|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | | |  |
|  | **Huawei StreamCQL Syntax Manual** | |  |
|  | |
| **Issue** | **01** |
| **Date** | **2016-08-15** |
| HUAWEI TECHNOLOGIES CO., LTD. | |

Contents

[1 About This Document 1](#_Toc498933359)

[2 System Function List 2](#_Toc498933360)

[3 Key Concepts Definitions 4](#_Toc498933361)

[4 Process of Developing CQL Applications 5](#_Toc498933362)

[5 CQL Syntax Conventions 7](#_Toc498933363)

[6 Data Types 9](#_Toc498933364)

[6.1 Basic Data Types 9](#_Toc498933365)

[6.2 Null 12](#_Toc498933366)

[7 CQL Syntax Definitions 13](#_Toc498933367)

[7.1 Application Operation Statements 13](#_Toc498933368)

[7.1.1 Drop 13](#_Toc498933369)

[7.1.2 Submit 14](#_Toc498933370)

[7.1.3 Explain 15](#_Toc498933371)

[7.1.4 Load 15](#_Toc498933372)

[7.1.5 Deactive 16](#_Toc498933373)

[7.1.6 Active 16](#_Toc498933374)

[7.1.7 Rebalance 17](#_Toc498933375)

[7.2 Stream Operation Statements 18](#_Toc498933376)

[7.2.1 Create Input Stream 18](#_Toc498933377)

[7.2.2 Create Output Stream 20](#_Toc498933378)

[7.3 Query Statements 23](#_Toc498933379)

[7.3.1 Insert 23](#_Toc498933380)

[7.3.2 SELECT 25](#_Toc498933381)

[7.3.2.1 Select Clause 26](#_Toc498933382)

[7.3.2.2 From Clause 26](#_Toc498933383)

[7.3.2.3 Where Clause 32](#_Toc498933384)

[7.3.2.4 Group By Clause 33](#_Toc498933385)

[7.3.2.5 Having Clause 34](#_Toc498933386)

[7.3.2.6 Order By Clause 34](#_Toc498933387)

[7.3.2.7 Limit Clause 35](#_Toc498933388)

[7.3.2.8 Parallel Clause 36](#_Toc498933389)

[7.3.3 Subquery Clause 36](#_Toc498933390)

[7.3.4 DataSource 37](#_Toc498933391)

[7.3.4.1 Create DataSource 37](#_Toc498933392)

[7.3.4.2 DataSourceQuery 38](#_Toc498933393)

[7.3.5 Customized Operator 40](#_Toc498933394)

[7.3.5.1 Create User Operator 40](#_Toc498933395)

[7.3.5.2 User Operator Query 42](#_Toc498933396)

[7.4 Command Statements 43](#_Toc498933397)

[7.4.1 Set/Get 43](#_Toc498933398)

[7.4.2 Add Jar 44](#_Toc498933399)

[7.4.3 Add File 44](#_Toc498933400)

[7.4.4 Create Function 45](#_Toc498933401)

[7.4.5 Drop Function 46](#_Toc498933402)

[7.4.6 Show Applications 46](#_Toc498933403)

[8 Window 48](#_Toc498933404)

[8.1 rows window 52](#_Toc498933405)

[8.2 range window 54](#_Toc498933406)

[8.3 naturalDay window 58](#_Toc498933407)

[9 Expression 60](#_Toc498933408)

[9.1 Common Expression 64](#_Toc498933409)

[9.2 Expression Constant 65](#_Toc498933410)

[9.3 Special Expression 65](#_Toc498933411)

[9.3.1 Cast 65](#_Toc498933412)

[9.3.2 Case when 66](#_Toc498933413)

[9.3.3 Like 67](#_Toc498933414)

[9.3.4 Between 68](#_Toc498933415)

[9.3.5 In 68](#_Toc498933416)

[9.4 Rules for Converting Expression Data Types 69](#_Toc498933417)

[9.4.1 Operation Expression 69](#_Toc498933418)

[9.4.2 CAST Expression 69](#_Toc498933419)

[10 Customized Interface 71](#_Toc498933420)

[10.1 Data Serialization and Deserialization 71](#_Toc498933421)

[10.2 Streaming Data Obtaining 79](#_Toc498933422)

[10.3 Stream Data Writing 85](#_Toc498933423)

[10.4 UDF Function 90](#_Toc498933424)

[10.5 Data Source 92](#_Toc498933425)

[10.6 Customized Operator 98](#_Toc498933426)

[11 Internal System Interface 104](#_Toc498933427)

[11.1 Serialization and Deserialization 104](#_Toc498933428)

[11.1.1 SimpleSerde 104](#_Toc498933429)

[11.1.2 KeyValueSerDe 105](#_Toc498933430)

[11.1.3 CSVSerDe 106](#_Toc498933431)

[11.1.4 BinarySerDe 107](#_Toc498933432)

[11.2 Data Reading 110](#_Toc498933433)

[11.2.1 KafkaInput 110](#_Toc498933434)

[11.2.2 TCPClientInput 111](#_Toc498933435)

[11.2.3 RamdomGen 111](#_Toc498933436)

[11.3 Data Writing 113](#_Toc498933437)

[11.3.1 KafkaOutput 113](#_Toc498933438)

[11.3.2 TCPClientOutput 114](#_Toc498933439)

[11.3.3 ConsoleOutput 114](#_Toc498933440)

[11.4 Function 115](#_Toc498933441)

[11.4.1 Character Processing Function 115](#_Toc498933442)

[11.4.2 Time Processing Function 118](#_Toc498933443)

[11.4.3 Mathematics Function 125](#_Toc498933444)

[11.4.4 Type Conversion Function 127](#_Toc498933445)

[11.4.5 Aggregation Function 132](#_Toc498933446)

[11.5 Data Source 140](#_Toc498933447)

[11.5.1 RDB Data Source 140](#_Toc498933448)

[12 Other CQL Configuration Parameters 143](#_Toc498933449)

[13 CQL Error Code 147](#_Toc498933450)

[14 CQL Java Development API 153](#_Toc498933451)

[15 CQL Key Word List 154](#_Toc498933452)

# About This Document

Continuous Query Language (CQL) is a query language used for data streams. Compared with traditional SQL, CQL introduces the concept of window. Data is stored in memory so that in-memory computing can be quickly implemented. CQL query results are computing results at a specific moment of data streams.

CQL is a Storm-based SQL query language. It resolves the problem that original Storm APIs are complex and difficult to use and some basic functions are unavailable. CQL improves usability of the Storm component.

During CQL syntax design, it is found that syntax of existing CEP products includes not only SQL statements but requires client code. This forces users to learn client APIs, which improves complexity and difficulty.

The CQL design objective is to use SQL statements and certain commands to execute and release all tasks so that the tasks can be distributed by using SQL interfaces. In this way, client interfaces are unified. Users who are familiar with SQL statements can develop operational CQL programs only by learning certain special CQL syntax, such as syntax of window or stream definitions. This reduces difficulty.

# System Function List

The following table describes system functions.

CQL function list

| Module | Function | Description |
| --- | --- | --- |
| Data type | DataType | Supports the basic types int, String, long, float, double, and boolean, time types time, date, and timestamp, and the decimal digit type. |
| Application operation statement | Create Input Stream | Creates an input stream, including a data capture mode and data deserialization method. |
| Create Output Stream | Creates an output stream, including a data sending mode and data serialization method. |
| Explain | Views and exports an execution plan. |
| Load Application | Loads an execution plan. |
| Submit Application | Submits an application. |
| Drop Application | Deletes an application. |
| Deactive Application | Deactivates an application. |
| Active Application | Deactivates an application. |
| Rebalance Application | Re-allocates an application. |
| Query statement | Select | Query statement. |
| Insert as select | Redirects a query result to another stream. |
| multiInsert | Processes data of a stream and sends the data to other streams. |
| Window | Window. |
| Join | Supports Join types inner, left, right, full, and cross. |
| Group by | Group. |
| Order by | Sorts output data of a same batch. |
| Filter Before Window | Filters data before the data enters a window. |
| Where | Filters data after the data enters a window. |
| Having | Filters aggregated data. |
| Limit | Limits output volumes. |
| DataSource | Data source. |
| User Operator | Customized operator. |
| subQuery | Sub query. |
| Command. | Show applications | Views submitted applications. |
| Set/Get | Set or obtains configured attributes. |
| Add jar | Adds a jar package. |
| Add file | Adds a file. |
| Create function | Registers a function. |
| Drop Function | Deletes a function. |
| Operator | Expression. | Supports And, OR, IS NULL, and NULL.  Supports <','>','<=','>=', '=','==','!=',like, and case when. |
| Customized data serialization rule | Parses data. |
| Customized data capture rule | Obtains various files or data. |
| Customized data recording rule | Sends computing results to different devices. |

# Key Concepts Definitions

* Stream: A set of (infinite) elements, each of which belongs to a same schema. Each element is related to logic time. That is, streams have the tuple and time attributes. Any elements can be expressed in the format of **Element<tuple,Time>**, in which **tuple** includes data structures and content, and **Time** is the logic time of data.
* Window: A way for processing unbounded and streaming events. A window sets an event stream to a static view at a moment for various query operations on database tables. Two types of window are defined on a stream, that is, time-based and row-based windows. Both types of window support slide and tumble.
* Expression: A set of symbols and operators. The CQL parsing engine processes an expression to obtain single values. A simple expression can be a constant, variable, or function. Two or more simple expressions can be combined as a complex expression by using operators.

# Process of Developing CQL Applications

CQL application development is different from development by using SQL statements. In the SQL-statement-based development process, data is imported and then queried, and the SQL engine is executed each time when a SQL statement is submitted and obtains a query result. In the process of developing a CQL application, each **Submit** command indicates completion of an application. One application contains at least one input stream, one output stream, and one data computing statement and can contain customized parameters, files, jar packages, and functions. In the CQL application development process, once the **Submit** command is run, the system locally compiles and then submits an application. After the application is submitted, all local user operation records are deleted, including configured parameters and submitted user jar packages. When a CQL statement is submitted: if a syntax error occurs, a user can modify the statement and then submit the statement again. If this happens, entered content will be deleted, and the user must re-submit the entire application. Syntax errors usually occur when an explain or submit statement is run.

The sequence for submitting CQL statements is not strictly specified, but streams (functions, or parameters) must be defined before being used.

The following table describes the general sequence for compiling a CQL application.

| No. | Operation | Description |
| --- | --- | --- |
| 1 | Add a customized file.  Add a customized jar package. | Submit the customized file and jar package.  This step is necessary only when customized content is used. |
| 2 | Customize a function. | Define the customized function.  This step is necessary only when customized content is used. |
| 3 | Create an input stream.  Create an output stream.  Create a customized input stream, output stream, serialization and deserialization. | You can use built-in system functions or customized functions to create input and output streams. |
| 4 | Compile a logic computing statement. | Use a stream to process service computing logic. |
| 5 | Submit an application. | Submits an application. |

The following is a CQL application example:

|  |
| --- |
| *--Add a jar package implemented by using a user-customization interface.*  add jar "/opt/streaming/example/example.jar";  *--Create an input stream by using a user-customization interface.*  CREATE INPUT STREAM S1(C1 STRING, C2 STRING)  SOURCE "com.huawei.streaming.example.userdefined.operator.input.FileInput"  PROPERTIES ("fileinput.path" = "/opt/streaming/example/input.txt");  *--Create an output stream.*  CREATE OUTPUT STREAM rs(C1 STRING,C2 STRING)  SINK kafkaOutput  PROPERTIES("operator.kafka.topic"="ttopic");  *--Application logic.*  insert into stream rs select \* from S1;  *--Submits an application.*  submit application force example; |



Except strings (quoted by single quotation marks or double quotation marks) and application names, other content in a CQL statement are case insensitive, such as key words and parameter names.

# CQL Syntax Conventions

| Convention | Description | Example |
| --- | --- | --- |
| Uppercase | Indicates a CQL syntax keyword. | CREATE |
| *Italic* | Indicates a CQL syntax parameter provided by users. | *stream\_name* |
| |(vertical bar) | Separates syntax items. You can use one of them. | INT |STRING |
| *[]* (italicsquarebrackets) | A symbol used in CQL syntax for defining windows and data sources. You need to differentiate between italicsquare brackets and square brackets. The latter indicates optional syntax. | S *[*ROWS *row\_number* SLIDE [EXCLUDE NOW] *]* |
| [ ] (squarebrackets) | Indicatesoptional syntax. The syntax does not contain square brackets. | *column\_name* [ AS *alias*] |
| ,... | Indicates that the previous item can be repeated. Items are separated by commas (,). | {*column\_name* <data\_type> [ <cql\_comment>]} ,… |
| … | Indicates that the previous item can be repeated. Items are separated by spaces. | <multi\_insert>… |
| ; | Indicates a CQL syntax block terminator. It is used to terminate a syntax definition. | DROP APPLICATION [IF EXISTS] *application\_name;* |
| <label>::= | Indicates the name of a syntax block. It is used to group and mark a too long syntax sectionor syntax unit that can be used in multiple positions in a statement. | <data\_type>::=  INT|STRING; |
| <label> | References a syntax block. | <column\_list> |
| *--Comment* | Describes a syntax block or CQL example functions. | *--Completes yntax definition of an input stream* |

# Data Types

[6.1 Basic Data Types](#_EN-US_TOPIC_0029436558)

[6.2 Null](#_EN-US_TOPIC_0029436549)

## Basic Data Types

Basic CQL data types

| Type | Value Range | Description |
| --- | --- | --- |
| Boolean | True or False | Constant: true or false  For details about serialization and deserialization rules for string output, see related chapters. |
| Int | -231 to 231-1  That is, -2147483648 to 2147483647. | Digits of the Int type are 1, 2, 3, and so on, and negative numbers include -1, -2, and so on. |
| Long | -263 to 263-1  That is,  -9223372036854775808 to 9223372036854775807. | Data of the Long type is in the same format of data of the Int type, and constants are prefixed with L, case insensitive, such as 1L or -1l. |
| Float | 1.4E-45 to 3.4028235E38 | Data of the Float type is in the same format of data of the Int type, but data of the Float type must be prefixed with F, case insensitive, such as -1.0F or 2f. |
| Double | 4.9E-324 to 1.7976931348623157E308 | Data of the Double type is in the same format of data of the Int type, but data of the Double type must be prefixed with D, case insensitive, such as -1.0d or 2D. |
| String |  | Characters of the string type are quoted by using single quotation marks or double quotation marks, such as "abc" or 'abc". |
| Timestamp | Value range of data of the Java.lang.Long type | Input format: yyyy-[M]M-[d]d [H]H:[m]m:[s]s[.SSS][ Z]  In the format, [.SSS] is optional, and its value can be numbers ranging from 1 to 3, accurate to nanosecond (ns).  Output format: yyyy-MM-dd HH:mm:ss.SSS Z,accurate to millisecond.  For details about the formatting methods and parameter meanings, visit:  <http://docs.oracle.com/javase/6/docs/api/java/text/SimpleDateFormat.html>.  Character strings are in the format of'1900-01-01 00:00:00'. CQL supports direct input of strings, and the system converts such strings into a specific type.  The TimeStamp type does not support constant input. |
| Date | Value range of data of the Java.lang.Long type | Date type, including a year (yyyy), month,and day.  Input format: yyyy-[M]M-[d]d  Output format: yyyy-MM-dd  For details about the formatting methods and parameter meanings, visit:  <http://docs.oracle.com/javase/6/docs/api/java/text/SimpleDateFormat.html>.  Character strings are in the format of'1900-01-01'. CQL supports direct input of strings, and the system converts such strings into a specific type.  The Date type adopts GMT+00:00 in CQL internal computing.  The Date type does not support constant input. |
| Time | Value range of data of the Java.lang.Long type | Time in a day, including an hour, minute,and second.  Input format: [H]H:[m]m:[s]s  Output format: HH:mm:ss  For details about the formatting methods and parameter meanings, visit:  <http://docs.oracle.com/javase/6/docs/api/java/text/SimpleDateFormat.html>.  Character strings are in the format of'00:00:00'. CQL supports direct input of strings, and the system converts such strings into a specific type.  The Time type adopts GMT+00:00 in CQL internal computing.  The Time type does not support constant input. |
| Decimal | A number of the Decimal type consists of a non-scale integer of any precision and a 32-digit scale integer. For zero or a positive number, the scale is the number of digits next to the decimal point. For a negative number, the matching formula is as follows: unscaledValue ?? 10-scale. | Computing for amount data types does not cause precision loss.  Constants must be prefixed with BD, case insensitive, such as 1BD or -1bd. |



Definitions of all data types in CQL statements are case insensitive, such as int, Int, iNt, and INT.

If built-in CQL serialization is used for input and output of all data types, input numbers do not need to be prefixed with F, D, or BD. Constants in CQL expressions must be prefixed.

Ranges of data of the Floating and Double types adopt scientific notation, such as 1.99714E13=19971400000000.

Values +0, -0, and +0 and +0.0 and -0.0 are not differentiated. That is, +0, -0, and +0 indicate 0, and +0.0 and -0.0 indicate 0.0.

When strings are formatted to data of the Floating or Double type, such data can be suffixed with F or D respectively. However, output data of the Float or Double type is not suffixed with F or D respectively.

The system does not check whether data overflows during data deserialization, and therefore, users must ensure that entered data is within the scope of a specific type.

In the time zone settings, +**0000** and -**0000** indicate the same time zone.



Computing and expression of data of the Float and Double types will cause precision loss, and therefore, data of the Float and Double types is not recommended for high-precision computing. Instead, data of the Decimal type is recommended.

## Null

NULL:

The null value does not belong to any data types. Data of any types may have the null value.

# CQL Syntax Definitions

[7.1 Application Operation Statements](#_EN-US_TOPIC_0029436561)

[7.2 Stream Operation Statements](#_EN-US_TOPIC_0029436552)

[7.3 Query Statements](#_EN-US_TOPIC_0029436530)

[7.4 Command Statements](#_EN-US_TOPIC_0029436494)

## Application Operation Statements

### Drop

* **Syntax**

|  |
| --- |
| DROP APPLICATION [IF EXISTS] *application\_name;* |

* **Parameter**

*application\_name*: indicates the name of an application. The value can contain letters, digits, or underscores (\_).

* **Function**

Deletes a system application.

If IF EXISTS is specified, the system asks whether the application to be deleted exists. If the application exists, the system deletes the application; otherwise, the system does not delete the application.

If IF EXISTS is not specified, an error occurs if the application to be deleted does not exist.



Users can delete all applications that the users can view in either a security or non-security environment.

* **Example**

|  |
| --- |
| DROP APPLICATION transformEvents; |
| DROP APPLICATION IF EXISTS transformEvents; |

### Submit

* **Syntax**

|  |
| --- |
| SUBMIT APPLICATION [FORCE] *application\_name [path] ;* |

* **Parameter**

*application\_name*: indicates the name of an application. The value can contain letters, digits, or underscores (\_).

*path*: indicates the path to an execution plan file. The value can be quoted with single quotation marks or double quotation marks. This parameter is optional. If this parameter is used, an execution plan is obtained from a specific path and then submitted. The value of appName will overwrite an application name in an execution plan.

* **Function**

The SUBMIT APPLICATION statement submits an application, and then the system starts parsing and releasing the application.

If the statement contains FORCE, the system checks whether the application to be submitted exists. If the application already exists, the system stops the application, and then submits the current application. If the statement does not contain FORCE, the system submits an application without checking whether the application exists. If the application already exists, an error is reported.



Once an application is submitted, all configured parameters and submitted SQL statements in the current CQL user thread will be deleted, and the current client connection thread will be reset so that subsequent applications can be submitted.

If remaining system resources are insufficient to meet application process requirements when an application is submitted, the application will be successfully submitted and enters the **active** state. However, other necessary resources will join a resource application queue, and new processes will run the application after being released.

If no system resource is left when an application is submitted, the application will be successfully submitted and enter the **active** state, but no process will run this application. The application will join a resource application queue and wait for a system resource to be released.

In a security environment, users can view applications submitted only by themselves. Therefore, users may fail to view applications of the same names. Administrator users can view all applications. Administrator users are specified by Storm cluster configuration.

If the path parameter is set, all entered content will be deleted, and then a task is submitted.

The value of path can be a relative path in the startup path of an application, that is, in the current path for starting the CQL command line client or CQL process.

* **Example**

|  |
| --- |
| SUBMIT APPLICATION transformEvents; |
| SUBMIT APPLICATION FORCE transformEvents; |

### Explain

* **Syntax**

|  |
| --- |
| EXPLAIN APPLICATION *application\_name* [*path*]; |

* **Parameter**

*application\_name*: indicates the name of an application. The value can contain letters, digits, or underscores (\_).

*path*: indicates the path to an execution plan file. The value can be quoted with single quotation marks or double quotation marks.



1. The value of path can be a relative path in the startup path of an application, that is, in the current path for starting the CQL command line client or CQL process.

* **Function**

Displays query plans of all submitted statements.

Displays the number of processing units for a query operation and displays data output and input for CQL result parsing and query optimization.

Specifies a file path for exporting query plans to specific files.

Submits new CQL statements if submit application is not executed. However, CQL statements, such as query and insertion statements, cannot be repeatedly submitted; otherwise, parsing results may be incorrect.

* **Example**

|  |
| --- |
| EXPLAIN APPLICATION transform '/opt/cqlplan/transform.xml'; |

### Load

* **Syntax**

|  |
| --- |
| LOAD APPLICATION *path;* |

* **Parameter**

*path*: indicates the path to an execution plan file. The value can be quoted with single quotation marks or double quotation marks.

* **Function**

Loads a query plan to memory. By default, all submitted SQL statements and parameter settings will be deleted before a query plan is loaded.

* **Example**

|  |
| --- |
| LOAD APPLICATION '/opt/cqlplan/transform.xml'; |

### Deactive

* **Syntax**

|  |
| --- |
| DEACTIVE APPLICATION *application\_name;* |

* **Parameter**

*application\_name*: indicates the name of an application. The value can contain letters, digits, or underscores (\_).

* **Function**

Deactivates (suspends) an application.



The Deactive operation does not force an application to quit, but the application runs until the next data message is processed. Then an input operator stop receiving messages. If the application contains a time slide window or time tumble window, output data may also be generated. If the application process unexpectedly quits, the Streaming cluster will automatically start the process, but the process is still in the **Inactive** state.

* **Example**

|  |
| --- |
| DEACTIVE APPLICATION transformEvents; |

### Active

* **Syntax**

|  |
| --- |
| ACTIVE APPLICATION *application\_name;* |

* **Parameter**

*application\_name*: indicates the name of an application. The value can contain letters, digits, or underscores (\_).

