdpcqlstop command to stop a query group

1. The `sdpcqlstop` command is used to send a query group stop notice.



2. When the output stream runs out of data, an exception is thrown in response to the data reception application's calling of the `get` or `getAll` method. In this way, data processing completion can be recognized.



Legend:

----> : Flow of method calls and the exception that occurs when a method is called

—> : Command execution flow

The details of the flow shown in the figure are explained below. The numbers correspond to the numbers in the figure.

1. When the `sdpcqlstop` command issues a query group stop notification, the stream data processing engine stops accepting input stream data. It continues to process the data already in the input stream and then terminates processing.
Even after the command is executed, if query results remain in the output stream, the data reception application continues to receive the data using the `get` or `getAll` method.
2. When the output stream runs out of data, the exception `SDPClientQueryGroupStopException` is thrown the next time the data

reception application calls the `get` or `getAll` method. Using this exception as the trigger, the data reception application can be stopped.

6.4.2 Specifying time information in the data source mode

To manage the time information for the tuples in the stream data when using the data source mode, you must specify time information in the data transmission application. Furthermore, the time information of the tuples being sent must be set so that the tuples are always in ascending order.

This subsection explains the items that must be implemented in the data transmission application in order to specify time information in tuples. It also explains the processing by the stream data processing engine following data transmission.

Note that the processes explained here are not necessary when using the server mode.

(1) Specifying time information in tuples

When using the data source mode, you must specify time information as one of the data columns in the tuples to be sent.

To specify time information, you must set it in the data transmission application so it can be referenced in the query definition.

Query definition

Define the time information as a `TIMESTAMP` data type in the schema specification character string of the stream definition that describes the query definition file. A definition example follows:

```
REGISTER STREAM s1(t1 TIMESTAMP, id INT, name VARCHAR(10));
```

Packaging in the data transmission application

In the data transmission application, create a Java object of the `java.sql.Timestamp` class for the time information to be stored as column data in the tuple. An implementation example follows:

```

        :
        :
// Create a tuple object.
Object[] data = new Object[]{
    new Timestamp(System.currentTimeMillis()),
    new Integer(100),
    new String("data1")
};

// Set the object in the tuple.
StreamTuple tuple1 = new StreamTuple(data);
        :
        :

```

Note: When creating an actual data transmission application, extract the data source's time information that is contained in a log file or the like and specify it for new Timestamp.

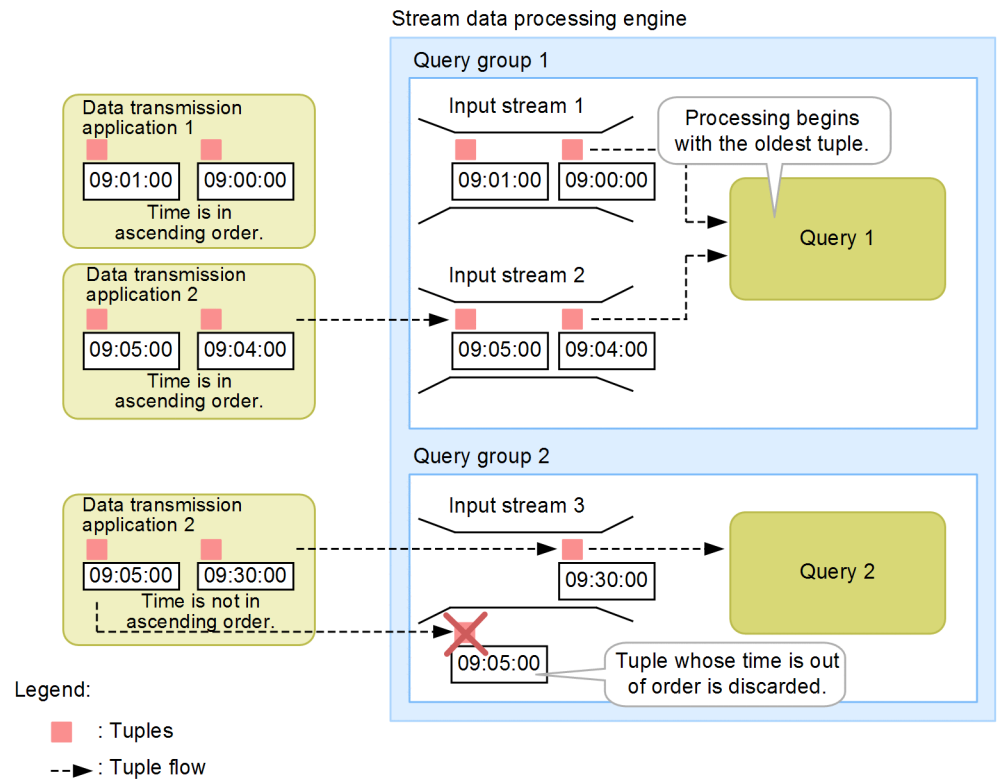
(2) Controlling the time information when sending tuples

When using the data source mode, the time information in the tuples being sent must be controlled by the data transmission application to be in ascending order.

For each input stream, the stream data processing engine processes the tuples it has received in ascending time order. Note that tuples sent in the wrong order are discarded by the SDP server.

The figure below shows the processing sequence when tuples are sent from multiple data transmission applications to multiple input streams. In this figure, input stream 1 and input stream 2 are part of the same query group. The numeric value (such as 09:00:00) indicated below the tuple in the figure is the time information specified for that tuple.

Figure 6-7: Example of processing sequence when tuples are sent from multiple data transmission applications to multiple input streams



When a single query group contains multiple input streams, the stream data processing engine processes streams beginning with the one containing the tuple with the oldest time information. In the example in the figure, the tuples in *input stream 1* are processed first, followed by tuples in *input stream 2*. In order for a query to be executed, all streams, except those for which a processing termination notification was sent using the `putEnd` method, must contain one or more tuples. If even one stream is empty, the stream containing the oldest tuple cannot be determined, and so no processing is performed by the stream data processing engine.

If tuples arrive out of time sequence, the stream data processing engine discards them. In the example in the figure, the time sequence of the tuples in *input stream 3* is incorrect, and so the tuples that arrived later are discarded.

Reference note:

If data is sent from multiple data transmission applications when using the data source mode, an error may cause the data arrival at the stream data processing engine to be out of order, so the processing order may not be in ascending order of the time stamp. In this case, if you want to process the out-of-order files instead of discarding them, you need to use the timestamp adjustment function to adjust the time. For details about how to use the timestamp adjustment function, see the explanation on adjusting the time stamp of tuples in the manual *uCosminexus Stream Data Platform - Application Framework Setup and Operation Guide*.

(3) Processing after all tuples are input

When all tuples have been input from the data transmission application, send a data transmission completion notification to the stream data processing engine.

When using the data source mode, the stream data processing engine processes data in time sequence based on the tuple's time information. Once all tuples have been input from the data transmission application and the input stream runs out of tuples, the operation time ceases to advance in the stream data processing engine. Since some query processing, such as a window operation, is only executed when the operation time advances, stream data may be stranded in the stream data processing engine when there are no more tuples in the input stream.

To prevent such a situation, the data transmission application must explicitly indicate when all data has been transmitted. When a data transmission completion notification is received, the stream data processing engine processes all remaining tuples and outputs the results to the output stream.

For details about how to send a data transmission completion notification, see 6.4.1 *Detecting the trigger for terminating a data reception application (data processing termination notification)*.

6.4.3 Preventing queue overflow

Queues are used to exchange tuples between the stream data processing engine and a custom adaptor. For the input stream, it is called the *input stream queue*, and for the output stream, the *output stream queue*. The tuples in these queues are processed on a FIFO basis.

The maximum number of tuples that can be in the input stream queue or the output stream queue is defined in one of the following files:

- `system_config.properties`
- Query group property file
- Stream property file

If more tuples than the specified value are sent, a queue overflow occurs.

This section explains what happens when a queue overflow occurs in the input stream queue or the output stream queue, and an implementation method to prevent queue overflow.

(1) What happens when a queue overflows

Exactly what happens when a queue overflows differs depending on whether it occurs in the input stream queue or in the output stream queue.

The following table shows what happens when the input stream queue overflows.

Table 6-1: What happens when the input stream queue overflows

No.	Timestamp mode	Timestamp adjustment function	Action
1	Server mode	--	<ul style="list-style-type: none"> The exception <code>SDPClientFreeInputQueueSizeLackException</code> is thrown when the data transmission application calls the <code>put</code> method. The stream data processing engine compares the number of tuples sent from the data transmission application with the free size in the input stream queue, and does not register any of the sent tuples in the input stream queue if it would overflow. The stream data processing engine does not, however, prevent the query group from being processed.
2	Data source mode	Valid	<ul style="list-style-type: none"> The exception <code>SDPClientException</code> is thrown when the data transmission application calls the <code>put</code> method. The stream data processing engine prevents the query group from being processed. Also, because the query group is not processed, all input tuples in the input stream queue are discarded. This processing, however, does not discard any tuples already in the output stream queue.
3		Invalid	<ul style="list-style-type: none"> The exception <code>SDPClientFreeInputQueueSizeLackException</code> is thrown when the data transmission application calls the <code>put</code> method. The stream data processing engine compares the number of tuples sent from the data transmission application with the free size in the input stream queue, and does not register any of the sent tuples in the input stream queue if it would overflow. The stream data processing engine does not, however, prevent the query group from being processed.

Legend:

--: Not applicable

The table below shows what happens when the output stream queue overflows. The action varies depending on the specification of the `querygroup.sleepOnOverStoreRetryCount` parameter in the definition file (`system_config.properties` or the query group property file).

Table 6-2: What happens when the output stream queue overflows

No.	Definition file specification	Action
1	If <code>querygroup.sleepOnOverStoreRetryCount != 0</code> is specified	The number of output tuples to be registered by the query group is compared with the amount of free space in the output stream queue, and execution of the query group is temporarily stopped if the output stream queue would overflow (the thread for the executing query is put to sleep). After that, if the output stream queue would overflow if the output tuples were registered in the output stream queue, the query group is held. Also, because the query group is held, all input tuples registered in the input stream queue are discarded. This hold process, however, does not discard the tuples registered in the output stream queue.
2	If <code>querygroup.sleepOnOverStoreRetryCount = 0</code> is specified	Holds the query group. Also, because the query group is held, all input tuples registered in the input stream queue are discarded. This hold process, however, does not discard the tuples registered in the output stream queue.

(2) Preventing the input stream queue from overflowing

This subsection explains how to implement a procedure to prevent the input stream queue from overflowing in a data transmission application.

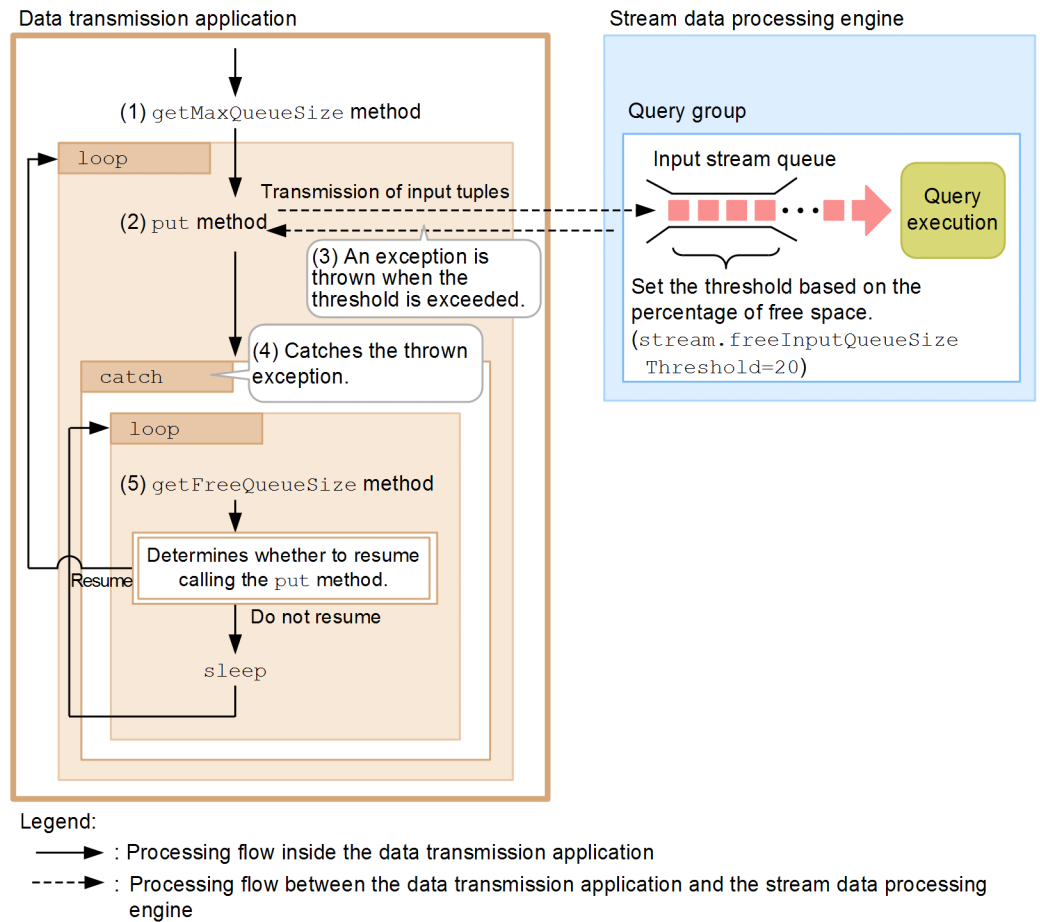
Hint:

To enable the process explained here, you must specify the `stream.freeInputQueueSizeThreshold` parameter in the SDP server definition file (`system_config.properties`, the query group property file, or the stream property file) in advance.

In the example described below, `stream.freeInputQueueSizeThreshold=20` is specified.

The following figure provides an overview of a way to implement the data transmission application in order to prevent a queue overflow.

Figure 6-8: Coding the data transmission application to prevent the input stream queue from overflowing



The details of the processes shown in the figure are explained below. The numbers in the explanation correspond to the numbers in the figure.

1. The data transmission application calls the `getMaxQueueSize` method and acquires the maximum size in the input stream queue. This value is used in step 5 to calculate the remaining percentage of free space in the input stream queue.
2. The data transmission application calls the `put` method to send the input tuples to the stream data processing engine. The sent tuples are stored in the input stream queue managed by the query group.
3. If the percentage of free space in the input stream queue falls to or below the threshold specified in the definition file (the value specified in the

`stream.freeInputQueueSizeThreshold` parameter), the stream data processing engine sends the exception `SDPClientFreeInputQueueSizeThresholdOverException` to the data transmission application.

