Oracle® Fusion Middleware

CQL Language Reference for Oracle Event Processing

11*g* Release 1 (11.1.1.9)

E12048-11

February 2015

Documentation for developers that provides a reference to Oracle Continuous Query Language (Oracle CQL), an SQL-like language for querying streaming data in Oracle Event Processing applications.

Oracle Fusion Middleware CQL Language Reference for Oracle Event Processing, 11*g* Release 1 (11.1.1.9)

E12048-11

Copyright © 2007, 2015, Oracle and/or its affiliates. All rights reserved.

Primary Author: Oracle Corporation

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

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

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

3 Pseudocolumns

4 Operators

5 Expressions

6 Conditions

7 Common Oracle CQL DDL Clauses

Part II Functions

8 Built-In Single-Row Functions

9 Built-In Aggregate Functions

10 Colt Single-Row Functions

12 java.lang.Math Functions

16 Oracle Spatial

17 Oracle Event Processing JDBC Data Cartridge

18 Oracle Event Processing Hadoop Data Cartridge

19 Oracle Event Processing NoSQL Database Data Cartridge

Part IV Using Oracle CQL

20 Oracle CQL Queries, Views, and Joins

22 Oracle CQL Statements

List of Examples

List of Figures

List of Tables

Preface

This reference contains a complete description of the Oracle Continuous Query Language (Oracle CQL), a query language based on SQL with added constructs that support streaming data. Using Oracle CQL, you can express queries on data streams to perform event processing using Oracle Event Processing. Oracle CQL is a new technology but it is based on a subset of SQL99.

Oracle Event Processing (formally known as the WebLogic Event Server) is a Java server for the development of high-performance event driven applications. It is a lightweight Java application container based on Equinox OSGi, with shared services, including the Oracle Event Processing Service Engine, which provides a rich, declarative environment based on Oracle Continuous Query Language (Oracle CQL) - a query language based on SQL with added constructs that support streaming data - to improve the efficiency and effectiveness of managing business operations. Oracle Event Processing supports ultra-high throughput and microsecond latency using JRockit Real Time and provides Oracle Event Processing Visualizer and Oracle Event Processing IDE for Eclipse developer tooling for a complete real time end-to-end Java Event-Driven Architecture (EDA) development platform.

Audience

This document is intended for all users of Oracle CQL.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Related Documents

For more information, see the following:

- *Oracle Fusion Middleware Getting Started Guide for Oracle Event Processing*
- *Oracle Fusion Middleware Administrator's Guide for Oracle Event Processing*

- *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*
- *Oracle Fusion Middleware Visualizer User's Guide for Oracle Event Processing*
- *Oracle Fusion Middleware Java API Reference for Oracle Event Processing*
- *Oracle Fusion Middleware EPL Language Reference for Oracle Event Processing*
- *Oracle Database SQL Language Reference* at `http://download.oracle.com/docs/cd/B28359_01/server.111/b28286/toc.htm`
- SQL99 Specifications (ISO/IEC 9075-1:1999, ISO/IEC 9075-2:1999, ISO/IEC 9075-3:1999, and ISO/IEC 9075-4:1999)
- Oracle Event Processing Forum: `http://forums.oracle.com/forums/forum.jspa?forumID=820`
- *Oracle Event Processing Samples*: `http://www.oracle.com/technologies/soa/complex-event-processing.html`

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Syntax Diagrams

Syntax descriptions are provided in this book for various Oracle CQL, SQL, PL/SQL, or other command-line constructs in graphic form or Backus Naur Form (BNF). See "How to Read Syntax Diagrams" in the *Oracle Database SQL Language Reference* for information about how to interpret these descriptions.

What's New in This Guide

This guide has been updated in several ways. The following table lists the sections that have been added or changed.

For a list of known issues (release notes), see the "Known Issues for Oracle SOA Products and Oracle AIA Foundation Pack" at `http://www.oracle.com/technetwork/middleware/docs/soa-aiafp-knownissuesindex-364630.html`.

Sections	Changes Made	February 2013
Entire Guide	Product renamed to Oracle Event Processing	X
Chapter 9 Built-In Aggregate Functions		
listagg	Section added to describe the listagg function, which you can use to aggregate the values of its argument into a `java.util.List`.	X
Chapter 13 User-Defined Functions		
User-Defined Function Datatypes	Rewritten to note that all built-in Oracle CQL datatypes are supported in user-defined functions.	X
Chapter 18 Oracle Event Processing Hadoop Data Cartridge		
Chapter 18, "Oracle Event Processing Hadoop Data Cartridge"	Chapter added to describe how to access Hadoop data sources from Oracle CQL code.	X
Chapter 19 Oracle Event Processing NoSQL Data Cartridge		
Chapter 19, "Oracle Event Processing NoSQL Database Data Cartridge"	Chapter added to describe how to access NoSQL data sources from Oracle CQL code.	X
Chapter 22 Oracle CQL Statements		
BINARY Example: UNION and UNION ALL	Section updated to illustrate how to use the UNION ALL operator with two input streams to produce an output stream.	X
BINARY Example: INTERSECT	Section updated to include an example of the INTERSECT operator.	X

Sections	Changes Made	February 2013
BINARY Example: EXCEPT and MINUS	Section updated with descriptive text and EXCEPT operator example.	X
ORDER BY Query Example and ORDER BY ROWS Query Example	Section updated with more description and examples.	X

Part I

Understanding Oracle CQL

This part contains the following chapters:

1

Introduction to Oracle CQL

This chapter introduces Oracle Continuous Query Language (Oracle CQL), a query language based on SQL with added constructs that support streaming data. Using Oracle CQL, you can express queries on data streams with Oracle Event Processing.

Oracle Event Processing (formally known as the WebLogic Event Server) is a Java server for the development of high-performance event driven applications. It is a lightweight Java application container based on Equinox OSGi, with shared services, including the Oracle Event Processing Service Engine, which provides a rich, declarative environment based on Oracle CQL to improve the efficiency and effectiveness of managing business operations. Oracle Event Processing supports ultra-high throughput and microsecond latency using JRockit Real Time and provides Oracle Event Processing Visualizer and Oracle Event Processing IDE for Eclipse developer tooling for a complete real time end-to-end Java Event-Driven Architecture (EDA) development platform.

This chapter includes the following sections:

- Fundamentals of Oracle CQL
- Oracle CQL Statements
- Oracle CQL and SQL Standards
- Oracle Event Processing Server and Tools Support

Fundamentals of Oracle CQL

Databases are best equipped to run queries over finite stored data sets. However, many modern applications require long-running queries over continuous unbounded sets of data. By design, a stored data set is appropriate when significant portions of the data are queried repeatedly and updates are relatively infrequent. In contrast, data streams represent data that is changing constantly, often exclusively through insertions of new elements. It is either unnecessary or impractical to operate on large portions of the data multiple times.

Many types of applications generate data streams as opposed to data sets, including sensor data applications, financial tickers, network performance measuring tools, network monitoring and traffic management applications, and clickstream analysis tools. Managing and processing data for these types of applications involves building data management and querying capabilities with a strong temporal focus.

To address this requirement, Oracle introduces Oracle Event Processing, a data management infrastructure that supports the notion of streams of structured data records together with stored relations.

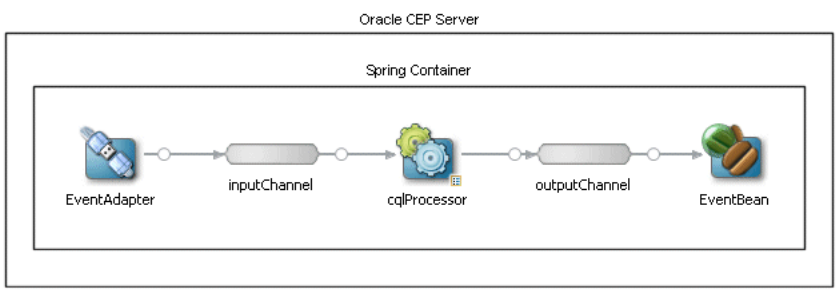
To provide a uniform declarative framework, Oracle offers Oracle Continuous Query Language (Oracle CQL), a query language based on SQL with added constructs that support streaming data.

Oracle CQL is designed to be:

- Scalable with support for a large number of queries over continuous streams of data and traditional stored data sets.
- Comprehensive to deal with complex scenarios. For example, through composability, you can create various intermediate views for querying.

Figure 1–1 shows a simplified view of the Oracle Event Processing architecture. Oracle Event Processing server provides the light-weight Spring container for Oracle Event Processing applications. The Oracle Event Processing application shown is composed of an event adapter that provides event data to an input channel. The input channel is connected to an Oracle CQL processor associated with one or more Oracle CQL queries that operate on the events offered by the input channel. The Oracle CQL processor is connected to an output channel to which query results are written. The output channel is connected to an event Bean: a user-written Plain Old Java Object (POJO) that takes action based on the events it receives from the output channel.

Figure 1–1 Oracle Event Processing Architecture



Using Oracle Event Processing, you can define event adapters for a variety of data sources including JMS, relational database tables, and files in the local filesystem. You can connect multiple input channels to an Oracle CQL processor and you can connect an Oracle CQL processor to multiple output channels. You can connect an output channel to another Oracle CQL processor, to an adapter, to a cache, or an event Bean.

Using Oracle Event Processing IDE for Eclipse and Oracle Event Processing Visualizer, you:

- Create an Event Processing Network (EPN) as Figure 1–1 shows.
- Associate one more Oracle CQL queries with the Oracle CQL processors in your EPN.
- Package your Oracle Event Processing application and deploy it to Oracle Event Processing server for execution.

Consider the typical Oracle CQL statements that Example 1–1 shows.

Example 1–1 Typical Oracle CQL Statements

```
<?xml version="1.0" encoding="UTF-8"?>
<n1:config xsi:schemaLocation="http://www.bea.com/ns/wlevs/config/application wlevs_application_config.xsd"
xmlns:n1="http://www.bea.com/ns/wlevs/config/application"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<processor>
    <name>cqlProcessor</name>
```

```
    <rules>
        <view id="lastEvents" schema="cusip bid ask"><![CDATA[
            select cusip, bid, srcId, bidQty, ask, askQty, seq
            from inputChannel[partition by srcId, cusip rows 1]
        ]]></view>
        <view id="bidask" schema="cusip bid ask"><![CDATA[
            select cusip, max(bid), min(ask)
            from lastEvents
            group by cusip
        ]]></view>
            <view ...><![CDATA[
                ...
        ]]></view>
        ...
        <view id="MAXBIDMINASK" schema="cusip bidseq bidSrcId bid askseq askSrcId ask bidQty askQty"><![CDATA[
            select bid.cusip, bid.seq, bid.srcId as bidSrcId, bid.bid, ask.seq, ask.srcId as askSrcId, ask.ask,
bid.bidQty, ask.askQty
            from BIDMAX as bid, ASKMIN as ask
            where bid.cusip = ask.cusip
        ]]></view>
        <query id="BBAQuery"><![CDATA[
            ISTREAM(select bba.cusip, bba.bidseq, bba.bidSrcId, bba.bid, bba.askseq, bba.askSrcId, bba.ask,
                bba.bidQty, bba.askQty, "BBAStrategy" as intermediateStrategy, p.seq as correlationId, 1 as
priority
            from MAXBIDMINASK as bba, inputChannel[rows 1] as p where bba.cusip = p.cusip)
        ]]></query>
    </rules>
</processor>
```

This example defines multiples views (the Oracle CQL-equivalent of subqueries) to create multiple relations, each building on previous views. Views always act on an inbound channel such as `inputChannel`. The first view, named `lastEvents`, selects directly from `inputChannel`. Subsequent views may select from `inputChannel` directly or select from previously defined views. The results returned by a view's select statement remain in the view's relation: they are not forwarded to any outbound channel. That is the responsibility of a query. This example defines query `BBAQuery` that selects from both the `inputChannel` directly and from previously defined views. The results returned from a query's select clause are forwarded to the outbound channel associated with it: in this example, to `outputChannel`. The `BBAQuery` uses a tuple-based stream-to-relation operator (or sliding window).