* **Function**

Activates an application to enable a suspended application to run.

* **Example**

|  |
| --- |
| ACITVE APPLICATION transformEvents; |

### Rebalance

* **Syntax**

|  |
| --- |
| REBALANCE APPLICATION *application\_name* SET WORKER *int\_number;* |

* **Parameter**

*application\_name*: indicates the name of an application. The value can contain letters, digits, or underscores (\_).

*int\_number*: indicates a positive integer.

* **Function**

Re-allocates applications and changes the number of workers of an application. After the number of workers of an application is changed, the application restarts. This command is used only to change the number of workers but it cannot be used to calculate concurrency of operators of each type.



If rebalance fails due to insufficient resources, an error is reported, indicating that user resources are insufficient.

If rebalance fails because the number of resources applied is greater than the maximum number of concurrent resources supported by an application, an error is reported, indicating that user resources are insufficient.

If there are applications of higher priorities in a resource application queue during Rebalance, resources may be occupied by such applications. If this happens, Rebalance is successfully executed, but the number of resources obtained is different from the number of resources applied. In this case, the application is in the Active state and waits for other system resources to be released.

* **Example**

|  |
| --- |
| *--Set the number of processes of an application to 4.*  REBALANCE APPLICATION transformEvents SET WORKER 4; |

## Stream Operation Statements

### Create Input Stream

* **Syntax**

|  |
| --- |
| *--Definition of the entire syntax of an input stream*  CREATE INPUT STREAM *stream\_name*  <column\_list> [ <cql\_comment> ]  [ <serde\_clause> ] <source\_clause>  [ <parallel\_clause> ]  ;  <column\_list>::=  ( <column\_defined> ,… )  ;  *--Name, type, and comments of a data column*  <column\_defined>::=  *column\_name* <data\_type> [ <cql\_comment>]  ;  *--Deserialization component and related attribute configuration*  <serde\_clause>::=  SERDE *serde\_name* [<property\_list>]  ;  *--Component for obtaining source data and related attribute configuration*  <source\_clause>::=  SOURCE *source\_name* [<property\_list>]  ;  *--Setting of the maximum number of concurrent resources*  <parallel\_clause>::=  PARALLEL *int\_number*  ;  *--Data type*  <data\_type>::=  INT  | LONG  | FLOAT  | DOUBLE  | BOOLEAN  | STRING  | DATE  | TIME  | TIMESTAMP  | DECIMAL  ;  *--Attribute list*  <property\_list>::=  PROPERTIES (<property\_pair> , ... )  ;  <property\_pair>::=  *property\_name* = *property\_value*  ;  *--Comments of an input or output stream*  <cql\_comment>::=  COMMENT *comment\_string*  ; |

* **Parameter**

*stream\_name*: indicates the name of a stream. The value is a string, without being quoted by single quotation marks or double quotation marks.

*column\_name*: indicates the name of a column. The value is a string, without being quoted by single quotation marks or double quotation marks.

*serde\_name*: indicates a serialization name. If serialization is internally implemented in CQL, the name can be an abbreviation, such as SimpleSerDe. If serialization is customized serialization, the name must be quoted by single quotation marks or double quotation marks.

*source\_name*: indicates an input class name. If a source is internally implemented in CQL, the name can be an abbreviation, such as KafkaInput. If a source is customized source, the name must be quoted by single quotation marks or double quotation marks.

*int\_number*: indicates a positive integer.

*property\_name*: indicates the name of an attribute. The value is a string quoted by single quotation marks or double quotation marks.

*property\_value*: indicates the value of an attribute. The value is a string quoted by single quotation marks or double quotation marks.

*comment\_string*: indicates comment content. The value is a string quoted by single quotation marks or double quotation marks.

* **Function**

Defines the data column name, mode for obtaining data, and data deserialization mode of an input stream.

SERDE: defines a data deserialization mode, that is, the mode for parsing data obtained from inputStream into Schema of an entire stream. Default serialization and deserialization are configured in the system. If the default deserialization is used, the SERDE substatement and related attribute configuration can be ignored.

SOURCE: defines the mode for obtaining data. For example, data is obtained from Kafka or a file. The SOURCE statement is mandatory.

ParallelClause: specifies the maximum number of concurrent input operators.

* **Example**

|  |
| --- |
| *--Use system built-in deserialization to obtain operator examples.*  CREATE INPUT STREAM example  (  eventId INT,  eventDesc STRING  )  COMMENT "this is an example of create input stream."  SERDE simpleSerDe  PROPERTIES ("separator" = "|")  SOURCE TCPClientInput  PROPERTIES ( "server" = "127.0.0.1",  "port = "9999" )  Parallel 2; |
| CREATE INPUT STREAM example  (  eventId INT,  eventDesc STRING  )  COMMENT "this is an example of create input stream."  SOURCE "com.huawei.streaming.example.source.localFileReader"  PROPERTIES ("reader.path" = "/opt/streaming/example"); |

### Create Output Stream

* **Syntax**

|  |
| --- |
| *--Definition of the entire syntax of an output stream*  CREATE OUTPUT STREAM *stream\_name*  <column\_list> [ <cql\_comment> ]  [ <serde\_clause> ] <sink\_clause>  [ <parallel\_clause> ]  ;  *--Definition column items*  <column\_list>::=  ( <column\_defined> ,… )  ;  *--Name, type, and comments of a data column*  <column\_defined>::=  *column\_name* <data\_type> [ <cql\_comment>]  ;  *--Deserialization component and related attribute configuration*  <serde\_clause>::=  SERDE *serde\_name* [<property\_list>]  ;  *--Component for recording stream data calculation results and related attribute configuration*  <sink\_clause>::=  SINK *sink\_name* [<property\_list>]  ;  *--Setting of the maximum number of concurrent resources*  <parallel\_clause>::=  PARALLEL *int\_number*  ;  *--Data type*  <data\_type>::=  INT  | LONG  | FLOAT  | DOUBLE  | BOOLEAN  | STRING  | DATE  | TIME  | TIMESTAMP  | DECIMAL  ;  *--Attribute list*  <property\_list>::=  PROPERTIES (<property\_pair> , ... )  ;  <property\_pair>::=  *property\_name* = *property\_value*  ;  *--Comments of an input or output stream*  <cql\_comment>::=  COMMENT *comment\_string*  ; |

* **Parameter**

*stream\_name*: indicates the name of a stream. The value is a string, without being quoted by single quotation marks or double quotation marks.

*column\_name*: indicates the name of a column. The value is a string, without being quoted by single quotation marks or double quotation marks.

*serde\_name*: indicates a serialization name. If serialization is internally implemented in CQL, the name can be an abbreviation, such as SimpleSerDe. If serialization is customized serialization, the name must be quoted by single quotation marks or double quotation marks.

*sink\_name*: indicates an output class name. If a source is internally implemented in CQL, the name can be an abbreviation, such as KafkaOutput. If a source is customized source, the name must be quoted by single quotation marks or double quotation marks.

*int\_number*: indicates a positive integer.

*property\_name*: indicates the name of an attribute. The value is a string quoted by single quotation marks or double quotation marks.

*property\_value*: indicates the value of an attribute. The value is a string quoted by single quotation marks or double quotation marks.

*comment\_string*: indicates comment content. The value is a string quoted by single quotation marks or double quotation marks.

* **Function**

Defines the data column name, mode for writing data, and data serialization mode of an output stream.

SERDE: defined a data serialization mode, that is the mode for serializing calculated stream data into data of a specific format, such as the CSV or binary format. Data columns are the same as those defined in output Schema. Default serialization and deserialization are configured in the system. If the default deserialization is used, the SERDE substatement and related attribute configuration can be ignored.

SINK: defines the mode for recording processed data. For example, data is recorded into Kafka or a file. The SINK statement is mandatory.

<parallel\_clause>: specifies the maximum number of concurrent input operators.

* **Example**

|  |
| --- |
| *--Use a system built-in Kafka message queue as an event output source.*  *--Format data into data of the CSV format and provide CSV data.*  CREATE OUTPUT STREAM transformOutput  (  cnt INT  )  SERDE csvSerDe  SINK kafkaOutput  PROPERTIES ("operator.kafka.topic" = "example")  parallel 2; |
| *--Customized serialization*  *--Submit a customized program in add jar mode before using the program.*  CREATE OUTPUT STREAM transformOutput  (  cnt INT  )  SERDE "com.huawei.streaming.example.serde.binarySerDe"  PROPERTIES ("binary.compress" = "true")  SINK kafkaOutput  PROPERTIES ("operator.kafka.topic" = "example")  parallel 2; |

## Query Statements

### Insert

* **Syntax**

|  |
| --- |
| *--Single insert syntax*  <insert\_clause> <select\_statement>;  <insert\_clause>::=  INSERT INTO STREAM *stream\_name*  ;  *--Definition of syntax of the select statement*  <selectStatement>::=  [ <select\_clause> ]  [ <from\_clause> ]  [ <where\_clause> ]  [ <groupby\_clause> ]  [ <having\_clause> ]  [ <orderby\_clause> ]  [ <limit\_clause> ]  [ <parallel\_clause> ]  ; |
| *-- MultiInsert syntax*  *--Send data in a stream to different streams based on different processing rules.*  *--From statements allow simple streams only, and other syntax elements, such as window, are forbidden.*  <from\_clause> <multi\_insert> … [ <parallel\_clause> ]  ;  *--MultiInsert body syntax*  <multi\_insert>::=  <insert\_clause> <multi\_select>  ;  *-- Definition of syntax of the select statement in MultiInsert scenarios*  *--Compared with other select statements, this select statement does not contain only From and parallel substatements.*  <multi\_select>::=  [ <select\_clause> ]  [ <where\_clause> ]  [ <groupby\_clause> ]  [ <having\_clause> ]  [ <orderby\_clause> ]  [ <limit\_clause> ]  ; |

* **Parameter**

*stream\_name*: indicates the name of a stream. The value is a string, without being quoted by single quotation marks or double quotation marks.

<select\_statement>: indicates a select substatement. For details, see 7.3.2 SELECT.

<parallel\_clause>: indicates a substatement for setting concurrency. This substatement is optional.For details, see 7.3.2.8 Parallel Clause.

<select\_clause>: indicates a select query statement that specifies query content. This statement is mandatory. For details, see 7.3.2.1 Select Clause.

<where\_clause>: indicates a where condition filtering substatement. This substatement is optional. For details, see 7.3.2.3 Where Clause.

<groupby\_clause>: indicates a group by substatement for grouping. This substatement is optional. For details, see 7.3.2.4 Group By Clause.

<having\_clause>: indicates a filtering substatement after having convergence. This substatement is optional. For details, see 7.3.2.5 Having Clause.

<orderby\_clause>: indicates an order by sorting substatement. This substatement is optioal. For details, see 7.3.2.6 Order By Clause.

<limit\_clause>: indicates an output limitation substatement. This substatement is optional. For details, see 7.3.2.7 Limit Clause.

* **Function**

Imports data into an undefined stream. If data of multiple streams needs to be imported into a new stream, names and types of schema columns generated by multiple import statements must be the same.

Imports select results into a stream that does not exist. The system automatically creates this stream, which is neither an input stream nor an output stream.

The MultiInsert statement is usually used to split data in a stream. The statement distributes data in a stream to specific streams based on different processing rules after data processing so that one stream changes to multiple streams.

The MultiInsert statement has the From statement only, which allows only definitions of simple streams rather than complex syntax such as window.

* **Example**

|  |
| --- |
| INSERT INTO STREAM transformTemp SELECT \* FROM transform; |
| *--MultiInsert statement that does not contain sub query*  FROM teststream  INSERT INTO STREAM s1 SELECT \*  INSERT INTO STREAM s2 SELECT a  INSERT INTO STREAM s3 SELECT id, name WHERE id> 10  Parallel 4; |
| *--MultiInsert statement that contains sub query*  FROM  (  SELECT count(id) as id, 'sss' as name  FROM testStream(id>5 )[RANGE 1 SECONDS SLIDE]  GROUP BY ss  )  INSERT INTO STREAM s1 SELECT \*  INSERT INTO STREAM s2 SELECT a  INSERT INTO STREAM s3 SELECT id,name WHERE id> 10; |

### SELECT

A Select statement consists of the following substatements:

|  |
| --- |
| *--A select statement must be used together with an insert statement.*  *--Therefore, a syntax block is defined.*  *--Other substatements are also defined in syntax blocks.*  <select\_statement>::=  [ <select\_clause> ]  [ <from\_clause> ]  [ <where\_clause> ]  [ <groupby\_clause> ]  [ <having\_clause> ]  [ <orderby\_clause> ]  [ <limit\_clause> ]  [ <parallel\_clause> ]  ; |

#### Select Clause

* **Syntax**

|  |
| --- |
| <select\_clause>::=  SELECT <select\_list>,…  ;  <select\_list>::=  [ *stream\_name* | *stream\_alias*.] \*  | [ *stream\_name* | *stream\_alias*.] *column\_name* [ [ AS ] *column\_alias* ]  | <expression> [ [ AS ] *column\_alias* ]  ; |

* **Parameter**

*stream\_name*: indicates a stream name, which is the same as the stream name defined in the create input stream and create output stream.

*column\_name*: indicates a column name, which is the same as the column name defined in the create input stream and create output stream.

*stream\_alias*: indicates a stream alias, which is defined in a from substatement and quoted in a select substatement.

*column\_alias*: indicates a column alias, which is defined in a where substatement, having substatement or group by substatement and quoted in a select substatement.

<expression>: indicates a CQL expression substatement.For details, see 9 Expression.

* **Function**

Defines a mode for calculating output data.

The Distinct syntax cannot be directly used in a select substatement. Instead, the syntax can be used only in the UDAF function, such as count(distinct id).

* **Example**

|  |
| --- |
| SELECT \* FROM transform[RANGE UNBOUNDED SLIDE]; |
| SELECT eventid FROM transform[RANGE UNBOUNDED]; |
| SELECT eventid, eventcode1 + transform.eventcde2 FROM transform; |
| SELECT eventid, sum(DISTINCT cde) FROM transform[RANGE UNBOUNDED SLIDE] GROUP BY eventtype; |

#### From Clause

* **Syntax**

|  |
| --- |
| *--From substatement syntax definition*  <from\_clause>::=  FROM<common\_from\_clause> | <join\_clause> |  <datasource\_clause> | <combine\_clause>  ;  *-- Common from substatement syntax definition*  <common\_from\_clause>::=  <common\_from\_stream> [ <filter\_before\_window> ]  [ <window\_source> ] [ [ AS ] *stream\_alias* ]  ;  *--Join substatement syntax definition*  *--Cross join does not contain the on condition.*  *--Only bidirectional join is supported.*  <join\_clause>::=  <common\_with\_unidirection> <common\_join> <common\_with\_unidirection> ON <expression>  | <common\_with\_unidirection> <corss\_join> <common\_with\_unidirection>  ;  *--Unidirection syntax definition for unidirectional join*  <common\_with\_unidirection>::=  <common\_from\_clause> [UNIDIRECTION]  ;  *--Data source reference definition*  <datasource\_clause>::=  <common\_from\_clause> , <datasource\_body>  ;  *--Combine substatement syntax definition*  <combine\_clause>::=  <combine\_body> <combine\_condition>  ;  <common\_from\_stream>::=  *stream\_name* | <subquery>  ;  <common\_join>::=  <inner\_join>  | <left\_join>  | <right\_join>  | <full\_join>  ;  <inner\_join>::=  [ INNER ] JOIN  ;  <left\_join>::=  LEFT [ OUTER ] JOIN  ;  <right\_join>::=  RIGHT [ OUTER ] JOIN  ;  <full\_join>::=  FULL [ OUTER ] JOIN  ;  <corss\_join>::=  , | CROSS JOIN  ;  *--Optional alias quoted by a data source*  <datasource\_body>::=  DATASOURCE *datasource\_name* <datasource\_arguments> *datasource\_alias*  ;  <datasource\_arguments>::=  ( <datasource\_schema> , <datasource\_query> )  ;  *--Schema used in a data source*  <datasource\_schema>::=  SCHEMA ( <column\_defined> ,… )  ;  *--Data source query statement in which all CQL parameters are defined in this query statement*  <datasource\_query>::=  QUERY ( <expression> ,… )  ;  *--Name, type, and comments of a data column*  <column\_defined>::=  *column\_name* <data\_type> [ <cql\_comment> ]  ;  <combine\_body>::=  *stream\_name* ,...  ;  *-- Combine statement condition definition*  <combine\_condition>::=  COMBINE ( <expression>, ... )  ;  *--Sub query definition*  <subquery>::=  ( <select\_statement> )  ;  *--Comments of an input or output stream*  <cql\_comment>::=  COMMENT *comment\_string*  ; |

* **Parameter**

*stream\_name*: indicates a stream name, which is the same as the stream name defined in the create input stream.

*stream\_alias*: indicates a stream alias. which is the same as the stream name defined in the create input stream.

*datasource\_name*: indicates a data source name, which is the same as the data source name defined in an input stream.

*datasource\_alias*: indicates a data source alias. which is the same as the stream name defined in the create input stream.

*column\_name*: indicates a column name, which is the same as the column name defined in the create input stream and create output stream.

*comment\_string*: indicates comment content. The value is a string quoted by single quotation marks or double quotation marks.

<window\_source>: indicates a window syntax block. For details, see 8 Window.

<expression>: indicates a CQL expression substatement.For details, see 9 Expression.

<select\_statement>: indicates a select substatement.For details, see 7.3.2 SELECT.

<data\_type>: indicates a data type syntax block. For details, see 7.2 Stream Operation Statements.

* **Function**

The From substatement quotes streams and subquery. If a substatement is included, the substatement must be renamed.

The Fiterbeforewindow substatement filters data before the data enters window. The Fiterbeforewindow substatement filters data before the where substatement filters data. The FilterBeforeWindow substatement effectively reduces the amount of data that enters window.

The Combine statement combines output results of multiple streams. The statement maps, combines, and provide output of streams based on specific columns. The statement is applicable to scenarios in which one input stream generates multiple middle streams and calculation results of the middle streams must be combined. When the Combine operation is performed, each middle stream must contain a field that can be matched. When the Combine operation is performed, tables must be separated with commas (,), and keywords such as Join cannot be used.

The Join substatement supports inner join, left outer join, right outer join, full outer join, and cross join. If two streams are joined by a comma (<), cross join is executed by default. Mapping rules and output results of Join are similar to those of SQL.

The Join statement processes connections for data in the left-stream and right-stream windows based on certain conditions.

Description about various Join types

| Type | Description | Example |
| --- | --- | --- |
| [INNER] JOIN | Sends combination of streams of both sides that meet conditions. | From S1[range 20 seconds batch]  join S2[range unbounded]  on s1.id=s2.type;  from S1[range 20 seconds batch]  inner join S2[range unbounded]  on s1.id=s2.type; |
| LEFT [OUTER] JOIN | Displays data lines that meet conditions and data lines in left data streams that do not meet conditions. NULL is displayed if no matching item exists in right data streams. | from S1[range 20 seconds batch]  left outer join S2[range unbounded]  on s1.id=s2.type;  from S1[range 20 seconds batch]  left join S2[range unbounded]  on s1.id=s2.type; |
| RIGHT [OUTER] JOIN | Displays data lines that meet conditions and data lines in left data streams that do not meet conditions. NULL is displayed if no matching item exists in right data streams. | from S1[range 20 seconds batch]  right outer join S2[range unbounded]  on s1.id=s2.type;  from S1[range 20 seconds batch]  right join S2[range unbounded]  on s1.id=s2.type; |
| FULL [OUTER] JOIN | Displays data lines that meet conditions and data lines in either the left or right data stream that does not meet conditions. | from S1[range 20 seconds batch]  full outer join S2[range unbounded]  on s1.id=s2.type;  from S1[range 20 seconds batch]  full join S2[range unbounded]  on s1.id=s2.type; |
| CROSS JOIN | Displays Cartesian product of two connected tables. The number of lines in the product is the multiple of numbers of lines in both tables. | From S1[range 20 seconds batch],  S2[range unbounded];  from S1[range 20 seconds batch]  cross join S2[range unbounded]; |



Currently, CQL supports dual-stream Join, equal-value Join, and attribute expressions only. For example, s1 inner join s2 on s1.id=s2.id.

A Unidirection keyword specifies single-direction Join. During internal Join of a stream processing system, Join is triggered by two streams if both streams generate data. If one stream includes unidirection and the other stream generates data, Join is not triggered. Join is triggered only when the stream that includes Unidirection generates data.

The From substatement includes a DataSource syntax definition. For details about DataSource, see related chapters.

* **Example**

|  |
| --- |
| *--Subquery example*  SELECT \*  FROM  (  SELECT id, name, type  FROM transform  ) AS transformbak; |
| *--Combine statement example*  INSERT INTO STREAM realresult  SELECT transform.\*, transDetail.describe  FROM transform, transDetail  COMBINE( transform.eventid, transDetail.eventid ); |
| *--Pre-filtering example*  SELECT \*  FROM transform ( evnetid> 10 )[range UNBOUNDED]  WHERE eventtype =1; |
| *--Data source example*  insert into rs select rdb.id,s.id,count(rdb.id),sum(s.id) from S[rows 10 slide],  DATASOURCE rdbdatasource  [  SCHEMA (id int,name String,type int),  QUERY("select rid as id,rname,rtype from rdbtable where id = ? ",s.id)  ] rdb  where rdb.name like '%hdd%'  group by rdb.id,s.id; |
| *--Inner join example*  insert into stream rs select \* from S1[range 20 seconds batch] join S2[range unbounded] on s1.id=s2.type where s1.id> 5;  insert into stream rs select \* from S1[range 20 seconds batch] inner join S2[range unbounded] on s1.id=s2.type where s1.id> 5; |
| *--Left join example*  insert into stream rs select \* from S1[range 20 seconds batch] left outer join S2[range unbounded] on s1.id=s2.type where s1.id> 5;  insert into stream rs select \* from S1[range 20 seconds batch] left join S2[range unbounded] on s1.id=s2.type where s1.id> 5; |
| *--Right join example*  insert into stream rs select \* from S1[range 20 seconds batch] right outer join S2[range unbounded] on s1.id=s2.type where s1.id> 5;  insert into stream rs select \* from S1[range 20 seconds batch] right join S2[range unbounded] on s1.id=s2.type where s1.id> 5; |
| *--Full join example*  insert into stream rs select \* from S1[range 20 seconds batch] full outer join S2[range unbounded] on s1.id=s2.type where s1.id> 5;  insert into stream rs select \* from S1[range 20 seconds batch] full join S2[range unbounded] on s1.id=s2.type where s1.id> 5; |
| *--Cross join example*  insert into stream rs select \* from S1[range 20 seconds batch], S2[range unbounded] where s1.id> 5;  insert into stream rs select \* from S1[range 20 seconds batch] cross join S2[range unbounded] where s1.id> 5; |

#### Where Clause

* **Syntax**

|  |
| --- |
| <where\_clause>::=  WHERE <expression>  ; |

* **Parameter**

<expression>: indicates a CQL expression substatement.For details, see 9 Expression.