4. The notification from the stream data processing engine causes the data transmission application to catch the exception `SDPClientFreeInputQueueSizeThresholdOverException` that is thrown by the `put` method.

If `stream.freeInputQueueSizeThresholdOutputMessage=true` is specified in the definition file, the following message is issued to the message log:

```
KFSP42032-W The available space in the stream queue is now at or below the
threshold. Stream name = ..., number of elements = ..., upper limit value =
..., threshold (%) = ...
```

5. The data transmission application calls the `getFreeQueueSize` method and acquires the amount of free space in the input stream queue. This value is used to determine whether calling of the `put` method can be resumed.

If it is determined that there is enough space, calling of the `put` method is resumed.

If it is determined that there is not enough space, the process is put to sleep for a set amount of time, after which the `getFreeQueueSize` method is called again to determine whether calling of the `put` method can be resumed.

While this determination process is going on, tuples are not sent until enough free space can be secured in the input stream queue.

Note:

If the input stream queue overflows even though you specified `stream.freeInputQueueSizeThreshold`, the stream data processing engine throws the exception `SDPClientFreeInputQueueSizeLackException` to the data transmission application. If this happens, the exception `SDPClientFreeInputQueueSizeThresholdOverException` is not thrown.

An example of an implementation that prevents the input stream queue from overflowing is described below. In the implementation example, select either Process 1 or Process 2 according to the evaluation method used.

Implementation example

:

```

try {
    streamInput = connector.openStreamInput(TARGET_QUERYGROUP,
TARGET_STREAM);

    // Acquire the maximum queue size.
    int maxQueueSize = streamInput.getMaxQueueSize();

    for (int i=0; i<1000; i++){
        Object[] data = new Object[] {
            new Integer(i),
            new String("data:"+i)
        };
        StreamTuple tuple = new StreamTuple(data);

        try {
            streamInput.put(tuple);
        } catch (SDPClientFreeInputQueueSizeThresholdOverException
sce) {
            // The free size in the queue has fallen to 20% or less.
            // (When stream.freeInputQueueSizeThreshold=20
            // is specified).
            while (true) {
                int freeQueueSize = streamInput.getFreeQueueSize();

                // Process 1: Check the free size in the queue to see
                // if you can resume using the put method.
                // When the free size in the queue reaches 500 or more,
                // resume the put method.
                if (freeQueueSize >= 500) {
                    break;
                } else {
                    Thread.sleep(100);
                }
                // End of Process 1

                // Process 2: Check the ratio between the free size in
                // the queue.
                // and the maximum size to see if you can resume using
                // the put method.
                double freePerMax = ((double)freeQueueSize /
(double)maxQueueSize) * 100;
                // When the free size in the queue reaches 500 or more,
                // resume the put method.
                if (freePerMax >= 50) {
                    break;
                } else {
                    Thread.sleep(100);
                }
            }
        }
    }
}

```

```

        }
        //End of Process 2

    }

    } catch (SDPClientFreeInputQueueSizeLackException sce2) {
        // There is no free size in the queue.
        // (Tuples sent using the put method were not accepted by
        // the stream data processing engine).
        // Therefore, the process is put to sleep before resending
        // tuple.
        Thread.sleep(100);
        try {
            // Resend tuples.
            streamInput.put(tuple);
        } catch (SDPClientFreeInputQueueSizeThresholdOverException
sce3) {
            // Since tuples were successfully resent, ignore
            // the threshold exception here.
        }
    }
}
streamInput.putEnd();
} catch (SDPClientFreeInputQueueSizeLackException sce4) {
    // There is no free size in the queue.
    // (Tuples sent using the put method were not accepted by
    // the stream data processing engine).
    :(omitted)
    // Stop the transmission.
    streamInput.putEnd();
    :(omitted)
} catch (...) {
    // Implement the processes that happen when other exceptions
    // are caught.
    :(omitted)
}

```

(3) Preventing the output stream queue from overflowing

To prevent the output stream queue from overflowing, specify the `querygroup.sleepOnOverStoreRetryCount` and `querygroup.sleepOnOverStore` parameters in the definition file. If these parameters are specified, execution of the query group is temporarily stopped (put to sleep) if the output stream queue is about to overflow.

Furthermore, if the implementation described in (2) *Preventing the input stream queue from overflowing* is used, sending of input tuples to the input stream queue can be suppressed when the execution of the query group is temporarily stopped. This allows the system's processing to return to normal without causing a queue overflow.

The figure below provides an overview of a process that prevents the output stream

queue from overflowing. Coding of the data transmission application is the same as in (2) *Preventing the input stream queue from overflowing*.

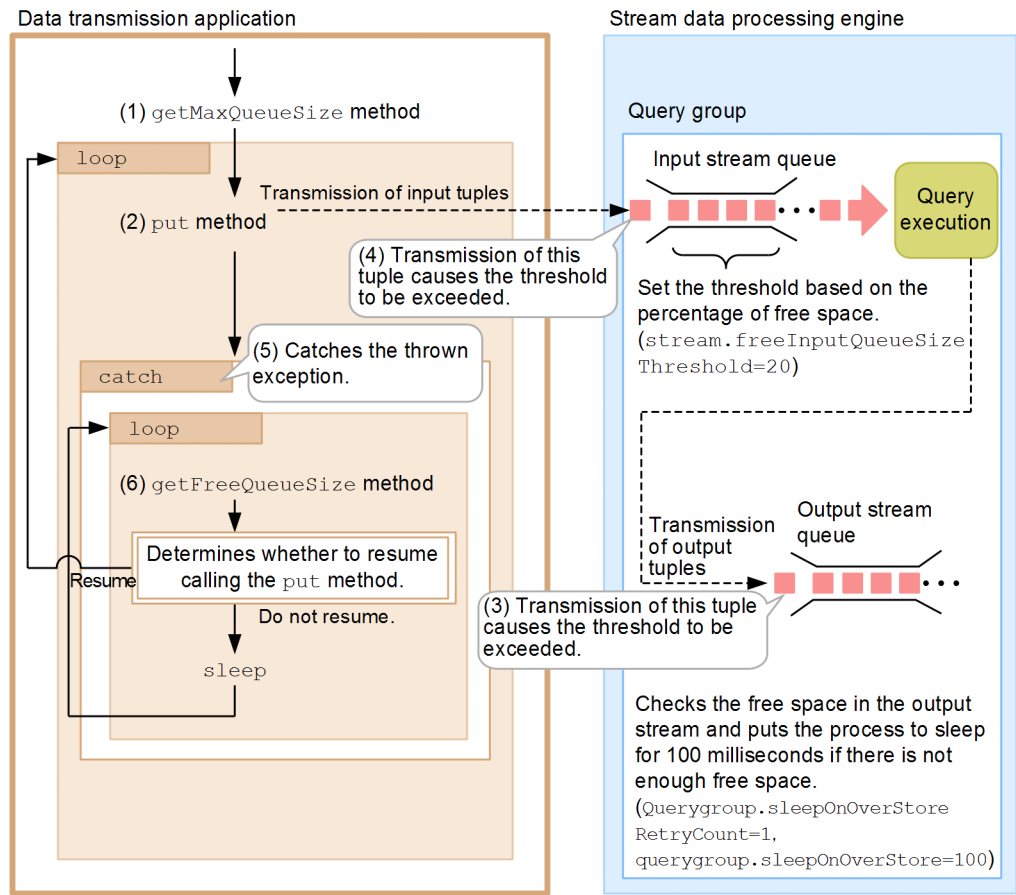
Hint:

To enable the process explained here, you must specify the following parameters in the SDP server definition file:

- `stream.freeInputQueueSizeThreshold`
- `querygroup.sleepOnOverStoreRetryCount`
- `querygroup.sleepOnOverStore`

In the example, `stream.freeInputQueueSizeThreshold=20`, `querygroup.sleepOnOverStoreRetryCount=1`, and `querygroup.sleepOnOverStore=100` are specified.

Figure 6-9: Overview of a process that prevents the output stream queue from overflowing



Legend:

—→ : Processing flow inside the data transmission application

-----→ : Processing flow between the data transmission application and the stream data processing engine

The details of the processes shown in the figure are explained below. The numbers in the explanation correspond to the numbers in the figure.

1. The data transmission application calls the `getMaxQueueSize` method and acquires the maximum size of the input stream queue. This value is used to calculate the remaining percentage of free space in the input stream queue.
2. The data transmission application calls the `put` method to send tuples to

the stream data processing engine. The sent tuples are stored in the input stream queue managed by the query group.

3. If the stream data processing engine detects that the output tuples sent by query execution will cause the output stream queue to overflow, it temporarily stops (puts to sleep) execution of the query group. Whether the query group is put to sleep and how long it will be kept sleeping are specified in the `querygroup.sleepOnOverStoreRetryCount` and `querygroup.sleepOnOverStore` parameters.

If the query group goes to sleep, the stream data processing engine outputs the following message to the message log:

```
KFSP42033-W The system is retrying registration of a tuple in the stream queue. Query
group name = ..., stream name = ..., sleep time until retry = ..., retry iterations =
...
```

Note that the status of the sleeping query group is still set to *Executing* in this case. The data transmission application is still allowed to send tuples to the input stream queue, and the data reception application is still allowed to acquire results from the output stream queue.

4. While execution of the query group is sleeping, if the data transmission application continues to call the `put` method, unprocessed tuples continue to accumulate in the input stream queue. When the percentage of free space in the input stream queue falls to or below the threshold specified in the definition file (the value specified in the `stream.freeInputQueueSizeThreshold` parameter), the stream data processing engine sends the exception `SDPClientFreeInputQueueSizeThresholdOverException` to the data transmission application.
5. The notification from the stream data processing engine causes the data transmission application to catch the exception `SDPClientFreeInputQueueSizeThresholdOverException` that is thrown by the `put` method.
6. The data transmission application calls the `getFreeQueueSize` method and acquires the amount of free space in the input stream queue. This value is used to determine whether calling of the `put` method can be resumed.

If it is determined that there is enough space in the input stream queue, calling of the `put` method is resumed.

If it is determined that there is not enough space in the input stream queue, the process is put to sleep for a set amount of time, after which the `getFreeQueueSize` method is called again to determine whether calling of the `put` method can be resumed.

In this way, the sending of input tuples using the `put` method is suppressed until a sufficient amount of free space is available in the input stream queue. When a sufficient amount of free space is available in the output stream queue and the execution of the query group is resumed to process the tuples in the input stream queue, input tuples can be sent again, and the system returns to its normal state.

6.5 Compilation procedure

To compile a custom adaptor:

1. In preparation, define an environment variable.

Add the path shown below to the operating user's `PATH` environment variable, which was specified when Stream Data Platform - AF was set up. This step is not required during subsequent compile operations.

```
installation-directory\psb\jdk\bin
```

2. Execute the compile.

Execute the following command:

```
javac -cp installation-directory\lib\sdp.jar Java source program
```

3. Create a `.jar` file (in the case of an in-process connection custom adaptor)

In the case of an in-process connection custom adaptor, create a `.jar` file that groups together the class files generated during the compile.

The following is an example of a command to group together class files `A.class` and `B.class` in the `test` directory into a file called `test.jar`.

```
jar cf test.jar test\A.class test\B.class
```

6.6 Notes on creating custom adaptors

This section explains other details about creating custom adaptors.

Use the following policies when creating a custom adaptor's data transmission application and data reception application:

- Do not connect to the same stream more than once.

Connect to each stream only once.

Although a single `SDPConnector` type object cannot connect to the same stream more than once, multiple `SDPConnector` type objects can be used to connect to the same stream multiple times. However, making multiple connections lowers the throughput of the stream data processing engine.

- If the output frequency of query result data to the output stream is high compared to the processing load of the data reception application, use either of the following methods to reduce the communication overhead of the data reception application:
 - If you are using the polling method to receive multiple query results, use the `getAll` method to receive all of them in a single batch. This can reduce the communication overhead for data transfer.
 - Create the data reception application as an in-process connection custom adaptor so that it runs in the same process as the SDP server. This can reduce communication overhead between processes.

By reducing the communication overhead, you can prevent tuples from being stranded in the output stream queue, thereby preventing a queue overflow.

- If the output frequency for query results is low, we recommend that you create the data reception application as an in-process connection custom adaptor that runs in the same process as the SDP server, and that you use the callback receive method for receiving data. This can reduce CPU usage.
- When creating an RMI connection custom adaptor, declare the `main` method as follows:

```
public static void main(String[])
```

The command (`sdpstartap`) you executed to start the RMI connection custom adaptor executes the main class, `main` method.

- When creating an RMI connection custom adaptor, do not use 1 as the application termination code.

Chapter

7. APIs for Sending and Receiving Data

This chapter explains the syntax of the APIs for sending and receiving data, used when creating custom adaptors.

- 7.1 Syntax of APIs for sending and receiving data
- 7.2 List of APIs for sending and receiving data
- 7.3 SDPConnector interface (common API)
- 7.4 SDPConnectorFactory class (for RMI connection)
- 7.5 StreamEventListener interface (for in-process connection)
- 7.6 StreamInprocessUP interface (for in-process connection)
- 7.7 StreamInput interface (common API)
- 7.8 StreamOutput interface (common API)
- 7.9 StreamTime class (common API)
- 7.10 StreamTuple class (common API)
- 7.11 Exception class (common API)

7.1 Syntax of APIs for sending and receiving data

This section explains the syntax of the APIs for sending and receiving data.

The items listed below are explained for each API. Some of the items are not explained for some of the APIs.

Format

Shows the API's syntax.

Explanation

Explains the API's function.

Parameters

Explains the parameters used with the API.

Exceptions

Explains the exceptions that can occur when using the API.

Return value

Explains the API's return value.

Notes

Explains other information you might need related to using the API.

7.2 List of APIs for sending and receiving data

The APIs for sending and receiving data can be classified into the three types described below. Use the one best suited to the custom adaptor being created.

- Common APIs

These APIs are used for creating both RMI connection custom adaptors and in-process connection custom adaptors.

- RMI connection API

This API is used for creating RMI connection custom adaptors.

- In-process connection APIs

These APIs are used for creating in-process connection custom adaptors.

The interfaces and classes of the APIs for sending and receiving data are listed below.