For more information on these elements, see:

- Section , "Streams and Relations"
- Section , "Relation-to-Relation Operators"
- Section , "Stream-to-Relation Operators (Windows)"
- Section , "Relation-to-Stream Operators"
- Section , "Stream-to-Stream Operators"
- Section , "Queries, Views, and Joins"
- Section , "Pattern Recognition"
- Section , "Event Sources and Event Sinks"
- Section , "Functions"
- Section , "Data Cartridges"
- Section , "Time"
- Section , "Oracle CQL Statements"

- Section , "Lexical Conventions"
- Section , "Documentation Conventions"

For more information on Oracle Event Processing server and tools, see:

- Section , "Oracle Event Processing Server and Tools Support."
- *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*
- *Oracle Fusion Middleware Visualizer User's Guide for Oracle Event Processing*
- *Oracle Fusion Middleware Administrator's Guide for Oracle Event Processing*

Streams and Relations

This section introduces the two fundamental Oracle Event Processing objects that you manipulate using Oracle CQL:

- Streams
- Relations

Using Oracle CQL, you can perform the following operations with streams and relations:

- Relation-to-Relation Operators: to produce a relation from one or more other relations
- Stream-to-Relation Operators (Windows): to produce a a relation from a stream
- Relation-to-Stream Operators: to produce a stream from a relation
- Stream-to-Stream Operators: to produce a stream from one or more other streams

Streams

A stream is the principle source of data that Oracle CQL queries act on.

Stream `S` is a bag (or multi-set) of elements `(s,T)` where `s` is in the schema of `S` and `T` is in the time domain.

Stream elements are tuple-timestamp pairs, which can be represented as a sequence of timestamped tuple insertions. In other words, a stream is a sequence of timestamped tuples. There could be more than one tuple with the same timestamp. The tuples of an input stream are required to arrive at the system in the order of increasing timestamps. For more information, see Section , "Time".

A stream has an associated schema consisting of a set of named attributes, and all tuples of the stream conform to the schema.

The term "tuple of a stream" denotes the ordered list of data portion of a stream element, excluding timestamp data (the `s` of `<s,t>`). Example 1–2 shows how a stock ticker data stream might appear, where each stream element is made up of `<timestamp value>`, `<stock symbol>`, and `<stock price>`:

Example 1–2 Stock Ticker Data Stream

```
...
<timestampN>    NVDA,4
<timestampN+1>  ORCL,62
<timestampN+2>  PCAR,38
<timestampN+3>  SPOT,53
<timestampN+4>  PDCO,44
<timestampN+5>  PTEN,50
...
```

In the stream element <timestampN+1> `ORCL,62`, the tuple is `ORCL,62`.

By definition, a stream is unbounded.

This section describes:

- Section , "Streams and Channels"
- Section , "Channel Schema"
- Section , "Querying a Channel"
- Section , "Controlling Which Queries Output to a Downstream Channel"

For more information, see:

- Section , "Event Sources and Event Sinks"
- Section , "Introduction to Oracle CQL Queries, Views, and Joins"
- "Channels Representing Streams and Relations" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*

Streams and Channels Oracle Event Processing represents a stream as a channel as Figure 1–2 shows. Using Oracle Event Processing IDE for Eclipse, you connect the stream event source (`PriceAdapter`) to a channel (`priceStream`) and the channel to an Oracle CQL processor (`filterFanoutProcessor`) to supply the processor with events. You connect the Oracle CQL processor to a channel (`filteredStream`) to output Oracle CQL query results to down-stream components (not shown in Figure 1–2).

Figure 1–2 Stream in the Event Processing Network

> **Note:** In Oracle Event Processing, you must use a channel to connect an event source to an Oracle CQL processor and to connect an Oracle CQL processor to an event sink. A channel is optional with other Oracle Event Processing processor types.

Channel Schema The event source you connect to a stream determines the stream's schema. In Figure 1–2, the `PriceAdapter` adapter determines the `priceStream` stream's schema. Example 1–3 shows the `PriceAdapter` Event Processing Network (EPN) assembly file: the `eventTypeName` property specifies event type `PriceEvent`. The `event-type-repository` defines the property names and types for this event.

Example 1–3 Channel Schema Definition

```
...
<wlevs:event-type-repository>
    <wlevs:event-type type-name="PriceEvent">
        <wlevs:properties>
            <wlevs:property name="cusip" type="char" />
            <wlevs:property name="bid" type="double" />
            <wlevs:property name="srcId" type="char" />
            <wlevs:property name="bidQty" type="int" />
            <wlevs:property name="ask" type="double" />
            <wlevs:property name="askQty" type="int" />
            <wlevs:property name="seq" type="bigint" />
```

```
            <wlevs:property name="sector" type="char" />
        </wlevs:properties>
    </wlevs:event-type>
</wlevs:event-type-repository>

<wlevs:adapter id="PriceAdapter" provider="loadgen">
    <wlevs:instance-property name="port" value="9011"/>
    <wlevs:listener ref="priceStream"/>
</wlevs:adapter>

<wlevs:channel id="priceStream" event-type="PriceEvent">
    <wlevs:listener ref="filterFanoutProcessor"/>
</wlevs:channel>

<wlevs:processor id="filterFanoutProcessor" provider="cql">
    <wlevs:listener ref="filteredStream"/>
</wlevs:processor>

...
```

Querying a Channel Once the event source, channel, and processor are connected as Figure 1–2 shows, you can write Oracle CQL statements that make use of the stream. Example 1–4 shows the component configuration file that defines the Oracle CQL statements for the `filterFanoutProcessr`.

Example 1–4 filterFanoutProcessor Oracle CQL Query Using priceStream

```
<processor>
    <name>filterFanoutProcessor</name>
    <rules>
        <query id="Yr3Sector"><![CDATA[
            select cusip, bid, srcId, bidQty, ask, askQty, seq
            from priceStream where sector="3_YEAR"
        ]]></query>
        <query id="Yr2Sector"><![CDATA[
            select cusip, bid, srcId, bidQty, ask, askQty, seq
            from priceStream where sector="2_YEAR"
        ]]></query>
        <query id="Yr1Sector"><![CDATA[
            select cusip, bid, srcId, bidQty, ask, askQty, seq
            from priceStream where sector="1_YEAR"
        ]]></query>
    </rules>
</processor>
```

Controlling Which Queries Output to a Downstream Channel If you specify more than one query for a processor as Example 1–4 shows, then all query results are output to the processor's out-bound channel (`filteredStream` in Figure 1–2).

Optionally, in the component configuration file, you can use the `channel` element `selector` attribute to control which query's results are output as Example 1–5 shows. In this example, query results for query `Yr3Sector` and `Yr2Sector` are output to filteredStream but not query results for query `Yr1Sector`. For more information, see "Connecting EPN Stages Using Channels" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

Example 1–5 Using channel Element selector Child Element to Control Which Query Results are Output to a Channel

```
<channel>
    <name>filteredStream</name>
```

```
    <selector>Yr3Sector Yr2Sector</selector>
</channel>
```

You may configure a `channel` element with a `selector` before creating the queries in the upstream processor. In this case, you must specify query names that match the names in the `selector`.

For more information, see "Controlling Which Queries Output to a Downstream Channel" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

Relations

Time varying relation `R` is a mapping from the time domain to an unbounded bag of tuples to the schema of `R`.

A relation is an unordered, time-varying bag of tuples: in other words, an instantaneous relation. At every instant of time, a relation is a bounded set. It can also be represented as a sequence of timestamped tuples that includes insertions, deletions, and updates to capture the changing state of the relation.

Like streams, relations have a fixed schema to which all tuples conform.

Oracle Event Processing supports both base and derived streams and relations. The external sources supply data to the base streams and relations.

A base (explicit) stream is a source data stream that arrives at an Oracle Event Processing adapter so that time is non-decreasing. That is, there could be events that carry same value of time.

A derived (implicit) stream/relation is an intermediate stream/relation that query operators produce. Note that these intermediate operators can be named (through views) and can therefore be specified in further queries.

A base relation is an input relation.

A derived relation is an intermediate relation that query operators produce. Note that these intermediate operators can be named (through views) and can therefore be specified in further queries.

In Oracle Event Processing, you do not create base relations yourself. The Oracle Event Processing server creates base relations for you as required.

When we say that a relation is a time-varying bag of tuples, time refers to an instant in the time domain. Input relations are presented to the system as a sequence of timestamped updates which capture how the relation changes over time. An update is either a tuple insertion or deletion. The updates are required to arrive at the system in the order of increasing timestamps.

For more information, see:

- "Channels Representing Streams and Relations" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*
- Section , "Time"

Relations and Oracle Event Processing Tuple Kind Indicator

By default, Oracle Event Processing includes time stamp and an Oracle Event Processing tuple kind indicator in the relations it generates as Example 1–6 shows.

Example 1–6 Oracle Event Processing Tuple Kind Indicator in Relation Output

```
Timestamp   Tuple Kind  Tuple
```

```
 1000:      +          ,abc,abc
 2000:      +          hihi,abchi,hiabc
 6000:      -          ,abc,abc
 7000:      -          hihi,abchi,hiabc
 8000:      +          hi1hi1,abchi1,hi1abc
 9000:      +          ,abc,abc
13000:      -          hi1hi1,abchi1,hi1abc
14000:      -          ,abc,abc
15000:      +          xyzxyz,abcxyz,xyzabc
20000:      -          xyzxyz,abcxyz,xyzabc
```

The Oracle Event Processing tuple kind indicators are:

- + for inserted tuple
- - for deleted tuple
- `U` for updated tuple indicated when invoking `com.bea.wlevs.ede.api.RealtionSink` method `onUpdateEvent` (for more information, see *Oracle Fusion Middleware Java API Reference for Oracle Event Processing*).

Relation-to-Relation Operators

The relation-to-relation operators in Oracle CQL are derived from traditional relational queries expressed in SQL.

Anywhere a traditional relation is referenced in a SQL query, a relation can be referenced in Oracle CQL.

Consider the following examples for a stream `CarSegStr` with schema: `car_id integer`, `speed integer`, `exp_way integer`, `lane integer`, `dir integer`, and `seg integer`.

In Example 1–7, at any time instant, the output relation of this query contains the set of vehicles having transmitted a position-speed measurement within the last 30 seconds.

Example 1–7   Relation-to-Relation Operation

```
<processor>
    <name>cqlProcessor</name>
    <rules>
        <view id="CurCarSeg" schema="car_id exp_way lane dir seg"><![CDATA[
            select distinct
                car_id, exp_way, lane, dir, seg
            from
                CarSegStr [range 30 seconds]
        ]]></query>
    </rules>
</processor>
```

The `distinct` operator is the relation-to-relation operator. Using `distinct`, Oracle Event Processing returns only one copy of each set of duplicate tuples selected. Duplicate tuples are those with matching values for each expression in the select list. You can use distinct in a `select_clause` and with aggregate functions.