* **Function**

Filtering conditions in the Where statement are used after data enters window and before data aggregation. Therefore, where expressions do not contain aggregation functions.

An expression can be a customized expression, but the return value must be of the Boolean type, such as is null.

Multiple expressions can be connected by AND or other dyadic-comparison expressions.

* **Example**

|  |
| --- |
| WHERE a.size = 1 and a.size + b.size =2; |
| WHERE a.size = 1 and ( a.size + b.size )>= 2; |
| WHERE a.size = 1 and cast(a.isnative as boolean)=true; |

#### Group By Clause

* **Syntax**

|  |
| --- |
| <groupby\_clause>::=  GROUP BY <expression> , ...  ; |

* **Parameter**

<expression>: indicates a CQL expression substatement.For details, see 9 Expression.

* **Function**

Distributes data based on the value of **groupBy** and groups operators in group by mode for aggregation.

The Group by operation is performed before window. That is, after data is grouped in group by mode, the window operation is performed for attributes of data groups.

A group by condition is a UDF expression result. That is, data is grouped based on UDF function results.



The Group by expression supports only data columns in streams but does not support other expressions, such as functions.

* **Example**

|  |
| --- |
| SELECT s.age,count(1) as cc  FROM schagetransformEvent[RANGE 1 SECOND BATCH] as s  GROUP BY s.age,s.departmentid; |

#### Having Clause

* **Syntax**

|  |
| --- |
| <having\_clause>::=  HAVING <expression>  ; |

* **Parameter**

<expression>: indicates a CQL expression substatement.For details, see 9 Expression.

* **Function**

Filters aggregated result data.

The having substatement can be separately defined because a CQL stream query statement contains aggregation window by default.

A having statement generally serves as a filtering condition of an aggregation operation, such as having count(\*)> 10.



Having-based filtering occurs after aggregation, and therefore filtering conditions do not include stream names and column names of an input stream in a From substatement because the original data cannot be found.

* **Example**

|  |
| --- |
| SELECT  schages.age,  count(1) as cnt  FROM schagetransformEvent[RANGE 20 SECONDS BATCH]  GROUP BY schages.age  HAVING schages.age=10 AND cnt=2;  *--The having statement includes the s.age=10 condition, but the s.age=10 condition will be run after count is executed because it is included in the having statement.* |

#### Order By Clause

* **Syntax**

|  |
| --- |
| <orderby\_clause>::=  ORDER BY <orderby\_column>, ...  ;  <orderby\_column>::=  <expression> [ ASC | DESC ]  ; |

* **Parameter**

<expression>: indicates a CQL expression substatement.For details, see 9 Expression.

* **Function**

Sorts one or more fields. Fields are sorted in ascending order by default.

Sorts fields in ASC order.

Sorts fields in DESC order.

If concurrency is not 1 and Orderby is used, fields in an operator are sorted.



Order by occurs after aggregation, and therefore filtering conditions do not include stream names and column names of an input stream in a From substatement because the original data cannot be found.

* **Example**

|  |
| --- |
| SELECT \*  FROM schagetransformEvent[RANGE 20 SECONDS BATCH]  ORDER BY id DESC; |

#### Limit Clause

* **Syntax**

|  |
| --- |
| LIMIT *int\_number;* |

* **Parameter**

*int\_number*: indicates a positive integer.

* **Description**

Limits the maximum volume of output data in a batch.

If multiple pieces of data are generated in a batch, the data volume is limited from the beginning. If the volume of data in a batch does not reach the upper limit, data of the entire batch is generated.

* **Example**

|  |
| --- |
| SELECT \* FROM ticketsRecorder[RANGE 20 SECONDS BATCH] LIMIT 10; |

#### Parallel Clause

* **Syntax**

|  |
| --- |
| PARALLEL *int\_number*; |

* **Parameter**

*int\_number*: indicates a positive integer.

* **Description**

Sets the maximum number of concurrent resources. If the Parallel statement contains a query substatement, subquery concurrency is also affected because concurrency of external subquery overwrites subquery concurrency.

This statement is optional, and the default concurrency is 1.

* **Example**

|  |
| --- |
| SELECT \* FROM ticketsRecorder[RANGE 20 SECONDS BATCH] LIMIT 10 Parallel 2; |
| INSERT INTO STREAM rs  SELECT \*, count(a)  FROM (  SELECT id as a,id as b, id c from S  )[RANGE 20 SECONDS BATCH]  WHERE a> 5  GROUP BY c  PARALLEL 4; |

### Subquery Clause

* **Syntax**

|  |
| --- |
| *--Subquery syntax definition*  <subquery>::=  ( <select\_statement> )  ; |

* **Parameter**

<select\_statement>: indicates a parameter whose definition is the same as the definition of the select substatement. For details, see 7.3.2 SELECT

* **Function**

Subquery can be quoted in a From substatement.

Subquery in a From statement can be quoted only when an alias is specified.

* **Example**

|  |
| --- |
| SELECT a.id  FROM (  SELECT count(tid) as id  FROM schagetransformEvent[RANGE UNBOUNDED]  ) AS a; |

### DataSource

Data sources are containers for storing data, and data sources are not provided in streams. Instead, data in data sources is queried based on conditions in triggering mode.

A data source can serve as a database. Each time an event occurs, data is queried in a data source, and query results are displayed.

Data source usage is classified into data source definitions and data source query.

#### Create DataSource

* **Syntax**

|  |
| --- |
| *--Data source definition syntax*  CREATE DATASOURCE *datasource\_name*  SOURCE *class\_name*  <property\_list>  ;  *--Attribute list*  <property\_list>::=  PROPERTIES (< property\_pair> , ... )  ;  <property\_pair>::=  *property\_name* = *property\_value*  ; |

* **Parameter**

*datasource\_name*: indicates the name of a data source. The value is a string, without being quoted by single quotation marks or double quotation marks.

*class\_name*: indicates the class of a data source. The value is a string quoted by single quotation marks or double quotation marks. If className is internally implemented in CQL, the value can be an abbreviation, such as RDBDatasource.

*property\_name*: indicates the name of an attribute.The value is a string quoted by single quotation marks or double quotation marks.

*property\_value*: indicates the value of an attribute. The value is a string quoted by single quotation marks or double quotation marks.

* **Function**

Provides data source definition syntax. Data source usage is classified into the definition and use phases. This part describes data source definition syntax. For details about data source usage syntax, see definitions of From substatements.

A data source provides a mode for accessing external data, such as accessing databases of the relationship type in real time and accessing components such as the HBase. Data of multiple rows can be obtained from original data of one row.

Data source definitions are irrelevant to data source usage. One data source can be defined in a topology, and the data source can be used in other scenarios. For example, a defined database data source can be used for query by using different query statements in different scenarios.

* **Example**

|  |
| --- |
| CREATE DATASOURCE rdbdatasource  SOURCE RDBDataSource  PROPERTIES (  "url" = "jdbc:postgresql://127.0.0.1:1521/streaming",  "userName" = "55B5B07CF57318642D38F0CEE0666D26",  "password" = "55B5B07CF57318642D38F0CEE0666D26"  ); |

#### DataSourceQuery

* **Syntax**

DataSourceQuery is used together with From substatements. dataource\_body syntax is also defined in From statements.

|  |
| --- |
| *--Optional alias quoted by a data source*  <datasource\_body>::=  DATASOURCE *datasource\_name* <datasource\_arguments> *datasource\_alias*  ;  <datasource\_arguments>::=  ( <datasource\_schema> , <datasource\_query> )  ;  *--Schema used in a data source*  <datasource\_schema>::=  SCHEMA ( <column\_defined> ,… )  ;  *--Data source query statement in which all CQL parameters are defined in this query statement*  <datasource\_query>::=  QUERY ( <expression> ,… )  ;  *--Name, type,and comments of a data column*  <column\_defined>::=  *column\_name* <data\_type> [ <cql\_comment> ]  ;  *--Comments of an input or output stream*  <cql\_comment>::=  COMMENT *comment\_string*  ; |

* **Parameter**

*datasource\_name*: indicates a data source name, which is the same as the name defined in a data source.

*datasource\_alias*: indicates the alias of a data source. The value is a string not quoted by double quotation marks and single quotation marks.

*column\_name*: indicates the name of a column. The value is a string, without being quoted by single quotation marks or double quotation marks.

*comment\_string*: indicates comment content. The value is a string quoted by single quotation marks or double quotation marks.

<expression>: indicates a CQL expression substatement.For details, see 9 Expression.

<data\_type>: indicates a data type syntax block. For details, see 7.2 Stream Operation Statements.

* **Function**

Query parameters of a data source include Schema definition query and data source query parameters.

The Scheme definition is the same as the Scheme definition in Create Input Stream. The definition is used to specify the number of columns, name, and type in data source query results.

* **Example**

|  |
| --- |
| *--RDB data capture, supporting capture of multiple rows of data, CQL UDF, window, and aggregation calculation*  *--Fixed sequence of parameters inside QUERY and parameters varying depending on data sources*  *--If an RDB SQL statement does not include Where, multiple rows of records can be queried in one time. The data source output mode is the same as the output mode of Join.*  insert into rs select rdb.id,s.id,count(rdb.id),sum(s.id) from S[rows 10 slide],  DATASOURCE rdbdatasource  [  SCHEMA (id int,name String,type int),  QUERY("select rid as id,rname,rtype from rdbtable where id = ? ", s.id)  ] rdb  where rdb.name like '%hdd%'  group by rdb.id,s.id; |

### Customized Operator

If SQL statements cannot describe complex user service logic, customized operators can be used. After implementing an interface for customized operators, users can use customized operators, input or output operators, or other convergence operators.

Customized operators support single input schema or single output schema only.

Data source usage includes data source definitions and query.

#### Create User Operator

* **Syntax**

|  |
| --- |
| *--Customized operator syntax*  CREATE OPERATOR *operator\_name* AS *class\_name*  <input\_schema\_statement> <output\_schema\_statement>  [ <property\_list>]  ;  *--Input schema of the customized operator*  <input\_schema\_statement>::=  INPUT ( <column\_list> )  ;  *--Output schema of the customized operator*  <output\_schema\_statement>::=  OUTPUT ( <column\_list> )  ;  *--Data column list definition*  <column\_list>::=  ( <column\_defined> ,… )  ;  *--Name, type, and comments of a data column*  <column\_defined>::=  *column\_name* <data\_type> [ <cql\_comment>]  ;  *--Data type*  <data\_type>::=  INT  | LONG  | FLOAT  | DOUBLE  | BOOLEAN  | STRING  | DATE  | TIME  | TIMESTAMP  | DECIMAL  ;  *--Attribute list*  <property\_list>::=  PROPERTIES (<property\_pair> , ... )  ;  <property\_pair>::=  *property\_name* = *property\_value*  ;  *--Comments of an input or output stream*  <cql\_comment>::=  COMMENT *comment\_string*  ; |

* **Parameter**

*operator\_name*: indicates the name of a customized operator. The value is a string, without being quoted by single quotation marks or double quotation marks.

*column\_name*: indicates the name of a column. The value is a string, without being quoted by single quotation marks or double quotation marks.

*class\_name*: indicates the class of a data source. The value is a string quoted by single quotation marks or double quotation marks.

*property\_name*: indicates the name of an attribute.The value is a string quoted by single quotation marks or double quotation marks.

*property\_value*: indicates the value of an attribute. The value is a string quoted by single quotation marks or double quotation marks.

*comment\_string*: indicates comment content. The value is a string quoted by single quotation marks or double quotation marks.

* **Function**

Defines syntax of a customized operator. In the syntax, input and output schema of an operator and used parameters and classes are defined.

User input and output schema is verified before a topology is submitted. If data types or the column count are inconsistent, check will fail, and submitting the topology will fail.

* **Example**

|  |
| --- |
| create operator userOp as "com.huawei.streaming.example.userdefined.operator.UserOperator"  input (id int, name string)  output (newID string, name string, type int)  properties ("userop.filename" = "/home/omm/kv.properties"); |

#### User Operator Query

* **Syntax**

User Operator Query is used together with From substatements. dataSourceBody syntax is also defined in From statements.

|  |
| --- |
| *--Query syntax definition of the customized operator*  <insert\_clause> <using\_statement>  ;  <using\_statement>::=  USING OPERATOR *operator\_name* FROM *stream\_name*[ distribute\_clause ] [ parallel\_clause ]  ;  <distribute\_clause>::=  DISTRIBUTE BY *column\_name*  ;  *--Parallel settings*  <parallel\_clause>::=  PARALLEL *int\_number*  ; |

* **Parameter**

*operator\_name*: indicates the name of a customized operator. The value is a string, without being quoted by single quotation marks or double quotation marks.

*stream\_name*: indicates the name of a stream. The value is a string, without being quoted by single quotation marks or double quotation marks.

*column\_name*: indicates the name of a column. The value is a string, without being quoted by single quotation marks or double quotation marks.

*int\_number*: indicates a positive integer.

* **Function**

Combines customized operators and other input or output operators or functional operators into a topology to implement user functions.

* **Example**

|  |
| --- |
| insert into rs using operator userOp from s distribute by id parallel 2; |

## Command Statements

All system commands take effect immediately after being delivered, and command output can be immediately viewed.

After a submit application operation is performed, all modified parameters, customized functions, and windows will be reset.

### Set/Get

* **Syntax**

|  |
| --- |
| SET *property\_name*= *property\_value;*  GET *property\_name;* |

* **Parameter**

*property\_name*: indicates the name of an attribute. The value is a string quoted by single quotation marks or double quotation marks.

*property\_value*: indicatesthe value of an attribute. The value is a string quoted by single quotationmarks or double quotation marks.

* **Function**

The Set command sets a system variable or parameter, whereas the Get command obtains a system variable or parameter.

Parameter names are characters that are case insensitive. Special parameter names can be strings quoted by single quotation marks or double quotation marks, and content in strings is case sensitive.

Variable values are case-sensitive strings.

The Set and Get commands can contain system variables, environment variables, and other defined attributes.

* **Example**

Set "x"="1": sets the system variable X, which can be used in any scenarios in which system attributes are set.

Get "x": obtains the value of the variable x. If the variable is not included, NULL is displayed.

|  |
| --- |
| *--General set command*  set 'streaming.serde.default' = 'com.huawei.streaming.serde.SimpleSerDe'; |
| *--Set command string quoted by single quotation marks or double quotation marks*  set "serde.simpleserde.separator"=','; |
| *--Attribute value quoted in the value of an attribute configured by using the set command*  set "x" = "abc";  set "x" = "${conf:x}def";  *--Get value abcdef*  get "x"; |
| *--Set using a system variable*  set "streaming.template.directory" = "${system:java.io.tmpdir}"; |

### Add Jar

* **Syntax**

|  |
| --- |
| ADD JAR *path*; |

* **Parameter**

*path*: indicates a file path. The value is a string quoted by single quotation marks or double quotation marks.

* **Function**

User interfaces are always used in the system. For example, users use customized serialization rules, input and output, and functions in CQL. Users can package such programs as jar packages and submit the packages by running the **add jar** command. Then, the packages can be used.

After connections are disabled, manually added jar files are uninstalled from a local process so that subsequent connections are not affected.

* **Example**

|  |
| --- |
| Add jar '/opt/huawei/streaming/test/test.jar'; |

### Add File

* **Syntax**

|  |
| --- |
| ADD FILE *path*; |

* **Parameter**

*path*: indicates a file path. The value is a string quoted by single quotation marks or double quotation marks.

* **Function**

User interfaces are always used in the system. For example, users use customized serialization rules, input and output, and functions in CQL. Configuration files or resource files may be used in such classes, and users can submit such files by running the **add File** command. Then, the applications can be accessed after being submitted to remote clusters.

* **Example**

|  |
| --- |
| ADD FILE '/opt/huawei/streaming/test/test.conf'; |

### Create Function

* **Syntax**

|  |
| --- |
| CREATE FUNCTION *function\_name* AS *class\_name* [ <property\_list> ]  *--Attribute list*  <property\_list>::=  PROPERTIES (<property\_pair> , ... )  ;  <property\_pair>::=  *property\_name* = *property\_value*  ; |

* **Parameter**

*function\_name*: indicates the name of a function. The value is a string, without being quoted by single quotation marks or double quotation marks.

*class\_name*: indicates the class of a data source. The value is a string quoted by single quotation marks or double quotation marks.

*property\_name*: indicates the name of an attribute. The value is a string quoted by single quotation marks or double quotation marks.

*property\_value*: indicates the value of an attribute. The value is a string quoted by single quotation marks or double quotation marks.

* **Function**

Registers customized functions. Names of customized functions must be different from names of system functions; otherwise, a system error is reported. Customized functions can be repeatedly defined.

All function names are case insensitive.

The function does not need to be deregistered and will expire after an application is submitted. When the function expires, registered jar packages and files will be uninstalled from a local process.

* **Example**

|  |
| --- |
| CREATE FUNCTION rdbCompare AS 'com.huawei.streaming.example.udf.RDBCompare'; |
| create function udf1 as "com.huawei.streaming.cql.functions.DirectOutputUDF"  properties ("cql.test.direct.output" = "a"); |
| create function udf2 as "com.huawei.streaming.cql.functions.DirectOutputUDF"  properties ("cql.test.direct.output" = "b"); |
| create function udf3 as "com.huawei.streaming.cql.functions.DirectOutputUDF"; |

### Drop Function

* **Syntax**

|  |
| --- |
| DROP FUNCTION [IF EXISTS] *function\_name*; |

* **Parameter**

*function\_name*: indicates the name of a function. The syntax is the same as syntax of create Function.

* **Function**

Deletes a function from the system. Function names are case insensitive.

If the function does not exist in a scenario excluding IF EXISTS, an error is reported.

* **Example**

|  |
| --- |
| DROP FUNCTION rdbCompare; |

### Show Applications

* **Definition**

|  |
| --- |
| SHOW APPLICATIONS [ *matched\_application\_name* ]; |

* **Parameter**

*matched\_application\_name*: indicates the name of an application, which is optional. The value is a string quoted by single quotation marks or double quotation marks and supports fuzzy search.

* **Function**

Queries application information.

Query results are determined by a lower-layer operating platform, generally including application names, application status, and application start time.

Filters applications based on application names.

In the security environment, common users can view applications submitted only by themselves, but administrator users can view all applications. Administrator users are specified by Storm cluster configuration.

* **Example**

|  |
| --- |
| Show applications;  *--Querying all applications that run in the system* |
| Show applications filter;  *--Displaying applications that run in the system and whose names contain filter* |
| Show applications 'filter\*';  *--Querying applications that run in the system and whose names start with filter* |
| Show applications '\*filter\*';  *--Querying applications that run in the system and whose names contain filter* |
| Show applications '\*filter';  -*-Querying applications that run in the system and whose names end with filter* |

# Window

* **Syntax**

|  |
| --- |
| *--Window syntax definition*  *stream\_name* ( <filter\_before\_window> ) *[* <window\_source> *]*  *--Syntax definitions fordifferent types of windows*  <window\_source>::=  <range\_window>  | <rows\_window>  | <range\_today>  ;  *--Rows window*  <rows\_window>::=  ROWS *int\_number* <window\_properties> <window\_determiner>  ;  *-- Time range window*  <range\_window>::=  RANGE <range\_bound> [<window\_properties>] <window\_determiner>  ;  *--A window with the time range of the current day*  <range\_today>::=  RANGE TODAY <expression> <window\_determiner>  ;  *--Time range definition*  <range\_bound>::=  <range\_time>  | <range\_unbound>  ;  *--An unlimited window*  <range\_unbound>::=  UNBOUNDED  ;  *--Time range definition*  *--The default unit is millisecond. Value combinations are allowed, and at least one value must exist.*  <range\_time>::=  [<range\_day>] [<range\_hour>] [<range\_minutes>] [<range\_seconds>] [<range\_milliseconds>]  ;  *--Time range in days*  <range\_day>::=  *int\_number* DAY  | *int\_number* DAYS  ;  *--Time range in hours*  <range\_hour>::=  *int\_number* HOUR  | *int\_number* HOURS  ;  *--Time range in minutes*  <range\_minutes>::=  *int\_number* MINUTE  | *int\_number* MINUTES  ;  *--Time range in seconds*  <range\_seconds>::=  *int\_number* SECOND  | *int\_number* SECONDS  ;  *--Time range in milliseconds*  <range\_milliseconds>::=  *int\_number* MILLISECOND  | *int\_number* MILLISECONDS  ;  *--Window attribute. A window can be a slide or tumble window.*  <window\_properties>::=  SLIDE  | BATCH  ;  *--Window qualifier, such as partition by, sort by,trigger by, or whether to exclude the current event*  <window\_determiner>::=  [<partitionby\_determiner>]  [<sortby\_determiner>]  [<triggerby\_determiner>]  [<excludenow\_determiner>]  ;  *--Window event partition mode*  <partitionby\_determiner>::=  PARTITION BY <expression>  ;  *--Window event sorting mode*  <sortby\_determiner>  SORT BY <expression>  ;  *--Window event triggering mode*  <triggerby\_determiner>::=  TRIGGER BY <expression>  ;  *--Whether to exclude the current event during calculation*  <excludenow\_determiner>::=  EXCLUDE NOW  ; |

* **Parameter**

*stream\_name*: indicates the name of a stream. The value is a string, without being quoted by single quotation marks or double quotation marks.

*int\_number*: indicates a positive integer.

<filter\_before\_window>: indicates the syntax block of the filter expression before data enters a window. For details, see 7.3.2.2 From Clause.

<expression>: indicates a CQL expression substatement.For details, see 9 Expression.

* **Function**

A window stored stream data generated in a specified period and generates expired events.

Window names are case insensitive.

A window consists of a window body, an attribute, and qualifiers. A window body specifies whether a window is a time-based window or row-based window. A window attribute specifies whether data output of a window is implemented in the slide or tumble mode. A qualifier specifies the mode for updating data in a window, such as a field for grouping and whether current events are included in aggregation calculation.

The time unit of a time-based window can be millisecond(s), second(s), minute(s), hour(s), or day(s).

Multiple qualifiers are supported, but they must be defined by sequence. The sequence for using qualifiers is as follows: Partition by, Sort by, Trigger by, Exclude Now

A CQL window consists of a window body, an attribute, and qualifiers. One window has only one window body and one attribute but can have multiple qualifiers.