Table 7-1: List of APIs for sending and receiving data (interface)

No.	Package name	Interface name	Function	Type
1	jp.co.Hitachi. soft.sdp.api	<i>SDPConnector</i>	Interface for the connector that attaches a custom adaptor to the SDP server. Used for connecting to an input stream or output stream.	Common API
2		<i>StreamInput</i>	Interface used for sending stream data to the SDP server and for disconnecting from the input stream	Common API
3		<i>StreamOutput</i>	Interface used for receiving query results from the SDP server, for registering or unregistering listener objects for callback, and for disconnecting from the output stream	Common API
4	jp.co.Hitachi. soft.sdp.api.i nprocess	<i>StreamEventListener</i>	Interface for implementing the process to be used for callbacks	In-process connection API

No.	Package name	Interface name	Function	Type
5		<i>StreamInprocessUP</i>	Interface for implementing the main process, which is equivalent to a custom adaptor's main method. The implementing class of the <code>StreamInprocessUP</code> interface is loaded and executed when the SDP server is started.	In-process connection API

Table 7-2: List of APIs for sending and receiving data (class)

No.	Package name	Class name	Function	Type
1	<code>jp.co.Hitachi.sof t.sdp.api</code>	<i>SDPConnectorFactory</i>	Class for generating an <code>SDPConnector</code> type object	RMI connection API
2	<code>jp.co.Hitachi.sof t.sdp.common.util</code>	<i>StreamTime</i>	Class for specifying date and time	Common API
3	<code>jp.co.Hitachi.sof t.sdp.common.data</code>	<i>StreamTuple</i>	Class for expressing a stream tuple	Common API

These interfaces and classes are explained in alphabetic order below.

7.3 SDPConnector interface (common API)

Explanation

This interface functions as the connector used for attaching to the SDP server. A connector is generated as an `SDPConnector` type object.

A connector is used to attach to a query group's input or output stream that has been defined in the SDP server.

The `openStreamInput` or `openStreamOutput` method of the `SDPConnector` interface is used to connect to a stream. However, a single connector cannot connect to a stream with the same name more than once.

You generate a connector using the following process, depending on the data connection method between the SDP server and a custom adaptor that you use.

- Generating a connector that connects to the SDP server using an RMI connection

Calling the `connect` method of the `SDPConnectorFactory` class generates a connector that attaches to the SDP server using an RMI connection.

- Generating a connector that connects to the SDP server using an in-process connection

When the `sdpstartinpro` command is executed, a connector is generated that attaches to the SDP server using an in-process connection. This adaptor receives the generated `SDPConnector` type object as an `execute` method parameter from the `StreamInprocessUP` interface.

Field

None.

Constructor

None.

Method

The following table shows the `SDPConnector` interface method list.

Return value	Method name	Function
void	<i>close()</i>	Closes the connector.
boolean	<i>isClosed()</i>	Checks whether the connector is closed.
StreamInput	<i>openStreamInput(String group_name,String stream_name)</i>	Connects to the input stream that corresponds to the group name and stream name specified in the parameters. An <code>SDPConnector</code> type object cannot connect to a stream with the same name more than once.

Return value	Method name	Function
StreamOutput	<i>openStreamOutput(String group_name,String stream_name)</i>	Connects to the output stream that corresponds to the group name and stream name specified in the parameters. An <code>SDPConnector</code> type object cannot connect to a stream with the same name more than once.

Notes

None.

close() method

Format`void close()`**Explanation**

Closes the connector.

Parameters

None.

Exceptions

The following table shows the exceptions and the conditions for their generation.

Exception	Generation condition
<code>SDPClientCommunicationException</code>	A communication exception occurred during RMI connection.
<code>SDPClientException</code>	The connector was already closed.

Return value

None.

isClosed() method

Format`boolean isClosed()`**Explanation**

Checks whether the connector is closed.

Parameters

None.

Exceptions

None.

Return value

- true

The connector is closed.

- false

The connector is open.

openStreamInput(String group_name,String stream_name) method

Format

```
StreamInput openStreamInput(String group_name,String
stream_name)
```

Explanation

Connects to the input stream that corresponds to the group name and stream name specified in the parameters. An `SDPConnector` type object cannot connect to a stream with the same name more than once.

Parameters

- group_name

Specifies a query group name.

- stream_name

Specifies a stream name.

Exceptions

The following table shows the exceptions and the conditions for their generation.

Exception	Generation condition
<code>NullPointerException</code>	null is specified for the parameter.
<code>SDPClientCommunicationException</code>	A communication exception occurred during RMI connection.

Exception	Generation condition
<code>SDPClientException</code>	<ul style="list-style-type: none"> • A nonexistent group name or stream name is specified. • The same <code>SDPConnector</code> type object was already used to connect to the same input stream. • A stream that is not an input stream is specified. • The connector was already closed.

Return value

The input stream (`StreamInput` type object) that was connected

`openStreamOutput(String group_name,String stream_name)` method

Format

```
StreamOutput openStreamOutput (String group_name,String
stream_name)
```

Explanation

Connects to the output stream that corresponds to the group name and stream name specified in the parameters. An `SDPConnector` type object cannot connect to a stream with the same name more than once.

Parameters

- `group_name`
Specifies a query group name.
- `stream_name`
Specifies a stream name.

Exceptions

The following table shows the exceptions and the conditions for their generation.

Exception	Generation condition
<code>NullPointerException</code>	null is specified for the parameter.
<code>SDPClientCommunicationException</code>	A communication exception occurred during RMI connection.
<code>SDPClientException</code>	<ul style="list-style-type: none"> • A nonexistent group name or stream name is specified. • The same <code>SDPConnector</code> type object was already used to connect to the same output stream. • The connector was already closed.
<code>SDPClientRelationStateException</code>	The specified stream is in the relation state.

Return value

The output stream (`StreamOutput` type object) that was connected

7.4 SDPConnectorFactory class (for RMI connection)

Class hierarchy

```
jp.co.Hitachi.soft.sdp.api.SDPConnectorFactory
```

Explanation

This is a factory class that implements the process for generating an `SDPConnector` type object. Executing the `connect` method in this class generates an `SDPConnector` type object that attaches to the SDP server using RMI communication.

Method

The following table shows the `SDPConnectorFactory` class method list.

Return value	Method name	Function
<code>SDPConnector</code>	<i><code>connect()</code></i>	Generates an <code>SDPConnector</code> type object that is connected to the SDP server using RMI communication.

Notes

The `connect` method is a static method.

connect() method

Format

```
SDPConnector connect()
```

Explanation

Generates an `SDPConnector` type object that connects to the SDP server using RMI communication.

Parameters

None.

Exceptions

The following table shows the exceptions and the conditions for their generation.

Exception	Generation condition
<code>SDPClientCommunicationException</code>	A communication exception occurred during RMI connection.
<code>SDPClientException</code>	Parsing of <code>system_config.properties</code> failed.

Return value

The connector (`SDPConnector` type object) to the SDP server

7.5 StreamEventListener interface (for in-process connection)

Explanation

This is the interface that implements the process for the method that issues a callback when a tuple is generated on the SDP server.

In the custom adaptor, create a class in which the `StreamEventListener` interface is implemented and enter the processing that you want to take place during a callback in the `onEvent` method.

Once the `registerForNotification` method of the `StreamOutput` interface is used to register an object of the class in which the `StreamEventListener` interface is implemented, the registered object's `onEvent` method issues a callback the next time a tuple is generated on the SDP server.

Method

The following table shows the `StreamEventListener` interface method list.

Return value	Method name	Function
void	<i>onEvent(StreamTuple tuple)</i>	This method issues a callback when a tuple is generated on the SDP server.

Notes

The `StreamEventListener` interface inherits the `java.rmi.Remote` class.

onEvent(StreamTuple tuple) method

Format

```
void onEvent(StreamTuple tuple)
```

Explanation

This method is called back when a tuple is generated on the SDP server.

Parameters

- `tuple`
Specifies the generated tuple.

Exceptions

None.

Return value

None.

7.6 StreamInprocessUP interface (for in-process connection)

Explanation

For a custom adaptor to connect to the SDP server using an in-process connection, you must create a class in which the `StreamInprocessUP` interface is implemented.

The `StreamInprocessUP` interface provides the `execute` method and the `stop` method.

For the `execute` method, specify the main process (which is equivalent to the `main` method of the custom adaptor). For the `stop` method, specify the termination process for the custom adaptor.

When the `sdpstartinpro` command is executed to start an in-process connection custom adaptor, the SDP server loads a class in which the `StreamInprocessUP` interface is implemented and generates a thread (custom adaptor execution thread) in which the `execute` method runs. The `execute` method is called from this thread. Specify the implemented class name to be loaded by the SDP server and the path name for the `jar` file that stores the implemented class in the property file `user_app.in-process-connection-custom-adaptor-name.properties`. For details about property files, see the manual *uCosminexus Stream Data Platform - Application Framework Setup and Operation Guide*.

The `stop` method is called from an SDP server thread when the `sdpstopinpro` command is executed to stop the in-process connection custom adaptor.

Method

The following table shows the `StreamInprocessUP` interface method list.

Return value	Method name	Function
void	<code>execute(SDPConnector con)</code>	Defines the custom adaptor's main process. This method is called by the custom adaptor execution thread generated by the SDP server when the <code>sdpstartinpro</code> command is executed.
void	<code>stop()</code>	Defines the custom adaptor's termination process. This method is called by an SDP server thread when the <code>sdpstopinpro</code> command is executed.

Notes

- A user thread started by the `execute` method of the `StreamInprocessUP` interface must be stopped using the `stop` method. If a stop process is not defined in the `stop` method, or a process such as an endless loop that cannot be stopped by the `stop` method included in the user thread, the user thread may not be

stopped.

- If a process such as an endless loop is started by the `execute` method of the `StreamInprocessUP` interface, the execution thread of the `execute` method may also not be stopped in some cases.

execute(SDPConnector con) method

Format

```
void execute(SDPConnector con)
```

Explanation

Defines the custom adaptor's main process.

This method is called by the custom adaptor execution thread generated by the SDP server when the `sdpstartinpro` command is executed.

Parameters

- `con`
Specifies an `SDPConnector` type object.

Exceptions

None.

Return value

None.

stop() method

Format

```
void stop()
```

Explanation

Defines the custom adaptor's termination process.

This method is called by an SDP server thread when the `sdpstopinpro` command is executed.

Parameters

None.

Exceptions

None.

Return value

None.

7.7 StreamInput interface (common API)

Explanation

This interface is used by a custom adaptor to send tuples to the SDP server.

The `put` method is used for sending tuples. Two types of `put` methods are available: one for sending multiple tuples in a batch and the other for sending a single tuple.

Method

The following table shows the `StreamInput` interface method list.

Return value	Method name	Function
void	<i>close()</i>	Closes the connection to the input stream queue.
int	<i>getFreeQueueSize()</i>	Acquires the amount of free space in the input stream queue.
int	<i>getMaxQueueSize()</i>	Acquires the maximum size of the input stream queue.
boolean	<i>isStarted()</i>	In the data source mode, this method checks whether the input stream has started accepting new stream data.
void	<i>put(ArrayList<StreamTuple> tuple_list)</i>	Sends multiple tuples.
void	<i>put(StreamTuple tuple)</i>	Sends a single tuple.
void	<i>putEnd()</i>	Notifies the input stream that the stream data input has ended. After an input completion notification is sent by this method and before a query group is resumed, if the <code>put</code> or <code>putEnd</code> method is executed, an exception is returned.

Notes

None.

close() method

Format

```
void close()
```

Explanation

Close the connection to an input stream.

Parameters

None.

Exceptions

The following table shows the exception and the condition for its generation.

Exception	Generation condition
<code>SDPClientException</code>	The input stream was already closed.

Return value

None.

getFreeQueueSize() method**Format**

```
int getFreeQueueSize()
```

Explanation

Acquires the amount of free space in the input stream queue.

The free space in the input stream queue is determined by subtracting the amount being used from the value specified in the `engine.maxQueueSize` parameter of `system_config.properties`.

Parameters

None.

Exceptions

The following table shows the exceptions and the conditions for their generation.

Exception	Generation condition
<code>SDPClientCommunicationException</code>	A communication exception occurred during RMI connection.
<code>SDPClientException</code>	The input stream is already closed.
<code>SDPClientQueryGroupHoldException</code>	The query group is held. (Detailed exception of <code>SDPClientQueryGroupStateException</code>)
<code>SDPClientQueryGroupNotExistException</code>	The query group was deleted.
<code>SDPClientQueryGroupStateException</code>	The query group is not being executed.

Exception	Generation condition
<code>SDPClientQueryGroupStopException</code>	The query group is stopped. (Detailed exception of <code>SDPClientQueryGroupStateException</code>)

Return value

Amount of free space in the input stream queue (`int`)

getMaxQueueSize() method**Format**

```
int getMaxQueueSize()
```

Explanation

Acquires the maximum size of the input stream queue.

The maximum size of the input stream queue is the value specified in the `engine.maxQueueSize` parameter of `system_config.properties`.

Parameters

None.

Exceptions

The following table shows the exceptions and the conditions for their generation.

Exception	Generation condition
<code>SDPClientException</code>	The input stream is already closed.
<code>SDPClientCommunicationException</code>	A communication exception occurred during RMI connection.

Return value

The maximum size of the input stream queue (`int`)

isStarted() method**Format**

```
boolean isStarted()
```

Explanation

In the data source mode, this method checks whether the input stream has started accepting new stream data.

Parameters

None.

Exceptions

The following table lists the exceptions.

Exception	Generation condition
<code>SDPClientCommunicationException</code>	A communication exception occurred during RMI connection.
<code>SDPClientException</code>	<ul style="list-style-type: none"> The input stream is already closed. The stream is running in the server mode.
<code>SDPClientQueryGroupHoldException</code>	The query group is held.
<code>SDPClientQueryGroupNotExistException</code>	The query group was deleted.
<code>SDPClientQueryGroupStateException</code>	The query group is not being executed.
<code>SDPClientQueryGroupStopException</code>	The query group is stopped.

Return value

- `true`

The input stream has started to accept stream data.

- `false`

The input stream has not started to accept stream data.

`put(ArrayList<StreamTuple> tuple_list)` method

Format

```
void put(ArrayList<StreamTuple> tuple_list)
```

Explanation

Sends multiple tuples.

Parameters

- `tuple_list`

Specifies a list of tuples to be sent (`StreamTuple` type object).

Exceptions

The following table shows the exceptions and the conditions for their generation.