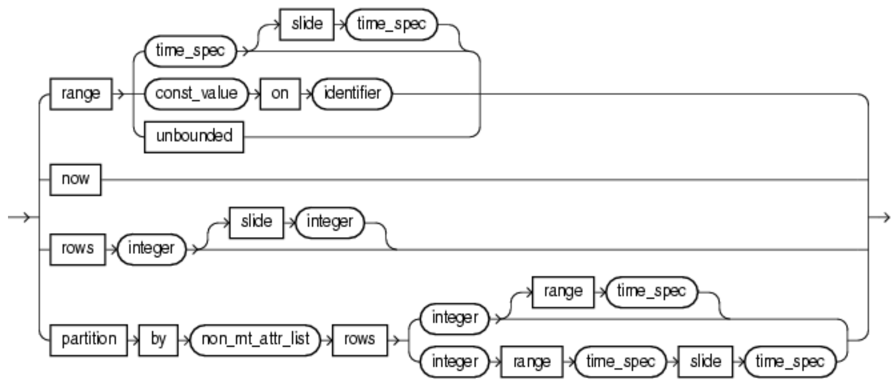
For more information on `distinct`, see:

- Chapter 9, "Built-In Aggregate Functions"
- *select_clause::=* on page 22-3

Stream-to-Relation Operators (Windows)

Oracle CQL supports stream-to-relation operations based on a sliding window. In general, `S[W]` is a relation. At time `T` the relation contains all tuples in window `W` applied to stream `S` up to `T`.

***window_type*::=**



Oracle CQL supports the following built-in window types:

- Range: time-based

  `S[Range T]`, or, optionally,

  `S[Range T1 Slide T2]`

- Range: time-based unbounded

  `S[Range Unbounded]`

- Range: time-based now

  `S[Now]`

- Range: constant value

  `S[Range C on ID]`

- Tuple-based:

  `S[Rows N]`, or, optionally,

  `S[Rows N1 Slide N2]`

- Partitioned:

  `S[Partition By A1 ... Ak Rows N]` or, optionally,

  `S[Partition By A1 ... Ak Rows N Range T]`, or

  `S[Partition By A1 ... Ak Rows N Range T1 Slide T2]`

This section describes the following stream-to-relation operator properties:

- Section , "Range, Rows, and Slide"
- Section , "Partition"
- Section , "Default Stream-to-Relation Operator"

For more information, see:

- "Range-Based Stream-to-Relation Window Operators" on page 4-6
- "Tuple-Based Stream-to-Relation Window Operators" on page 4-14

- "Partitioned Stream-to-Relation Window Operators" on page 4-19

Range, Rows, and Slide

The keywords `Range` and `Rows` specify how much data you want to query:

- `Range` specifies as many tuples as arrive in a given time period
- `Rows` specifies a number of tuples

The Slide keyword specifies how frequently you want to see output from the query, while the Range keyword specifies the time range from which to query events. Using Range and Slide together results in a set of events from which to query, with that set changing based on where the query window slides to. So the set time is the time from which events get drawn for the query.

So the time interval is the actual amount of time (as measured by event timestamps) divided by the amount of time specified for sliding. If the remainder from this is 0, then the set time is the time interval multiplied by the amount of time specified for the slide. If the remainder is greater than 0, then the set time is the time interval + 1 multiplied by the amount of time specified for the slide.

Another way to express this is the following formula:

timeInterval = actualTime / slideSpecification

if((actualTime % slideSpecification) == 0)   // No remainder

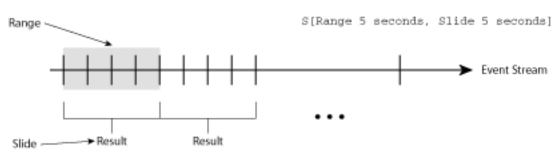
   setTime = timeInterval * slideSpecification

else

   setTime = (timeInterval + 1) * slideSpecification

In Figure 1–3, the `Range` specification indicates "I want to look at 5 seconds worth of data" and the `Slide` specification indicates "I want a result every 5 seconds". In this case, the query returns a result at the end of each `Range` specification (except for certain conditions, as "Range, Rows, and Slide at Query Start-Up and for Empty Relations" on page 1-11 describes).

Figure 1–3   Range and Slide: Equal (Steady-State Condition)



In Figure 1–4, the `Range` specification indicates "I want to look at 10 seconds worth of data" and the `Slide` specification indicates "I want a result every 5 seconds". In this case, the query returns a result twice during each `Range` specification (except for certain conditions, as Section , "Range, Rows, and Slide at Query Start-Up and for Empty Relations" describes)

Figure 1–4   Range and Slide: Different (Steady-State Condition)

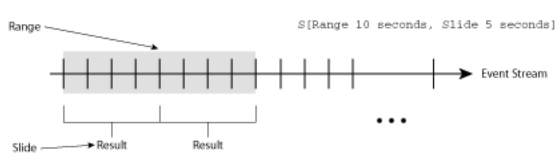


Table 1–1 lists the default Range, Range unit, and Slide (where applicable) for range-based and tuple-based stream-to-relation window operators:

Table 1–1 Default Range and Tuple-Based Stream-to-Relation Operators

Window Operator	Default Range	Default Range Unit	Default Slide
Range-Based Stream-to-Relation Window Operators	`Unbounded`	`seconds`	1 nanosecond
Tuple-Based Stream-to-Relation Window Operators	N/A	N/A	1 tuple

Range, Rows, and Slide at Query Start-Up and for Empty Relations The descriptions for Figure 1–3 and Figure 1–4 assume a steady-state condition, after the query has been running for some time. Table 1–2 lists the behavior of Range, Rows, and Slide for special cases such as query start-up time and for an empty relation.

Table 1–2 Range, Rows, and Slide at Query Start-Up and Empty Relations

Operator or Function	Result
`COUNT(*)` or `COUNT(`*`expression`*`)`	Immediately returns 0 for an empty relation (when there is no `GROUP BY`), before `Range` or `Rows` worth of data has accumulated and before the first `Slide`.
`SUM(`*`attribute`*`)` and other aggregate functions	Immediately returns null for an empty relation, before `Range` or `Rows` worth of data has accumulated and before the first `Slide`.

For more information and detailed examples, see:

- "Range-Based Stream-to-Relation Window Operators" on page 4-6
- "Tuple-Based Stream-to-Relation Window Operators" on page 4-14
- "Partitioned Stream-to-Relation Window Operators" on page 4-19
- Section , "Functions"
- Section , "Using count With *, identifier.*, and identifier.attr"

Partition

The keyword `Partition By` logically separates an event stream `S` into different substreams based on the equality of the attributes given in the `Partition By` specification. For example, the `S[Partition By A,C Rows 2]` partition specification creates a sub-stream for every unique combination of `A` and `C` value pairs and the `Rows` specification is applied on these sub-streams. The `Rows` specification indicates "I want to look at 2 tuples worth of data".

For more information, see Section , "Range, Rows, and Slide".

Default Stream-to-Relation Operator

When you reference a stream in an Oracle CQL query where a relation is expected (most commonly in the `from` clause), a `Range Unbounded` window is applied to the stream by default. For example, the queries in Example 1–8 and Example 1–9 are identical:

Example 1–8 Query Without Stream-to-Relation Operator

```
<query id="q1"><![CDATA[
    select * from InputChannel
]]></query>
```

Example 1–9 Equivalent Query

```
<query id="q1"><![CDATA[
    IStream(select * from InputChannel[RANGE UNBOUNDED])
]]></query>
```

For more information, see Section , "Relation-to-Stream Operators".

Relation-to-Stream Operators

You can convert the result of a stream-to-relation operation back into a stream for further processing.

In Example 1–10, the select will output a stream of tuples satisfying the filter condition (`viewq3.ACCT_INTRL_ID = ValidLoopCashForeignTxn.ACCT_INTRL_ID`). The now window converts the `viewq3` into a relation, which is kept as a relation by the filter condition. The `IStream` relation-to-stream operator converts the output of the filter back into a stream.

Example 1–10 Relation-to-Stream Operation

```
<processor>
    <name>cqlProcessor</name>
    <rules>
        <query id="q3Txns"><![CDATA[
            IStream(
                select
                    TxnId,
                    ValidLoopCashForeignTxn.ACCT_INTRL_ID,
                    TRXN_BASE_AM,
                    ADDR_CNTRY_CD,
                    TRXN_LOC_ADDR_SEQ_ID
                from
                    viewq3[NOW], ValidLoopCashForeignTxn
                where
                    viewq3.ACCT_INTRL_ID = ValidLoopCashForeignTxn.ACCT_INTRL_ID
            )
        ]]></query>
    </rules>
</processor>
```

Oracle CQL supports the following relation-to-stream operators:

- `IStream`: insert stream.

  `IStream(R)` contains all `(r,T)` where `r` is in `R` at time `T` but `r` is not in R at time `T-1`.

  For more information, see "IStream Relation-to-Stream Operator" on page 4-25.

- `DSteam`: delete stream.

  `DStream(R)` contains all `(r,T)` where `r` is in `R` at time `T-1` but `r` is not in `R` at time `T`.

  For more information, see "DStream Relation-to-Stream Operator" on page 4-26.

- `RStream`: relation stream.

  `RStream(R)` contains all `(r,T)` where `r` is in `R` at time `T`.

  For more information, see "RStream Relation-to-Stream Operator" on page 4-27.

By default, Oracle Event Processing includes an operation indicator in the relations it generates so you can identify insertions, deletions, and, when using `UPDATE SEMANTICS`, updates. For more information, see Section , "Relations and Oracle Event Processing Tuple Kind Indicator".

Default Relation-to-Stream Operator

Whenever an Oracle CQL query produces a relation that is monotonic, Oracle CQL adds an `IStream` operator by default.

A relation `R` is monotonic if and only if `R(t1)` is a subset of `R(t2)` whenever `t1 <= t2`.

Oracle CQL use a conservative static monotonicity test. For example, a base relation is monotonic if it is known to be append-only: `S[Range Unbounded]` is monotonic for any stream `S`; and the join of two monotonic relations is also monotonic.

If a relation is not monotonic (for example, it has a window like `S[range 10 seconds]`), it is impossible to determine what the query author intends (`IStream`, `DStream`, or `RStream`), so Oracle CQL does not add a relation-to-stream operator by default in this case.

Stream-to-Stream Operators

Typically, you perform stream to stream operations using the following:

- A stream-to-relation operator to turn the stream into a relation. For more information, see Section , "Stream-to-Relation Operators (Windows)".
- A relation-to-relation operator to perform a relational filter. For more information, see Section , "Relation-to-Relation Operators".
- A relation-to-stream operator to turn the relation back into a stream. For more information, see Section , "Relation-to-Stream Operators".

However, some relation-relation operators (like filter and project) can also act as stream-stream operators. Consider the query that Example 1–11 shows: assuming that the input `S` is a stream, the query will produce a stream as an output where stream element `c1` is greater than 50.

Example 1–11 Stream-to-Stream Operation

```
<processor>
    <name>cqlProcessor</name>
    <rules>
        <query id="q0"><![CDATA[
            select * from S where c1 > 50
        ]]></query>
    </rules>
</processor>
```

This is a consequence of the application of the default stream-to-relation and relation-to-stream operators. The stream `S` in Example 1–11 gets a default `[Range Unbounded]` window added to it. Since this query then evaluates to a relation that is monotonic, an `IStream` gets added to it.

For more information, see:

- Section , "Default Stream-to-Relation Operator"
- Section , "Default Relation-to-Stream Operator"

In addition, Oracle CQL supports the following direct stream-to-stream operators:

- `MATCH_RECOGNIZE`: use this clause to write various types of pattern recognition queries on the input stream. For more information, see Section , "Pattern Recognition".
- `XMLTABLE`: use this clause to parse data from the `xmltype` stream elements using XPath expressions. For more information, see Section , "XMLTABLE Query".