Window body types

|  |  |  |
| --- | --- | --- |
| Window Body | Syntax | Description |
| rows window | ROWS *n1* | Row windows whose quantity is *n1*. Each row window contains a maximum of *n1* rows of data. |
| range window | RANGE *t1* | Time window within the *t1* time range. |
| naturalDay window | RANGE TODAY *exp1* | A natural-day window is a tumble window storing data on the current day. The return value of *exp1* is a time type, that is, number type (int,long), timestamp, Date. |

Window attribute

|  |  |  |
| --- | --- | --- |
| Window Attribute | Syntax | Description |
| slide window | SLIDE | Window data output is implemented in slide mode, and one piece of data is provided each time. |
| tumble window | BATCH | Window data output is implemented in tumble mode. The volume of each data output depends on window capacity. |

Window qualifiers

|  |  |  |
| --- | --- | --- |
| Window Qualifier | Syntax | Description |
| grouped window | PARTITION BY *exp1* | Data in a window is grouped based on *exp1*.  A group window supports single expressions only. |
| sorted window | SORT BY *column\_name* | Data in a window is sorted based on the time sequence specified in *exp1*.  Sort by sorts columns only but does not sort expressions such as functions. |
| eventDrive window | TRIGGER BY *exp1* | Data in a window is updated depending on updates of external events rather than system rules. The *exp1* expression supports number type (int,long), timestamp, Date. |
| excludeNow Window | EXCLUDE NOW | Indicates whether a window excludes current events.  Data in the current row is excluded in a window. |



All batch windows do not support the EXCLUDE NOW and sort by functions.

The TRIGGER BY operation does not support the time data type.

The EXCLUDE NOW operation excludes current rows, and therefore, Exclude now query statements cannot contain current rows but contain aggregation functions or group columns such as count and sum.

Output window data includes new and old data. Generally only CQL window data is new output data.

SORT BY windows resolves disorder of distributed data, and data of such windows is old data only.

During a Join operation, if the Sort by window is at the same side as Unidirection, output data is old data; otherwise, output data is new data.

The Join operation does not support scenarios in which one side is a Sort by window and the other side is a window of another type because new data and old data cannot be calculated together.

A group window groups data and data output is performed based on the grouping sequence rather than the sequence by which data enters a window. Data in groups is traversed in hash mode in the system, irrelevant to the mode for grouping and sorting data.

It is recommended that a group window be used together with the Group by syntax.

[8.1 rows window](#_EN-US_TOPIC_0029436495)

[8.2 range window](#_EN-US_TOPIC_0029436483)

[8.3 naturalDay window](#_EN-US_TOPIC_0029436579)

## rows window

* **Syntax**

|  |
| --- |
| ROWS *int\_number* <window\_properties> <window\_determiner>; |

* **Parameter**

*int\_number*: indicates the number of rows. The value is a positive integer.

<window\_properties>: indicates a window attribute, that is, slide window or tumble window. For details, see 8 Window.

<window\_determiner>: indicates a window qualifier, such as partition by, trigger by, or sort by. For details, see 8 Window.

* **Description**

A rowswindow stores events in a window and generates new and old events based window attributes in different time periods. The maximum number of events that can be stored in a rowswindow is specified.

rowswindow

| Syntax | Window Name | Description |
| --- | --- | --- |
| S*[* ROWS *n1* SLIDE [ EXCLUDE NOW ] *]* | Length slide window | A maximum of *n1* events can be stored in a window. When new events are generated, existing events expire in sequence. Expired events are in different batches. |
| S*[* ROWS *n1* BATCH *]* | Length tumble window | A maximum of *n1* events can be stored in a window. When new events are generated, existing events expire in sequence. All events that expire at the same time are in a same batch.  This window does not support EXCLUDE NOW. |
| S*[* ROWS *n1* SLIDE PARTITION BY *exp1* [ EXCLUDE NOW ] *]* | Row-based length slide window | Length slide window that contains groups to which events belong. Each group has a maximum of *n1* events, which will expire in sequence. |
| S*[* ROWS *n1* BATCH PARTITION BY *exp1]* | Row-based length tumble window | Length tumble window that contains groups to which events belong. Each group has a maximum of *n1* events, which will expire in sequence. Each group belongs to a batch.  This window does not support EXCLUDE NOW. |
| S*[* ROWS *n1* SLIDE SORT BY *exp1* *]* | Element sorting window | Data in an element sorting window can be sorted based on the size of a specific field or a fixed sequence specified by a time stamp field. Data in such a window will expire based on the sorting result.  This window does not support EXCLUDE NOW. |

* **Example**

|  |
| --- |
| Select \* from transformEvent[rows 10 slide];  *--Slide window whose capacity is 10.* |
| Select \* from transformEvent[rows 10 slide partition by type];  *--Data in such a window is grouped by type, and each group contains a maximum of 10 pieces of data.* |
| Select \* from transformEvent[rows 1000 batch partition by type]; |
| Select \* from transformEvent[rows 1000 slide sort by ts]; |

## range window

* **Syntax**

|  |
| --- |
| RANGE <range\_bound> [<window\_properties>] <window\_determiner>; |

* **Parameter**

<range\_bound>: indicates a window event range. For details, see 8 Window.

<window\_properties>: indicates a window attribute, that is, slide window or tumble window. For details, see 8 Window.

<window\_determiner>: indicates a window qualifier, such as partition by, trigger by, or sort by. For details, see 8 Window.

* **Function**

A rangewindow stores events in a window and generates new and old events based window attributes in different time periods. A range window can store events only of a specific time range.

rangewindow

| Syntax | Window Name | Description |
| --- | --- | --- |
| S*[* RANGE UNBOUNDED [ EXCLUDE NOW ] *]* | Unlimited rangew indow | The volume of data stored in this window is not limited, and the data will never expire. |
| S*[* RANGE *t1* SLIDE *]* | Time slide window | A time slide window stores data of the latest *t1*, which is a time unit. Time units, such as Seconds, can be added. Events in a time slide window will expire in sequence, and each expired event belongs to a unique batch.  This window does not support EXCLUDE NOW. |
| S*[* RANGE *t1* BATCH *]* | Time tumble window | A time slide window stores data of the latest *t1*, which is a time unit. Time units, such as seconds, can be added. Events stored in a time tumble window expire by batch, and expired data belongs to a same batch.  This window does not support EXCLUDE NOW. |
| S*[* RANGE *t1* SLIDE PARTITION BY *exp1* *]* | Group time slide window | Time slide window that contains groups to which events belong. The time range of each group is *t1*.  This window does not support EXCLUDE NOW. |
| S*[* RANGE *t1* BATCH PARTITION BY *exp1* *]* | Group time tumble window | Time tumble window that contains groups to which events belong. The time range of each group is *t1*.  This window does not support EXCLUDE NOW. |
| S*[* RANGE *t1* SLIDE SORT BY *exp1* *]* | Time sorting window | A time sorting window stores data of a specific time range. Each piece of newly generated data is compared with existing data in the window, output is performed for expired data, and newly generated data is inserted into specific positions in sequence.  This window does not support EXCLUDE NOW. |
| S*[* RANGE *t1* SLIDE TRIGGER BY *exp1* [ EXCLUDE NOW ] *]* | Event-driven time slide window | An event-driven time slide stores data of the latest *t1* time unit. *exp1* is an expression whose return value is a time type (long,timestamp,Date). Each piece of newly generated data is compared with existing data in the window, and output is performed for data generated outside *t1*. Output is performed for only one piece of data each time. |
| S*[* RANGE *t1* BATCH TRIGGER BY *exp1* *]* | Event-driven time tumble window | An event-driven time tumble stores data of the latest *t1* time unit. *exp1* is an expression whose return value is a time type (long,timestamp,Date). Each piece of newly generated data is compared with existing data in the window, and collection and output are performed for events generated outside *t1*.  This window does not support EXCLUDE NOW. |
| S*[* RANGE *t1* SLIDE PARTITION BY *exp1* TRIGGER BY EXP2 [ EXCLUDE NOW ] *]* | Event-driven group time slide window | An event-driven group time slide window is an event-driven slide window that contains groups. Event sliding occurs on each group. |
| S*[* RANGE *t1* BATCH PARTITION BY *exp1* TRIGGER BY *exp2* *]* | Event-driven group time tumble window | An event-driven group time tumble window is an event-driven time slide window that contains groups. Event tumbling occurs on each group.  This window does not support EXCLUDE NOW. |



During calculation, a time sorting window is compared with time in a cluster. If the sum of event time and window time is later than the system time, output is performed. If the sum of event time and window time is earlier than the system time, an event has occurred, and event output will occur.

1. **Example**

|  |
| --- |
| Select \* from transformEvent[range unbounded] as s1 inner join transformBak s2 on S1.id=S2.id; |
| Select \* from transformEvent[range 10 seconds slide]; |
| Select \* from transformEvent[range 1 day 6 hours slide]; |
| Select \* from transformEvent[range 10 seconds slide partition by type]; |
| Select \* from transformEvent[range 1000 milliseconds batch];  Define a time tumble window whose length is 1 second. |
| Select \* from transformEvent[range 1000 milliseconds batch partition by id]; |
| Select \* from transformEvent[range 1000 milliseconds sort by dte]; |
| insert into stream output\_where\_event\_tbatch  select sum(OrderPrice),avg(OrderPrice),count(OrderPrice)  from input\_where\_event\_tbatch[range 10 seconds slide trigger by TS exclude now]  where OrderPrice=100 or OrderPrice>=700; |
| insert into stream output\_where\_event\_tbatch  select sum(OrderPrice),avg(OrderPrice),count(OrderPrice)  from input\_where\_event\_tbatch[range 10 seconds slide partition by id trigger by TS exclude now]  where OrderPrice=100 or OrderPrice>=700; |
| insert into stream output\_where\_event\_tbatch  select sum(OrderPrice),avg(OrderPrice),count(OrderPrice)  from input\_where\_event\_tbatch[range 10 seconds batch trigger by TS exclude now]  where OrderPrice=100 or OrderPrice>=700; |
| insert into stream output\_where\_event\_tbatch  select sum(OrderPrice),avg(OrderPrice),count(OrderPrice)  from input\_where\_event\_tbatch[range 10 seconds batch partition by type trigger by TS exclude now]  where OrderPrice=100 or OrderPrice>=700; |

## naturalDay window

* **Syntax**

|  |
| --- |
| RANGE TODAY <expression> <window\_determiner>; |

* **Parameter**

<expression>: indicates a CQL expression substatement.For details, see 9 Expression.

<window\_determiner>: indicates a window qualifier, such as partition by, trigger by, or sort by. For details, see 8 Window.

* **Function**

A naturalDaywindow stores events of a certain quantity and generates new and old events based window attributes in different time periods. The maximum number of events that can be stored in a window is specified.

naturalDaywindow

| Syntax | Window Name | Description |
| --- | --- | --- |
| S*[*RANGE TODAY *exp1* [ EXCLUDE NOW ] *]* | naturalDaywindow | A naturalDay window stores data of an entire day. Whether an entire natural day has passed is determined based on a newly generated event. If an entire natural day has passed, data of the previous natural day in the window will expire. A natualDay window is a slide window.  *exp1* is a time stamp expression used to determine whether a time point is on the current day. |
| S*[*RANGE TODAY *exp1* PARTITION BY *exp2* [ EXCLUDE NOW ] *]* | Group naturalDaywindow | A group naturalDay window is a naturalDay window that contains groups. A group naturalDay window is used together with a naturalDay window.  *exp1* is a time stamp expression used to determine whether a time point is on the current day. |

* **Example**

|  |
| --- |
| insert into stream rs  select id,name,count(id)  from transformEvent[range today ts]  where id> 5  group by type  having id>10; |
| insert into stream rs  select id,name,count(id)  from transformEvent[range today ts partition by type]  where id> 5  group by type  having id>10; |

# Expression

* **Syntax**

|  |
| --- |
| *--Expression definition*  <expression>::=  <logic\_or\_expression>  ;  *--Or logic expression definition*  <logic\_or\_expression>::=  <logic\_and\_expression> OR <logic\_and\_expression> ...  ;  *--And logic expression definition*  <logic\_and\_expression>::=  <logic\_not\_expression> AND <logic\_not\_expression> ...  ;  *--Not logic expression definition*  <logic\_not\_expression>::=  [NOT ] <is\_null\_like\_in\_expressions>  ;  *--Definition of is null,like,and in expressions*  <is\_null\_like\_in\_expressions>::=  <binary\_expression>  [IS <null\_condition>  <expression\_like>  <expression\_between>  |<expression\_in>  ]  ;  *--Between expression definition*  <expression\_between>::=  [NOT ] BETWEEN <bit\_expression> AND <bit\_expression>  ;  *--Binary expression definition*  <binary\_expression>::=  <bit\_expression> <relation\_expression>\*  ;  *--Relation expression definition*  <relation\_expression>::=  <relation\_operator> <bit\_expression>  ;  *--Relation symbol definition*  <relation\_operator>::=  =  |==  |<=>  |<>  |!=  |>  |>  |>=  |<=  ;  *--Previous* *expression definition*  <expression\_previous>::=  PREVIOUS ( <expression>)  ;  *--In expression definition*  <expression\_in>::=  [NOT] IN ( <expression>,... )  ;  *--Like expression definition*  <expression\_like>::=  [NOT] LIKE <bit\_expression>  ;  *--Addition operation definition*  <bit\_expression>::=  <arithmetic\_star\_expression> <arithmetic\_plus\_operator> <arithmetic\_star\_expression> ...  ;  *--Definition of addition and subtraction operators*  <arithmetic\_plus\_operator>::=  + |-  ;  *--Definition of multiplication and division operations*  <arithmetic\_star\_expression>::=  <field\_expression> <arithmetic\_star\_operator> <field\_expression> ...  ;  *--Definition of multiplication and division operators*  <arithmetic\_star\_operator>::=  \* |/ | %  ;  *--Field expression definition*  <field\_expression>::=  [*stream\_name\_or\_stream\_alias*.]<atom\_expression>  ;  *--Atom expression definition*  <atom\_expression>::=  NULL  |<constant>  |<expression\_previous>  |<function>  |<cast\_expression>  |<case\_expression>  |<when\_expression>  |*column\_name*  | (<expression> )  ;  *--Constant definition*  <constant>::=  <unary\_constant>  | *const\_string\_value*  | <boolean\_value>  ;  *--Definition of positive and negative numeric constants*  <unary\_constant>::=  [ +| - ]  *const\_integer\_value*  | *const\_long\_value*  | *const\_float\_value*  | *const\_double\_value*  | *const\_bigdecimal\_value*  ;  *--Boolean constant definition*  <boolean\_value>::=  TRUE  |FALSE  ;  *--Constant definition*  <function>::=  *function\_name* ( [ DISTINCT <expression> ,... ] )  ;  *--Cast expression definition*  <cast\_expression>::=  CAST ( <expression> AS<data\_type> )  ;  *--Case expression definition*  <case\_expression>::=  CASE <expression> <when\_then\_expression>... [ ELSE <expression> ] END  ;  *--When expression definition*  <when\_expression>::=  CASE <when\_then\_expression>... [ ELSE <expression>] END  ;  *--Definition of the when then expression in the case when expression*  <when\_then\_expression>::=  WHEN <expression> THEN <expression>  ; |

* **Parameter**

*function\_name*: indicates the name of a function. The value is a string, without being quoted by single quotation marks or double quotation marks.

*column\_name*: indicates a column name, which is the same as the column name defined in the create input stream and create output stream.

*const\_integer\_value*:indicates an integer constant. The positive integer is expressed as 1, 2, 3... , and the negative integer is expresses as –1, –2...

*const\_long\_value*: indicates a long-type constant. Long-type constants are expressed in the similar way to that of integer constants. Letter L (case insensitive) must be suffixed to a constant,for example, 1L and –1l.

*const\_float\_value*: indicates a float-type constant. Float-type constants are expressed in the similar way to that of integer constants. Letter F (case insensitive) must be suffixed to a constant,for example, –1.0F and 2f.

*const\_double\_value*: indicates a double-type constant. Double-type constants are expressed in the similar way to that of integer constants. Letter D (case insensitive) must be suffixed to a constant,for example, –1.0d and 2D.

*const\_bigdecimal\_value:* indicates a decimal constant.Decimal constants do not compromise precision during operation. Letters BD(case insensitive) must be suffixed to a constant, for example, 1BD and –1bd.

*const\_string\_value*: indicates a string-type constant quoted by single quotation marks or double quotation marks.

*stream\_name\_or\_stream\_alias*: indicates the name of a stream or alias. The value is a string, without being quoted by single quotation marks or double quotation marks.

<data\_type>: indicates a data type syntax block. For details, see 7.2 Stream Operation Statements.

* **Function**

An expression is a set of symbols and operators. The CQL parsing engine processes an expression to obtain single values. A simple expression can be a constant, variable, or function. Two or more simple expressions can be combined as a complex expression by using operator.

[9.1 Common Expression](#_EN-US_TOPIC_0029436564)

[9.2 Expression Constant](#_EN-US_TOPIC_0029436587)

[9.3 Special Expression](#_EN-US_TOPIC_0029436584)

[9.4 Rules for Converting Expression Data Types](#_EN-US_TOPIC_0029436492)

## Common Expression

Common expressions may be included in select, where, or having substatements.

The following table describes expression priorities from the highest to the lowest.

List of common expressions

| Expression | Description |
| --- | --- |
| +, - | Unary operator. |
| \*, / | Multiply or divide. |
| +, - | Binary operator. |
| =, <>, <,>, <=,>=, != | Comparison operator. |
| IS [NOT] NULL, [NOT] LIKE, [NOT] BETWEEN,[NOT] IN | Judgment expression used to check whether a value is blank. |
| AND, OR | Logic expression: The AND or OR relationship exists between conditions. |



The Date, time, and timestamp data types do not support operation such as plus, minus, multiply, divide, and compare.

Operation, such as plus, minus, multiply, divide, and compare, is unavailable to strings.

Unary operators, such as + and -, are available only to constant numbers, such as -1 and -0.1f.

The return value of Null and all common expression, except AND and OR, are null.

Return values of Null And True, Null Or true, Null Or false, and Null and false are Null, true, null, and false respectively.

## Expression Constant

Constants in CQL statements can be used in expressions only. The expressions include select, where, and having substatements.

String: indicates a string quoted by single quotation marks or double quotation marks, such as "Streaming".

Int: indicates an integer, such as 1 or -1.

Long: indicates a string ended with non-case-sensitive L, such as 1l and 2L.

Float: indicates a string ended with non-case-sensitive F, such as 1.0f and -1.0F.

Double: indicates a string ended with non-case-sensitive D, such as 1.0D and -1.0D.

Decimal: indicates a string ended with non-case-sensitive BD, such as 1.0BD and -1.0BD.

Boolean: true or false.

Time: not supported, and it must be converted by using UDF functions.

Date: not supported, and it must be converted by using UDF functions.

Timestamp: not supported, and it must be converted by using UDF functions.

## Special Expression

### Cast

* **Syntax**

|  |
| --- |
| *--Cast expression definition*  <cast\_expression>::=  CAST ( <expression> AS <data\_type> )  ; |

* **Parameter**

<expression>: indicates a CQL expression substatement.For details, see 9 Expression.

<data\_type>: indicates a data type syntax block. For details, see 7.2 Stream Operation Statements.

* **Return value type**

The return value type is the same as the data type of datatype in input parameters.

* **Function**

Converts original data into data of another type.

* **Example**

|  |
| --- |
| INSERT INTO STREAM rs  SELECT  CAST(id AS int),  CAST(id AS long),  CAST(id AS float),  CAST(id AS double),  CAST(id AS String),  CAST(id AS boolean)  FROM s  WHERE NOT id != 'a'  AND type != 1; |

### Case when

* **Syntax**

|  |
| --- |
| *--Case expression definition*  <case\_expression>::=  CASE <expression> <when\_then\_expression>... [ ELSE <expression> ] END  ;  *--When expression definition*  <when\_expression>::=  CASE <when\_then\_expression>... [ ELSE <expression>] END  ;  *--Definition of the when then expression in the case when expression*  <when\_then\_expression>::=  WHEN <expression> THEN <expression>  ; |

* **Parameter**

<expression>: indicates a CQL expression substatement.For details, see 9 Expression.

* **Return value type**

Calculation results of the then or else expression are returned.

* **Function**

Provides different results based on different conditions.

* **Example**

|  |
| --- |
| CASE WHEN sex = 1 THEN Population ELSE 0 END; |
| case key when '1' then 1 when '2' then 2 when '3' then 3 else 99 end; |

### Like

* **Syntax**

|  |
| --- |
| *--Like expression definition*  <expression\_like>::=  [NOT] LIKE <bit\_expression>  ; |

* **Parameter**

<bit\_expression>: indicates the addition operation syntax block.For details, see 9 Expression.

* **Return value type**

Boolean

* **Description**

This expression can be a regular expression that checks whether an expression matches. If yes, true is returned. If no, false is returned.

If an expression contains NOT, the return value is the opposite value.

* **Example**

1. % is a string that contains zero or more characters.

Example: WHERE title LIKE '%computer% ' (querying all book names that at any places and contain the word computer)

2. The underscore (\_) indicates any characters.

Example: WHERE au\_fname LIKE '\_ean ' (querying a name that contains four letters and end with ean, such as Dean and Sean)

3. [] indicates any character within a specific range.

Example: WHERE au\_lname LIKE '[C-P]arsen ' (querying an author first name that end with arsen and start with any letter from C to P, such as Carsen, Larsen, and Karsen)

4. [^] indicates any character outside a specific range.

Example: WHERE au\_lname LIKE 'de[^l]% ' (querying all author first names that start with de which is not followed with l)

### Between

* **Syntax**

|  |
| --- |
| *--Between expression definition*  <expression\_between>::=  [ NOT ] BETWEEN <bit\_expression> AND <bit\_expression>  ; |

* **Parameter**

<bit\_expression>: indicates the addition operation syntax block.For details, see 9 Expression.

* **Return value type**

boolean

* **Function**

Checks whether a field is in a specific range. In this expression, a larger value is located before a larger value, and the return value is of the boolean type.

The data type of columns must be the same as data types of the two values.

If an expression contains NOT, the return value is the opposite value.

* **Example**

|  |
| --- |
| Size between 1 and 30; |

### In

* **Syntax**

|  |
| --- |
| *--In expression definition*  <expression\_in>::=  [NOT] IN ( <expression>,... )  ; |

* **Parameter**

<expression>: indicates a CQL expression substatement.For details, see 9 Expression.