Exception	Generation condition
<code>ClassCastException</code>	An element other than <code>StreamTuple</code> was included in the list in the parameter.
<code>NullPointerException</code>	<ul style="list-style-type: none"> • <code>null</code> or a list containing <code>null</code> was specified for the parameter. • A <code>StreamTuple</code> type object containing <code>null</code> was specified in one of the data object arrays constituting an element of the tuple list.
<code>SDPClientCommunicationException</code>	A communication exception occurred during RMI connection.
<code>SDPClientException</code>	<ul style="list-style-type: none"> • The data type of an element in the tuple list is different from the stream definition. • The input stream was already closed.
<code>SDPClientQueryGroupHoldException</code>	The query group is held.
<code>SDPClientQueryGroupNotExistException</code>	The query group was deleted.
<code>SDPClientQueryGroupStateException</code>	The query group is not being executed.
<code>SDPClientQueryGroupStopException</code>	The query group is stopped.

Return value

None.

put(StreamTuple tuple) method

Format

```
void put(StreamTuple tuple)
```

Explanation

Sends a single tuple.

Parameters

■ `tuple`

Specifies the tuple to be sent (`StreamTuple` type object).

Exceptions

The following table shows the exceptions and the conditions for their generation.

Exception	Generation condition
<code>NullPointerException</code>	<ul style="list-style-type: none"> <code>null</code> is specified for the parameter. A <code>StreamTuple</code> type object containing <code>null</code> was specified in the data object array.
<code>SDPClientCommunicationException</code>	A communication exception occurred during RMI connection.
<code>SDPClientException</code>	<ul style="list-style-type: none"> The data type of the tuple element is different from the stream definition. The input stream was already closed.
<code>SDPClientFreeInputQueueSizeLackException</code>	There is not enough free space in the input stream queue (in this case, the tuple is not loaded into the input stream queue).
<code>SDPClientFreeInputQueueSizeThresholdOverException</code>	The amount of free space in the input stream queue has fallen to or below the threshold specified in the <code>stream.freeInputQueueSizeThreshold</code> parameter (in this case, the tuple is loaded into the input stream queue).
<code>SDPClientQueryGroupHoldException</code>	The query group is held.
<code>SDPClientQueryGroupNotExistException</code>	The query group was deleted.
<code>SDPClientQueryGroupStateException</code>	The query group is not being executed.
<code>SDPClientQueryGroupStopException</code>	The query group is stopped.

Return value

None.

putEnd() method

Format

```
void putEnd()
```

Explanation

Notifies the input stream that there is no more stream data.

After an input completion notification is sent by this method and before a query group is resumed, if the `put` or `putEnd` method is executed, an exception is returned.

Parameters

None.

Exceptions

The following table shows the exceptions and the conditions for their generation.

Exception	Generation condition
<code>SDPClientCommunicationException</code>	A communication exception occurred during RMI connection.
<code>SDPClientException</code>	<ul style="list-style-type: none"> • The input stream is already closed. • The input stream has already stopped accepting stream data.
<code>SDPClientQueryGroupHoldException</code>	The query group is held.
<code>SDPClientQueryGroupNotExistException</code>	The query group was deleted.
<code>SDPClientQueryGroupStateException</code>	The query group is not being executed.
<code>SDPClientQueryGroupStopException</code>	The query group is stopped.

Return value

None.

7.8 StreamOutput interface (common API)

Explanation

This interface is used by a custom adaptor to receive tuples from the SDP server.

The following two types of APIs are available for receiving tuples:

- Polling method
- Callback method

The polling method is used by a custom adaptor to dynamically acquire tuples from the SDP server. The `get` method and `getAll` method are available. The `get` method is for receiving a single tuple, and the `getAll` method is for receiving multiple tuples in a batch.

The callback method is called to passively acquire tuples as they are generated in the SDP server. To receive data using the callback method, you must pre-register an object (listener object) in the SDP server that defines the method to be called by the SDP server when a tuple is generated.

A listener object is an object in which the `StreamEventListener` interface is implemented. The `StreamOutput` interface provides the `registerForNotification` method for registering a listener object in the SDP server. The `unregisterForNotification` method is available to cancel registration of a listener object.

Method

The following table shows the `StreamOutput` interface method list.

Return value	Method name	Function
<code>void</code>	<code>close()</code>	Closes the connection to the output stream.
<code>StreamTuple</code>	<code>get()</code>	Acquires a single tuple registered in the SDP server.
<code>ArrayList<StreamTuple></code>	<code>get(int count)</code>	Acquires the number of tuples specified by the <code>count</code> parameter from the SDP server.
<code>ArrayList<StreamTuple></code>	<code>get(int count, long timeout)</code>	Acquires the number of tuples specified by the <code>count</code> parameter from the SDP server. If there is no result data in the SDP server, this method waits until result data arrives or until the time specified in the <code>timeout</code> parameter elapses.
<code>ArrayList<StreamTuple></code>	<code>getAll()</code>	Acquires all tuples registered in the SDP server.

Return value	Method name	Function
<code>ArrayList<StreamTuple></code>	<i><code>getAll(long timeout)</code></i>	Acquires all tuples registered in the SDP server. If there is no result data in the SDP server, this method waits until result data arrives or until the time specified in the <code>timeout</code> parameter elapses.
<code>int</code>	<i><code>getFreeQueueSize()</code></i>	Acquires the amount of free space in the output stream queue.
<code>int</code>	<i><code>getMaxQueueSize()</code></i>	Acquires the maximum size of the output stream queue.
<code>void</code>	<i><code>registerForNotification(StreamEventListener n)</code></i>	Registers a listener object for callback.
<code>void</code>	<i><code>unregisterForNotification(StreamEventListener n)</code></i>	Cancels a registered listener object to prevent further execution of the callback process.

Notes

None.

close() method

Format`void close()`**Explanation**

Closes the connection to the output stream.

Parameters

None.

Exceptions

The following table shows the exceptions and the conditions for their generation.

Exception	Generation condition
<code>SDPClientCommunicationException</code>	A communication exception occurred during RMI connection.
<code>SDPClientException</code>	The output stream was already closed.

Return value

None.

get() method

Format

```
StreamTuple get()
```

Explanation

Acquires a single tuple registered in the SDP server.

If there are no tuples in the output stream queue, `null` is returned.

If a stop notification has been issued to the query group when this method is called (a data transmission termination notification has been sent by a method, or a query group stop notification has been sent by a command), one of the following processes takes place:

- For the first method call after the stop notification
`null` is returned.
- For the second or subsequent method call after the stop notification
An exception is thrown.

This method is a polling method.

Parameters

None.

Exceptions

The following table shows the exceptions and the conditions for their generation.

Exception	Generation condition
<code>SDPClientCommunicationException</code>	A communication exception occurred during RMI connection.
<code>SDPClientEndOfStreamException</code>	Processing of the transmitted data is finished.
<code>SDPClientException</code>	<ul style="list-style-type: none"> • The output stream is already closed. • A listener object for callback is already registered.
<code>SDPClientQueryGroupHoldException</code>	The query group is held.
<code>SDPClientQueryGroupNotExistException</code>	The query group was deleted.
<code>SDPClientQueryGroupStateException</code>	The query group is not being executed.
<code>SDPClientQueryGroupStopException</code>	There is no result data or the query group is stopped.

Return value

Acquired tuple (`StreamTuple` type object)

get(int count) method

Format

```
ArrayList<StreamTuple> get(int count)
```

Explanation

Acquires the number of tuples specified by the `count` parameter from the SDP server.

If there is no result data in the SDP server, an empty `ArrayList` type object is returned.

If a stop notification has been issued to the query group when this method is called (a data transmission termination notification has been sent by a method, or a query group stop notification has been sent by a command), one of the following processes takes place:

- For the first method call after the stop notification
An empty `ArrayList` type object is returned.
- For the second or subsequent method call after the stop notification
An exception is thrown.

This method is a polling method.

Parameters

- `count`

Specifies the number of data items to be acquired from the SDP server. You can specify between 1 and 1,048,576 data items.

The actual number of data items that will be acquired varies as shown below, depending on the value specified in this parameter and the state of the output stream queue.

Condition	Number that can be acquired
Value specified in <code>count</code> is less than or equal to the number of data items in the output stream queue.	Value specified in the <code>count</code> parameter
Value specified in <code>count</code> is greater than the number of data items in the output stream queue.	Number of data items in the output stream queue
Value specified in <code>count</code> is greater than the maximum size of the output stream queue.	

Exceptions

The following table shows the exceptions and the conditions for their generation.

Exception	Generation condition
<code>SDPClientCommunicationException</code>	A communication exception occurred during RMI connection.
<code>SDPClientEndOfStreamException</code>	Processing of the transmitted data is finished.
<code>SDPClientException</code>	<ul style="list-style-type: none"> • The output stream is already closed. • A listener object for callback is already registered. • A parameter is invalid.
<code>SDPClientQueryGroupHoldException</code>	The query group is held. (Detailed exception of <code>SDPClientQueryGroupStateException</code>)
<code>SDPClientQueryGroupNotExistException</code>	The query group was deleted.
<code>SDPClientQueryGroupStateException</code>	The query group is not being executed.
<code>SDPClientQueryGroupStopException</code>	There is no result data or the query group is stopped. (Detailed exception of <code>SDPClientQueryGroupStateException</code>)

Return value

A tuple list (`ArrayList<StreamTuple>` type object)

`get(int count, long timeout)` method

Format

```
ArrayList<StreamTuple> get(int count, long timeout)
```

Explanation

Acquires the number of tuples specified by the `count` parameter from the SDP server.

If there is no result data in the SDP server, this method waits until result data arrives or until the time specified in the `timeout` parameter elapses. If result data arrives while the method is waiting, an `ArrayList` type object containing the new data is returned. If the time specified in the `timeout` parameter elapses or if an interrupt occurs in the waiting thread, one of the following objects is returned:

- If there is data registered:
An `ArrayList` type object containing the data is returned.
- If there is no data registered:
An empty `ArrayList` type object is returned.

If a stop notification has been issued to the query group when this method is called (a data transmission termination notification has been sent by a method, or a query group stop notification has been sent by a command), one of the following processes takes place:

- For the first method call after the stop notification
An empty `ArrayList` type object is returned.
- For the second or subsequent method call after the stop notification
An exception is thrown.

This method is a polling method.

Parameters

■ `count`

Specifies the number of data items to be acquired from the SDP server. You can specify between 1 and 1,048,576 data items.

The actual number of data items that will be acquired varies as shown below, depending on the value specified in this parameter and the state of the output stream queue.

Condition	Number that can be acquired
Value specified in <code>count</code> is less than or equal to the number of data items in the output stream queue.	Value specified in the <code>count</code> parameter
Value specified in <code>count</code> is greater than the number of data items in the output stream queue.	Number of data items in the output stream queue
Value specified in <code>count</code> is greater than the maximum size of the output stream queue.	

■ `timeout`

Specifies the maximum amount of time (in milliseconds) to wait when there is no data.

Depending on the specified value, one of the following processes takes place:

Specified value	Process that is executed
Negative number	No waiting
0	Waits until result data arrives or until the stream ends.
Positive number	Waits until result data arrives or until the specified time elapses.

Exceptions

The following table shows the exceptions and the conditions for their generation.

Exception	Generation condition
<code>SDPClientCommunicationException</code>	A communication exception occurred during RMI connection.
<code>SDPClientEndOfStreamException</code>	Processing of the transmitted data is finished.
<code>SDPClientException</code>	<ul style="list-style-type: none"> • The output stream is already closed. • A listener object for callback is already registered. • A parameter is invalid.
<code>SDPClientQueryGroupHoldException</code>	The query group is held. (Detailed exception of <code>SDPClientQueryGroupStateException</code>)
<code>SDPClientQueryGroupNotExistException</code>	The query group was deleted.
<code>SDPClientQueryGroupStateException</code>	The query group is not being executed.
<code>SDPClientQueryGroupStopException</code>	There is no result data or the query group is stopped. (Detailed exception of <code>SDPClientQueryGroupStateException</code>)

Return value

A tuple list (`ArrayList<StreamTuple>` type object)

getAll() method

Format

```
ArrayList<StreamTuple> getAll()
```

Explanation

Acquires all tuples registered in the SDP server.

If there is no result data in the SDP server, an empty `ArrayList` type object is returned.

If a stop notification has been issued to the query group when this method is called (a data transmission termination notification has been sent by a method, or a query group stop notification has been sent by a command), one of the following processes takes place:

- For the first method call after the stop notification
An empty `ArrayList` type object is returned.
- For the second or subsequent method call after the stop notification

An exception is thrown.

This method is a polling method.

Parameters

None.

Exceptions

The following table shows the exceptions and the conditions for their generation.

Exception	Generation condition
<code>SDPClientCommunicationException</code>	A communication exception occurred during RMI connection.
<code>SDPClientEndOfStreamException</code>	Processing of the transmitted data is finished.
<code>SDPClientException</code>	<ul style="list-style-type: none"> The output stream is already closed. A listener object for callback is already registered.
<code>SDPClientQueryGroupHoldException</code>	The query group is held.
<code>SDPClientQueryGroupNotExistException</code>	The query group was deleted.
<code>SDPClientQueryGroupStateException</code>	The query group is not being executed.
<code>SDPClientQueryGroupStopException</code>	There is no result data or the query group is stopped.

Return value

A tuple list (`ArrayList<StreamTuple>` type object)

getAll(long timeout) method

Format

```
ArrayList<StreamTuple> getAll(long timeout)
```

Explanation

Acquires all tuples registered in the SDP server.

If there is no result data in the SDP server, this method waits until result data arrives or until the time specified in the `timeout` parameter elapses. If result data arrives while the method is waiting, an `ArrayList` type object containing the new data is returned. If the time specified in the `timeout` parameter elapses or if an interrupt occurs in the waiting thread, one of the following objects is returned:

- If there is data registered:

An `ArrayList` type object containing the data is returned.

- If there is no data registered:

An empty `ArrayList` type object is returned.

If a stop notification has been issued to the query group when this method is called (a data transmission termination notification has been sent by a method, or a query group stop notification has been sent by a command), one of the following processes takes place:

- For the first method call after the stop notification

An empty `ArrayList` type object is returned.

- For the second or subsequent method call after the stop notification

An exception is thrown.

This method is a polling method.

Parameters

- `timeout`

Specifies the maximum amount of time (in milliseconds) to wait when there is no data.

One of the following processes takes place depending on the specified value:

Specified value	Process that is executed
Negative number	No waiting
0	Waits until result data arrives or until the stream ends.
Positive number	Waits until result data arrives or until the specified time elapses.

Exceptions

The following table shows the exceptions and the conditions for their generation.

Exception	Generation condition
<code>SDPClientCommunicationException</code>	A communication exception occurred during RMI connection.
<code>SDPClientEndOfStreamException</code>	Processing of the transmitted data is finished.
<code>SDPClientException</code>	<ul style="list-style-type: none"> • The output stream is already closed. • A listener object for callback is already registered.
<code>SDPClientQueryGroupHoldException</code>	The query group is held. (Detailed exception of <code>SDPClientQueryGroupStateException</code>)
<code>SDPClientQueryGroupNotExistException</code>	The query group was deleted.