Queries, Views, and Joins

An Oracle CQL query is an operation that you express in Oracle CQL syntax and execute on an Oracle Event Processing CQL processor to retrieve data from one or more streams, relations, or views. A top-level SELECT statement that you create in a <query> element is called a **query**. For more information, see Section , "Queries".

An Oracle CQL view represents an alternative selection on a stream or relation. In Oracle CQL, you use a view instead of a subquery. A top-level SELECT statement that you create in a <view> element is called a **view**. For more information, see Section , "Views".

Each query and view must have an identifier unique to the processor that contains it. Example 1–12 shows a query with an id of q0. The id value must conform with the specification given by *identifier::=* on page 7-17.

Example 1–12 Query and View id Attribute

```
<processor>
    <name>cqlProcessor</name>
    <rules>
        <query id="q0"><![CDATA[
            select * from S where c1 > 50
        ]]></query>
    </rules>
</processor>
```

A **join** is a query that combines rows from two or more streams, views, or relations. For more information, see Section , "Joins".

For more information, see Chapter 20, "Oracle CQL Queries, Views, and Joins".

Pattern Recognition

The Oracle CQL MATCH_RECOGNIZE construct is the principle means of performing pattern recognition.

A sequence of consecutive events or tuples in the input stream, each satisfying certain conditions constitutes a pattern. The pattern recognition functionality in Oracle CQL allows you to define conditions on the attributes of incoming events or tuples and to identify these conditions by using String names called correlation variables. The pattern to be matched is specified as a regular expression over these correlation variables and it determines the sequence or order in which conditions should be satisfied by different incoming tuples to be recognized as a valid match.

For more information, see Chapter 21, "Pattern Recognition With MATCH_ RECOGNIZE".

Event Sources and Event Sinks

An Oracle Event Processing event source identifies a producer of data that your Oracle CQL queries operate on. An Oracle CQL event sink identifies a consumer of query results.

This section explains the types of event sources and sinks you can access in your Oracle CQL queries and how you connect event sources and event sinks.

Event Sources

An Oracle Event Processing event source identifies a producer of data that your Oracle CQL queries operate on.

In Oracle Event Processing, the following elements may be event sources:

- adapter (JMS, HTTP, and file)
- channel
- processor
- table
- cache

Note: In Oracle Event Processing, you must use a channel to connect an event source to an Oracle CQL processor and to connect an Oracle CQL processor to an event sink. A channel is optional with other Oracle Event Processing processor types. For more information, see Section , "Streams and Relations".

Oracle Event Processing event sources are typically push data sources: that is, Oracle Event Processing expects the event source to notify it when the event source has data ready.

Oracle Event Processing relational database table and cache event sources are pull data sources: that is, Oracle Event Processing polls the event source on arrival of an event on the data stream.

For more information, see:

- Section , "Table Event Sources"
- Section , "Cache Event Sources"

Event Sinks

An Oracle CQL event sink connected to a CQL processor is a consumer of query results.

In Oracle Event Processing, the following elements may be event sinks:

- adapter (JMS, HTTP, and file)
- channel
- processor
- cache

You can associate the same query with more than one event sink and with different types of event sink.

Connecting Event Sources and Event Sinks

In Oracle Event Processing, you define event sources and event sinks using Oracle Event Processing IDE for Eclipse to create the Event Processing Network (EPN) as Figure 1–5 shows. In this EPN, adapter `PriceAdapter` is the event source for channel `priceStream`; channel `priceStream` is the event source for Oracle CQL processor `filterFanoutProcessor`. Similarly, Oracle CQL processor `filterFanoutProcessor` is the event sink for channel `priceStream`.

Figure 1–5 Event Sources and Event Sinks in the Event Processing Network

For more information, see:

- Section , "Streams and Relations"
- Section , "Introduction to Oracle CQL Queries, Views, and Joins"
- *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*

Table Event Sources

Using Oracle CQL, you can access tabular data, including:

- Section , "Relational Database Table Event Sources"
- Section , "XML Table Event Sources"
- Section , "Function Table Event Sources"

For more information, see Section , "Event Sources and Event Sinks"

Relational Database Table Event Sources

Using an Oracle CQL processor, you can specify a relational database table as an event source. You can query this event source, join it with other event sources, and so on.

For more information, see Section , "Oracle CQL Queries and Relational Database Tables".

XML Table Event Sources

Using the Oracle CQL `XMLTABLE` clause, you can parse data from an `xmltype` stream into columns using XPath expressions and conveniently access the data by column name.

For more information, see Section , "XMLTABLE Query".

Function Table Event Sources

Use the `TABLE` clause to access, as a relation, the multiple rows returned by a built-in or user-defined function, as an array or `Collection` type, in the `FROM` clause of an Oracle CQL query.

For more information, see:

- Section , "Function TABLE Query"
- Section , "Functions"

Cache Event Sources

Using an Oracle CQL processor, you can specify an Oracle Event Processing cache as an event source. You can query this event source and join it with other event sources using a `now` window only.

For more information, see:

- Section , "Event Sources and Event Sinks"

- Section , "Cache Query"
- "S[now]" on page 4-7

Functions

Functions are similar to operators in that they manipulate data items and return a result. Functions differ from operators in the format of their arguments. This format enables them to operate on zero, one, two, or more arguments:

```
function(argument, argument, ...)
```

A function without any arguments is similar to a pseudocolumn (refer to Chapter 3, "Pseudocolumns"). However, a pseudocolumn typically returns a different value for each tuple in a relation, whereas a function without any arguments typically returns the same value for each tuple.

Oracle CQL provides a wide variety of built-in functions to perform operations on stream data, including:

- single-row functions that return a single result row for every row of a queried stream or view
- aggregate functions that return a single aggregate result based on group of tuples, rather than on a single tuple
- single-row statistical and advanced arithmetic operations based on the Colt open source libraries for high performance scientific and technical computing.
- aggregate statistical and advanced arithmetic operations based on the Colt open source libraries for high performance scientific and technical computing.
- statistical and advanced arithmetic operations based on the `java.lang.Math` class

If Oracle CQL built-in functions do not provide the capabilities your application requires, you can easily create user-defined functions in Java by using the classes in the `oracle.cep.extensibility.functions` package. You can create aggregate and single-row user-defined functions. You can create overloaded functions and you can override built-in functions.

If you call an Oracle CQL function with an argument of a datatype other than the datatype expected by the Oracle CQL function, then Oracle Event Processing attempts to convert the argument to the expected datatype before performing the Oracle CQL function.

Note: Function names are case sensitive:

- Built-in functions: lower case.
- User-defined functions: `welvs:function` element `function-name` attribute determines the case you use.

For more information, see:

- Chapter 8, "Built-In Single-Row Functions"
- Chapter 9, "Built-In Aggregate Functions"
- Chapter 10, "Colt Single-Row Functions"
- Chapter 11, "Colt Aggregate Functions"
- Chapter 12, "java.lang.Math Functions"

- Chapter 13, "User-Defined Functions"
- Section , "Datatype Conversion"

Data Cartridges

The Oracle CQL data cartridge framework allows you to tightly integrate arbitrary domain objects with the Oracle CQL language and use domain object fields, methods, and constructors within Oracle CQL queries in the same way you use Oracle CQL native types.

Currently, Oracle Event Processing provides the following data cartridges:

- Oracle Java data cartridge: this data cartridge exposes Java types, methods, fields, and constructors that you can use in Oracle CQL queries and views as you would Oracle CQL native types.

  See Chapter 15, "Oracle Java Data Cartridge".

- Oracle Spatial: this data cartridge exposes Oracle Spatial types, methods, fields, and constructors that you can use in Oracle CQL queries and views as you would Oracle CQL native types.

  See Chapter 16, "Oracle Spatial".

- Oracle JDBC data cartridge: this data cartridge allows you to incorporate arbitrary SQL functions against multiple tables and data sources in Oracle CQL queries and views as you would Oracle CQL native types.

  See Chapter 17, "Oracle Event Processing JDBC Data Cartridge".

For more information, see:

- Section , "Understanding Data Cartridges"
- Section , "Oracle CQL Data Cartridge Types"

Time

Timestamps are an integral part of an Oracle Event Processing stream. However, timestamps do not necessarily equate to clock time. For example, time may be defined in the application domain where it is represented by a sequence number. Timestamps need only guarantee that updates arrive at the system in the order of increasing timestamp values.

Note that the timestamp ordering requirement is specific to one stream or a relation. For example, tuples of different streams could be arbitrarily interleaved.

Oracle Event Processing can observe application time or system time.

To configure application timestamp or system timestamp operation, see child element `application-timestamped` in "wlevs:channel" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

For system timestamped relations or streams, time is dependent upon the arrival of data on the relation or stream data source. Oracle Event Processing generates a heartbeat on a system timestamped relation or stream if there is no activity (no data arriving on the stream or relation's source) for more than a specified time: for example, 1 minute. Either the relation or stream is populated by its specified source or Oracle Event Processing generates a heartbeat every minute. This way, the relation or stream can never be more than 1 minute behind.

To configure a heartbeat, see "heartbeat" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

For system timestamped streams and relations, the system assigns time in such a way that no two events will have the same value of time. However, for application timestamped streams and relations, events could have same value of time.

If you know that the application timestamp will be strictly increasing (as opposed to non-decreasing) you may set `wlevs:channel` attribute `is-total-order` to `true`. This enables the Oracle Event Processing engine to do certain optimizations and typically leads to reduction in processing latency.

To configure `is-total-order`, see "wlevs:application-timestamped" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

The Oracle Event Processing scheduler is responsible for continuously executing each Oracle CQL query according to its scheduling algorithm and frequency.

For more information on the scheduler, see "scheduler" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

Oracle CQL Statements

Oracle CQL provides statements for creating queries and views.

This section describes:

- Section , "Lexical Conventions"
- Section , "Syntactic Shortcuts and Defaults"
- Section , "Documentation Conventions"

For more information, see:

- Chapter 20, "Oracle CQL Queries, Views, and Joins"
- Chapter 22, "Oracle CQL Statements"

Lexical Conventions

Using Oracle Event Processing IDE for Eclipse or Oracle Event Processing Visualizer, you write Oracle CQL statements in the XML configuration file associated with an Oracle Event Processing CQL processor. This XML file is called the configuration source.

The configuration source must conform with the `wlevs_application_config.xsd` schema and may contain only `rule`, `view`, or `query` elements as Example 1–13 shows.