* **Return value type**

Boolean

* **Function**

Checks whether a column value is within a specific range. Data type of columns must be the same as data types of all constants in IN expressions.

If an expression contains NOT, the return value is the opposite value.

* **Example**

|  |
| --- |
| Type in (1,2,3);  Type in ('a','b','c'); |

## Rules for Converting Expression Data Types

### Operation Expression

Operation expressions are used for operations such as plus, minus, multiply, divide, and compare.

When operation is performed for data of different types, automatic upcasting is performed so that special function conversion is not required.

Rules for converting types of data in CQL operation expressions are as follows, in ascending order:

Int < Long < Float < Double < Decimal

Operation is unavailable to the String, Boolean, Date, Time, and TimeStamp data types.

### CAST Expression

CAST expressions can convert data of various types. The following table describes rules for CQL CAST expressions to convert data of a source type to data of a target type.

List of supported data types that can be converted by CAST expressions

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | String | Boolean | Int | Long | Float | Double | Decimal | Time | Date | TimeStamp |
| String | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y |
| Boolean | Y | Y |  |  |  |  |  |  |  |  |
| Int | Y | Y | Y | Y | Y | Y | Y |  |  |  |
| Long | Y | Y | Y | Y | Y | Y | Y |  |  |  |
| Float | Y | Y | Y | Y | Y | Y | Y |  |  |  |
| Double | Y | Y | Y | Y | Y | Y | Y |  |  |  |
| Decimal | Y | Y | Y | Y | Y | Y |  |  |  |  |
| Time | Y | Y | Y | Y | Y | Y |  |  |  |  |
| Date | Y | Y | Y | Y | Y | Y |  |  |  |  |
| TimeStamp | Y | Y | Y | Y | Y | Y |  |  |  |  |



The CAST type conversion function is the same as the function of the type conversion function.

For time type conversion, the default time zone for Streaming clusters is used. CQL does not support specified time zones.

The boolean type of numbers greater than 1 is true, and the boolean type of numbers less than 1 is false.

# Customized Interface

[10.1 Data Serialization and Deserialization](#_EN-US_TOPIC_0029436517)

[10.2 Streaming Data Obtaining](#_EN-US_TOPIC_0029436528)

[10.3 Stream Data Writing](#_EN-US_TOPIC_0029436592)

[10.4 UDF Function](#_EN-US_TOPIC_0029436568)

[10.5 Data Source](#_EN-US_TOPIC_0029436545)

[10.6 Customized Operator](#_EN-US_TOPIC_0029436491)

## Data Serialization and Deserialization

* **Description**

Only the com.huawei.streaming.serde.StreamSerDe interface implements data deserialization.

The com.huawei.streaming.serde.BaseSerDe class can also be inherited because the com.huawei.streaming.serde.BaseSerDe class provides common implementation methods, such as creating blank objects and converting data types.

Interfaces for data serialization and deserialization

| Method Name | Method Description | Parameter | Return Value |
| --- | --- | --- | --- |
| setConfig | Sets serialization attributes.  This method is invoked during compilation. | StreamingConfig | Void |
| getConfig | Obtains serialization attributes.  This method is invoked during compilation. | None | StreamingConfig |
| initialize | Initialization interface.  This method is invoked during compilation. | StreamingConfig | Void |
| setSchema | Sets input or output schema.  This method is invoked during compilation.  This method is invoked later than the setConfig method. Therefore, schema cannot be obtained from the setConfig method. | TupleEventType  Input or output schema | Void |
| deSerialize | Converts external input data into data of a type that can be identified by the system.  This method is invoked during operating. | Object  Data that can be identified by an external third-party system, such as strings. | List<Object[]>  Data type that can be internally identified.  The list size specifies the number of data rows.  The number of columns in data rows is the same as the length of the Object array, and the sequence of data in the Object array is the same as the sequence specified in data rows. |
| serialize | Converts system computing results into data that can be identified by other external systems.  This method is invoked during operating. | List<Object[]>  Internal computing result  The list size specifies the number of data rows.  The number of columns in data rows is the same as the length of the Object array, and the sequence of data in the Object array is the same as the sequence specified in data rows. | Object  Data that can be identified by an external third-party system, such as strings. |

* **Example**

|  |
| --- |
| package com.huawei.streaming.example.userdefined.serde;  import java.util.List;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import com.google.common.base.Splitter;  import com.google.common.collect.Lists;  import com.huawei.streaming.config.StreamingConfig;  import com.huawei.streaming.event.TupleEventType;  import com.huawei.streaming.exception.StreamSerDeException;  import com.huawei.streaming.exception.StreamingException;  import com.huawei.streaming.serde.StreamSerDe;  import com.huawei.streaming.util.DataTypeUtils;  /\*\*  \* Serialization and deserialization example  \*/  public class ExampleSerDe implements StreamSerDe  {  private static final long serialVersionUID = -8447913584480461044L;  private static final Logger LOG = LoggerFactory.getLogger(ExampleSerDe.class);  private TupleEventType schema;  private String separator = ",";  private StringBuilder sb = new StringBuilder();  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setConfig(StreamingConfig arg0)  throws StreamingException  {  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public StreamingConfig getConfig()  {  //Obtain attributes.  return null;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setSchema(TupleEventType outputSchema)  {  schema = outputSchema;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public TupleEventType getSchema()  {  return schema;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void initialize()  throws StreamSerDeException  {  }  /\*\*  \* Deserialization  \* Parse data obtained from an input stream to a data type that can be identified by the system.  \*  \* Data obtained from the @param data input stream  \* Input operators vary depending on data types in the deserialization type.  \* Character string type  \* In the example program, parsing is performed based on strings.  \* Data parsed by @return  \* Data of multiple rows may be generated from one row, and therefore, return values are arrays.  \* @throws StreamSerDeException serialization exception  \* @see [Class, Class#method, class#member]  \*/  @Override  public List<Object[]> deSerialize(Object data)  throws StreamSerDeException  {  if (data == null)  {  return null;  }  String stringValue = (String)data;  List<Object[]> splitResults = Lists.newArrayList();  Object[] values = Lists.newArrayList(Splitter.on(separator).split(stringValue)).toArray();  splitResults.add(values);  return createEventsInstance(splitResults);  }  /\*\*  \* Serialization method  \* Serialize an input event to an object that can be identified by output.  \* Character string or data of another type  \*  \* Event generated or processed by the @param events system. The event is an array.  \* Data serialized by @return. The data can be strings.  \* @throws StreamSerDeException serialization exception  \* @see [Class, Class#method, class#member]  \*/  @Override  public Object serialize(List<Object[]> events)  throws StreamSerDeException  {  if (events == null)  {  LOG.info("Input event is null");  return null;  }  clearTmpString();  for (Object[] event : events)  {  serializeEvent(event);  }  return removeLastChar();  }  /\*\*  \* Data type generated based on data columns in schema  \*/  private List<Object[]> createEventsInstance(List<Object[]> events)  throws StreamSerDeException  {  if (events == null || events.size() == 0)  {  return Lists.newArrayList();  }  List<Object[]> list = Lists.newArrayList();  for (Object[] event : events)  {  Object[] eventInstance = createEventInstance(event);  list.add(eventInstance);  }  return list;  }  /\*\*  \* Create an event instance.  \*/  private Object[] createEventInstance(Object[] event)  throws StreamSerDeException  {  validateColumnSize(event);  Object[] arr = new Object[schema.getAllAttributes().length];  for (int i = 0; i <schema.getAllAttributeTypes().length; i++)  {  arr[i] = createInstance(schema.getAllAttributeTypes()[i], event[i]);  }  return arr;  }  private String removeLastChar()  {  return sb.substring(0, sb.length() - 1);  }  /\*\*  \* Create a data instance based on a specific type.  \*/  private Object createInstance(Class<?> clazz, Object value)  throws StreamSerDeException  {  try  {  return DataTypeUtils.createValue(clazz, value.toString());  }  catch (StreamingException e)  {  throw new StreamSerDeException(e.getMessage(), e);  }  }  /\*\*  \* Serialization event in a row  \*/  private void serializeEvent(Object[] event)  {  for (Object column : event)  {  appendToTmpString(column);  }  sb.replace(sb.length() - 1, sb.length() - 1, "");  }  /\*\*  \* Add a value to the end of a temporary string.  \*/  private void appendToTmpString(Object val)  {  if (val != null)  {  sb.append(val.toString() + separator);  }  else  {  sb.append(separator);  }  }  /\*\*  \* Clear a temporary string before serialization.  \*/  private void clearTmpString()  {  sb.delete(0, sb.length());  }  /\*\*  \* Check whether the number of columns is the same as the number of columns in schema.  \*/  private void validateColumnSize(Object[] columns)  throws StreamSerDeException  {  if (columns.length != schema.getAllAttributeTypes().length)  {  LOG.error("deserializer result array size is not equal to the schema column size, "  + "schema size :{}, deserializer size :{}", schema.getAllAttributeTypes().length, columns.length);  throw new StreamSerDeException(  "deserializer result array size is not equal to the schema column size, schema size :"  + schema.getAllAttributeTypes().length + ", deserializer size :" + columns.length);  }  }  } |
| *-Example:*  *--Upload a jar package to any directory, such as /opt/streaming/example.jar.*  *--Run the following CQL statements:*  add jar "/opt/streaming/example.jar ";  create input stream S1  (  C1 STRING,  C2 STRING  )  SERDE "com.huawei.streaming.example.userdefined.serde.ExampleSerDe"  SOURCE kafkainput  PROPERTIES ( "operator.kafka.groupid" = "gidkpi\_1\_1","operator.kafka.topic"="agg\_1\_1\_1"  );  create output stream rs  (  C1 STRING,  C2 STRING  )  SERDE "com.huawei.streaming.example.userdefined.serde.ExampleSerDe"  SINK ConsoleOutput  PROPERTIES("printFrequence" = "10");  insert into stream rs select \* from S1;  submit application force read; |

## Streaming Data Obtaining

* **Description**

The com.huawei.streaming.operator.IInputStreamOperator interface is used to obtain data from external event sources.

Interface for obtaining stream data

| Method Name | Method Description | Parameter | Return Value |
| --- | --- | --- | --- |
| setConfig | Sets operator attributes.  This method is invoked during compilation. | StreamingConfig | Void |
| getConfig | Get operator attributes.  This method is invoked during compilation. | None | StreamingConfig |
| setSerDe | Sets a serialization or deserialization class.  This method is invoked during operating. | StreamSerDe | Void |
| getSerDe | Get a serialization or deserialization class.  This method is invoked during operating. | None | StreamSerDe |
| setEmitter | Sets an Emitter object.  This method is invoked during operating. | IEmitter | Void |
| initialize | Initialization interface invoked during operating. | None | Void |
| execute | Execution interface, invoked during operating. | None | Void |
| destroy | Destroy interface, invoked during operating. | None | Void |

* **Example**

|  |
| --- |
| package com.huawei.streaming.example.userdefined.operator.input;  import java.io.File;  import java.io.IOException;  import java.nio.charset.Charset;  import java.util.List;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import com.google.common.base.Strings;  import com.google.common.io.Files;  import com.google.common.io.LineProcessor;  import com.huawei.streaming.config.StreamingConfig;  import com.huawei.streaming.exception.StreamSerDeException;  import com.huawei.streaming.exception.StreamingException;  import com.huawei.streaming.exception.StreamingRuntimeException;  import com.huawei.streaming.operator.IEmitter;  import com.huawei.streaming.operator.IInputStreamOperator;  import com.huawei.streaming.serde.StreamSerDe;  /\*\*  \* File obtaining example  \*/  public class FileInput implements IInputStreamOperator  {  private static final long serialVersionUID = 1145305812403368160L;  private static final Logger LOG = LoggerFactory.getLogger(FileInput.class);  private static final String CONF\_FILE\_PATH = "fileinput.path";  private static final Charset CHARSET = Charset.forName("UTF-8");  /\*\*  \* File path  \*/  private String filePath;  private IEmitter emitter;  private StreamSerDe serde;  private StreamingConfig config;  /\*\*  \* File initialized by the initialize interface  \* Invoked during operating  \* Serialization transfer is not required.  \*/  private transient File file;  private transient FileLineProcessor processor;  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setConfig(StreamingConfig conf)  throws StreamingException  {  this.filePath = conf.getStringValue(CONF\_FILE\_PATH);  this.config = conf;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setEmitter(IEmitter iEmitter)  {  this.emitter = iEmitter;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setSerDe(StreamSerDe streamSerDe)  {  this.serde = streamSerDe;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public StreamingConfig getConfig()  {  return this.config;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public StreamSerDe getSerDe()  {  return this.serde;  }  /\*\*  \* Initialization  \*  \* @throws StreamingException initialization exception  \*/  @Override  public void initialize()  throws StreamingException  {  if (Strings.isNullOrEmpty(filePath))  {  LOG.error("file path is null.");  throw new StreamingException("file path is null.");  }  file = new File(filePath);  if (!file.exists())  {  LOG.error("file in path is not exists.");  throw new StreamingException("file in path is not exists.");  }  if (!file.isFile())  {  LOG.error("file in path is not a file type.");  throw new StreamingException("file in path is not a file type.");  }  processor = new FileLineProcessor();  }  /\*\*  \* destroy interface invoked during operating  \*  \* @throws StreamingException stream processing exception  \* @see [Class, Class#method, class#member]  \*/  @Override  public void destroy()  throws StreamingException  {  }  /\*\*  \* Input operator execution interface  \*  \* @throws StreamingException stream processing exception  \* @see [Class, Class#method, class#member]  \*/  @Override  public void execute()  throws StreamingException  {  try  {  Files.readLines(file, CHARSET, processor);  }  catch (IOException e)  {  throw StreamingException.wrapException(e);  }  }  private class FileLineProcessor implements LineProcessor<Object>  {  /\*\*  \* {@inheritDoc}  \*/  @Override  public boolean processLine(String line)  throws IOException  {  List<Object[]> deserResults = null;  try  {  deserResults = serde.deSerialize(line);  }  catch (StreamSerDeException e)  {  //Ignore a deserialization excemption.  LOG.warn("Ignore a serde exception.", e);  return false;  }  for (Object[] event : deserResults)  {  try  {  emitter.emit(event);  }  catch (StreamingException e)  {  //Direct emit exception, processed based on an external troubleshooting mechanism  throw new StreamingRuntimeException(e);  }  }  return true;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public Object getResult()  {  return null;  }  }  } |
| *--Procedure:*  *--Upload a jar package to any directory, such as /opt/streaming/example.jar.*  *--Run the following CQL statements:*  add jar "/opt/streaming/example/example.jar";  create input stream S1  (  C1 STRING,  C2 STRING  )  SOURCE "com.huawei.streaming.example.userdefined.operator.input.FileInput"  PROPERTIES (  "fileinput.path" = "/opt/streaming/example/input.txt"  );  create output stream rs  (  C1 STRING,  C2 STRING  )  SINK "com.huawei.streaming.example.userdefined.operator.output.FileOutput"  PROPERTIES (  "fileOutput.path" = "/opt/streaming/example/output.txt"  );  insert into stream rs select \* from S1;  submit application force example; |

## Stream Data Writing

* **Description**

The com.huawei.streaming.operator. IOutputStreamOperator interface writes operation results to external event sources.

Interface for writing stream data

| Method Name | Method Description | Parameter | Return Value |
| --- | --- | --- | --- |
| setConfig | Sets operator attributes.  This method is invoked during compilation. | StreamingConfig | Void |
| getConfig | Get operator attributes.  This method is invoked during compilation. | None | StreamingConfig |
| setSerDe | Sets a serialization or deserialization class.  This method is invoked during operating. | StreamSerDe | Void |
| getSerDe | Get a serialization or deserialization class.  This method is invoked during operating. | None | StreamSerDe |
| initialize | Initialization interface invoked during operating. | None | Void |
| execute | Execution interface, invoked during operating. | String streamName: indicates a stream name.  TupleEvent: indicates group data. | Void |
| destroy | Destroy interface, invoked during operating. | None | Void |

* **Example**

|  |
| --- |
| package com.huawei.streaming.example.userdefined.operator.output;  import java.io.File;  import java.io.IOException;  import java.nio.charset.Charset;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import com.google.common.base.Strings;  import com.google.common.io.Files;  import com.huawei.streaming.config.StreamingConfig;  import com.huawei.streaming.event.TupleEvent;  import com.huawei.streaming.exception.StreamSerDeException;  import com.huawei.streaming.exception.StreamingException;  import com.huawei.streaming.operator.IOutputStreamOperator;  import com.huawei.streaming.serde.BaseSerDe;  import com.huawei.streaming.serde.StreamSerDe;  /\*\*  \* Example of a customized interface  \* File output example  \*/  public class FileOutput implements IOutputStreamOperator  {  private static final long serialVersionUID = 153729627538127379L;  private static final Logger LOG = LoggerFactory.getLogger(FileOutput.class);  private static final String CONF\_FILE\_PATH = "fileOutput.path";  private static final Charset CHARSET = Charset.forName("UTF-8");  /\*\*  \* File path  \*/  private String filePath;  private StreamSerDe serde;  private StreamingConfig config;  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setConfig(StreamingConfig conf)  throws StreamingException  {  this.filePath = conf.getStringValue(CONF\_FILE\_PATH);  this.config = conf;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setSerDe(StreamSerDe streamSerDe)  {  this.serde = streamSerDe;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public StreamingConfig getConfig()  {  return this.config;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public StreamSerDe getSerDe()  {  return this.serde;  }  /\*\*  \* Initialization  \*  \* @throws StreamingException initialization exception  \*/  @Override  public void initialize()  throws StreamingException  {  }  /\*\*  \* {@inheritDoc}  \*  \* @param streamName  \* @param event  \*/  @Override  public void execute(String streamName, TupleEvent event)  throws StreamingException  {  if (Strings.isNullOrEmpty(filePath))  {  LOG.error("file path is null.");  throw new StreamingException("file path is null.");  }  File file = new File(filePath);  if (!file.exists())  {  LOG.error("file in path is not exists.");  throw new StreamingException("file in path is not exists.");  }  if (!file.isFile())  {  LOG.error("file in path is not a file type.");  throw new StreamingException("file in path is not a file type.");  }  String result = serializeEvent(event);  writeEvent(file, result);  }  /\*\*  \* destroy interface invoked during operating  \*  \* @throws com.huawei.streaming.exception.StreamingException stream processing exception  \* @see [Class, Class#method, class#member]  \*/  @Override  public void destroy()  throws StreamingException  {  }  private String serializeEvent(TupleEvent event)  {  String result = null;  try  {  result = (String)serde.serialize(BaseSerDe.changeEventsToList(event));  }  catch (StreamSerDeException e)  {  LOG.error("failed to serialize data ", e);  }  return result;  }  private void writeEvent(File file, String result)  throws StreamingException  {  try  {  Files.append(result, file, CHARSET);  }  catch (IOException e)  {  throw StreamingException.wrapException(e);  }  }  } |
| *--Upload a jar package to any directory, such as <b>/opt/streaming/example.jar</b>.*  *--Run the following CQL statements:*  add jar "/opt/streaming/example/example.jar";  create input stream S1  (  C1 STRING,  C2 STRING  )  SOURCE "com.huawei.streaming.example.userdefined.operator.input.FileInput"  PROPERTIES (  "fileinput.path" = "/opt/streaming/example/input.txt"  );  create output stream rs  (  C1 STRING,  C2 STRING  )  SINK "com.huawei.streaming.example.userdefined.operator.output.FileOutput"  PROPERTIES (  "fileOutput.path" = "/opt/streaming/example/output.txt"  );  insert into stream rs select \* from S1;  submit application force example; |

## UDF Function

* **Description**

Customized functions can be used in CQL expressions to process one or more columns and then provide specific results.

UDF function interfaces

| Method Name | Method Description | Parameter | Return Value |
| --- | --- | --- | --- |
| Constructor | UDF function construction method. | Map<String, String>  Configuration parameter. | None |
| evaluate | Function execution method. | Customized UDF function. The parameter type must be a data type supported by CQL. During CQL application, the parameter sequence must be the same as the sequence defined in the evaluate method. | Customized UDF function. The parameter type must be a data type supported by CQL and cannot be Object. |

Customized UDF function, inherited from the com.huawei.streaming.udfs.UDF. The function has only one construction function Config, and all parameters necessary for the function are stored in Config.

Parameters can be set by using the set command.

The UDF function implements the evaluate method, method parameters and return values are determined by users, but the data type must be supported by the system.

* **Example**

|  |
| --- |
| package com.huawei.streaming.example.udf;  import java.util.Map;  import com.huawei.streaming.udfs.UDF;  /\*\*  \* Delete special characters in the front of and at the end of a string.  \*/  public class UDFTrim extends UDF  {  private static final long serialVersionUID = 4793756788804334850L;  /\*\*  \* <Default construction function>  \*  \* Parameter necessary for the @param config udf function  \* Parameters set by the  \* in CQL  \*/  public UDFTrim(Map<String, String> config)  {  super(config);  }  /\*\*  \* UDF function execution method  \* Method name evaluate  \*  \* @param s Character string  \* @return Delete strings after spaces.  \* @see [Class, Class#method, class#member]  \*/  public String evaluate(String s)  {  if (s == null)  {  return null;  }  return s.trim();  }  } |

## Data Source

* **Description**

The com.huawei.streaming.datasource IDataSource interface obtains data from an external data source, such as a database.