Exception	Generation condition
<code>SDPClientQueryGroupStateException</code>	The query group is not being executed.
<code>SDPClientQueryGroupStopException</code>	There is no result data or the query group is stopped. (Detailed exception of <code>SDPClientQueryGroupStateException</code>)

Return value

A tuple list (`ArrayList<StreamTuple>` type object)

`getFreeQueueSize()` method

Format

```
int getFreeQueueSize()
```

Explanation

Acquires the amount of free space in the output stream queue.

The free space in the output stream queue is determined by subtracting the space being used from the value specified in the `engine.maxQueueSize` parameter of `system_config.properties`.

Parameters

None.

Exceptions

The following table shows the exceptions and the conditions for their generation.

Exception	Generation condition
<code>SDPClientCommunicationException</code>	A communication exception occurred during RMI connection.
<code>SDPClientException</code>	The output stream is already closed.
<code>SDPClientQueryGroupHoldException</code>	The query group is held. (Detailed exception of <code>SDPClientQueryGroupStateException</code>)
<code>SDPClientQueryGroupNotExistException</code>	The query group was deleted.
<code>SDPClientQueryGroupStateException</code>	The query group is not being executed.

Return value

Amount of free space in the output stream queue (`int`)

getMaxQueueSize() method

Format

```
int getMaxQueueSize()
```

Explanation

Acquires the maximum size of the output stream queue.

The maximum size of the output stream queue is the value specified in the `engine.maxQueueSize` parameter of `system_config.properties`.

Parameters

None.

Exceptions

The following table shows the exceptions and the conditions for their generation.

Exception	Generation condition
<code>SDPClientCommunicationException</code>	A communication exception occurred during RMI connection.
<code>SDPClientException</code>	The output stream is already closed.

Return value

The maximum size of the output stream queue (`int`)

registerForNotification(StreamEventListener n) method

Format

```
void registerForNotification(StreamEventListener n)
```

Explanation

Registers a listener object for callback.

Once this method is executed, a polling method (`get` or `getAll` method) cannot be executed.

Only one listener object can be registered for each stream. Also, a listener object that is already registered cannot be registered for another stream.

Parameters

- `n`

Specifies a listener object.

Exceptions

The following table shows the exceptions and the conditions for their generation.

Exception	Generation condition
<code>NullPointerException</code>	<code>null</code> is specified for the parameter.
<code>SDPClientException</code>	<ul style="list-style-type: none"> • The output stream is already closed. • A listener object is already registered for a stream. • The specified listener object is already registered for another stream. • This method is called during RMI connection.
<code>SDPClientQueryGroupNotExistException</code>	The query group was deleted.

Return value

None.

unregisterForNotification(StreamEventListener n) method

Format

```
void unregisterForNotification(StreamEventListener n)
```

Explanation

Cancels a registered listener object to prevent further execution of the callback process.

Parameters

- `n`

Specifies a listener object.

Exceptions

The following table shows the exceptions and the conditions for their generation.

Exception	Generation condition
<code>NullPointerException</code>	<code>null</code> is specified for the parameter.

Exception	Generation condition
<code>SDPClientException</code>	<ul style="list-style-type: none">• No listener object is defined.• The output stream is already closed.• This method is called during RMI connection.

Return value

None.

7.9 StreamTime class (common API)

Class hierarchy

```
java.lang.Object
| - java.util.Date
|   - jp.co.Hitachi.soft.sdp.common.util.StreamTime
```

Explanation

The `StreamTime` class is an expansion of the `Date` class.

This class is used to process time information for a tuple.

Method

The following table shows the `StreamTime` class method list.

Return value	Method name	Function
boolean	<i>equals(StreamTime when)</i>	Determines if a tuple's own time matches the time indicated by the <code>StreamTime</code> type object specified in the parameter.
long	<i>getTimeMillis()</i>	Acquires the time (in milliseconds) from a tuple.
int	<i>hashCode()</i>	Acquires the hash code that indicates the tuple.
java.lang. .String	<i>toString()</i>	Converts the time information in a tuple into a string and acquires it.

Notes

None.

equals(StreamTime when) method

Format

```
boolean equals(StreamTime when)
```

Explanation

Determines whether the tuple's own time matches the time indicated by the `StreamTime` type object specified in the parameter.

Parameters

■ `when`

Specifies the `StreamTime` type object to be compared.

Exceptions

The following table shows the exception and the condition for its generation.

Exception	Generation condition
<code>NullPointerException</code>	<code>null</code> is specified for the parameter.

Return value

- `true`

The parameter specifies a `StreamTime` type object and its time matches the tuple's own time.

- `false`

Either the object specified by the parameter is not a `StreamTime` type object or its time does not match the time in the tuple.

`getTimeMillis()` method

Format

```
long getTimeMillis()
```

Explanation

Acquires the time (in milliseconds) from the tuple.

Parameters

None.

Exceptions

None.

Return value

Time (in milliseconds) from the tuple (`long`)

`hashCode()` method

Format

```
int hashCode()
```

Explanation

Acquires the hash code that indicates the tuple.

Parameters

None.

Exceptions

None.

Return valueHash code that indicates the tuple (`int`)

toString() method

Format`java.lang.String toString()`**Explanation**

Converts the time information in the tuple to a string and acquires it.

Parameters

None.

Exceptions

None.

Return valueString expression obtained by converting the time in the tuple to the format shown below (`java.lang.String` type. `Δ` indicates a single-byte space)`aaa Δ bbb Δ cc Δ dd:ee:ff Δ ggg Δ hhhh [timeMillis:iii]`

The following table describes the output.

Output item	Description	Output format (range)
<i>aaa</i>	Day of the week	Sun, Mon, Tue, Wed, Thu, Fri, or Sat
<i>bbb</i>	Month	Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec
<i>cc</i>	Day	A two-digit decimal number (01 to 31)
<i>dd</i>	Hour	A two-digit decimal number (00 to 23)
<i>ee</i>	Minute	A two-digit decimal number (00 to 59)
<i>ff</i>	Second	A two-digit decimal number (00 to 59)

Output item	Description	Output format (range)
<i>ggg</i>	Time zone	The time zone if specified. Spaces if no time zone is specified
<i>hhhh</i>	Year	A four-digit decimal number
<i>iii</i>	The time (in milliseconds) kept by the object	The time (in milliseconds) from the object

An output example follows:

```
"Thu Jan 01 09:00:01 GMT 1970 [timeMillis:1234] "
```

7.10 StreamTuple class (common API)

Class hierarchy

```
java.lang.Object
| -jp.co.Hitachi.soft.sdp.common.data.StreamTuple
```

Explanation

The `StreamTuple` class is used to express tuples.

Constructor

The following table shows the `StreamTuple` class constructor list.

Constructor name	Function
<code>StreamTuple(Object[] dataArray)</code>	Generates a new <code>StreamTuple</code> class instance (<code>StreamTuple</code> type object) in the data object array specified in the parameter.

Method

The following table shows the `StreamTuple` class method list.

Return value	Method name	Function
boolean	<code>equals(Object obj)</code>	Determines whether the data object array and times of the <code>StreamTuple</code> type object specified in the parameter match those in the data object array of the tuple's <code>StreamTuple</code> type object.
Object[]	<code>getDataArray()</code>	Acquires the data object array of the <code>StreamTuple</code> type object.
StreamTime	<code>getSystemTime()</code>	Acquires the system time.
int	<code>hashCode()</code>	Acquires a hash code.
java.lang.String	<code>toString()</code>	Acquires the tuple information from the <code>StreamTuple</code> type object and returns it as a character string.

Notes

The tuple's time is automatically set to the system time when a tuple arrives at the SDP server (`StreamTime` type object). If a tuple is generated by a custom adaptor, the initial value of the time object is set to `null`.

Usage example

In the example shown below, data in a Java data type is used to generate a `StreamTuple` type object.

In this example, the generated tuples have elements that are `Integer` type data and `String` type data.

```
Object[] data = new Object[] {
    new Integer (1),
    new String ("AAA")
};
tuple = new StreamTuple(data);
```

StreamTuple(Object[] dataArray) constructor

Format

```
public StreamTuple(Object[] dataArray)
```

Explanation

Generates a new `StreamTuple` class instance (`StreamTuple` type object) in the data object array specified in the parameter.

Parameters

- `dataArray`

Specifies a data object array that will contain the tuple. Specify data in the Java data type.

Exceptions

The following table shows the exception and the condition for its generation.

Exception	Generation condition
<code>NullPointerException</code>	<code>null</code> is specified for the parameter.

equals(Object obj) method

Format

```
boolean equals(Object obj)
```

Explanation

Determines whether the data object array and times of the `StreamTuple` type object specified in the parameter match those in the data object array of the tuple's `StreamTuple` type object.

Parameters

- `obj`

Specifies the `StreamTuple` type object to be compared.

Exceptions

The following table shows the exception and the condition for its generation.

Exception	Generation condition
<code>NullPointerException</code>	<code>null</code> is specified for the parameter.

Return value

- `true`

All of the tuple content (content and times of data object array) is the same.

- `false`

The tuple content is not the same (`false` is returned if even one item is different).

getDataArray() method

Format

```
Object [] getDataArray()
```

Explanation

Acquires the data object array of the `StreamTuple` type object.

Parameters

None.

Exceptions

None.

Return value

A data object array containing Java data type data (`Object []` type)

getSystemTime() method

Format

```
StreamTime getSystemTime()
```

Explanation

Acquires the system time.

Parameters

None.

Exceptions

None.

Return value

An object that indicates the system time (`StreamTime` type object)

hashCode() method

Format

```
int hashCode()
```

Explanation

Acquires a hash code.

This method allows you to use a hash table provided by the `java.util.Hashtable` class.

Parameters

None.

Exceptions

None.

Return value

The hash code of the `StreamTuple` type object (`int`)

toString() method

Format

```
java.lang.String toString()
```

Explanation

Acquires the tuple information from the `StreamTuple` type object and returns it as a character string. This method converts the following data types into character strings and returns the result:

- Data object array (`Object []` type object)
- System time (`StreamTime` type object)

Parameters

None.

Exceptions

None.

Return value

A character string showing tuple information (`java.lang.String` type)

7.11 Exception class (common API)

Class hierarchy

```

java.lang.Object
├─ java.lang.Throwable
│   └─ java.lang.Exception
│       ├── jp.co.Hitachi.soft.sdp.common.exception.SDPCClientException
│       ├── jp.co.Hitachi.soft.sdp.common.exception.
│       │   SDPCClientFreeInputQueueSizeThresholdOverException
│       └── jp.co.Hitachi.soft.sdp.common.exception.
│           SDPCClientFreeInputQueueSizeLackException

```

SDPCClientException class (common API)

Class hierarchy

```

java.lang.Object
├─ java.lang.Throwable
│   └─ java.lang.Exception
│       └─ jp.co.Hitachi.soft.sdp.common.exception.SDPCClientException

```

Explanation

The exception class returns the content of any exception detected by the SDP server of the stream data processing system to the custom adaptor.

Field

None.

Constructor

None.

Method

Since this class inherits the `java.lang.Exception` class, it can use `java.lang.Exception` class methods such as the `getMessage` method.

Subclasses

The following table shows the subclasses of the `SDPCClientException` class.

Table 7-3: Subclasses of the SDPClientException class

No.	Subclass	Explanation
1	SDPClientQueryGroupException (common API)	This exception class returns a query group exception, detected by the SDP server while the custom adaptor is executing, to the custom adaptor.
2	SDPClientQueryGroupNotExistException (common API)	This exception class is generated if the specified query group does not exist.
3	SDPClientQueryGroupStateException (common API)	This exception class is generated if the specified query group is not being executed.
4	SDPClientQueryGroupHoldException (common API)	This exception class is generated if a method is called for a query group that is being held.
5	SDPClientQueryGroupStopException (common API)	This exception class is generated if a method is called for a stopped query group.
6	SDPClientEndOfStreamException (common API)	This exception class is generated when the input stream data ends.
7	SDPClientRelationStateException (common API)	This exception class is generated when the specified stream is in a relation state.
8	SDPClientCommunicationException (for RMI connection)	This exception class returns an RMI communication exception that occurs in the stream data processing system to the custom adaptor.

The class hierarchy of the subclasses is shown below.

```

jap.co.Hitachi.soft.sdp.common.exception.SDPClientException
├─jap.co.Hitachi.soft.sdp.common.exception.SDPClientQueryGroupException
│   ├──jap.co.Hitachi.soft.sdp.common.exception.SDPClientQueryGroupNotExistException
│   ├──jap.co.Hitachi.soft.sdp.common.exception.SDPClientQueryGroupStateException
│   ├──jap.co.Hitachi.soft.sdp.common.exception.SDPClientQueryGroupHoldException
│   └─jap.co.Hitachi.soft.sdp.common.exception.SDPClientQueryGroupStopException
├─jap.co.Hitachi.soft.sdp.common.exception.SDPClientEndOfStreamException
├─jap.co.Hitachi.soft.sdp.common.exception.SDPClientRelationStateException
└─jap.co.Hitachi.soft.sdp.common.exception.SDPClientCommunicationException

```

SDPClientFreeInputQueueSizeThresholdOverException class (common API)

Class hierarchy

```

java.lang.Object
  ↳ java.lang.Throwable
      ↳ java.lang.Exception
          ↳ jp.co.Hitachi.soft.sdp.common.exception.
              SDPClientFreeInputQueueSizeThresholdOverException
  
```

Explanation

This exception class indicates that the amount of free space in the input stream queue has fallen to or below the threshold specified by the `stream.freeInputQueueSizeThreshold` key of `system_config.properties`.

Field

None.

Constructor

None.

Subclass

None.

SDPClientFreeInputQueueSizeLackException class (common API)

Class hierarchy

```

java.lang.Object
  ↳ java.lang.Throwable
      ↳ java.lang.Exception
          ↳ jp.co.Hitachi.soft.sdp.common.exception.
              SDPClientFreeInputQueueSizeLackException
  
```

Explanation

This exception class indicates that there is insufficient free space in the input stream queue.

Field

None.

Constructor

None.

Subclass

None.

Chapter

8. Sample Programs Using APIs for Sending and Receiving Data

This chapter provides sample programs that use the APIs for sending and receiving data.

After installing the sample programs explained here, you can execute them after a user ID has been registered and a working directory has been created. For details about the procedure up to the point of creating a working directory, see the *uCosminexus Stream Data Platform - Application Framework Setup and Operation Guide*.