Example 1–13 Typical Oracle CQL Processor Configuration Source File

```
<?xml version="1.0" encoding="UTF-8"?>
<n1:config xsi:schemaLocation="http://www.bea.com/ns/wlevs/config/application wlevs_
application_config.xsd"
    xmlns:n1="http://www.bea.com/ns/wlevs/config/application"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<processor>
    <name>cqlProcessor</name>
    <rules>
        <view id="lastEvents" schema="cusip bid srcId bidQty ask askQty seq"><![CDATA[
            select cusip, bid, srcId, bidQty, ask, askQty, seq
            from inputChannel[partition by srcId, cusip rows 1]
        ]]></view>
        <view id="bidask" schema="cusip bid ask"><![CDATA[
```

```
            select cusip, max(bid), min(ask)
            from lastEvents
            group by cusip
        ]]></view>
            <view ...><![CDATA[
                ...
        ]]></view>
        ...
        <view id="MAXBIDMINASK" schema="cusip bidseq bidSrcId bid askseq askSrcId ask bidQty
askQty"><![CDATA[
            select bid.cusip, bid.seq, bid.srcId as bidSrcId, bid.bid, ask.seq, ask.srcId as
askSrcId, ask.ask, bid.bidQty, ask.askQty
            from BIDMAX as bid, ASKMIN as ask
            where bid.cusip = ask.cusip
        ]]></view>
        <query id="BBAQuery"><![CDATA[
            ISTREAM(select bba.cusip, bba.bidseq, bba.bidSrcId, bba.bid, bba.askseq,
                bba.askSrcId, bba.ask, bba.bidQty, bba.askQty, "BBAStrategy" as
intermediateStrategy,
                p.seq as correlationId, 1 as priority
            from MAXBIDMINASK as bba, inputChannel[rows 1] as p where bba.cusip = p.cusip)
        ]]></query>
    </rules>
</processor>
```

When writing Oracle CQL queries in an Oracle CQL processor component configuration file, observe the following rules:

- You may specify one Oracle CQL statement per `view` or `query` element.
- You must *not* terminate Oracle CQL statements with a semicolon (`;`).
- You must enclose each Oracle CQL statement in `<![CDATA[` and `]]>` as Example 1–13 shows.
- When you issue an Oracle CQL statement, you can include one or more tabs, carriage returns, or spaces anywhere a space occurs within the definition of the statement. Thus, Oracle Event Processing evaluates the Oracle CQL statement in Example 1–14 and Example 1–15 in the same manner.

Example 1–14 Oracle CQL: Without Whitespace Formatting

```
<processor>
    <name>cqlProcessor</name>
    <rules>
        <query id="QTollStr"><![CDATA[
            RSTREAM(select cars.car_id, SegToll.toll from CarSegEntryStr[now] as cars,
SegToll
                where (cars.exp_way = SegToll.exp_way and cars.lane = SegToll.lane
                    and cars.dir = SegToll.dir and cars.seg = SegToll.seg))
        ]]></query>
    </rules>
</processor>
```

Example 1–15 Oracle CQL: With Whitespace Formatting

```
<processor>
    <name>cqlProcessor</name>
    <rules>
        <query id="QTollStr"><![CDATA[
            RSTREAM(
                select
                    cars.car_id,
                    SegToll.toll
                from
```

```
                    CarSegEntryStr[now]
                as
                    cars, SegToll
                where (
                    cars.exp_way = SegToll.exp_way and
                    cars.lane = SegToll.lane and
                    cars.dir = SegToll.dir and
                    cars.seg = SegToll.seg
                )
            )
        ]]></query>
    </rules>
</processor>
```

- Case is insignificant in reserved words, keywords, identifiers and parameters. However, case is significant in function names, text literals, and quoted names.

  For more information, see:

  – Section , "Functions"

  – Section , "Literals"

  – Section , "Schema Object Names and Qualifiers"

- Comments are not permitted in Oracle CQL statements. For more information, see Section , "Comments".

Note: Throughout the *Oracle Fusion Middleware CQL Language Reference for Oracle Event Processing*, Oracle CQL statements are shown only with their `view` or `query` element for clarity.

Syntactic Shortcuts and Defaults

When writing Oracle CQL queries, views, and joins, consider the syntactic shortcuts and defaults that Oracle CQL provides to simplify your queries.

For more information, see:

- Section , "Default Stream-to-Relation Operator"
- Section , "Default Relation-to-Stream Operator"
- "HelloWorld Example" in the *Oracle Fusion Middleware Getting Started Guide for Oracle Event Processing*

Documentation Conventions

All Oracle CQL statements in this reference (see Chapter 22, "Oracle CQL Statements") are organized into the following sections:

Syntax The syntax diagrams show the keywords and parameters that make up the statement.

Caution: Not all keywords and parameters are valid in all circumstances. Be sure to refer to the "Semantics" section of each statement and clause to learn about any restrictions on the syntax.

Purpose The "Purpose" section describes the basic uses of the statement.

Prerequisites The "Prerequisites" section lists privileges you must have and steps that you must take before using the statement.

Semantics The "Semantics" section describes the purpose of the keywords, parameter, and clauses that make up the syntax, and restrictions and other usage notes that may apply to them. (The conventions for keywords and parameters used in this chapter are explained in the Preface of this reference.)

Examples The "Examples" section shows how to use the various clauses and parameters of the statement.

Oracle CQL and SQL Standards

Oracle CQL is a new technology but it is based on a subset of SQL99.

Oracle strives to comply with industry-accepted standards and participates actively in SQL standards committees. Oracle is actively pursuing Oracle CQL standardization.

Oracle Event Processing Server and Tools Support

Using the Oracle Event Processing server and tools, you can efficiently create, package, deploy, debug, and manage Oracle Event Processing applications that use Oracle CQL.

Oracle Event Processing Server

Oracle Event Processing server provides the light-weight Spring container for Oracle Event Processing applications and manages server and application lifecycle, provides a JRockit real-time JVM with deterministic garbage collection, and a wide variety of essential services such as security, Jetty, JMX, JDBC, HTTP publish-subscribe, and logging and debugging.

For more information on Oracle Event Processing server, see *Oracle Fusion Middleware Administrator's Guide for Oracle Event Processing*.

Oracle Event Processing Tools

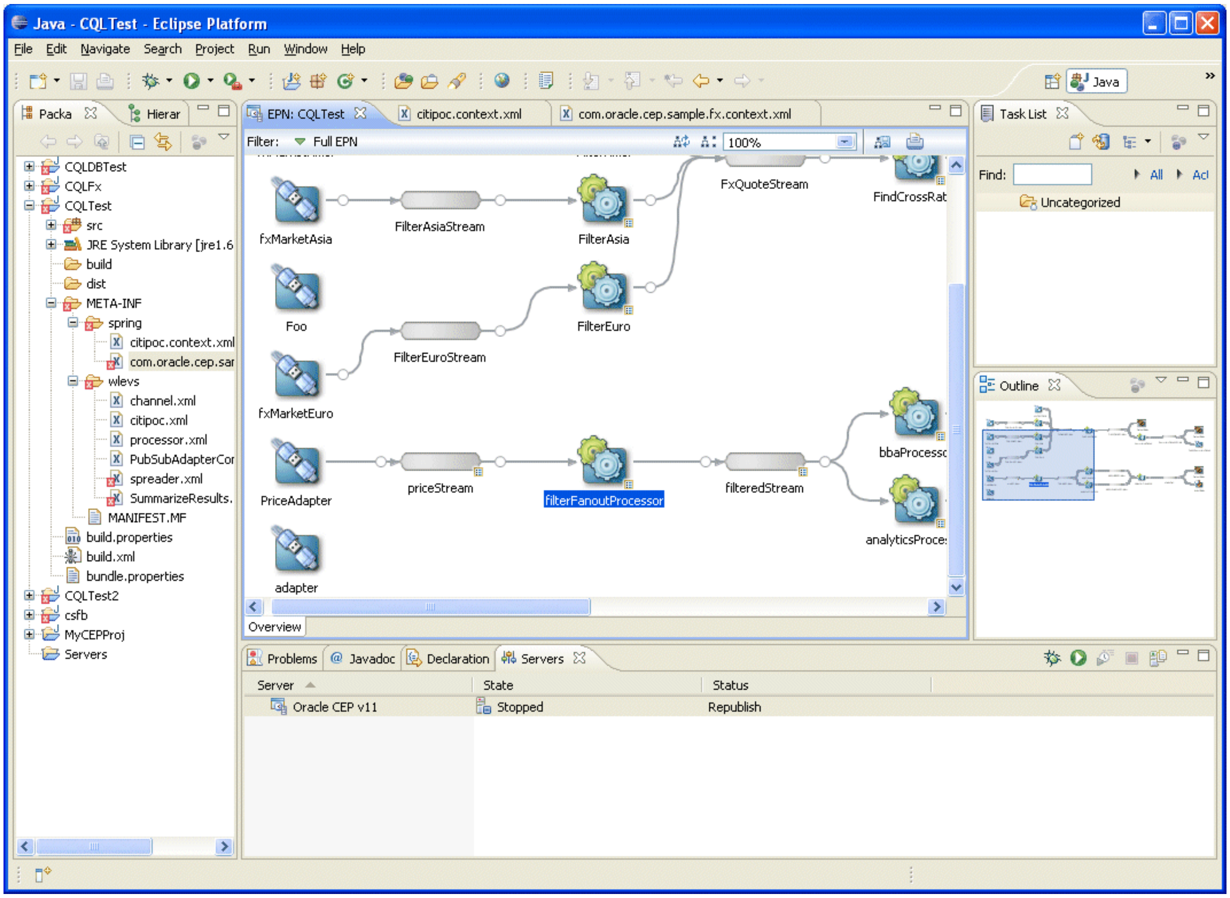
Oracle Event Processing provides the following tools to facilitate your Oracle CQL development process:

- Section , "Oracle Event Processing IDE for Eclipse"
- Section , "Oracle Event Processing Visualizer"

Oracle Event Processing IDE for Eclipse

Oracle Event Processing IDE for Eclipse is targeted specifically to programmers that want to develop Oracle Event Processing applications as Figure 1–6 shows.

Figure 1–6 Oracle Event Processing IDE for Eclipse



The Oracle Event Processing IDE for Eclipse is a set of plugins for the Eclipse IDE designed to help develop, deploy, and debug Oracle Event Processing applications.

The key features of Oracle Event Processing IDE for Eclipse are:

- Project creation wizards and templates to quickly get started building event driven applications.
- Advanced editors for source files including Java and XML files common to Oracle Event Processing applications.
- Integrated server management to seamlessly start, stop, and deploy to Oracle Event Processing server instances all from within the IDE.
- Integrated debugging.
- Event Processing Network (EPN) visual design views for orienting and navigating in event processing applications.
- Integrated support for the Oracle Event Processing Visualizer so you can use the Oracle Event Processing Visualizer from within the IDE (see Section , "Oracle Event Processing Visualizer").