Data source interfaces

| Method Name | Method Description | Parameter | Return Value |
| --- | --- | --- | --- |
| setConfig | Sets data source parameters. This method is invoked during compilation. | StreamingConfig  Query parameter. | void |
| setSchema | Sets data source query schema. This method is invoked during compilation. | TupleEventType  Schema matching a data source query result. | void |
| initialize | Initialization method, invoked during operating. |  | void |
| Execute | Data source query method, invoked during operating. | List<Object>  Replaced query parameter, which is the calculation result of an express.  This array parameter is a query parameter defined in a From substatement. This parameter is stored in a list based on a defined sequence and has been replaced during invoking. | List<Object[]>  Query result, in which the list size is the number of rows of query results, object array is the data result in each row, and data types correspond to scheme data types in the setSchema method. |
| destroy | Destroys objects inside data sources. This method is invoked during operating. |  | void |

* **Example**

|  |
| --- |
| package com.huawei.streaming.example.userdefined.datasource;  import java.io.BufferedReader;  import java.io.File;  import java.io.IOException;  import java.nio.charset.Charset;  import java.util.List;  import java.util.Properties;  import org.apache.storm.shade.org.apache.commons.io.IOUtils;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import com.google.common.base.Strings;  import com.google.common.collect.Lists;  import com.google.common.io.Files;  import com.huawei.streaming.config.StreamingConfig;  import com.huawei.streaming.datasource.IDataSource;  import com.huawei.streaming.event.TupleEventType;  import com.huawei.streaming.exception.ErrorCode;  import com.huawei.streaming.exception.StreamingException;  /\*\*  \* Data source mapping an attribute  \* Obtain local files from the node where the worker process exists.  \* Read mapping between keys and values in a configuration file.  \* Send return values.  \*/  public class PropertyMatchDataSource implements IDataSource  {  public static final Charset CHARSET = Charset.forName("UTF-8");  private static final long serialVersionUID = 8056232432674642637L;  private static final Logger LOG = LoggerFactory.getLogger(PropertyMatchDataSource.class);  private static final String CONF\_FILE\_PATH = "example.datasource.path";  private String propertyFilePath;  private Properties properties;  private StreamingConfig config;  /\*\*  \* Output schema of data source query, matching the schema definition in a query statement  \* Unused in an example  \*/  private TupleEventType schema;  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setConfig(StreamingConfig conf)  throws StreamingException  {  config = conf;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setSchema(TupleEventType tuple)  {  schema = tuple;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void initialize()  throws StreamingException  {  initConfig();  if (Strings.isNullOrEmpty(propertyFilePath))  {  LOG.error("file path is null.");  throw new StreamingException("file path is null.");  }  File file = new File(propertyFilePath);  validateFile(file);  loadProperties(file);  }  private void initConfig()  throws StreamingException  {  //Initialization used for obtaining attributes  if (config.containsKey(CONF\_FILE\_PATH))  {  this.propertyFilePath = config.get(CONF\_FILE\_PATH).toString();  }  else  {  LOG.error("can not found {} from configuration.", CONF\_FILE\_PATH);  throw new StreamingException(ErrorCode.CONFIG\_NOT\_FOUND, CONF\_FILE\_PATH);  }  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public List<Object[]> execute(List<Object> replacedQueryArguments)  throws StreamingException  {  validateArgs(replacedQueryArguments);  return evaluateValue(replacedQueryArguments.get(0));  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void destroy()  throws StreamingException  {  //Used for destroying opened resources, such as disabling streams or connections  }  private List<Object[]> evaluateValue(Object replacedQueryArgument)  {  String key = replacedQueryArgument.toString();  if (properties.containsKey(key))  {  Object[] values = {properties.get(key)};  List<Object[]> results = Lists.newArrayList();  results.add(values);  return results;  }  return null;  }  private void validateFile(File file)  throws StreamingException  {  if (!file.exists())  {  LOG.error("file in path is not exists.");  throw new StreamingException("file in path is not exists.");  }  if (!file.isFile())  {  LOG.error("file in path is not a file type.");  throw new StreamingException("file in path is not a file type.");  }  }  private void loadProperties(File file)  throws StreamingException  {  properties = new Properties();  BufferedReader reader = null;  try  {  reader = Files.newReader(file, CHARSET);  properties.load(Files.newReader(file, CHARSET));  }  catch (IOException e)  {  LOG.error("failed to read property files.", e);  throw new StreamingException("failed to read property files.");  }  finally  {  IOUtils.closeQuietly(reader);  }  }  private void validateArgs(List<Object> replacedQueryArguments)  throws StreamingException  {  if (replacedQueryArguments.size() != 1)  {  LOG.error("rdb dataSource query arguments are not equal to 1, args size : {}", replacedQueryArguments.size());  throw new StreamingException("rdb dataSource query arguments are not equal to 1, args size : "  + replacedQueryArguments.size());  }  }  } |
| *--Procedure:*  *--Upload a jar package to any directory, such as /opt/streaming/example.jar.*  *--Run add jar '/opt/streaming/example.jar';.*  *--Run the following CQL statements:*  add jar '/opt/streaming/example.jar';  create input stream S  (id int,name String)  SOURCE KafkaInput  PROPERTIES (  "operator.kafka.groupid" = "gidkpi\_1\_1",  "operator.kafka.topic" = "0912"  );  CREATE DATASOURCE propertyDataSource  SOURCE "com.huawei.streaming.example.datasource.PropertyMatchDataSource"  PROPERTIES (  "example.datasource.path" = "/opt/example.properties");  create output stream rs  (type String, id String, tag String)  SINK consoleOutput;  insert into rs select s.id,s.name,ds.value from S,  DATASOURCE propertyDataSource  [  SCHEMA (value String),  QUERY(s.id)  ] ds;  submit application datasourcetest; |

## Customized Operator

* **Description**

The com.huawei.streaming.operator.IFunctionStreamOperator interface supports single-stream input and output only.

Interfaces of customized operators

| Method Name | Method Description | Parameter | Return Value |
| --- | --- | --- | --- |
| setConfig(StreamingConfig conf) | StreamingConfig: indicates a configuration parameter | Void | Sets configuration parameters. This method is invoked during compilation. |
| getConfig() |  | Void | Get configuration parameters. This method is invoked during compilation. |
| setEmitter(Map<String, IEmitter> emitterMap) | In the Map, key indicates a stream name, and value indicates an emitter instance. | void | Sets multiple emiters. Each stream corresponds to one emiter instance. This method is invoked during operating. |
| Initialize() |  | Void | Initialization interface invoked during operating. |
| execute(String streamName, TupleEvent event) | StreamingName: indicates the name of a stream in which group data exists.  TupleEvent: indicates group data. | Void | Operating interface that processes data. |
| Destroy() | Destroy operator. | Void | Destroy operator inner objects.  This method is invoked during operating. |

* **Example**

|  |
| --- |
| package com.huawei.streaming.example.userdefined.operator;  import java.io.BufferedReader;  import java.io.File;  import java.io.IOException;  import java.nio.charset.Charset;  import java.util.List;  import java.util.Properties;  import org.apache.storm.shade.org.apache.commons.io.IOUtils;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import com.google.common.base.Strings;  import com.google.common.collect.Lists;  import com.google.common.io.Files;  import com.huawei.streaming.config.StreamingConfig;  import com.huawei.streaming.datasource.IDataSource;  import com.huawei.streaming.event.TupleEventType;  import com.huawei.streaming.exception.ErrorCode;  import com.huawei.streaming.exception.StreamingException;  /\*\*  \* Customized operator  \* If sent data exists in a file, replace the data with content in the file.  \*  \* Input schema: a string, b int  \* Output schema: c string, d int, e float  \*/  public class UserOperator implements IFunctionStreamOperator  {  private static final Logger LOG = LoggerFactory.getLogger(UserOperator.class);  public static final Charset CHARSET = Charset.forName("UTF-8");  private static final long serialVersionUID = -4438239751340766284L;  private static final String CONF\_FILE\_NAME = "userop.filename";  private String fileName;  private Properties properties;  private Map<String, IEmitter> emitters = null;  private StreamingConfig config;  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setConfig(StreamingConfig conf)  throws StreamingException  {  if (!conf.containsKey(CONF\_FILE\_NAME))  {  LOG.error("can not found config value {}.", CONF\_FILE\_NAME);  throw new StreamingException("can not found config value " + CONF\_FILE\_NAME + ".");  }  fileName = conf.getStringValue(CONF\_FILE\_NAME);  this.config = conf;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public StreamingConfig getConfig()  {  return this.config;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setEmitter(Map<String, IEmitter> emitterMap)  {  if (emitterMap == null || emitterMap.isEmpty())  {  LOG.error("can not found emitter.");  throw new StreamingRuntimeException("can not found config value " + CONF\_FILE\_NAME + ".");  }  emitters = emitterMap;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void initialize()  throws StreamingException  {  File file = new File(fileName);  validateFile(file);  loadProperties(file);  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void execute(String streamName, TupleEvent event)  throws StreamingException  {  Object[] values = event.getAllValues();  Object[] result = new Object[3];  if (properties.containsKey(String.valueOf(values[0])))  {  result[0] = properties.get(String.valueOf(values[0]));  result[1] = 1;  result[2] = 1.0f;  }  else  {  result[0] = "NONE";  result[1] = 1;  result[2] = 1.0F;  }  for (IEmitter emitter : emitters.values())  {  emitter.emit(result);  }  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void destroy()  throws StreamingException  {  }  private void validateFile(File file)  throws StreamingException  {  if (!file.exists())  {  LOG.error("file in path is not exists.");  throw new StreamingException("file in path is not exists.");  }  if (!file.isFile())  {  LOG.error("file in path is not a file type.");  throw new StreamingException("file in path is not a file type.");  }  }  private void loadProperties(File file)  throws StreamingException  {  properties = new Properties();  BufferedReader reader = null;  try  {  reader = Files.newReader(file, CHARSET);  properties.load(Files.newReader(file, CHARSET));  }  catch (IOException e)  {  LOG.error("failed to read property files.", e);  throw new StreamingException("failed to read property files.");  }  finally  {  IOUtils.closeQuietly(reader);  }  }  } |
| *-Procedure:*  *--Upload a jar package to any directory, such as /opt/streaming/example.jar.*  *--Run add jar '/opt/streaming/example.jar';.*  *--Run the following CQL statements:*  add jar '/opt/streaming/example.jar';  create input stream s (id int, name string)  SOURCE KafkaInput  PROPERTIES ( "operator.kafka.groupid" = "gidkpi\_1\_1","operator.kafka.topic"="agg\_1\_1");  create output stream rs  (id string, name string, type int)  SINK KafkaOutput properties("operator.kafka.topic"="agg\_2");  create operator userOp as "com.huawei.streaming.example.userdefined.operator.UserOperator"  input (id int, name string)  output (newID string, name string, type int)  properties ("userop.filename" = "/home/omm/kv.properties");  insert into rs using operator userOp from s distribute by id parallel 2;  submit application force tt; |

# Internal System Interface

This chapter describes implementation and parameters of internal system interfaces that are used in CQL.

Default values of parameters of internal interfaces are reference only.



Names of all parameters of internal system interfaces are case insensitive.

[11.1 Serialization and Deserialization](#_EN-US_TOPIC_0029436591)

[11.2 Data Reading](#_EN-US_TOPIC_0029436547)

[11.3 Data Writing](#_EN-US_TOPIC_0029436533)

[11.4 Function](#_EN-US_TOPIC_0029436540)

[11.5 Data Source](#_EN-US_TOPIC_0029436525)

## Serialization and Deserialization

### SimpleSerde

* **Function**

This interface splits data based on a specific separator and deserializes data based on schema types.



This interface deserializes data of the boolean type. The true string (case insensitive) is converted into true, and false (case insensitive) is converted into false. Other characters are converted into null.

* **Parameter**

Parameters of the default serialization and deserialization interface

| Parameter | Default Value | Description |
| --- | --- | --- |
| separator | , | Separator between columns. |
| timezoneForTimestamp | ${system:user.timezone} | Data serialization and deserialization time zone setting. The client time zone is used by default.  The parameter format can be either of the following modes:  GMT+08:00  Asia/Shanghai |

* **Example**

|  |
| --- |
| CREATE INPUT STREAM S  (id INT, name STRING )  SERDE simpleSerde  PROPERTIES ( "separator" = "," )  SOURCE KafkaInput  PROPERTIES (  "operator.kafka.groupid" = "gidkpi\_1\_1",  "operator.kafka.topic" = "simple\_1"  ); |

### KeyValueSerDe

* **Function**

This interface serializes data as data in the KeyValue mode, such as key1=value1 and key2=value2.



This interface deserializes data of the boolean type. The true string (case insensitive) is converted into true, and false (case insensitive) is converted into false. Other characters are converted into null.

During Keyvalue serialization and deserialization, keys and values are connected by using equal marks (=) which cannot be replaced. Therefore, during serialization and deserialization of data of the keyvalue format, keys and values cannot contain equal marks (=); otherwise, results are incorrect.

* **Parameter**

Parameters of the interface for serializing and deserializing data of the KeyValue format

| Parameter | Default Value | Description |
| --- | --- | --- |
| separator | , | Separator between columns. |
| timezoneForTimestamp | ${system:user.timezone} | Data serialization and deserialization time zone setting. The client time zone is used by default.  The parameter format can be either of the following modes:  GMT+08:00  Asia/Shanghai |

* **Example**

|  |
| --- |
| CREATE INPUT STREAM S  (id INT, name STRING )  SERDE keyValueSerde  PROPERTIES ( "separator" = "," )  SOURCE KafkaInput  PROPERTIES (  "operator.kafka.groupid" = "gidkpi\_1\_1",  "operator.kafka.topic" = "simple\_1"  ); |

### CSVSerDe

* **Function**

This interface deserializes data of the CSV format.



This interface deserializes data of the boolean type. The true string (case insensitive) is converted into true, and false (case insensitive) is converted into false. Other characters are converted into null.

If a column contains separators, the column must be quoted by quotations. One double quotation marks in the CSV format are replaced with two double quotation marks.

Examples of data of the CSV format

| Source Data | CSV Data |
| --- | --- |
| 1,a,b | 1,a,b |
| 2,"a","b" | 2,"""a""","""b""" |
| 3,a,b,"a,b" | 3,"a,b","""a,b""" |

* **Parameter**

|  |  |  |
| --- | --- | --- |
| timezoneForTimestamp | ${system:user.timezone} | Data serialization and deserialization time zone setting. The client time zone is used by default.  The parameter format can be either of the following modes:  GMT+08:00  Asia/Shanghai |

* **Example**

|  |
| --- |
| CREATE INPUT STREAM S  (id INT, name STRING )  SERDE CSVSerDe  SOURCE KafkaInput  PROPERTIES (  "operator.kafka.groupid" = "gidkpi\_1\_1",  "operator.kafka.topic" = "simple\_1"  ); |

### BinarySerDe

* **Function**

This interface serializes and deserializes byte codes and generally be used with TCP-related output and input. This interface obtains each attribute from byte codes based on lengths.

A fixed length is defined for each data type. If length of data is less than the fixed length, obtaining the data fails.

Binary length of each data type

| Data Type | Length (Eight-Digit Byte) |
| --- | --- |
| Boolean | 1 |
| Int | 4 |
| Long | 8 |
| Float | 4 |
| Double | 8 |
| Decimal | Minimum length 5 |
| Time | Minimum length 8 for data of the string format and 8 for data of the long type |
| Date | Minimum length 10 for data of the string format and 8 for data of the long type |
| Timestamp | Minimum length 19 for data of the string format and 8 for data of the long type |
| String | Minimum length 1 |



This interface deserializes data of the boolean type. The true string (case insensitive) is converted into true, and false (case insensitive) is converted into false. Other characters are converted into null.

Int, long, flat, and double data types require symbols, whereas the C type does not require symbols. Therefore, positive and negative relations must be considered during cross-platform access.

* **Parameter**

Parameters of the interface for serializing and deserializing binary data

| Parameter | Default Value | Description |
| --- | --- | --- |
| attributesLength | None | Length of each attribute field. |
| timeType | Long | Time type.  String or Long type. The default type is Long.  The value is case-insensitive.  If this parameter is set, serialization and deserialization of date, time, and timestamp of all binaryserde data in an operator are affected, and other operators are not affected. |
| decimalType | decimal | decimal type.  String or decimal type. The default type is decimal.  The value is case-insensitive.  If this parameter is set, serialization and deserialization of binaryserde decimal data in an operator are affected, and other operators are not affected. |
| timezoneForTimestamp | ${system:user.timezone} | Data serialization and deserialization time zone setting. The client time zone is used by default.  The parameter format can be either of the following modes:  GMT+08:00  Asia/Shanghai |

* **Example**

|  |
| --- |
| CREATE INPUT STREAM call  (  CallingNumber STRING,  CalledNumber STRING,  InitTime LONG  )  SERDE BinarySerDe  PROPERTIES ( "attributesLength" = "26,26,8" )  SOURCE TCPClientInput  PROPERTIES ( "server" = "127.0.0.1", "port" = "9999" ); |

## Data Reading

### KafkaInput

* **Function**

This operator obtains data from Kafka message queues.

* **Parameter**

Parameters of the Kafka input operator

| Parameter | Default Value | Description |
| --- | --- | --- |
| operator.kafka.groupid | None | Group ID, which is a client ID. |
| operator.kafka.topic | None | Type of a message. Messages of different types use different topics. |
| operator.kafka.zookeepers | zookeeper address used in Kafka in a cluster. | zookeeper address for obtaining kafka data. Ports are added to zookeeper addresses. Multiple addresses are separated with commas (,), such as 192.168.0.20:2181, 192.168.0.21:2181/kafka. |
| operator.kafka.zksessiontimeout | 20000 | Timeout interval for a zookeeper connection. |
| operator.kafka.zksynctime | 20000 | kafka zookeeper time synchronization parameter. |
| operator.kafka.messageserializerclass | kafka.serializer.StringEncoder | Class for serializing data obtained from kafka consumers. |
| operator.kafka.read.frombeginning | False | Indicates whether data is read from topic. If the value is false, data is read from the latest position found in kafka after kafka starts. |

* **Example**

|  |
| --- |
| CREATE INPUT STREAM S  (id INT, name STRING )  SERDE simpleSerde  PROPERTIES ( separator = "," )  SOURCE KafkaInput  PROPERTIES (  "operator.kafka.groupid" = "gidkpi\_1\_1",  "operator.kafka.topic" = "simple\_1",  "operator.kafka.zookeepers" = "localhost:2181,10.0.0.21:2181"  ); |

### TCPClientInput

* **Function**

This interface obtains data from a remote server on a TCP client.

* **Parameter**

Input parameters of a TCP client

| Parameter | Default Value | Description |
| --- | --- | --- |
| server | None | IP address of a TCP server. |
| port | None | Listening port of a TCP server. |
| sessionTimeout | 2000 | Timeout interval for a TCP client connection. The unit is millisecond. |

* **Example**

|  |
| --- |
| CREATE INPUT STREAM transaction  (  C1 LONG,  C2 STRING  )  SOURCE TCPClientInput  PROPERTIES (  "server" = "127.0.0.1",  "port" = "9999" ); |

### RamdomGen

* **Function**

This interface generates input sources of random numbers.

* **Parameter**

Parameters of the operator for generating random numbers

| Parameter | Default Value | Description |
| --- | --- | --- |
| timeUnit | SECONDS | Time unit. The default value is second. For example, MILLISECONDS, SECONDS, MINUTES, HOURS, and DAY. |
| period | 1 | Time period. The default value is one second. |
| eventNumPerPeriod | 1 | Number of pieces of event data sent in which time period. The default value is one. That is, one piece of data is sent in each second. |
| isSchedule | false | Indicates whether data generated at random is sent as scheduled. |
| totalNumber | 0 | Number of pieces of data to be sent. The value 0 indicates that the number of pieces of data to be sent is not limited. |
| delayTime | 0 | Delayed time for sending data. The value 0 indicates that data is immediately sent. |

* **Example**

|  |
| --- |
| CREATE INPUT STREAM S  (id INT, name STRING, type INT)  SOURCE randomgen  PROPERTIES (  "timeUnit" = "SECONDS",  "period" = "1",  "eventNumPerPeriod" = "1",  "isSchedule" = "true",  "totalNumber" = "20000",  "delayTime" = "0"); |

## Data Writing

### KafkaOutput

* **Function**

This operator writes data to kafka message queues.

* **Parameter**

Parameters of the Kafka output operator

| Parameter | Default Value | Description |
| --- | --- | --- |
| operator.kafka.topic | None | Type of a message. Messages of different types use different topics. |
| operator.kafka.zookeepers | zookeeper address used in Kafka in a cluster. | zookeeper address for obtaining kafka data. Ports are added to zookeeper addresses. Multiple addresses are separated with commas (,), such as 10.0.0.5:2181, 10.0.0.6:2181/kafka. |
| operator.kafka.brokers | brokers address used in Kafka in a cluster. | Address for consumers to obtain message metadata (including topics, partitions, and replicas). The value is in the format of host1:port1,host2:port2. |
| operator.kafka.zksessiontimeout | 20000 | Timeout interval for a zookeeper connection. |
| operator.kafka.zksynctime | 20000 | kafka zookeeper time synchronization parameter. |
| operator.kafka.messageserializerclass | kafka.serializer.StringEncoder | Class for serializing data obtained from kafka consumers. |

* **Example**

|  |
| --- |
| CREATE OUTPUT STREAM rs  (id STRING, name STRING)  SINK KafkaOutput  PROPERTIES (  "operator.kafka.topic" = "simple\_2",  "operator.kafka.zookeepers" = "10.0.0.12:2181",  "operator.kafka.brokers" = "10.0.0.12:9092"); |

### TCPClientOutput

* **Function**

This interface sends operation results from a TCP client to a specific server.

* **Parameter**

Parameters of the output operator of a TCP client

| Parameter | Default Value | Description |
| --- | --- | --- |
| server | None | IP address of a TCP server. |
| port | None | Listening port of a TCP server. |
| sessionTimeout | 2000 | Timeout interval for a TCP client connection. The unit is millisecond. |

* **Example**

|  |
| --- |
| CREATE OUTPUT STREAM FilterHO  (  IMSI STRING,  EventID STRING,  HOCount INT  )  SINK TCPClientOutput  PROPERTIES(  "server" = "127.0.0.1",  "port" = "6001",  "sessionTimeout"='2000' ); |

### ConsoleOutput

* **Function**

This interface prints received data on a console, but data output is not performed.

* **Parameter**

Parameters of the console output operator

| Parameter | Default Value | Description |
| --- | --- | --- |
| printFrequence | 1 | Frequency for printing data. Each piece of data is printed by default. If this parameter is set to 10, every 10 messages are printed. |

* **Example**

|  |
| --- |
| CREATE OUTPUT STREAM FilterHO  (  IMSI STRING,  EventID STRING,  HOCount INT  )  SINK ConsoleOutput  PROPERTIES("printFrequence" = "10"); |

## Function

Functions can be used in select, where, and having substatements. Currently, functions are not used in group by and order by substatements.

Usually used functions are defined in the system. Functions can be manually defined as required.

If names of customized functions are the same as names of built-in system functions, customized functions overwrite system functions in the current thread, but other users are not affected. Customized functions will not overwrite system functions when the process ends.

### Character Processing Function

This function processes characters, such as obtaining parameter value lengths, converting parameter types, combining parameters, and replacing certain content in parameters.