- 8.1 Sample program configuration
- 8.2 Sample program for an RMI connection custom adaptor
- 8.3 Sample program for an in-process connection custom adaptor

8.1 Sample program configuration

This section explains the configuration of the sample programs provided by Stream Data Platform - AF.

To execute a sample program, first copy the entire directory *installation-directory*\samples\api\, containing the files shown in the table below, to *working-directory*\samples\api\.

Note that you would use different sample program files depending on the custom adaptor's process configuration

Table 8-1: Sample program configuration

Directory	File	Custom adaptor's process configuration		Explanation
		RMI connection	In-process connection	
query\	Inprocess_QueryTest	N	Y	In-process connection query definition file
	RMI_QueryTest	Y	N	RMI connection query definition file
conf\	Inprocess_QueryGroupTest ^{#1}	N	Y	Property file for in-process connection query group
	jvm_client_options.cfg ^{#2}	Y	N	JavaVM options file for RMI connections
	jvm_options.cfg ^{#3}	Y	Y	JavaVM options file for SDP servers
	logger.properties	Y	Y	Log file output property file
	RMI_QueryGroupTest ^{#4}	Y	N	Property file for RMI connection query group
	system_config.properties	Y	Y	System configuration property file
	user_app.InprocessAPTest.properties ^{#5}	N	Y	In-process connection property file

Directory	File	Custom adaptor's process configuration		Explanation
		RMI connection	In-process connection	
src\samples\	Inprocess_Main.java	N	Y	In-process connection transmission/reception control application
	Inprocess_Receiver.java	N	Y	In-process connection data reception application
	Inprocess_Sender.java	N	Y	In-process connection data transmission application
	RMI_ReceiveTupleTest.java	Y	N	RMI connection data reception application
	RMI_SendTupleTest.java	Y	N	RMI connection data transmission application

Legend:

Y: Used. N: Not used.

#1

In `Inprocess_QueryGroupTest`, the path of the query definition file for in-process connection is set to:

```
querygroup.cqlFilePath=.\samples\api\query\Inprocess_QueryTest
```

#2

In `jvm_client_options.cfg`, the paths to the log file output destination (*working-directory*\logs\) and the class path are set to:

```
SDP_JVM_LOG=-XX:HitachiJavaLog:.\logs\SDPClientVM
SDP_CLASS_PATH=.\samples\api\src
```

#3

In `jvm_options.cfg`, the path of the log file output destination (*working-directory*\logs\) is set to:

```
SDP_JVM_LOG=-XX:HitachiJavaLog:.\logs\SDPServerVM
```

#4

In `RMI_QueryGroupTest`, the path of the query definition file for RMI connection is set to:

```
querygroup.cqlFilePath=.\samples\api\query\RMI_QueryTest
```

#5

In `user_app.InprocessAPTest.properties`, the paths to the main class name and class path for in-process connection are set to:

```
user_app.classname=samples.Inprocess_Main
```

```
user_app.classpath_dir=.\samples\api\src
```

8.2 Sample program for an RMI connection custom adaptor

This section explains how to execute the sample program for an RMI connection custom adaptor, and the details of the sample program.

We recommend that you first run the sample program according to the execution procedure, and then check how the definition files and source files that comprise the sample program have been defined and implemented.

8.2.1 Procedure for executing the sample program for an RMI connection custom adaptor

This subsection explains the procedure for executing the sample program for an RMI connection custom adaptor. Four console windows are used in this procedure. The individual console windows that are used are indicated below in parentheses (()). Open and run a new console window as needed.

1. Move to the working directory (console window 1).

```
cd working-directory
```

2. Copy the entire directory *installation-directory*\samples\api\, including the sample program, and place it directly under the *working-directory* (console window 1).

If you have already built a sample environment under the *working-directory*, this step and step 3 are not required. Proceed to step 4.

```
xcopy /EY installation-directory\samples\api .\samples\api\
```

3. Copy the sample program's *conf* directory and its entire contents to the *conf* directory directly under the working directory (console window 1).

Any property files that are already in *working-directory*\conf will be overwritten. If you want to save them, copy them to another directory before you perform this step.

```
xcopy /Y samples\api\conf .\conf\
```

4. Compile the source files (console window 1).

The source files must be compiled in an environment in which you can execute

the `javac` command.

```
javac# -classpath installation-directory\lib\sdp.jar
samples\api\src\samples\RMI*.java
```

#

Define the required environment variables before you compile as described in 6.5 *Compilation procedure*. Alternatively, specify the path for the environment variables in the `javac` command.

5. Start the SDP server (console window 1).

By default, the SDP server port number is assumed to be 20400.

To change the port number, change the value of the `rmi.serverPort` property in `working-directory\conf\system_config.properties` before starting the server.

```
.\bin\sdpstart
```

6. Register a query group (streams and queries) (console window 2).

Open a new console window and move to the working directory.

```
cd working-directory
```

Then, execute the following command:

```
.\bin\sdpcql RMI_QueryGroupTest
```

7. Start the query group (console window 2).

```
.\bin\sdpcqlstart RMI_QueryGroupTest
```

8. Start the data reception application (console window 3).

Open a new console window and move to the working directory.

```
cd working-directory
```

Then, execute the command shown below. For the `-clientcfg` option, specify the JavaVM options file for RMI connections provided with the sample program.

```
.\bin\sdpstartap -clientcfg .\conf\jvm_client_options.cfg  
samples.RMI_ReceiveTupleTest
```

9. Start the data transmission application (console window 4).

Open a new console window and move to the working directory.

```
cd working-directory
```

Then, execute the command shown below. For the `-clientcfg` option, specify the JavaVM options file for RMI connections provided with the sample program.

```
.\bin\sdpstartap -clientcfg .\conf\jvm_client_options.cfg  
samples.RMI_SendTupleTest
```

10. Check the execution result (console window 3).

Confirm that the values 0 to 9 are displayed as the values for ID1, VAL1, and VAL2 in the console window (console window 3) in which you started the data reception application.

```
Get Data:ID1=0, VAL1=data1:0, VAL2=data2:0, TIME=...  
Get Data:ID1=1, VAL1=data1:1, VAL2=data2:1, TIME=...  
Get Data:ID1=2, VAL1=data1:2, VAL2=data2:2, TIME=...  
:  
Get Data:ID1=9, VAL1=data1:9, VAL2=data2:9, TIME=...
```

11. Stop the query group (console window 2).

```
.\bin\sdpcqlstop RMI_QueryGroupTest
```

12. Stop the SDP server (console window 2).

```
.\bin\sdpstop
```

8.2.2 Query group definition content (RMI connection custom adaptor)

This subsection explains the content of the query group (RMI_QueryGroupTest) registered in step 6 under 8.2.1 *Procedure for executing the sample program for an RMI connection custom adaptor*.

In the property file for an RMI connection query group, two streams (named s1 and s2) and a query (named join) are defined. A tuple is generated from the stream data and is output to an output stream whose stream name is the query name join.

The content of the definition is shown below. Note that the uppercase and lowercase notations are different from the sample.

```
REGISTER STREAM s1(c1 INT, c2 VARCHAR(20));
REGISTER STREAM s2(c1 INT, c2 VARCHAR(20));

REGISTER QUERY join
ISTREAM(SELECT s1.c1 AS ID1, s1.c2 AS VAL1, s2.c2 AS VAL2
FROM s1[ROWS 3], s2[ROWS 2]
WHERE s1.c1 = s2.c1);
```

Data in streams s1 and s2 are joined, and an operation that generates a stream tuple is defined.

8.2.3 Content of the RMI connection data transmission application

This subsection explains the content of the RMI connection data transmission application (RMI_SendTupleTest.java) compiled in step 4 under 8.2.1 *Procedure for executing the sample program for an RMI connection custom adaptor*.

This application sends streams s1 and s2 to the SDP server. The StreamInput interface is used for sending data.

The source code is described below. Comments indicated by // [1], // ... [1] and the like correspond to the numbers in the explanation provided below. Note that these numbered comments are not in the actual sample program. Note also that some comments in the code below may differ from the comments in the actual sample program.

Sample program content

```
package samples;

import jp.co.Hitachi.soft.sdp.common.data.StreamTuple;
import
jp.co.Hitachi.soft.sdp.common.exception.SDPClientException;
import
jp.co.Hitachi.soft.sdp.common.exception.SDPClientQueryGroupHol
```

```

dException;
import
jp.co.Hitachi.soft.sdp.common.exception.SDPClientQueryGroupSto
pException;
import jp.co.Hitachi.soft.sdp.api.SDPConnector;
import jp.co.Hitachi.soft.sdp.api.SDPConnectorFactory;
import jp.co.Hitachi.soft.sdp.api.StreamInput;

public class RMI_SendTupleTest{
    //Specify the name of the stream to be sent.
    private static final String TARGET_STREAM1 = "s1";
    private static final String TARGET_STREAM2 = "s2";
    private static final String TARGET_QUERYGROUP =
"RMI_QueryGroupTest";

    private static SDPConnector connector;
    private static StreamInput streamInput1;
    private static StreamInput streamInput2;
    private static long interval=1000;

    //[1]
    public static void main(String[] args) {
        try {
            //Connect to the SDP server.
            connector = SDPConnectorFactory.connect();
            execute();
        } catch (SDPClientException sce) {
            System.err.println(sce.getMessage());
        }
    }
    //[1]

    //Perform data transmission.
    public static void execute() {
        try {
            //[2]
            //Connect to the stream to be sent.
            streamInput1 =
connector.openStreamInput(TARGET_QUERYGROUP, TARGET_STREAM1);
            streamInput2 =
connector.openStreamInput(TARGET_QUERYGROUP, TARGET_STREAM2);
            //[2]

            //Send data.
            for(int i=0; i<10; i++){
                try{
                    Thread.sleep(interval);
                } catch (InterruptedException e) {

```

```

        e.printStackTrace();
    }

    //[3]
    //Generate an object to be specified for the tuple.
    Object[] data1 = new Object[]{
        new Integer(i),
        new String("data1:"+i)
    };
    Object[] data2 = new Object[]{
        new Integer(i),
        new String("data2:"+i)
    };

    //Set the object in the tuple.
    StreamTuple tuple1 = new StreamTuple(data1);
    StreamTuple tuple2 = new StreamTuple(data2);
    //[3]

    //[4]
    //Send the tuple.
    streamInput1.put(tuple1);
    streamInput2.put(tuple2);
}
//Send a data transmission completion notification.
streamInput1.putEnd();
streamInput2.putEnd();
//[4]
} catch (SDPClientQueryGroupStopException sce) {
    //The query group is stopped (sdpcqlstop command)
    System.err.println(sce.getMessage());
} catch (SDPClientQueryGroupHoldException sce) {
    //The query group is held.
    System.err.println(sce.getMessage());
} catch (SDPClientException sce) {
    System.err.println(sce.getMessage());
}

//[5]
} finally {
    if (!connector.isClosed()) {
        try {
            //Close the connection to the input stream.
            streamInput1.close();
            streamInput2.close();
            //Close the connection to the SDP server.
            connector.close();
        } catch (SDPClientException sce2) {
            System.err.println(sce2.getMessage());
        }
    }
}

```



```

    }
    }
}
//... [5]
}

```

The content of the source code is explained as follows:

1. The `SDPConnectorFactory.connect` method is used to connect to the SDP server and acquire an `SDPConnector` type object.
2. The `SDPConnector` type object is used to call the `openStreamInput` method. This method connects to the input stream that is being transmitted and acquires a `StreamInput` type object.
3. An object is generated from the transmission data and then set to a tuple.
4. The `StreamInput` type object is used to call the `put` method and sends the data (tuples) to the input stream that you connected to in step 2. After all of the data has been transmitted, the `putEnd` method is used to send a transmission completion notification.
5. The `StreamInput` type object is used to call the `close` method and disconnect from the input stream. Afterwards, the `SDPConnector` type object is used to call the `close` method and close `SDPConnector`.

8.2.4 Content of the RMI connection data reception application

This subsection explains the content of the RMI connection data reception application (`RMI_ReceiveTupleTest.java`) compiled in step 4 under *8.2.1 Procedure for executing the sample program for an RMI connection custom adaptor*.

This application receives the result of joining streams `s1` and `s2`. The `StreamOutput` class is used for receiving the data.

The source code is described below. Comments indicated by `// [1]`, `// ... [1]` and the like correspond to the numbers in the explanation provided below. Note that these numbered comments are not in the actual sample program. Note also that some comments in the code below may differ from the comments in the actual sample program.