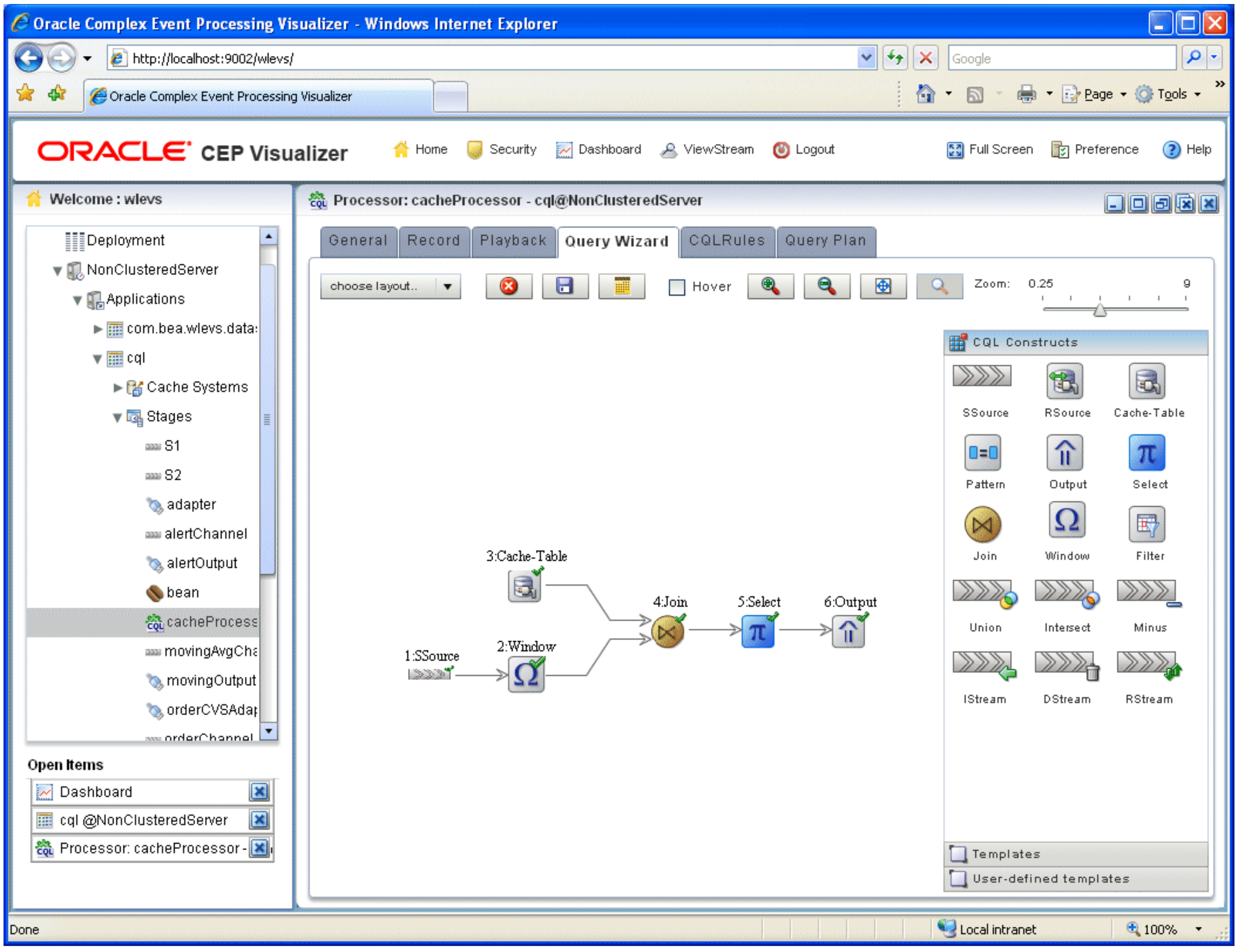
For details, see:

- *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*
- `http://www.oracle.com/technology/products/event-driven-architecture/cep-ide/11/index.html`

Oracle Event Processing Visualizer

Oracle provides an advanced run-time administration console called the Oracle Event Processing Visualizer as Figure 1–7 shows.

Figure 1–7 Oracle Event Processing Visualizer



Using Oracle Event Processing Visualizer, you can manage, tune, and monitor Oracle Event Processing server domains and the Oracle Event Processing applications you deploy to them all from a browser. Oracle Event Processing Visualizer provides a variety of sophisticated run-time administration tools, including support for Oracle CQL and EPL rule maintenance and creation.

For details, see *Oracle Fusion Middleware Visualizer User's Guide for Oracle Event Processing*

2

Basic Elements of Oracle CQL

This chapter provides a reference for fundamental parts of Oracle Continuous Query Language (Oracle CQL), including datatypes, literals, nulls, and more. Oracle CQL is the query language used in Oracle Event Processing applications.

This chapter includes the following sections:

- Datatypes
- Datatype Comparison Rules
- Literals
- Format Models
- Nulls
- Comments
- Aliases
- Schema Object Names and Qualifiers

Before using the statements described in Part IV, "Using Oracle CQL", you should familiarize yourself with the concepts covered in this chapter.

Datatypes

Each value manipulated by Oracle Event Processing has a datatype. The datatype of a value associates a fixed set of properties with the value. These properties cause Oracle Event Processing to treat values of one datatype differently from values of another. For example, you can add values of `INTEGER` datatype, but not values of `CHAR` datatype.When you create a stream, you must specify a datatype for each of its elements. When you create a user-defined function, you must specify a datatype for each of its arguments. These datatypes define the domain of values that each element can contain or each argument can have. For example, attributes with `TIMESTAMP` as datatype cannot accept the value February 29 (except for a leap year) or the values 2 or 'SHOE'.Oracle CQL provides a number of built-in datatypes that you can use. The syntax of Oracle CQL datatypes appears in the diagrams that follow.

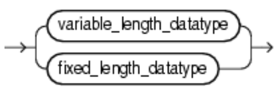
If Oracle CQL does not support a datatype that your events use, you can use an Oracle CQL data cartridge or a user-defined function to evaluate that datatype in an Oracle CQL query.

For more information, see:

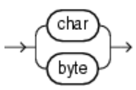
- Section , "Oracle CQL Built-in Datatypes"
- Section , "Handling Other Datatypes Using Oracle CQL Data Cartridges"

- Section , "Handling Other Datatypes Using a User-Defined Function"
- Section , "Datatype Comparison Rules"
- Section , "Literals"
- Section , "Format Models"
- Section , "How to Define a Data Type Alias Using the Aliases Element"

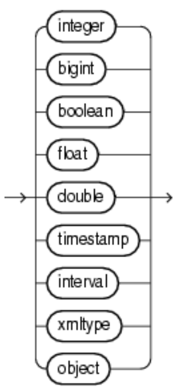
***datatype*::=**



***variable_length_datatype*::=**



***fixed_length_datatype*::=**



Oracle CQL Built-in Datatypes

Table 2–1 summarizes Oracle CQL built-in datatypes. Refer to the syntax in the preceding sections for the syntactic elements.

Consider these datatype and datatype literal restrictions when defining event types. For more information, see "Creating Oracle Event Processing Event Types" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

Table 2–1 Oracle CQL Built-in Datatype Summary

Oracle CQL Datatype	Description
`BIGINT`	Fixed-length number equivalent to a Java `Long` type. For more information, see Section , "Numeric Literals".
`BOOLEAN`	Fixed-length boolean equivalent to a Java `Boolean` type. Valid values are `true` or `false`.
`BYTE[(`*`size`*`)]`[1]	Variable-length character data of length *`size`* bytes. Maximum *`size`* is 4096 bytes. Default and minimum *`size`* is 1 byte. For more information, see Section , "Numeric Literals".
`CHAR[(`*`size`*`)]`[1]	Variable-length character data of length *`size`* characters. Maximum *`size`* is 4096 characters. Default and minimum *`size`* is 1 character. For more information, see Section , "Text Literals".

Table 2–1 (Cont.) Oracle CQL Built-in Datatype Summary

Oracle CQL Datatype	Description
DOUBLE	Fixed-length number equivalent to a Java `double` type. For more information, see Section , "Numeric Literals".
FLOAT	Fixed-length number equivalent to a Java `float` type. For more information, see Section , "Numeric Literals".
INTEGER	Fixed-length number equivalent to a Java `int` type. For more information, see Section , "Numeric Literals".
INTERVAL	Fixed-length `INTERVAL` datatype specifies a period of time. Oracle Event Processing supports `DAY TO SECOND`. Maximum length is 64 bytes. This corresponds to a Java `long` type. For more information, see Section , "Interval Literals".
TIMESTAMP	Fixed-length `TIMESTAMP` datatype stores a datetime literal that conforms to one of the `java.text.SimpleDateFormat` format models that Oracle CQL supports. Maximum length is 64 bytes. For more information, see Section , "Datetime Literals".
XMLTYPE	Use this datatype for stream elements that contain XML data. Maximum length is 4096 characters. `XMLTYPE` is a system-defined type, so you can use it as an argument of a function or as the datatype of a stream attribute. This corresponds to a Java `java.lang.String` type. For more information, see "SQL/XML (SQLX)" on page 5-16.
OBJECT	This stands for any Java object (that is, any subclass of `java.lang.Object`). We refer to this as opaque type support in Oracle Event Processing since the Oracle Event Processing engine does not understand the contents of an `OBJECT` field. You typically use this type to pass values, from an adapter to its destination, as-is; these values need not be interpreted by the Oracle Event Processing engine (such as `Collection` types or any other user-specific Java type) but that are associated with the event whose other fields are referenced in a query.

[1] Oracle CQL supports single-dimension arrays only.

Handling Other Datatypes Using Oracle CQL Data Cartridges

If your event uses a datatype that Oracle CQL does not support, you can use an Oracle CQL data cartridge to evaluate that datatype in an Oracle CQL query.

Oracle CQL includes the following data cartridges:

- Chapter 15, "Oracle Java Data Cartridge"
- Chapter 16, "Oracle Spatial"
- Chapter 17, "Oracle Event Processing JDBC Data Cartridge"

For more information, see Chapter 14, "Introduction to Data Cartridges".

Handling Other Datatypes Using a User-Defined Function

If your event uses a datatype that Oracle CQL does not support, you can create a user-defined function to evaluate that datatype in an Oracle CQL query.

Consider the `enum` datatype that Example 2–1 shows. The event that Example 2–2 shows uses this `enum` datatype. Oracle CQL does not support `enum` datatypes.

Example 2–1 Enum Datatype ProcessStatus

```
package com.oracle.app;

public enum ProcessStatus {
    OPEN(1),
    CLOSED(0)}
}
```

Example 2–2 Event Using Enum Datatype ProcessStatus

```
package com.oracle.app;

import com.oracle.capp.ProcessStatus;

public class ServiceOrder {
    private String serviceOrderId;
    private String electronicSerialNumber;
    private ProcessStatus status;
...
}
```

By creating the user-defined function that Example 2–3 shows and registering the function in your application assembly file as Example 2–4 shows, you can evaluate this `enum` datatype in an Oracle CQL query as Example 2–5 shows.

Example 2–3 User-Defined Function to Evaluate Enum Datatype

```
package com.oracle.app;

import com.oracle.capp.ProcessStatus;
public class CheckIfStatusClosed {
    public boolean execute(Object[] args) {
        ProcessStatus arg0 = (ProcessStatus)args[0];
        if (arg0 == ProcessStatus.OPEN)
            return Boolean.FALSE;
        else
            return Boolean.TRUE;
    }
}
```

Example 2–4 Registering the User-Defined Function in Application Assembly File

```
<wlevs:processor id="testProcessor">
    <wlevs:listener ref="providerCache"/>
    <wlevs:listener ref="outputCache"/>
    <wlevs:cache-source ref="testCache"/>
    <wlevs:function function-name="statusClosed" exec-method="execute" />
        <bean class="com.oracle.app.CheckIfStatusClosed"/>
    </wlevs:function>
</wlevs:processor>
```

Example 2–5 Using the User-Defined Function to Evaluate Enum Datatype in an Oracle CQL Query

```
<query id="rule-04"><![CDATA[
    SELECT
        meter.electronicSerialNumber,
        meter.exceptionKind
    FROM
        MeterLogEvent AS meter,
        ServiceOrder AS svco
    WHERE
        meter.electronicSerialNumber = svco.electronicSerialNumber and
        svco.serviceOrderId IS NULL OR statusClosed(svco.status)
]]></query>
```

For more information, see Chapter 13, "User-Defined Functions".

Datatype Comparison Rules

This section describes how Oracle Event Processing compares values of each datatype.

Numeric Values

A larger value is considered greater than a smaller one. All negative numbers are less than zero and all positive numbers. Thus, -1 is less than 100; -100 is less than -1.

Date Values

A later date is considered greater than an earlier one. For example, the date equivalent of `'29-MAR-2005'` is less than that of `'05-JAN-2006'` and `'05-JAN-2006 1:35pm'` is greater than `'05-JAN-2005 10:09am'`.

Character Values

Oracle CQL supports Lexicographic sort based on dictionary order.

Internally, Oracle CQL compares the numeric value of the `char`. Depending on the encoding used, the numeric values will differ, but in general, the comparison will remain the same. For example:

```
'a' < 'b'
'aa' < 'ab'
'aaaa' < 'aaaab'
```

Datatype Conversion

Generally an expression cannot contain values of different datatypes. For example, an arithmetic expression cannot multiply 5 by 10 and then add `'JAMES'`. However, Oracle Event Processing supports both implicit and explicit conversion of values from one datatype to another.