Character string processing function

| Function | Return Value | Description | Example |
| --- | --- | --- | --- |
| substr(string s, int from) | String | This function extracts sub strings from the string parameter.  In the function, from indicates the extraction start position.  The index count for strings starts from 1.  If the value of from is 0, processing is performed based on the value 1.  If the value of from is a positive number, all characters from from to the end are extracted.  If the value of from is negative number, the last n (absolute value of from) characters in a string are extracted. | substr("abc",2) --bc  substr("abc",0) --abc  substr("abc",1) --abc  substr("abc",-1) --c |
| substr(string s, int from, int count) | String | This function extracts sub strings from the string parameter.  In the function, from indicates the extraction start position.  The index count for strings starts from 1.  In the function, count indicates the length of an extracted sub string.  If the value of from is 0, processing is performed based on the value 1.  If the value of from is a positive number, the first count characters from from are extracted.  If the value of from is negative number, n (absolute value of from) characters from the nth character in a string are extracted.  If the value of count is less than 1, a blank string is returned. | substr("abc",0,2) --ab  substr("abc",1,2) --ab  substr("abc",-1,1) --c  substr("abc",2,-1) --Blank string |
| strlength(string s) | Int | Length of a string.  If the string is null, 0 is returned. | strlength(null) --0  strlength("") --0  strlength("abc") --3 |
| trim(String s) | String | Deletes special characters at the beginning and end of a string.  Special characters refer to characters whose ASCII code is less than 33, including spaces, table creation symbols, backspace, and Enter. For details, see the ASCII code table. | trim(null) --Blank string  trim("") --Blank string  trim(" ") --Blank string  trim("abc") --abc  trim("abc ") --abc  trim("abc") --abc  trim("abc\n") --trim Enter at the end of abc |
| concat(String s1, String s2) | String | Combine strings based on the parameter sequence. If any parameter is null, all results are null. | concat("a","b") --ab  concat("ab","") --ab  concat("ab",null) --null |
| concat(String s1, String s2, String s3) | String | Combine strings based on the parameter sequence. If any parameter is null, all results are null. | concat("a","b","c") --abc  concat("ab","","c") --abc  concat("ab",null,"c") --null |
| concat(String s1, String s2, String s3, String s4) | String | Combine strings based on the parameter sequence. If any parameter is null, all results are null. | concat("a","b","c","d") --abcd  concat("ab","","c","d") --abcd  concat("ab",null,"c","d") --null |
| concat(String s1, String s2, String s3, String s4, String s5) | String | Combine strings based on the parameter sequence. If any parameter is null, all results are null. | concat("a","b","c","d","e") --abcde  concat("ab","","c","d","e") --abcde  concat("ab",null,"c","d","e") --null |
| concat(String s1, String s2, String s3, String s4, String s5, String s6) | String | Combine strings based on the parameter sequence. If any parameter is null, all results are null. | concat("a","b","c","d","e","f") --abcdef  concat("ab","","c","d","e","f") --abcdef  concat("ab",null,"c","d","e","f") --null |
| concat(String s1, String s2, String s3, String s4, String s5, String s6, String s7) | String | Combine strings based on the parameter sequence. If any parameter is null, all results are null. | concat("a","b","c","d","e","f","g") --abcdefg  concat("ab","","c","d","e","f","g") --abcdefg  concat("ab",null,"c","d","e","f","g") --null |
| upper(String s) | String | Converts a string into an upper-case string.  The language is US. | upper("abc") --ABC  upper("aBc") --ABC |
| lower(String s) | String | Converts a string into a lower-case string.  The language is US. | lower("ABC") --abc  lower("aBc") --abc |

### Time Processing Function

This function performs operations on time data, including adding, reducing, and obtaining years, months, and days.

Time processing function

| Function | Return Value | Description | Example |
| --- | --- | --- | --- |
| currenttimemillis() | Long | Current time, in millisecond.  Time from 00:00:00.000 on 1970-00-00, in millisecond. | currenttimemillis() --1414036004463  --The from\_unixtime function can be used to verify the result. The from\_unixtime input parameter is accurate to second, and the return value of Currenttimemillis is accurate to millisecond. |
| day(String dateString) | Int | Number of days in the month specified by a time string.  The time string can be of the date or timestamp type, and the format is the same as the type of CQL data. | day("2013-02-28") --28  day("2013-02-29") --null  day("2013-02-29 09:00:01") --null  day("2013-02-28 09:00:01") --28  day("2013-02-28 09:00:01.0") –28  day("2013-02-28 09:00:01.0+0800") --28  day("2013-02-28 09:00:01.0 -0800") --1 |
| dayofmonth(String dateString) | Int | Number of days in the month specified by a time string.  The time string can be of the date or timestamp type, and the format is the same as the type of CQL data. | dayofmonth("2013-02-28") --28  dayofmonth("2013-02-29") --null  dayofmonth("2013-02-29 09:00:01")--null  dayofmonth("2013-02-28 09:00:01")--28  dayofmonth("2013-02-28 09:00:01.0") –28  dayofmonth("2013-02-28 09:00:01.0+0800") --28  dayofmonth("2013-02-28 09:00:01.0 -0800") --1 |
| month(String dateString) | Int | Month specified in a time string.  The time string can be of the date or timestamp type, and the format is the same as the type of CQL data. | month("2013-02-28") --2  month("2013-02-29") --null  month("2013-02-29 09:00:01")--null  month("2013-02-28 09:00:01") --2  month("2013-02-28 09:00:01.0") –2  month("2013-02-28 09:00:01.0+0800") --2  month("2013-02-28 09:00:01.0 -0800") --3 |
| year(String dateString) | Int | Year specified in a time string.  The time string can be of the date or timestamp type, and the format is the same as the type of CQL data. | year("2013-10-17 09:58:00.111")--2013  year("2013-10-17 09:58:00") –2013  year("2012-12-31 17:58:00.111-0800") --2013  year("2013-10-17 25:62:00")--null  year("2013-00-17") --null  year("2013-10-17") --2013  year("09:58:00")--null |
| hour(String dateString) | Int | Hour specified in a time string.  The time string can be of the time or timestamp type, and the format is the same as the type of CQL data. | hour("2013-10-17 09:58:00.111")--9  hour("2013-10-17 09:58:00") –9  hour("2013-10-17 09:58:00.111+0800") --9  hour("2013-10-17 09:58:00.111-0800") --1  hour("09:58:00") --9  hour("24:58:00") --null  hour("00:58:00") --0  hour("2013-10-17")--null |
| minute(String dateString) | Int | Minute specified in a time string.  The time string can be of the time or timestamp type, and the format is the same as the type of CQL data. | minute("2013-10-17 09:58:00.111")-- 58  minute("2013-10-17 09:58:00") –58  minute("2013-10-17 09:58:00.111+0030") --28  minute("2013-10-17 09:58:00.111-0030") --28  minute("09:58:00") -- 58  minute("09:60:00") -- null  minute("09:00:00") -- 0  minute("2013-10-17")-- null |
| second(String dateString) | Int | Second specified in a time string.  The time string can be of the time or timestamp type, and the format is the same as the type of CQL data. | second("2013-10-17 09:58:00.111")-- 0  second("2013-10-17 09:58:00") – 0  second("2013-10-17 09:58:01.111+0800") -- 1  second("09:58:00") -- 0  second("24:58:60") -- null  second("23:59:59") -- 59  second("24:00:00") -- null  second("2013-10-17")-- null |
| weekofyear(String dateString) | Int | Number of weeks in the year specified in a time string.  The time string can be of the date or timestamp type, and the format is the same as the type of CQL data. | weekofyear("2013-01-01 09:58:00.111") -- 1  weekofyear("2013-01-01 09:58:00")– 1  weekofyear("2013-01-01 02:58:20+0800") -- 1  weekofyear("2013-01-01 22:58:20-0800") -- 1  weekofyear("2013-12-28") -- 52  weekofyear("2013-12-52") -- null  weekofyear("2013-01-01 09:58:70") -- null |
| weekofyear(Date date) | Int | Number of weeks in a specified year. | weekofyear(todate("2013-1-1")) --1  weekofyear(todate("2013-01-01"))-- 1  weekofyear(todate("2013-12-28"))-- 52  weekofyear(todate("2013-12-52"))-- null |
| weekofyear(TimeStamp timestamp) | Int | Number of weeks in a specified year. | weekofyear(totimestamp("2013-01-01 09:58:00.111")) – 1  weekofyear(totimestamp("2013-01-01 02:58:20 +0800")) -- 1  weekofyear(totimestamp("2013-01-01 22:58:20 -0800")) -- 1  weekofyear(totimestamp("2013-01-01 09:58:00")) -- 1  weekofyear(totimestamp("2013-01-01 09:58:70")) -- null |
| from\_unixtime(int unixTime, String format) | String | Converts time to time of a specific format.  The time parameter is accurate to second.  Time formats include the following:  "yyyy-MM-dd HH:mm:ss"  "yyyy-MM-dd"  "HH:mm:ss"  "yyyy-MM-dd HH:mm:ss.SSS" | from\_unixtime(0, "yyyy-MM-dd") -- "1970-01-01" |
| from\_unixtime(long unixTime) | String | Converts time to time of a specific format.  The time parameter is accurate to second.  The default time format is as follows:  "yyyy-MM-dd HH:mm:ss" | from\_unixtime(0L) -- "1970-01-01 08:00:00"  from\_unixtime(1L) -- "1970-01-01 08:00:01"  from\_unixtime(100L) -- "1970-01-01 08:01:40"  from\_unixtime(-999999L) -- "1969-12-20 18:13:21" |
| from\_unixtime(long unixTime,String format) | String | Converts time to time of a specific format.  The time parameter is accurate to second.  Time formats include the following:  "yyyy-MM-dd HH:mm:ss"  "yyyy-MM-dd"  "HH:mm:ss"  "yyyy-MM-dd HH:mm:ss.SSS" | from\_unixtime(0L, "yyyy-MM-dd") -- "1970-01-01"  from\_unixtime(0L, "HH:mm:ss") -- "08:00:00"  from\_unixtime(0L, "yyyy-MM-dd HH:mm:ss") -- "1970-01-01 08:00:00"  from\_unixtime(0L, "yyyy-MM-dd HH:mm:ss.SSS") -- "1970-01-01 08:00:00.000" |
| datediff(String dateString1,String dateString2) | Int | Number of days between two time points.  The time string type can be date or timestmap. The type must be consistent with the CQL data type.  The string types of the output and input data must be the same. | datediff("2013-10-17 09:42:00","2013-10-17 09:42:01") – 0  datediff("2013-10-17 00:00:00","2013-10-16 23:59:59") -- 1  datediff("2013-10-17 09:42:00","2013-10-16 09:42:01") -- 1  datediff("2013-10-17 09:42:00","2013-10-16 09:41:01") – 1  datediff("2013-10-17 22:42:00 +0800","2013-10-17 22:41:01 -0800") -- -1  datediff("2013-10-17 09:42:00","2013-10-16 09:100:01") – null  datediff("2013-10-17","2013-10-16 09:41:01") -- 1  datediff("2013-10-17","2013-10-16") -- 1  datediff("2013-10-16","2013-10-17") -- -1  datediff("2013-10-16","2013") -- null |
| dateadd(String dateString,int days) | String | Adds days of a specific quantity to a specific date.  The time string type can be date or timestmap. The type must be consistent with the CQL data type.  The string types of the output and input data must be the same. | dateadd("2013-10-17 09:42:00.111", 1) -- "2013-10-18 09:42:00.111 +0800"  dateadd("2013-10-17 09:42:00", 1)-- "2013-10-18 09:42:00.000 +0800"  dateadd("2013-10-17 22:42:00-0800", 1) -- "2013-10-19 14:42:00.000 +0800"  dateadd("2013-10-17", 1) --"2013-10-18"  dateadd("2013-10-17", -1) --"2013-10-16"  dateadd("2013-10-17 09:100:00",1)-- null  dateadd("2013-10-17 09:100:00",10000) -- null |
| dateadd(TimeStamp timestamp,int days) | String | Adds days of a specific quantity to a specific date.  The output data type is the same as the CQL timestamp data type. | dateadd(totimestamp("2013-10-17 09:42:00"), 1) -- "2013-10-18 09:42:00.111 +0800"  dateadd(totimestamp("2013-10-1709:42:00"), -1) -- "2013-10-16 09:42:00.000 +0800"  " |
| datesub(String dateString,int days) | String | Deducts days of a specific quantity from a specific date.  The string types of the output and input data must be the same. | datesub("2013-10-17 09:42:00.111", 1) -- "2013-10-16 09:42:00.111 +0800"  datesub("2013-10-17 09:42:00", 1)-- "2013-10-16 09:42:00.000 +0800"  datesub("2013-10-17 09:42:00-0800", 1) -- "2013-10-17 01:42:00.000 +0800"  datesub("2013-10-17", 1) --"2013-10-16"  datesub("2013-10-17", -1) --"2013-10-18"  datesub("2013-10-17 09:100:00",1)-- null  datesub("2013-10-17 09:100:00",10000) -- null |
| datesub(TimeStamp timestamp,int days) | String | Deducts days of a specific quantity from a specific date.  A timestamp string is returned. | datesub(totimestamp("2013-10-17 09:42:00"), 1) -- "2013-10-16 09:42:00.000 +0800"  datesub(totimestamp("2013-10-1709:42:00"), -1) -- "2013-10-18 09:42:00.000 +0800" |

### Mathematics Function



Users must ensure that all mathematics operations do not overflow because the system does not check whether the mathematics operations overflow.

Mathematics functions

| Function | Return Value | Description | Example |
| --- | --- | --- | --- |
| abs(int number) | Int | Absolute value of the integer type.  Note: Users must ensure that data does not overflow because the system does not check whether data overflows. | abs(1) -- 1  abs(-1) -- 1  abs(0) -- 0  abs(-0) -- 0  abs(2147483647) -- 2147483647  abs(-2147483648) -- 2147483648 |
| abs(long number) | Long | Absolute value of the integer type.  Note: Users must ensure that data does not overflow because the system does not check whether data overflows. | abs(1L) -- 1  abs(-1L) -- 1  abs(0L) -- 0  abs(-0L) -- 0 |
| abs(float number) | float | Absolute value of the integer type.  Note: Users must ensure that data does not overflow because the system does not check whether data overflows. | abs(1.0f) -- 1.0  abs(-1.0f) -- 1.0  abs(0.0F) -- 0.0  abs(-0.0F) -- 0.0 |
| abs(double number) | double | Absolute value of the integer type.  Note: Users must ensure that data does not overflow because the system does not check whether data overflows. | abs(1.0d) -- 1.0  abs(-1.0d) -- 1.0  abs(0.0D) -- 0.0  abs(-0.0D) -- 0.0 |
| abs(decimal number) | decimal | Absolute value of the integer type.  Note: Users must ensure that data does not overflow because the system does not check whether data overflows. | abs(1.0BD) -- 1.0  abs(-1.0BD) -- 1.0  abs(0.0BD) -- 0.0  abs(-0.0BD) -- 0.0 |

### Type Conversion Function

Type conversion function

| Function | Return Value | Description | Example |
| --- | --- | --- | --- |
| tostring(object obj) | String | Converts transferred data into strings.  All data types are supported. | tostring(true) -- "true"  tostring(false) -- "false"  tostring(null) -- null  tostring(1) -- "1"  tostring(1L) -- "1"  tostring(1.0F) -- "1.0"  tostring(1.0D) -- "1.0"  tostring(toDecimal("1.0")) --"1.0"  tostring(toTime("15:40:00")) --"15:40:00"  tostring(toDate("2013-10-17")) --"2013-10-17"  tostring(toDate("2013-1-17")) --"2013-01-17"  tostring(toTimeStamp("2013-10-17 15:40:00.000")) -- "2013-10-17 15:40:00.000 +0800"  tostring(toTimeStamp("2013-10-17 15:40:00")) -- "2013-10-17 15:40:00.000.000 +0800" |
| toint(string number)  toint(int number)  toint(long number)  toint(float number)  toint(double number)  toint(date number)  toint(time number)  toint(timestamp number)  toint(decimal number) | int | Converts data of a specific type into data of the int type. | toint(1) -- 1  toint(1L) -- 1  toint(1.0f) -- 1  toint(1.4f) -- 1  toint(1.5f) -- 1  toint(1.6f) -- 1  toint(1.9f) -- 1  toint(1.9d) -- 1  toint("1") -- 1  toint("1.9") -- null  toint(toDecimal("1.9")) -- 1  toint(toDate("1970-01-01")) -- 0  toint(toTime("15:40:00")) -- 56400000  toint(toTimeStamp("1970-01-01 15:40:00.000")) -- 27600000 |
| tolong(string number)  tolong(int number)  tolong(long number)  tolong(float number)  tolong(double number)  tolong(date number)  tolong(time number)  tolong(timestamp number)  tolong(decimal number) | long | Converts data of a specific type into data of the long type. | tolong(1) -- 1L  tolong(1L) -- 1L  tolong(1.0f) -- 1L  tolong(1.4f) -- 1L  tolong(1.5f) -- 1L  tolong(1.6f) -- 1L  tolong(1.9f) -- 1L  tolong(1.9d) -- 1L  tolong("1") -- 1L  tolong("1.9") -- null  tolong(toDecimal("1.9")) -- 1L  tolong(toDate("1970-01-01")) --0L  tolong(toTime("15:40:00")) -- 56400000L  tolong(toTimeStamp("1970-01-01 15:40:00.000")) -- 27600000L |
| tofloat(string number)  tofloat(int number)  tofloat(long number)  tofloat(float number)  tofloat(double number)  tofloat(date number)  tofloat(time number)  tofloat(timestamp number)  tofloat(decimal number) | float | Converts data of a specific type into data of the float type. | tofloat(1) -- 1.0F  tofloat(1F) -- 1.0F  tofloat(1.0f) -- 1.0F  tofloat(1.4f) -- 1.4F  tofloat(1.5f) -- 1.5F  tofloat(1.6f) -- 1.6F  tofloat(1.9f) -- 1.9F  tofloat(1.9d) -- 1.9F  tofloat("1") -- 1.0F  tofloat("1.9") -- 1.9F  tofloat(toDecimal("1.9")) -- 1.9F  tofloat(toDate("1970-01-01")) --0.0F  tofloat(toTime("15:40:00")) -- 56400000.0F  tofloat(toTimeStamp("1970-01-01 15:40:00.000")) -- 27600000.0F |
| todouble(string number)  todouble(int number)  todouble(long number)  todouble(float number)  todouble(double number)  todouble(date number)  todouble(time number)  todouble(timestamp number)  todouble(decimal number) | double | Converts data of a specific type into data of the double type. | todouble(1) -- 1.0D  todouble(1F) -- 1.0D  todouble(1.0f) -- 1.0D  todouble(1.9d) -- 1.9D  todouble("1") -- 1.0D  todouble("1.9") -- 1.9D  todouble(toDecimal("1.9")) --1.9D  todouble(toDate("1970-01-01")) --0.0D  todouble(toTime("15:40:00")) -- 56400000.0D  todouble(toTimeStamp("1970-01-01 15:40:00.000")) -- 27600000.0D |
| toboolean(string value)  toboolean(int value)  toboolean(long value)  toboolean(float value)  toboolean(double value)  toboolean(date value)  toboolean(time value)  toboolean(timestamp value)  toboolean(decimal value)  toboolean(boolean value) | boolean | Converts data of a specific type into data of the boolean type.  If a number type or time type of the long type is not 0, true is returned.  The return value of all non-blank strings is true. | toboolean(0) -- false  toboolean(1) -- true  toboolean(0l) -- false  toboolean(1l) -- true  toboolean(0F) -- false  toboolean(1f) -- true  toboolean(0d) -- false  toboolean(1D) -- true);  toboolean(0.9d) -- false  toboolean(0.9f) -- false  toboolean(0BD) -- false  toboolean(0.0BD) -- false  toboolean(1BD) -- true  toboolean("") -- false  toboolean("0") -- true  toboolean("1") -- true  toboolean("true") -- true  toboolean("false") -- true  toboolean(totime("17:06:00")) --true  toboolean(todate("2013-10-17"))-- true  toboolean(totimestamp("2013-10-17 17:06:00.000")) -- true |
| todate(string s) | date | Converts data of a specific type into data of the date type.  Character strings in the yyyy-[M]M-[d]d format are supported. | todate("2014-09-25 17:07:00") -- null  todate("2014-09-25") -- "2014-09-25"  todate("17:07:00") -- -- null |
| todate(string s, string format) | date | Converts data of a specific type into data of the int type.  Time formats include the following:  "yyyy-MM-dd HH:mm:ss"  "yyyy-MM-dd"  "yyyy-MM-dd HH:mm:ss.SSS" | todate("2014-09-25 17:07:00", "yyyy-MM-dd HH:mm:ss") -- "2014-09-25"  todate("2014-09-25 17:07:01", "yyyy-MM-dd HH:mm:ss") -- "2014-09-25"  todate("2014-09-25", "yyyy-MM-dd") -- "2014-09-25" |
| totime(string s) | time | Converts data of a specific type into data of the time type. | totime("2014-09-25 17:07:00") -- null  totime("17:07:00") -- "17:07:00"  totime("17:07:90") -- null  totime("16:44:00") -- "16:44:00" |
| totimestamp(string s) | timestamp | Converts data of a specific type into data of the timestamp type. | totimestamp("2014-09-25 17:07:00") -- "2014-09-25 17:07:00.000 +0800"  totimestamp("2014-09-25 17:07:00.056") -- "2014-09-25 17:07:00.056.000 +0800"  totimestamp("abc") -- null  totimestamp("2014-09-25 17:07:00.56") -- "2014-09-25 17:07:00.56.000 +0800"  totimestamp("2014-09-25 17:07:00.999") -- "2014-09-25 17:07:00.56.999 +0800"  totimestamp("2014-09-25 17:07:00.9999999999") -- null  totimestamp("2014-9-9 7:7:5") --"2014-09-09 07:07:05.000 +0800"  totimestamp("2014-9-9 7:7:5.9") -- "2014-09-09 07:07:05.009 +0800" |
| todecimal(string number)  todecimal(int number)  todecimal(long number)  todecimal(double number)  todecimal(float number) | decimal | Converts data of a specific type into data of the decimal type. | todecimal(1) -- "1"  todecimal(1F) -- "1.0"  todecimal(1.0f) -- "1.0"  todecimal(1.9d) -- "1.9"  todecimal("1") -- "1.0"  todecimal("1.9") -- "1.9" |
| todecimal(long number, int scale) | decimal | Converts data of a specific type into data of the decimal type.  The scale parameter specifies a scale. | todecimal(1L,2) -- "0.01"  todecimal(100000L,2) -- "1000.00" |

### Aggregation Function

The aggregation processing function has one output value and multiple input values and combines multiple values into an output value.