Sample program content

```

package samples;

import java.util.Iterator;
import java.util.List;

```

```

import jp.co.Hitachi.soft.sdp.api.SDPConnectorFactory;
import jp.co.Hitachi.soft.sdp.api.SDPConnector;
import jp.co.Hitachi.soft.sdp.api.StreamOutput;
import jp.co.Hitachi.soft.sdp.common.data.StreamTuple;
import
jp.co.Hitachi.soft.sdp.common.exception.SDPClientCommunication
Exception;
import
jp.co.Hitachi.soft.sdp.common.exception.SDPClientException;
import
jp.co.Hitachi.soft.sdp.common.exception.SDPClientQueryGroupHol
dException;
import
jp.co.Hitachi.soft.sdp.common.exception.SDPClientQueryGroupSto
pException;
import
jp.co.Hitachi.soft.sdp.common.exception.SDPClientEndOfStreamEx
ception;
import jp.co.Hitachi.soft.sdp.common.util.StreamTime;
/**
 * Acquiring and Displaying Stream Data.
 *
 */

public class RMI_ReceiveTupleTest {
    //Specify the name of the stream to be acquired.
    private static final String STREAM_NAME = "JOIN";
    private static final String TARGET_QUERYGROUP =
"RMI_QueryGroupTest";

    private static SDPConnector connector;
    private StreamOutput streamOutput;
    private boolean run=true;
    private long interval=10;
    public static void main(String[] args) {

        //[1]
        try {
            //Connect to the SDP server.
            connector = SDPConnectorFactory.connect();

            //Begin the receiving process.
            RMI_ReceiveTupleTest receiver = new RMI_ReceiveTupleTest();
            receiver.execute();
        } catch (SDPClientException sce) {
            System.err.println(sce.getMessage());
        }
        //... [1]
    }
}

```

```

}

//Perform the receiving process.
public void execute() {
    try {
        //[2]
        //Connect to the result stream.
        streamOutput =
connector.openStreamOutput(TARGET_QUERYGROUP, STREAM_NAME);
        //[2]

        while (run) {
            try {
                Thread.sleep(interval);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            //[3]
            //Receive tuples.
            List<StreamTuple> tupleList = streamOutput.getAll();
            if (tupleList.isEmpty()) continue;
            //[3]

            for (Iterator iterator = tupleList.iterator();
iterator.hasNext();) {
                StreamTuple tuple = (StreamTuple) iterator.next();

                //[4]
                //Display the received tuples.
                Object[] data = tuple.getDataArray();
                Integer id1 = (Integer)data[0];
                String val1 = (String)data[1];
                String val2 = (String)data[2];
                StreamTime time = tuple.getSystemTime();
                System.out.println("Get Data: ID1="+id1+
                    ", VAL1="+val1+
                    ", VAL2="+val2+
                    ", TIME="+time.toString());
                //[4]
            }
        }
    } catch (SDPClientEndOfStreamException see) {
        //The transmission stream data has ended.
        System.out.println(see.getMessage());
    } catch (SDPClientQueryGroupStopException sce) {
        //The query group is stopped. (sdpcqlstop command)
        System.err.println(sce.getMessage());
    }
}

```

```

    } catch (SDPClientQueryGroupHoldException sce) {
        //The query group is held.
        System.err.println(sce.getMessage());
    } catch (SDPClientCommunicationException sce) {
        System.err.println(sce.getMessage());
    } catch (SDPClientException sce) {
        System.err.println(sce.getMessage());
    } finally {
        if (!connector.isClosed()) {
            //[5]
            try {
                //Close the connection to the output stream.
                streamOutput.close();
                //Close the connection to the SDP server.
                connector.close();
            } catch (SDPClientException sce) {
                System.err.println(sce.getMessage());
            }
            //[5]
        }
    }
}
}
}
}

```

The content of the source code is explained as follows:

1. The `SDPConnectorFactory.connect` method is used to connect to the SDP server and acquire an `SDPConnector` type object.
2. The `SDPConnector` type object is used to call the `openStreamOutput` method. This method connects to the output stream that is receiving data and acquires a `StreamOutput` type object.
3. The `StreamOutput` type object is used to call the `getAll` method and receive all of the tuples in the `StreamTuple` type object from the output stream that you connected to in step 2.
4. The `StreamTuple` type object is used to call the `getDataArray` method and acquires the tuple's object array. In the sample program, the tuple data as retrieved is in the `Integer` and `String` types, and it is output to standard output.
5. The `close` method of the `StreamOutput` type object is used to disconnect from the output stream. Then, the `close` method of the `SDPConnector` type object is used to close `SDPConnector`.

8.3 Sample program for an in-process connection custom adaptor

This section explains how to execute the sample program for an in-process connection custom adaptor, and the details of the sample program.

We recommend that you first run the sample program according to the execution procedure, and then check how the definition files and source files that comprise the sample program have been defined and implemented.

8.3.1 Procedure for executing the sample program for an in-process connection custom adaptor

This subsection explains the procedure for executing the sample program for an in-process connection custom adaptor. Two console windows are used in this procedure. The individual console windows that are used are indicated below in parentheses (()). Open and run a new console window as needed.

1. Move to the working directory (console window 1).

```
cd working-directory
```

2. Copy the entire directory *installation-directory*\samples\api\, including the sample program, and place it directly under the *working-directory* (console window 1).

If you have already built a sample environment directly under the *working-directory*, this step and step 3 are not required. Proceed to step 4.

```
xcopy /EY installation-directory\samples\api .\samples\api\
```

3. Copy the sample program's *conf* directory and its entire contents to the *conf* directory directly under the working directory (console window 1).

Any property files that are already in *working-directory*\conf will be overwritten. If you want to save them, copy them to another directory before you perform this step.

```
xcopy /Y samples\api\conf .\conf\
```

4. Compile the source files (console window 1).

The source files must be compiled in an environment in which you can execute

the `javac` command.

```
javac# -classpath installation-directory\lib\sdp.jar
samples\api\src\samples\Inprocess*.java
```

#

Define the required environment variables before you compile as described in 6.5 *Compilation procedure*. Alternatively, specify the path for the environment variables in the `javac` command.

5. Start the SDP server (console window 1).

```
.\bin\sdpstart
```

6. Register a query group (streams and queries) (console window 2).

Open a new console window and move to the working directory.

```
cd working-directory
```

Then, execute the following command:

```
.\bin\sdpctl Inprocess_QueryGroupTest
```

7. Start the query group (console window 2).

```
.\bin\sdpctlstart Inprocess_QueryGroupTest
```

8. Start the application for in-process connection (console window 2).

```
.\bin\sdpstartinpro InprocessAPTest
```

9. Check the execution result (console window 1).

Confirm that the results are displayed in the console window (console window 1) in which you started the SDP server. Since the data reception application in the sample program uses the polling method, the received results `["ad0", 0]` - `["ad24", 24]` are output to the console window.

```
Receiver : Tuple Get on FILTER1 [ VAL=ad0, ID=0, TIME=... ]
:
Receiver : Tuple Get on FILTER1 [ VAL=ad23, ID=23, TIME=... ]
Receiver : Tuple Get on FILTER1 [ VAL=ad24, ID=24, TIME=... ]
```

10. Stop the application for in-process connection (console window 2).

```
.\bin\sdpstopinpro InprocessAPTest
```

11. Stop the query group (console window 2).

```
.\bin\sdpqqlstop Inprocess_QueryGroupTest
```

12. Stop the SDP server (console window 2).

```
.\bin\sdpstop
```

8.3.2 Query group definition content (in-process connection custom adaptor)

This subsection explains the content of the property file for the in-process connection query group (Inprocess_QueryGroupTest) registered in step 6 under 8.3.1 *Procedure for executing the sample program for an in-process connection custom adaptor.*

In this query group, the name of the stream being transmitted is defined as data0, and the name of the stream that receives the data by means of the polling method is defined as filter1.

The content of the definition is shown below. Note that the uppercase and lowercase notations are different from the sample.

```
REGISTER STREAM data0(name VARCHAR(10), num BIGINT);
REGISTER QUERY filter1 ISTREAM(SELECT * FROM data0[ROWS 1] WHERE data0.num <= 24);
```

8.3.3 Content of the in-process connection transmission/reception control application

This subsection explains the content of the in-process connection transmission/

reception control application (`Inprocess_Main.java`) compiled in step 4 under *8.3.1 Procedure for executing the sample program for an in-process connection custom adaptor*. This program is the main program that controls the application for sending and receiving tuples.

The source code is described below. Comments indicated by `// [1]`, `// ... [1]` and the like correspond to the numbers in the explanation provided below. Note that these comments are not written in the actual sample program. Also, the code explained by these comments may differ in some cases from the code in the actual sample program.

Sample program content

```
package samples;

import jp.co.Hitachi.soft.sdp.api.SDPConnector;
import jp.co.Hitachi.soft.sdp.api.inprocess.StreamInprocessUP;
import
jp.co.Hitachi.soft.sdp.common.exception.SDPClientException;

public class Inprocess_Main implements StreamInprocessUP {

    // Thread object for reception
    Inprocess_Receiver receiver = null;

    // Thread object for transmission
    Inprocess_Sender sender = null;

    //Connector to the SDP server
    SDPConnector connector = null;

    // [1]
    public void execute(SDPConnector sc) {
        // ... [1]

        this.connector = sc;

        // [2]
        // Start the reception thread for polling.
        this.receiver = new Inprocess_Receiver(sc);
        receiver.start();

        // Start the transmission thread.
        this.sender = new Inprocess_Sender(sc);
        sender.start();
        // ... [2]
    }

    // [1]
```



```

public void stop() {
//... [1]

    try {
        //[3]
        // Stop the transmission thread.
        if(sender != null) {
            sender.terminate();
            sender.join();
        }
        // Stop the reception thread.
        if(receiver != null){
            receiver.terminate();
            receiver.join();
        }
    } catch (InterruptedException e) {
        System.err.println("Main      : " + e.getMessage());
    }

    try {
        // Close the connector.
        connector.close();
    } catch (SDPClientException sce) {
        System.err.println("Main      : " + sce.getMessage());
    }
//... [3]
}
}

```

The content of the source code is explained as follows:

1. In this application, the `Inprocess_Main` class is one in which the `StreamInprocessUP` interface is implemented. Therefore, the `execute` and `stop` methods defined in the `StreamInprocessUP` interface are implemented.
2. The `execute` method, which is executed when you start the application, starts the data reception thread and the data transmission thread.
3. The `stop` method, which is executed when you stop the application, stops the data reception thread and the data transmission thread started in step 2. Afterwards, the `close` method of the `SDPConnector` type object is called to close `SDPConnector`.

8.3.4 Content of the in-process connection data transmission application

This subsection explains the content of the in-process connection data transmission application (`Inprocess_Sender.java`) compiled in step 4 under *8.3.1 Procedure*

for executing the sample program for an in-process connection custom adaptor.

Reference note:

The sample program shows an example in which a thread class specifically dedicated for transmission is defined, and the client application's main class (the `Inprocess_Main` class in which the `StreamInprocessUP` interface is implemented) generates this thread to perform the transmission. A different method would be for the `Inprocess_Main` class itself to transmit tuples to the SDP server.

The source code is described below. Comments indicated by `// [1]`, `// . . . [1]` and the like correspond to the numbers in the explanation provided below. Note that these comments are not written in the actual sample program.

Sample program content

```
package samples;

import java.util.ArrayList;

import jp.co.Hitachi.soft.sdp.common.data.StreamTuple;
import
jp.co.Hitachi.soft.sdp.common.exception.SDPClientException;
import
jp.co.Hitachi.soft.sdp.common.exception.SDPClientQueryGroupHoldException;
import
jp.co.Hitachi.soft.sdp.common.exception.SDPClientQueryGroupStopException;
import jp.co.Hitachi.soft.sdp.api.SDPConnector;
import jp.co.Hitachi.soft.sdp.api.StreamInput;

public class Inprocess_Sender extends Thread{

    //[2]
    // Thread's startup status
    private volatile boolean running = true;
    //[... [2]

    //Connector to the SDP server
    private SDPConnector connector;

    // Generate a transmission thread.
    public Inprocess_Sender(SDPConnector c) {
        this.connector = c;
    }
}
```

```

ArrayList<StreamTuple> list = new ArrayList<StreamTuple>();

public void run() {

    final String group_name = "Inprocess_QueryGroupTest";
    final String stream_name = "DATA0";

    // Transmission stream object
    StreamInput si = null;

    // Connect to the transmission stream.
    try {
        si = connector.openStreamInput(group_name, stream_name);
    } catch (SDPClientException sce) {
        System.err.println("Sender      : " + sce.getMessage());
        running = false;
    }

    for(int i = 0; i < 50; i++){
        if(!running) {
            break;
        }

        // Generate the data to be transmitted.
        Object[] data = new Object[]{
            new String("ad"+i),
            new Long(i)
        };

        // Generate a tuple object.
        StreamTuple tuple = new StreamTuple(data);

        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            System.err.println("Sender      : Thread is interrupted");
            e.printStackTrace();
        }

        // Send a single tuple.
        try {
            si.put(tuple);
        } catch (SDPClientQueryGroupStopException sce) {
            //The query group is stopped. (sdpcqlstop command)
            //No action takes place and the processing continues.
        } catch (SDPClientQueryGroupHoldException sce) {
            //The query group is held.
            //No action takes place and the processing continues.
        }
    }
}

```

```

        } catch (SDPClientException sce) {
            System.err.println("Sender    :" + sce.getMessage());
            running = false;
        }
    }

    //[1]
    if(running) {
        try {
            si.putEnd();
        } catch (SDPClientQueryGroupStopException sce) {
            //The query group is stopped. (sdpcqlstop command)
            //No action takes place and the processing continues.
        } catch (SDPClientQueryGroupHoldException sce) {
            //The query group is held.
            //No action takes place and the processing continues.
        } catch (SDPClientException sce) {
            System.err.println("Sender    :" + sce.getMessage());
        }
    }

    try {
        if (si != null) {
            // Close the transmission stream.
            si.close();
        }
    } catch (SDPClientException sce) {
        System.err.println("Sender    :" + sce.getMessage());
    }

}

//[2]
public void terminate() {
    System.out.println("Sender    : terminate called");
    this.running = false;
}

//[... [2]
}
}

```

The content of the source code is explained as follows:

1. The data transmission method is the same as for an RMI connection. However, a data transmission completion notification is sent if the termination condition is not specified by the `terminate` method.
2. The process for stopping the transmission of tuples, that is, the termination condition for the `Inprocess_Sender` thread, is described in the `terminate`

method.

Since the `terminate` method is called from the SDP server thread, the `running` field of the `Inprocess_Receiver` class may be read from two threads at the same time in some cases. Therefore, the `volatile` attribute must be specified for the `running` field.

8.3.5 Content of the in-process connection data reception application (polling method)

This subsection explains the content of the in-process connection data reception application (`Inprocess_Receiver.java`) compiled in step 4 under 8.3.1 *Procedure for executing the sample program for an in-process connection custom adaptor*. In this program, the `Inprocess_Receiver` thread polls the SDP server at 100-millisecond intervals to acquire tuples. This thread is started by the `Inprocess_Main` class, which is a transmission/reception control program.

The source code is described below. Comments indicated by `// [1]`, `// ... [1]` and the like correspond to the numbers in the explanation provided below. Note that these comments are not written in the actual sample program.