Oracle recommends that you specify explicit conversions, rather than rely on implicit or automatic conversions, for these reasons:

- Oracle CQL statements are easier to understand when you use explicit datatype conversion functions.
- Implicit datatype conversion can have a negative impact on performance.
- Implicit conversion depends on the context in which it occurs and may not work the same way in every case.
- Algorithms for implicit conversion are subject to change across software releases and among Oracle products. Behavior of explicit conversions is more predictable.

This section describes:

- Section , "Implicit Datatype Conversion"
- Section , "Explicit Datatype Conversion"
- Section , "SQL Datatype Conversion"
- Section , "Oracle Data Cartridge Datatype Conversion"
- Section , "User-Defined Function Datatype Conversion"

Implicit Datatype Conversion

Oracle Event Processing automatically converts a value from one datatype to another when such a conversion makes sense.

Table 2–2 is a matrix of Oracle implicit conversions. The table shows all possible conversions (marked with an `X`). Unsupported conversions are marked with a `--`.

Table 2–2 Implicit Type Conversion Matrix

	to CHAR	to BYTE	to BOOLEAN	to INTEGER	to DOUBLE	to BIGINT	to FLOAT	to TIMESTAMP	to INTERVAL
from CHAR	--	--	--	--	--	--	--	X	--
from BYTE	X	--	--	--	--	--	--	--	--
from BOOLEAN	--	--	X	--	--	--	--	--	--
from INTEGER	X	--	--	--	X	X	X	--	--
from DOUBLE	X	--	--	--	X	--	--	--	--
from BIGINT	X	--	--	--	X	--	X	--	--
from FLOAT	X	--	--	--	X	--	--	--	--
from TIMESTAMP	X	--	--	--	--	--	--	--	--
from INTERVAL	X	--	--	--	--	--	--	--	--

The following rules govern the direction in which Oracle Event Processing makes implicit datatype conversions:

- During `SELECT FROM` operations, Oracle Event Processing converts the data from the stream to the type of the target variable if the select clause contains arithmetic expressions or condition evaluations.

  For example, implicit conversions occurs in the context of expression evaluation, such as `c1+2.0`, or condition evaluation, such as `c1 < 2.0`, where `c1` is of type `INTEGER`.
- Conversions from `FLOAT` to `BIGINT` are exact.
- Conversions from `BIGINT` to `FLOAT` are inexact if the `BIGINT` value uses more bits of precision that supported by the `FLOAT`.
- When comparing a character value with a `TIMESTAMP` value, Oracle Event Processing converts the character data to `TIMESTAMP`.
- When you use a Oracle CQL function or operator with an argument of a datatype other than the one it accepts, Oracle Event Processing converts the argument to the accepted datatype wherever supported.
- When making assignments, Oracle Event Processing converts the value on the right side of the equal sign (=) to the datatype of the target of the assignment on the left side.
- During concatenation operations, Oracle Event Processing converts from noncharacter datatypes to `CHAR`.
- During arithmetic operations on and comparisons between character and noncharacter datatypes, Oracle Event Processing converts from numeric types to `CHAR` as Table 2–2 shows.

Explicit Datatype Conversion

You can explicitly specify datatype conversions using Oracle CQL conversion functions. Table 2–3 shows Oracle CQL functions that explicitly convert a value from one datatype to another. Unsupported conversions are marked with a `--`.

Table 2–3 Explicit Type Conversion Matrix

	to CHAR	to BYTE	to BOOLEAN	to INTEGER	to DOUBLE	to BIGINT	to FLOAT	to TIMESTAMP	to INTERVAL
from CHAR	--	`hextoraw`	--	--	--	--	--	`to_timestamp`	--
from BYTE	--	`rawtohex`	--	--	--	--	--	--	--
from BOOLEAN									
from INTEGER	`to_char`	--	`to_boolean`	--	`to_double`	`to_bigint`	`to_float`	--	--
from DOUBLE	`to_char`	--	--	--	--	--	--	--	--
from LONG	--	--	--	--	--	--	--	`to_timestamp`	--
from BIGINT	`to_char`	--	`to_boolean`	--	`to_double`	--	`to_float`	--	--
from FLOAT	`to_char`	--	--	--	`to_double`	--	--	--	--
from TIMESTAMP	`to_char`	--	--	--	--	--	--	--	--
from INTERVAL	`to_char`	--	--	--	--	--	--	--	--

SQL Datatype Conversion

Using an Oracle CQL processor, you can specify a relational database table as an event source. You can query this event source, join it with other event sources, and so on. When doing so, you must observe the SQL and Oracle Event Processing data type equivalents that Oracle Event Processing supports.

For more information, see:

- Section , "Relational Database Table Query"
- "SQL Column Types and Oracle Event Processing Type Equivalents" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*

Oracle Data Cartridge Datatype Conversion

At run time, Oracle Event Processing maps between Oracle CQL and data cartridge datatypes according to the data cartridge's implementation.

For more information, see:

- Oracle Java data cartridge: Section , "Datatype Mapping"
- Oracle Spatial: Section , "Datatype Mapping"

User-Defined Function Datatype Conversion

At run time, Oracle Event Processing maps between the Oracle CQL datatype you specify for a user-defined function's return type and its Java datatype equivalent.

For more information, see Section , "User-Defined Function Datatypes".

Literals

The terms **literal** and **constant value** are synonymous and refer to a fixed data value. For example, `'JACK'`, `'BLUE ISLAND'`, and `'101'` are all text literals; 5001 is a numeric literal.

Oracle Event Processing supports the following types of literals in Oracle CQL statements:

- Text Literals
- Numeric Literals
- Datetime Literals
- Interval Literals

Text Literals

Use the text literal notation to specify values whenever `const_string`, `quoted_string_double_quotes`, or `quoted_string_single_quotes` appears in the syntax of expressions, conditions, Oracle CQL functions, and Oracle CQL statements in other parts of this reference. This reference uses the terms **text literal**, **character literal**, and **string** interchangeably.

Text literals are enclosed in single or double quotation marks so that Oracle Event Processing can distinguish them from schema object names.

You may use single quotation marks (`'`) or double quotation marks (`"`). Typically, you use double quotation marks. However, for certain expressions, conditions, functions, and statements, you must use the quotation marks as specified in the syntax given in other parts of this reference: either *`quoted_string_double_quotes`* or *`quoted_string_single_quotes`*.

If the syntax uses simply *`const_string`*, then you can use either single or double quotation marks.

If the syntax uses the term *`char`*, then you can specify either a text literal or another expression that resolves to character data. When *`char`* appears in the syntax, the single quotation marks are not used.

Oracle Event Processing supports Java localization. You can specify text literals in the character set specified by your Java locale.

For more information, see:

- Section , "Lexical Conventions"
- Section , "Schema Object Names and Qualifiers"
- *const_string::=* on page 7-13

Numeric Literals

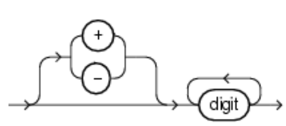
Use numeric literal notation to specify fixed and floating-point numbers.

Integer Literals

You must use the integer notation to specify an integer whenever *`integer`* appears in expressions, conditions, Oracle CQL functions, and Oracle CQL statements described in other parts of this reference.

The syntax of `integer` follows:

***integer*::=**



where *`digit`* is one of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

An integer can store a maximum of 32 digits of precision.

Here are some valid integers:

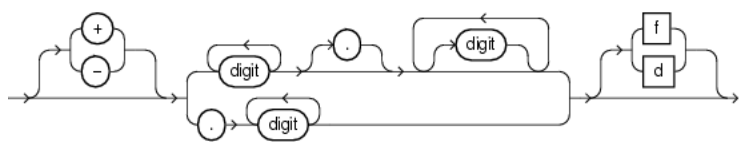
```
7
+255
```

Floating-Point Literals

You must use the number or floating-point notation to specify values whenever *number* or *n* appears in expressions, conditions, Oracle CQL functions, and Oracle CQL statements in other parts of this reference.

The syntax of *number* follows:

***number*::=**



where

- + or - indicates a positive or negative value. If you omit the sign, then a positive value is the default.
- *digit* is one of 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9.
- f or F indicates that the number is a 64-bit binary floating point number of type FLOAT.
- d or D indicates that the number is a 64-bit binary floating point number of type DOUBLE.

  If you omit f or F and d or D, then the number is of type INTEGER.

  The suffixes f or F and d or D are supported only in floating-point number literals, not in character strings that are to be converted to INTEGER. For example, if Oracle Event Processing is expecting an INTEGER and it encounters the string '9', then it converts the string to the Java Integer 9. However, if Oracle Event Processing encounters the string '9f', then conversion fails and an error is returned.

A number of type INTEGER can store a maximum of 32 digits of precision. If the literal requires more precision than provided by BIGINT or FLOAT, then Oracle Event Processing truncates the value. If the range of the literal exceeds the range supported by BIGINT or FLOAT, then Oracle Event Processing raises an error.

If your Java locale uses a decimal character other than a period (.), then you must specify numeric literals with *'text'* notation. In these cases, Oracle Event Processing automatically converts the text literal to a numeric value.

Note: You cannot use this notation for floating-point number literals.

For example, if your Java locale specifies a decimal character of comma (,), specify the number 5.123 as follows:

```
'5,123'
```

Here are some valid NUMBER literals:

```
25
```

```
+6.34
0.5
-1
```

Here are some valid floating-point number literals:

```
25f
+6.34F
0.5d
-1D
```

Datetime Literals

Oracle Event Processing supports datetime datatype `TIMESTAMP`.

Datetime literals must not exceed 64 bytes.

All datetime literals must conform to one of the `java.text.SimpleDateFormat` format models that Oracle CQL supports. For more information, see Section , "Datetime Format Models".

Currently, the `SimpleDateFormat` class does not support `xsd:dateTime`. As a result, Oracle CQL does not support XML elements or attributes that use this type.

For example, if your XML event uses an XSD like Example 2–6, Oracle CQL cannot parse the `MyTimestamp` element.

Example 2–6 Invalid Event XSD: xsd:dateTime is Not Supported

```
<xsd:element name="ComplexTypeBody">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="MyTimestamp" type="xsd:dateTime"/>
            <xsd:element name="ElementKind" type="xsd:string"/>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="node" type="SimpleType"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Oracle recommends that you define your XSD to replace `xsd:dateTime` with `xsd:string` as Example 2–7 shows.

Example 2–7 Valid Event XSD: Using xsd:string Instead of xsd:dateTime

```
<xsd:element name="ComplexTypeBody">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="MyTimestamp" type="xsd:string"/>
            <xsd:element name="ElementKind" type="xsd:string"/>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="node" type="SimpleType"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Using the XSD from Example 2–7, Oracle CQL can process events such as that shown in Example 2–8 as long as the `Timestamp` element's `String` value conforms to the `java.text.SimpleDateFormat` format models that Oracle CQL supports. For more information, see Section , "Datetime Format Models".

Example 2–8 XML Event Payload

```
<ComplexTypeBody xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ...>
```

```
  <MyTimestamp>2000-01-15T00:00:00</MyTimestamp>
  <ElementKind>plus</ElementKind>
  <name>complexEvent</name>
  <node>
    <type>complexNode</type>
    <number>1</number>
  </node>
</ComplexTypeBody>
```

For more information on using XML with Oracle CQL, see "SQL/XML (SQLX)" on page 5-16.