UDAF function

| Function | Return Value | Description | Example |
| --- | --- | --- | --- |
| count(Object value) | long | Calculates the volume of data in a window. Null is not calculated. | Count(1)  Count(\*)  Count(id) |
| count(Object value, expression) | long | Calculates the volume of data that meets expression conditions in a window.  Null is not calculated. | Count(id, type = "1") |
| sum(int number) | int | Calculates the sum of data in a window, in which data columns are of the int type.  Null is not calculated. | Sum(intprice) |
| sum(long number) | long | Calculates the sum of data in a window, in which data columns are of the long type.  Null is not calculated. | Sum(longprice) |
| sum(float number) | float | Calculates the sum of data in a window, in which data columns are of the float type.  Null is not calculated. | Sum(floatprice) |
| sum(double number) | double | Calculates the sum of data in a window, in which data columns are of the double type.  Null is not calculated. | Sum(doubleprice) |
| sum(decimal number) | Decimal | Calculates the sum of data in a window, in which data columns are of the decimal type.  Null is not calculated. | Sum(decimalprice) |
| sum(int number, expression) | Int | Calculates the sum of data that meets expression conditions in a window, in which data columns are of the int type.  Null is not calculated. | Sum(intprice, type = "1") |
| sum(long number, expression) | long | Calculates the sum of data that meets expression conditions in a window, in which data columns are of the long type.  Null is not calculated. | Sum(longprice, type = "1") |
| sum(float number, expression) | float | Calculates the sum of data that meets expression conditions in a window, in which data columns are of the float type.  Null is not calculated. | Sum(floatprice, type = "1") |
| sum(double number, expression) | double | Calculates the sum of data that meets expression conditions in a window, in which data columns are of the double type.  Null is not calculated. | Sum(doubleprice, type = "1") |
| Sum(decimal number, expression) | Decimal | Calculates the sum of data that meets expression conditions in a window, in which data columns are of the decimal type.  Null is not calculated. | Sum(decimalprice, type = "1") |
| avg(int number) | long | Calculates the average value of data in a window, in which data columns are of the int type.  Null is not calculated. | avg(intprice) |
| avg(long number) | long | Calculates the average value of data in a window, in which data columns are of the long type.  Null is not calculated. | avg(longprice) |
| avg(float number) | double | Calculates the average value of data in a window, in which data columns are of the float type.  Null is not calculated. | avg(floatprice) |
| avg(double number) | double | Calculates the average value of data in a window, in which data columns are of the double type.  Null is not calculated. | avg(doubleprice) |
| avg(decimal number) | decimal | Calculates the average value of data in a window, in which data columns are of the decimal type.  Null is not calculated. | avg(decimalprice) |
| avg(int number, expression) | long | Calculates the average value of data that meets expression conditions in a window, in which data columns are of the int type.  Null is not calculated. | avg(intprice, type = "1") |
| avg(long number, expression) | long | Calculates the average value of data that meets expression conditions in a window, in which data columns are of the long type.  Null is not calculated. | avg(longprice, type = "1") |
| avg(float number, expression) | double | Calculates the sum of data that meets expression conditions in a window, in which data columns are of the float type.  Null is not calculated. | avg(floatprice, type = "1") |
| avg(double number, expression) | double | Calculates the average value of data that meets expression conditions in a window, in which data columns are of the double type.  The Null value is not calculated in the average value. | avg(doubleprice, type = "1") |
| avg(decimal number, expression) | decimal | Calculates the average value of data that meets expression conditions in a window, in which data columns are of the decimal type.  The Null value is not calculated in the average value. | avg(decimalprice, type = "1") |
| max(int number) | Int | Calculates the maximum value of data in a window, in which data columns are of the int type.  Null is not calculated. | max(intprice) |
| max(long number) | Long | Calculates the maximum value of data in a window, in which data columns are of the long type.  Null is not calculated. | max(longprice) |
| max(float number) | Float | Calculates the maximum value of data in a window, in which data columns are of the float type.  Null is not calculated. | max(floatprice) |
| max(double number) | Double | Calculates the maximum value of data in a window, in which data columns are of the double type.  Null is not calculated. | max(doubleprice) |
| max(decimal number) | Decimal | Calculates the maximum value of data in a window, in which data columns are of the decimal type.  Null is not calculated. | max(decimalprice) |
| max(int number, expression) | Int | Calculates the maximum value of data that meets expression conditions in a window, in which data columns are of the int type.  Null is not calculated. | max(intprice, type = "1") |
| max(long number, expression) | Long | Calculates the maximum value of data that meets expression conditions in a window, in which data columns are of the long type.  Null is not calculated. | max(longprice, type = "1") |
| max(float number, expression) | Float | Calculates the maximum value of data that meets expression conditions in a window, in which data columns are of the float type.  Null is not calculated. | max(floatprice, type = "1") |
| max(double number, expression) | Double | Calculates the maximum value of data that meets expression conditions in a window, in which data columns are of the double type.  Null is not calculated. | max(doubleprice, type = "1") |
| max(decimal number, expression) | Decimal | Calculates the maximum value of data that meets expression conditions in a window, in which data columns are of the decimal type.  Null is not calculated. | max(decimalprice, type = "1") |
| min(int number) | Int | Calculates the minimum value of data in a window, in which data columns are of the int type.  Null is not calculated. | min(intprice) |
| min(long number) | Long | Calculates the minimum value of data in a window, in which data columns are of the long type.  Null is not calculated. | min(longprice) |
| min(float number) | Float | Calculates the minimum value of data in a window, in which data columns are of the float type.  Null is not calculated. | min(floatprice) |
| min(double number) | Double | Calculates the minimum value of data in a window, in which data columns are of the double type.  Null is not calculated. | min(doubleprice) |
| min(decimal number) | Decimal | Calculates the minimum value of data in a window, in which data columns are of the decimal type.  Null is not calculated. | min(decimalprice) |
| min(int number, expression) | Int | Calculates the minimum value of data that meets expression conditions in a window, in which data columns are of the int type.  Null is not calculated. | min(intprice, type = "1") |
| min(long number, expression) | Long | Calculates the minimum value of data that meets expression conditions in a window, in which data columns are of the long type.  Null is not calculated. | min(longprice, type = "1") |
| min(float number, expression) | Float | Calculates the minimum value of data that meets expression conditions in a window, in which data columns are of the float type.  Null is not calculated. | min(floatprice, type = "1") |
| min(double number, expression) | Double | Calculates the minimum value of data that meets expression conditions in a window, in which data columns are of the double type.  Null is not calculated. | min(doubleprice, type = "1") |
| min(decimal number, expression) | Decimal | Calculates the minimum value of data that meets expression conditions in a window, in which data columns are of the decimal type.  Null is not calculated. | min(decimalprice, type = "1") |
| PREVIOUS(number,column) | Consistent with the Column data type | Previous values in data columns, obtained from a window. If no data is obtained, Null is sent.  Number: number of previous values. The value **0** indicates the current row, and **1** indicates the previous row.  Column: name of a previous column to be queried. | **Previous(0,columnName)**: indicates the current row.  Previous(1,columnName): indicates the ColumnName column in the previous row of the current data in the current window. |



The Previous function does not support Join because results do not meet Join conditions.

The Previous function does not support sort windows, including length sort windows and time sort windows, because data in sort window may change, which results in inconsistency between data in windows and cache.

## Data Source

### RDB Data Source

* **Function**

Obtains data from a database.

RDB data sources support PreparedStatement, and data types of query parameters are supported by CQL.

Encrypts user names and passwords by using the cqlEncrypt.sh script on a CQL client in AES256 mode.

Data source parameters are classified into definition parameters and query parameters. Definition parameters are used in create datasource syntax, and query parameters are used in From substatements for query.

Multiple query parameters of RDB data sources are supported. The first parameter is used in query SQL statements by default, and other parameters are query parameters.



CQL does not include JDBC drives of databases, and therefore, users must run the add jar command to add JDBC drives.

* **Parameter**

Data source definition parameters

| Parameter | Default Value | Description |
| --- | --- | --- |
| driver | None | Database drive. |
| url | None | URL of a database, related to a database. |
| userName | None | Database user name. |
| password | None | Database user password. |
| decryptClass | com.huawei.streaming.encrypt. DefaultDecrypt | Data decryption. Users can also invoke the com.huawei.streaming.encrypt. StreamingDecrypt interface to decrypt data and run the **add jar** command for the decryption to take effect. |
| decryptType | ALL | Mode for encrypting database connection parameters.  USER,PASSWORD,NONE,ALL |

Table 36 Data source query parameters

|  |  |  |
| --- | --- | --- |
| Parameter | Default Value | Description |
| QueryString |  | Query SQL statement. The question mark (?) indicates necessary variables, and CQL automatically enters the variables based on the parameter definition sequence. The parameter definition is the same as the definition of the JDBC Prestatemnt interface. |
| queryArgs |  | Changeable parameter. There can be zero or multiple parameters, and this parameter is set in queryString based on the defined sequence. |

* **Example**

|  |
| --- |
| add jar "/opt/software/Test/gsjdbc4-V100R003C10SPC050.jar";  create input stream S  (id int,name String)  SOURCE KafkaInput  PROPERTIES (  "operator.kafka.groupid" = "gidkpi\_1\_1",  "operator.kafka.topic" = "0915"  );  CREATE DATASOURCE ds  SOURCE RDBDataSource  PROPERTIES (  "driver" = "org.postgresql.Driver",  "url" = "jdbc:postgresql://127.0.0.0:1521/streaming",  "userName" = "55B5B07CF57318642D38F0CEE0666D26",  "password"="55B5B07CF57318642D38F0CEE0666D26"  );  create output stream rs  (type String, id String, tag String)  SINK consoleOutput;  insert into rs select "datasoruceColumn", rdb.id, rdb.name from S,  DATASOURCE ds  [  SCHEMA (id String, name String),  QUERY("select id, name from testtb where id = ?", s.id)  ] rdb; |

# Other CQL Configuration Parameters

Except serialization, deserialization, input, and output parameters and configuration parameters used in UDF functions, there are also system configuration parameters and platform configuration parameters. Full names of such parameters are used actually.

Other CQL configuration parameters

| Parameter | Default Value | Description |
| --- | --- | --- |
| streaming.killapplication.overtime | 60 | Timeout interval for running the CQL kill application command. The unit is second. |
| streaming.common.istestmodel | False | Indicates whether the test mode is being used.  If the test mode is being used, the system only creates a topology but does not submit the topology.  This parameter is used only in CQL test scenarios. |
| streaming.common.parallel.number | 1 | Default concurrency. If it is not specified in CQL statements, the default value is used. |
| streaming.adaptor.application | com.huawei.streaming.storm.StormApplication | Lower-layer application platform. |
| streaming.template.directory | ${system:java.io.tmpdir} | Path to a temporary system directory. The default value is the tmp path. |
| streaming.storm.nimbus.host | 127.0.0.1 | nimbus address, which is invalid in Storm HA scenarios. |
| streaming.storm.nimbus.port | 6627 | nimbus port, which is invalid in Storm HA scenarios. |
| streaming.storm.ha.zkaddress | 127.0.0.1 | zookeeper cluster address of the storm HA. If there are multiple addresses, the addresses are separated with commas (,). |
| streaming.storm.ha.zkport | 24000 | zookeeper port of the storm HA. |
| streaming.storm.ha.zksessiontimeout | 30000 | Timeout interval for a session to connect the HA to the zookeeper. |
| streaming.storm.ha.zookeepersiontimeout | 30000 | Timeout interval for the HA to set up connections. |
| streaming.storm.thrift.transport.plugin | backtype.storm.security.auth.SimpleTransportPlugin | thrift protocol for connections between clients and servers. For security connections, use the following:  backtype.storm.security.auth.kerberos.KerberosSaslTransportPlugin |
| streaming.security.authentication | NONE | Indicates whether security is enabled.  The options are as follows: NONE and KERBEROS  The value is case-insensitive.  The default value is NONE, indicating that security is disabled. |
| streaming.security.zookeeper.principal | zookeeper/hadoop | principal of the Zookeeper server. |
| streaming.security.storm.principal | streaming/hadoop@HADOOP.COM | principal of the Zookeeper server. |
| streaming.security.user.principal | NONE | principal of a user. |
| streaming.security.keytab.path | NONE | Path to the keytab file of a user. |
| streaming.security.krbconf.path | NONE | Path to a krb5 file.  If the value is blank, the default paths are used.  The sequence for obtaining default paths is as follows:  1. ${JAVA\_HOME}/lib/security/krb5.conf  2. ${JAVA\_HOME}/krb5.ini  3. windows: %windir%\krb5.ini  linux: /etc/krb5.conf |
| streaming.storm.submit.islocal | False | Indicates whether a task is locally submitted. If yes, a CQL task is executed in a local process, and this mode is usually used for test. |
| streaming.localtask.alivetime.ms | 5000 | Life time of a local task. The unit is millisecond. |
| streaming.storm.killtask.waitseconds | 3 | Wait time for killing a task, in second. Storm parameters for killing a task are different from CQL parameters. |
| streaming.storm.worker.number | 1 | Number of CQL processes for submitting tasks. |
| streaming.serde.default | com.huawei.streaming.serde.SimpleSerDe | Default CQL serialization or deserialization class. |
| serde.simpleserde.separator | , | SimpleSerDe configuration parameter.  The default system serialization class is simpleSerDe, and therefore, configuration parameters must include parameters of this serialization class. |
| streaming.userfile.maxsize | 30 | Maximum size of a user file, in MB. |
| streaming.storm.rebalance.waitseconds | 10 | Wait time of a rebalance task. |
| streaming.operator.timezone | ${system:user.timezone} | Timezone settings in the CQL task execution. The client time zone is used by default. The parameter format can be either of the following modes:  GMT+08:00  Asia/Shanghai |

# CQL Error Code

CQL error code design

An error code consists of the following parts:

Prefix: CQL-

Class: two-digit number, starting from 00

Segmentation: three-digit number, starting from 000

Error code description

CQL-00: successful completion

CQL-01: warning

CQL-02: configuration attribute exception

CQL-03: stream processing platform exception

CQL-04: semantic analysis exception, including a syntax paring exception

CQL-05: topology exception, including a streaming operator parsing exception

CQL-06: function parsing exception

CQL-07: window parsing exception

CQL-08: security exception

CQL-99: unknown exception

CQL error codes

| Error Code | Description |
| --- | --- |
| CQL-00: successful completion | |
| CQL-00000 | Successful completion |
| CQL-01: warning | |
| CQL-01000 | Warning |
| CQL-02: configuration attribute exception | |
| CQL-02000 | Failure to find a configuration attribute |
| CQL-02001 | Failure to convert the data type of a configuration attribute |
| CQL-02002 | Incorrect configuration attribute value (correct data type but out-of-range or unsatisfied value) |
| CQL-03: stream processing platform exception | |
| CQL-03000 | Application deletion timeout  The timeout interval is specified by streaming.killapplication.overtime. |
| CQL-03001 | The application already exists. |
| CQL-03002 | The application does not exist. |
| CQL-03003 | Exception of the operator topology of an application |
| CQL-03004 | Failure to find an input operator |
| CQL-03005 | Application status exception resulting in a failure to run the active, deactive, or rebalance command |
| CQL-03006 | The number of application workers exceeds the number of available system workers. |
| CQL-03007 | Invalid nimbus service |
| CQL-03008 | Server communication exception |
| CQL-04: semantic analysis exception, including a syntax paring exception | |
| CQL-04000 | Syntax parsing exception |
| CQL-04001 | Unsupported syntax |
| CQL-04002 | Attribute expression supported only by the order by substatement |
| CQL-04003 | Failure to find an expression in the select substatement |
| CQL-04004 | The address file does not exist. |
| CQL-04005 | The address must be a file. |
| CQL-04006 | The file is not a jar file. |
| CQL-04007 | Failure to parse the jar address file |
| CQL-04008 | Failure to find the stream |
| CQL-04009 | Failure to convert a constant type |
| CQL-04010 | Unsupported data type |
| CQL-04011 | Failure to find the column from a specific stream |
| CQL-04012 | Failure to find the column from all streams |
| CQL-04013 | Failure to identify the column, which may exist in multiple streams |
| CQL-04014 | Failure to find the stream based on the name |
| CQL-04015 | Inconsistency between the number of output columns and the number of columns in the select substatement |
| CQL-04016 | Sub expressions of the like expression must be of the String type. |
| CQL-04017 | All return values of the not expression must be of the boolean type. |
| CQL-04018 | Parameters in the IN and BETWEEN expressions must be constants. |
| CQL-04019 | A CASE-WHEN expression must contain at least one WHEN-THEN expression. |
| CQL-04020 | A WHEN-THEN expression must contain both WHEN and THEN expressions. |
| CQL-04021 | Return values of the WHEN expression in the WHEN-THEN expression must be of the boolean type. |
| CQL-04022 | A CASE expression must contain an input expression. |
| CQL-04023 | Operator expressions support the number type only. |
| CQL-04024 | Logic expressions must be of the boolean type. |
| CQL-04025 | Failure to find the matching data source. |
| CQL-04026 | Failure to find the matching expression from data source parameters |
| CQL-04027 | Combine operations do not allow windows. |
| CQL-04028 | The input stream of a Combine operation does not match the number of streams in a combine condition. |
| CQL-04029 | combine operators support attribute expressions only. |
| CQL-04030 | In a combine statement, output columns of a stream must be placed together. |
| CQL-04031 | MultiInsert must not contain data sources. |
| CQL-04032 | MultiInsert must not contain GroupBy. |
| CQL-04033 | MultiInsert must not contain windows. |
| CQL-04034 | MultiInsert must not contain multiple input streams. |
| CQL-04035 | Multistream Join is not supported. |
| CQL-04036 | NATURAL JOIN is not supported. |
| CQL-04037 | In Join, the unidirection can be set only for one stream. |
| CQL-04038 | Failure to find the On condition in join statements |
| CQL-04039 | Failure to find the column matching an expression in a join condition from the input stream. |
| CQL-04040 | Unsupported Join condition |
| CQL-04041 | Failure to find the serialization and deserialization classes |
| CQL-04042 | Failure to find the class |
| CQL-04043 | Failure to initialize the serialization or deserialization class |
| CQL-04044 | The stream name already exists. |
| CQL-04045 | Types of two expressions do not match. |
| CQL-04046 | The total size of user files exceeds the upper limit.  The maximum number of files that can be uploaded by users is specified by streaming.userfile.maxsize.  The default value is 30MB. |
| CQL-04047 | The syntax parsing result is blank. |
| CQL-04048 | This window does not support Exclude Now. |
| CQL-04049 | The Previous expression cannot be used together with the sorting window. |
| CQL-04050 | The window does not support RStream output of old data.  This error code usually occurs in scenarios in which one window generates new data and the other window generates old data. |
| CQL-04051 | Failure to find the unidirection definition from the left stream or right stream.  This occurs in selfjoin which must be single-direction Join. |
| CQL-04052 | Failure to find a customized operator. |
| CQL-04053 | A customized operator supports only one input schema. |
| CQL-04054 | A customized operator supports only one output schema. |
| CQL-04055 | Failure to verify the input schema of a customized operator. |
| CQL-04056 | Failure to verify the output schema of a customized operator. |
| CQL\_40057 | The operator type does not match. |
| CQL-05: topology exception, including a streaming operator parsing exception | |
| CQL-05000 | Only one application can be submitted for a CQL file. |
| CQL-05001 | Failure to find an input stream for an operator |
| CQL-05002 | Failure to find an output stream for an operator |
| CQL-05003 | The application name is blank. |
| CQL-05004 | Failure to find a physical execution plan file. |
| CQL-05005 | The format of content in the physical execution plan file is incorrect. |
| CQL-05006 | Failure to initialize operator. |
| CQL-06: function parsing exception | |
| CQL-06000 | The function is not supported. |
| CQL-06001 | The system function cannot be removed. |
| CQL-06002 | The system function cannot be overwritten. |
| CQL-06003 | The customized function must be inherited from a specific class. |
| CQL-06004 | The previous function must be in the PREVIOUS(number, column) format. |
| CQL-06005 | Parameter type not supported by a UDF function |
| CQL-07: window parsing exception | |
| CQL-07000 | Unidentified window |
| CQL-07001 | The select list in an exclude now statement cannot include other non-group-by columns. |
| CQL-07002 | Incorrect window parameter, which results in a failure to create a window |
| CQL-07003 | The sorting window supports the slide type only. |
| CQL-08: security exception | |
| CQL-08000 | The security type is not supported. |
| CQL-08001 | Unauthorized |
| CQL-08002 | The keytab file address is incorrect. |
| CQL-08003 | Internal security error |
| CQL-08004 | Failure to authenticate a client user |
| CQL-99: unknown exception | |
| CQL-99000 | Common server exception |
| CQL-99999 | Unknown exception |

# CQL Java Development API

The com.huawei.streaming.cql.Driver class is invoked to submit CQL statements.

Description about the Driver method

CQL Java development interfaces

| Method Name | Method Description | Parameter | Return Value |
| --- | --- | --- | --- |
| Run | CQL statement execution. | String CQL statement | Void |
| getResult | Execution result. | None | CQLResult  Query results vary depending on statements. The output of the get command is the value of a query parameter. |
| Clean | Initialized user information deleted from driver.  This method is invoked after a user performs a submit operation or before an application is resubmitted. | None | Void |

# CQL Key Word List

CQL key word list

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| CREATE | SHOW | EXPLAIN | SET | GET |
| LOAD | EXPORT | DROP | ADD | SELECT |
| COMMENT | FORCE | SERDE | WITH | PROPERTIES |
| SOURCE | INPUT | STREAM | OUTPUT | SINK |
| SUBMIT | APPLICATION | DISTINCT | AND | OR |
| BETWEEN | IN | LIKE | RLIKE | REGEXP |
| CASE | WHEN | THEN | ELSE | END |
| CAST | EXISTS | IF | FALSE | AS |
| NULL | IS | TRUE | ALL | NOT |
| ASC | DESC | SORT | ORDER | GROUP |
| BY | HAVING | WHERE | FROM | ON |
| JOIN | FULL | PRESERVE | OUTER | CROSS |
| SEMI | LEFT | INNER | NATURAL | RIGHT |
| INTO | INSERT | OVERWRITE | LIMIT | UNION |
| APPLICATIONS | WINDOWS | EXTENDED | FUNCTIONS | FILE |
| INPATH | WINDOW | JAR | FUNCTION | COMBINE |
| UNIDIRECTION | PARALLEL | TRIGGER | PARTITION | SLIDE |
| BATCH | RANGE | ROWS | TODAY | UNBOUNDED |
| NOW | PREVIOUS | DATASOURCE | SCHEMA | QUERY |
| DEACTIVE | ACTIVE | WORKER | REBALANCE | DAYS |
| HOUR | HOURS | MINUTE | MINUTES | SECOND |
| SECONDS | MILLISECOND | MILLISECONDS | BOOLEAN | INT |
| LONG | FLOAT | DOUBLE | STRING | DATE |
| TIME | DECIMAL |  |  |  |