Sample program content

```
package samples;

import jp.co.Hitachi.soft.sdp.common.data.StreamTuple;
import
jp.co.Hitachi.soft.sdp.common.exception.SDPClientException;
import
jp.co.Hitachi.soft.sdp.common.exception.SDPClientQueryGroupHold
dException;
import
jp.co.Hitachi.soft.sdp.common.exception.SDPClientQueryGroupSto
pException;
import
jp.co.Hitachi.soft.sdp.common.exception.SDPClientEndOfStreamEx
ception;
import jp.co.Hitachi.soft.sdp.common.util.StreamTime;
import jp.co.Hitachi.soft.sdp.api.SDPConnector;
import jp.co.Hitachi.soft.sdp.api.StreamOutput;

public class Inprocess_Receiver extends Thread {

    //Connector to the SDP server
    private SDPConnector connector;

    // [2]
    // Thread's startup status
    private volatile boolean running = true;
```

```

//... [2]

// Generate a reception thread.
public Inprocess_Receiver(SDPConnector c) {
    this.connector = c;
}

public void run() {

    final String groupName = "Inprocess_QueryGroupTest";
    final String streamName = "FILTER1";

    // Result stream object
    StreamOutput so = null;

    // Connect to the result stream.
    try {
        so = connector.openStreamOutput(groupName, streamName);
    } catch (SDPClientException sce) {
        System.err.println("Receiver : " + sce.getMessage());
        running = false;
    }

    //[1]
    // Start receiving tuples.
    while(running){
    //... [1]
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            System.err.println("Receiver : Thread is interrupted");
            e.printStackTrace();
        }

        // Tuple object
        StreamTuple tuple = null;

        // Acquire tuples based on polling.
        try {
            tuple = so.get();
        } catch (SDPClientEndOfStreamException see) {
            //Transmission stream data has ended.
            System.out.println("Receiver : " + see.getMessage());
            break;
        } catch (SDPClientQueryGroupStopException sce) {
            //The query group is stopped. (sdpcqlstop command)
            System.err.println("Receiver : " + sce.getMessage());
            break;
        }
    }
}

```

```

    } catch (SDPClientQueryGroupHoldException sce) {
        //The query group is held.
        System.err.println("Receiver : " + sce.getMessage());
        break;
    } catch (SDPClientException sce) {
        System.err.println("Receiver : " + sce.getMessage());
        break;
    }

    //Display the received tuples.
    if (tuple != null) {
        Object[] data = tuple.getDataArray();
        String val = (String) data[0];
        Long id = (Long) data[1];
        StreamTime time = tuple.getSystemTime();
        System.out.println("Receiver : Tuple Get on " + streamName
            + " [ VAL="+val+", ID="+id+", TIME="+time.toString()+"
]");
    }

    try {
        if (so != null) {
            // Close the result stream.
            so.close();
        }
    } catch (SDPClientException sce) {
        System.err.println("Receiver : " + sce.getMessage());
    }

}

//[2]
public void terminate() {
    System.out.println("Receiver : terminate called");
    this.running = false;
}
//... [2]
}

```

The content of the source code is explained as follows:

1. The data reception process is executed until a termination condition is specified by the `terminate` method in step 2.
2. The process for stopping the reception of tuples, that is, the termination condition for the `Inprocess_Receiver` thread, is described in the `terminate` method.

Since the `terminate` method is called from the SDP server thread, the `running` field of the `Inprocess_Receiver` class may be read from two threads at the same time in some cases. Therefore, the `volatile` attribute must be specified for the `running` field.

Appendix

A. Reference Material for This Manual

A. Reference Material for This Manual

This appendix provides reference information, including various conventions, for this manual.

A.1 Related publications

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

- *uCosminexus Stream Data Platform - Application Framework Description* (3020-3-V01(E))

This manual provides an overview and a basic understanding of Stream Data Platform - AF.

We recommend that you read this manual to learn about the features and system configurations of Stream Data Platform - AF, and to acquire the basic knowledge needed to set up and operate a system.

- *uCosminexus Stream Data Platform - Application Framework Setup and Operation Guide* (3020-3-V02(E))

This manual explains how to design, set up, and operate Stream Data Platform - AF systems, and it provides details about the functions that can be specified when you set up a system.

We recommend that you read this manual to learn about how to analyze stream data through the design, setup, and operation of Stream Data Platform - AF systems.

- *uCosminexus Stream Data Platform - Application Framework Messages* (3020-3-V04(E))

This manual explains the messages output by Stream Data Platform - AF.

We recommend that you refer to this manual if necessary when a message is output.

A.2 Conventions: Abbreviations for product names

This manual uses the following abbreviations for product names and Java-related terms:

Abbreviation	Full name or meaning
jar	Java™ Archive
Java	Java™

Abbreviation	Full name or meaning
JavaVM	Java Virtual Machine
JDK	Java Development Kit
Stream Data Platform - AF	uCosminexus Stream Data Platform - Application Framework

A.3 Conventions: Acronyms

This manual also uses the following acronyms:

Acronym	Full name or meaning
AP	application program
API	application programming interface
CPU	central processing unit
CQL	Continuous Query Language
FIFO	first-in first-out
HTTP	Hyper Text Transfer Protocol
OS	operating system
RMI	Remote Method Invocation

A.4 Conventions: KB, MB, GB, and TB

This manual uses the following conventions:

- 1 KB (kilobyte) is 1,024 bytes.
- 1 MB (megabyte) is $1,024^2$ bytes.
- 1 GB (gigabyte) is $1,024^3$ bytes.
- 1 TB (terabyte) is $1,024^4$ bytes.

Index

Symbols

* 83, 91

A

abbreviations for products 224
acronyms 225
aggregate function 19, 90
 processing using, example of 22
alias 51, 89
 when specified in relation reference 51
 when specified in select expression 52
ALL 86
API for sending and receiving data 147
 sample program using 197
APIs, list of 149
application connection method 113
AVG 91

B

basic item 42
 how to specify 46
 specifying, in CQL 46

C

callback method 120, 123
carriage return code 47
cast specification 103
character code that can be used in CQL 44
character data 59
 comparison of 63
 that can be compared with numeric data after
 being cast, format of 64
character string
 defined as system reserved word 46
 showing dates 54
 showing time 55
 showing timestamp data 56
character string constant 52, 54

character that can be used in CQL 44
character-string-constant 106
classification, based on how data is sent and
received 113
close() method
 SDPConnector interface 152
 StreamInput interface 163
 StreamOutput interface 171
column specification 50
column specification list 93
column-name 89
common API 149
comparison predicate 100
comparison-operand 101
comparison-operator 101
compilation procedure 145
connect() method [SDPConnectorFactory class] 156
constant 52, 106
 notation for 52, 53
 type of 52
conventions
 abbreviations for products 224
 acronyms 225
 diagrams i
 fonts and symbols i
 KB, MB, GB, and TB 225
 version numbers ii
COUNT 91
CQL 10
 character code that can be used in 44
 character that can be used in 44
 specifying basic items in 46
CQL data, type of 58
CQL format 42
CQL list 71
CQL reference 69
CQL structure 10
CQL syntax, symbols used in explanation of 44
custom adaptor 112

- creating 111
- creating, note on 146
- other processes that may need to be implemented in 126
- positioning of 112
- type of 112
- what you can do when creating 8

D

- data comparison 63
 - note on 63
- data exchange methods for data transmission application and data reception application, using different 115
- data extraction 19
- data identifier 50
 - specification 50
 - valid range of 50
- data manipulation CQL 10, 80
- data manipulation CQL list 71
- data reception application 112
 - detecting trigger for terminating (data processing termination notification) 126
 - detecting trigger for terminating, method of 127
 - using data exchange methods, different from the one for data transmission application 115
- data transmission application 112
 - to prevent input stream queue from overflowing, coding 137
 - using data exchange methods, different from the one for data reception application 115
- data transmission termination notification
 - based on data transmission application calling method (putEnd method) 127
 - by data transmission application, flow for sending 128
- data type
 - general comparison 63
 - that can be compared, combinations of 63
- date
 - character string showing 54
 - comparison of 65
- date data 59

- DAY 97
- decimal constant 54
- DECIMAL type, note on 61
- definition CQL 10, 73
- definition CQL list 71
- delimiter 47
 - type of 47
 - usage example of 48
 - where, can be entered 48
 - where, cannot be entered 49
 - where, is positioned 48
- detecting trigger for terminating data reception application
 - data processing termination notification 126
 - method of 127
- diagram conventions i
- DSTREAM 27, 81

E

- environment variable 145
- equals(Object obj) method [StreamTuple class] 188
- equals(StreamTime when) method [StreamTime class] 183
- exception class 192
- execute(SDPConnector con) method [StreamInprocessUP interface] 161

F

- floating point constant 53
- flow, from introduction to operation 2
- font conventions i
- free format 42
- FROM clause 83

G

- GB meaning 225
- general-aggregate-function 91
- get() method [StreamOutput interface] 172
- get(int count) method [StreamOutput interface] 173
- get(int count, long timeout) method [StreamOutput interface] 174
- getAll() method [StreamOutput interface] 176

getAll(long timeout) method [StreamOutput interface] 177
 getDataArray() method [StreamTuple class] 189
 getFreeQueueSize() method
 StreamInput interface 164
 StreamOutput interface 179
 getMaxQueueSize() method
 StreamInput interface 165
 StreamOutput interface 180
 getSystemTime() method [StreamTuple class] 189
 getTimeMillis() method [StreamTime class] 184
 GROUP BY clause 85

H

hashCode() method
 StreamTime class 184
 StreamTuple class 190
 HAVING clause 85
 HOUR 97

I

in-process connection API 149
 in-process connection custom adaptor 113
 creating 120
 procedure for executing sample program for 211
 sample program for 211
 in-process connection data reception application (polling method), content of 219
 in-process connection data transmission application, content of 215
 in-process connection transmission/reception control application, content of 213
 input relation 11
 generating 12
 input stream 73
 input stream queue 134
 coding data transmission application to prevent from overflowing 137
 preventing, from overflowing 136
 what happens when overflowing 135
 inquiry 80
 integer constant 53
 isClosed() method [SDPConnector interface] 152

isStarted() method [StreamInput interface] 165
 ISTREAM 27, 81

J

jar 145
 Java data, type of 58
 javac 145
 jp.co.Hitachi.soft.sdp.api 149, 150
 jp.co.Hitachi.soft.sdp.api.inprocess 149
 jp.co.Hitachi.soft.sdp.common.data 150
 jp.co.Hitachi.soft.sdp.common.util 150

K

KB meaning 225
 keyword 42, 46
 specification example of 46
 specifying 46

L

limit values applicable to query definitions 66
 line feed code 48
 link 19
 linking process
 example of 20
 using ROWS window together with, note on 24
 listener object 170

M

manual, organization of 5
 mapping 58
 between CQL data types and Java data types 58
 MAX 91
 MB meaning 225
 memory usage, limiting 32
 mesh 32
 mesh interval 32
 MILLISECOND 97
 MIN 91
 MINUTE 97

N

name
 qualification of 49
 specifying 49
 notation for constant 52, 53
 NOW 96
 NOW window 13
 numeric constant 52, 53
 numeric data 58
 comparison of 65
 NUMERIC type, note on 61
 numeric value, specifying 47

O

onEvent(StreamTuple tuple) method
 [StreamEventListener interface] 158
 openStreamInput(String group_name,String
 stream_name) method [SDPConnector interface] 153
 openStreamOutput(String group_name,String
 stream_name) method [SDPConnector interface] 154
 operand 42
 operator 104
 output relation 11
 output stream 74
 output stream data, converting into 26
 output stream queue 134
 preventing, from overflowing 140
 process that prevents, from overflowing 142
 what happens when overflowing 136

P

PARTITION BY 96
 PARTITION BY window 13
 PATH environment variable 145
 polling method 116, 120, 122
 put(ArrayList<StreamTuple> tuple_list) method
 [StreamInput interface] 166
 put(StreamTuple tuple) method [StreamInput
 interface] 167
 putEnd() method [StreamInput interface] 168

Q

qualification 49

query definition 74
 example of 35
 limit values applicable to 66
 note on 66
 sample of 109
 query group stop notification, based on execution of
 sdpcqlstop command 129
 query result data
 receiving (in-process connection custom
 adaptor) 122
 receiving (RMI connection custom
 adaptor) 118
 query, what you can define in 7
 queue overflow
 preventing 134
 what happens when 135

R

RANGE 96
 range variable 51
 RANGE window 12, 32
 REGISTER QUERY clause 74
 REGISTER QUERY_ATTRIBUTE clause 75
 REGISTER STREAM clause 73
 registerForNotification(StreamEventListener n)
 method [StreamOutput interface] 180
 relation 11
 relation expression 81
 relation reference 93
 relation-name 94
 relational operation 19
 type of 19
 relationship
 between operations 11
 to related manuals 3
 RMI connection API 149
 RMI connection custom adaptor 113
 creating 116
 procedure for executing sample program
 for 201
 sample program for 201
 RMI connection data reception application, content
 of 207

RMI connection data transmission application, content of 204
 ROWS 96
 ROWS window 12
 using linking process together with, note on 24
 RSTREAM 28, 81

S

sample program
 configuration of 198
 for in-process connection custom adaptor 211
 for in-process connection custom adaptor, procedure for executing 211
 for RMI connection custom adaptor 201
 for RMI connection custom adaptor, procedure for executing 201
 using API for sending and receiving data 197
 SDPClientException class 192
 subclasses of 193
 SDPClientFreeInputQueueSizeLackException class 194
 SDPClientFreeInputQueueSizeThresholdOverException on class 194
 SDPConnector interface 151
 method list 151
 SDPConnectorFactory class 156
 method list 156
 sdpcqlstop command to stop query group, flow for using 130
 search condition 98
 SECOND 97
 select 19
 SELECT clause 82
 select expression 88
 selection list 88
 sign 103
 single-byte space 47
 specification, based on
 data group, example of 17
 number of data items, example of 15
 time stamp of arriving tuple, example of 17
 time, example of 16
 stop() method [StreamInprocessUP interface] 161

stream clause 80
 stream data
 receiving 116, 120
 sending 116, 120
 sending (in-process connection custom adaptor) 121
 sending (RMI connection custom adaptor) 116
 stream definition 73
 stream operation 26
 example of 28
 type of 27
 stream-name 94
 StreamEventListener interface 158
 method list 158
 StreamInprocessUP interface 160
 method list 160
 StreamInput interface 116, 120, 163
 method list 163
 StreamOutput interface 116, 120, 170
 method list 170
 StreamTime class 183
 method list 183
 StreamTuple class 187
 method list 187
 StreamTuple(Object[] dataArray) constructor 188
 SUM 91
 symbol conventions i
 symbol used in explanation of CQL syntax 44

T

tab code 48
 table ID 51
 TB meaning 225
 term 103
 time
 character string showing 55
 comparison of 65
 time data 60
 time division 32
 specified, definition example of 37
 specifying 32
 time division specification 75
 time information

Index

- controlling, when sending tuples 132
 - specifying, in data source mode 131
 - specifying, in tuples 131
- time specification 97
- timestamp data 60
 - character string showing 56
 - comparison of 65
- toString() method
 - StreamTime class 185
 - StreamTuple class 190
- tuple
 - added to output relation, example of outputting 28
 - deleted from output relation, example of outputting 29
 - in output relation at specified time interval, example of outputting all 30
 - processing after inputting all 134
 - sent from multiple data transmission applications to multiple input streams, processing sequence for 133

U

- UNION clause 86
- unregisterForNotification(StreamEventListener n) method [StreamOutput interface] 181

V

- valid range of data identifiers 50
- value expression 103
- value expression primary 103
- value rounding 61
- version number conventions ii

W

- WHERE clause 84
- window operation 12
 - type of 12
- window specification 95