Interval Literals

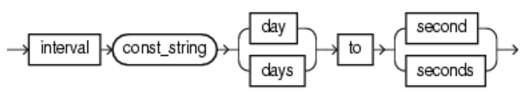
An interval literal specifies a period of time. Oracle Event Processing supports interval literal DAY TO SECOND. This literal contains a leading field and may contain a trailing field. The leading field defines the basic unit of date or time being measured. The trailing field defines the smallest increment of the basic unit being considered. Part ranges (such as only SECOND or MINUTE to SECOND) are not supported.

Interval literals must not exceed 64 bytes.

INTERVAL DAY TO SECOND

Specify DAY TO SECOND interval literals using the following syntax:

***interval_value*::=**



where *const_string* is a TIMESTAMP value that conforms to the appropriate datetime format model (see Section , "Datetime Format Models").

Examples of the various forms of INTERVAL DAY TO SECOND literals follow:

Form of Interval Literal	Interpretation
INTERVAL '4 5:12:10.222' DAY TO SECOND(3)	4 days, 5 hours, 12 minutes, 10 seconds, and 222 thousandths of a second.

You can add or subtract one DAY TO SECOND interval literal from another DAY TO SECOND literal and compare one interval literal to another as Example 2–9 shows. In this example, stream tkdata2_SIn1 has schema (c1 integer, c2 interval).

Example 2–9 Comparing Intervals

```
<query id="tkdata2_q295"><![CDATA
select * from tkdata2_SIn1 where (c2 + INTERVAL "2 1:03:45.10" DAY TO SECOND) > INTERVAL "6
12:23:45.10" DAY TO SECOND
]]></query>
```

Format Models

A **format model** is a character literal that describes the format of datetime or numeric data stored in a character string. When you convert a character string into a date or number, a format model determines how Oracle Event Processing interprets the string. The following format models are relevant to Oracle CQL queries:

- Number Format Models
- Datetime Format Models

Number Format Models

You can use number format models in the following functions:

- In the `to_bigint` function to translate a value of `int` datatype to `bigint` datatype.
- In the `to_float` function to translate a value of `int` or `bigint` datatype to `float` datatype

Datetime Format Models

Oracle CQL supports the format models that the `java.text.SimpleDateFormat` specifies.

Table 2–4 lists the `java.text.SimpleDateFormat` models that Oracle CQL uses to interpret `TIMESTAMP` literals. For more information, see Section , "Datetime Literals".

Table 2–4 Datetime Format Models

Format Model	Example
`MM/dd/yyyy HH:mm:ss Z`	`11/21/2005 11:14:23 -0800`
`MM/dd/yyyy HH:mm:ss z`	`11/21/2005 11:14:23 PST`
`MM/dd/yyyy HH:mm:ss`	`11/21/2005 11:14:23`
`MM-dd-yyyy HH:mm:ss`	`11-21-2005 11:14:23`
`dd-MMM-yy`	`15-DEC-01`
`yyyy-MM-dd'T'HH:mm:ss`	`2005-01-01T08:12:12`

You can use a datetime format model in the following functions:

- `to_timestamp`: to translate the value of a `char` datatype to a `TIMESTAMP` datatype.

Nulls

If a column in a row has no value, then the column is said to be **null**, or to contain null. Nulls can appear in tuples of any datatype that are not restricted by primary key integrity constraints. Use a null when the actual value is not known or when a value would not be meaningful.

Oracle Event Processing treats a character value with a length of zero as null. However, do not use null to represent a numeric value of zero, because they are not equivalent.

> **Note:** Oracle Event Processing currently treats a character value with a length of zero as null. However, this may not continue to be true in future releases, and Oracle recommends that you do not treat empty strings the same as nulls.

Any arithmetic expression containing a null always evaluates to null. For example, null added to 10 is null. In fact, all operators (except concatenation) return null when given a null operand.

For more information, see:

- "nvl" on page 8-8
- *null_spec::=* on page 22-5

Nulls in Oracle CQL Functions

All scalar functions (except `nvl` and `concat`) return null when given a null argument. You can use the `nvl` function to return a value when a null occurs. For example, the expression `NVL(commission_pct,0)` returns 0 if `commission_pct` is null or the value of `commission_pct` if it is not null.

Most aggregate functions ignore nulls. For example, consider a query that averages the five values `1000`, `null`, `null`, `null`, and `2000`. Such a query ignores the nulls and calculates the average to be `(1000+2000)/2 = 1500`.

Nulls with Comparison Conditions

To test for nulls, use only the null comparison conditions (see *null_conditions::=* on page 6-8). If you use any other condition with nulls and the result depends on the value of the null, then the result is `UNKNOWN`. Because null represents a lack of data, a null cannot be equal or unequal to any value or to another null. However, Oracle Event Processing considers two nulls to be equal when evaluating a decode expression. See *decode::=* on page 5-13 for syntax and additional information.

Nulls in Conditions

A condition that evaluates to `UNKNOWN` acts almost like `FALSE`. For example, a `SELECT` statement with a condition in the `WHERE` clause that evaluates to `UNKNOWN` returns no tuples. However, a condition evaluating to `UNKNOWN` differs from `FALSE` in that further operations on an `UNKNOWN` condition evaluation will evaluate to `UNKNOWN`. Thus, `NOT FALSE` evaluates to `TRUE`, but `NOT UNKNOWN` evaluates to `UNKNOWN`.

Table 2–5 shows examples of various evaluations involving nulls in conditions. If the conditions evaluating to `UNKNOWN` were used in a `WHERE` clause of a `SELECT` statement, then no rows would be returned for that query.

Table 2–5 Conditions Containing Nulls

Condition	Value of A	Evaluation
`a IS NULL`	`10`	`FALSE`
`a IS NOT NULL`	`10`	`TRUE`
`a IS NULL`	`NULL`	`TRUE`
`a IS NOT NULL`	`NULL`	`FALSE`
`a = NULL`	`10`	`FALSE`
`a != NULL`	`10`	`FALSE`
`a = NULL`	`NULL`	`FALSE`
`a != NULL`	`NULL`	`FALSE`
`a = 10`	`NULL`	`FALSE`
`a != 10`	`NULL`	`FALSE`

For more information, see Section , "Null Conditions".

Comments

Oracle CQL does not support comments.

Aliases

Oracle CQL allows you to define aliases (or synonyms) to simplify and improve the clarity of your queries.

This section describes:

- Section , "Defining Aliases Using the AS Operator"
- Section , "Defining Aliases Using the Aliases Element"

Defining Aliases Using the AS Operator

Using the AS operator, you can specify an alias in Oracle CQL for queries, relations, streams, and any items in the SELECT list of a query.

This section describes:

- Section , "Aliases in the relation_variable Clause"
- Section , "Aliases in Window Operators"

For more information, see Chapter 20, "Oracle CQL Queries, Views, and Joins".

Aliases in the relation_variable Clause

You can use the relation_variable clause AS operator to define an alias to label the immediately preceding expression in the select list so that you can reference the result by that name. The alias effectively renames the select list item for the duration of the query. You can use an alias in the ORDER BY clause (see Section , "Sorting Query Results"), but not other clauses in the query.

Example 2–10 shows how to define alias badItem for a stream element its.itemId in a SELECT list and alias its for a MATCH_RECOGNIZE clause.

Example 2–10 Using the AS Operator in the SELECT Statement

```
<query id="detectPerish"><![CDATA[
  select its.itemId as badItem
  from tkrfid_ItemTempStream MATCH_RECOGNIZE (
      PARTITION BY itemId
      MEASURES A.itemId as itemId
      PATTERN (A B* C)
      DEFINE
          A  AS  (A.temp >= 25),
          B  AS  ((B.temp >= 25) and (to_timestamp(B.element_time) - to_timestamp(A.element_
time) < INTERVAL "0 00:00:05.00" DAY TO SECOND)),
          C  AS  (to_timestamp(C.element_time) - to_timestamp(A.element_time) >= INTERVAL "0
00:00:05.00" DAY TO SECOND)
  ) as its
]]></query>
```

For more information, see Section , "From Clause".

Aliases in Window Operators

You can use the AS operator to define an alias to label the immediately preceding window operator so that you can reference the result by that name.

You may not use the AS operator within a window operator but you may use the AS operator outside of the window operator.

Example 2–11 shows how to define aliases bid and ask after partitioned range window operators.

Example 2–11 Using the AS Operator After a Window Operator

```
<query id="Rule1"><![CDATA[
SELECT
    bid.id as correlationId
    bid.cusip as cusip
    max(bid.b0) as bid0
    bid.srcid as bidSrcId,
    bid.bq0 as bid0Qty,
    min(ask.a0) as ask0,
    ask.srcid as askSrcId,
    ask.aq0 as ask0Qty
FROM
    stream1[PARTITION by bid.cusip rows 100 range 4 hours] as bid,
    stream2[PARTITION by ask.cusip rows 100 range 4 hours] as ask
GROUP BY
    bid.id, bid.cusip, bid.srcid,bid.bq0, ask.srcid, ask.aq0
]]></query>
```

For more information, see Section , "Stream-to-Relation Operators (Windows)".

Defining Aliases Using the Aliases Element

Aliases are required to provide location transparency. Using the aliases element, you can define an alias and then use it in an Oracle CQL query or view. You configure the aliases element in the component configuration file of a processor as Figure 2–12 shows.

Example 2–12 aliases Element in a Processor Component Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<n1:config xmlns:n1="http://www.bea.com/ns/wlevs/config/application">
  <processor>
    <name>processor1</name>
    <rules>
      <query id="q1">
       <![CDATA[
         select str(msg) from cqlInStream [rows 2];
       ]]>
     </query>
    </rules>
    <aliases>
        <type-alias>
            <source>str</source>
            <target>java.lang.String </target>
       </type-alias>
    </aliases>
  </processor>
</n1:config>
```

The scope of the aliases element is the queries and views defined in the rules element of the processor to which the aliases element belongs.

Note the following:

- If the alias already exists then, Oracle Event Processing will throw an exception.

- If a query or view definition references an alias, then the alias must already exist.

This section describes:

- Section , "How to Define a Data Type Alias Using the Aliases Element"

How to Define a Data Type Alias Using the Aliases Element

Using the `aliases` element child element `type-alias`, you can define an alias for a data type. You can create an alias for any built-in or data cartridge data type.

For more information, see Section , "Datatypes".

To define a type alias using the aliases element:

1. Edit the component configuration file of a processor.
2. Add an aliases element as Example 2–13 shows.

Example 2–13 Adding an aliases Element to a Processor

```
<?xml version="1.0" encoding="UTF-8"?>
<n1:config xmlns:n1="http://www.bea.com/ns/wlevs/config/application">
  <processor>
    <name>processor1</name>
    <rules>
      <query id="q1">
       <![CDATA[
         select str(msg) from cqlInStream [rows 2];
       ]]>
     </query>
    </rules>
    <aliases>
    </aliases>
  </processor>
</n1:config>
```

3. Add a `type-alias` child element to the aliases element as Example 2–14 shows.

Example 2–14 Adding a type-alias Element to a Processor

```
<?xml version="1.0" encoding="UTF-8"?>
<n1:config xmlns:n1="http://www.bea.com/ns/wlevs/config/application">
  <processor>
    <name>processor1</name>
    <rules>
      <query id="q1">
       <![CDATA[
         select str(msg) from cqlInStream [rows 2];
       ]]>
     </query>
    </rules>
    <aliases>
        <type-alias>
       </type-alias>
    </aliases>
  </processor>
</n1:config>
```

4. Add a `source` and `target` child element to the `type-alias` element as Example 2–15 shows, where:

- `source` specifies the alias.

  You can use any valid schema name. For more information, see Section , "Schema Object Names and Qualifiers"

- `target` specifies the data type the alias refers to.

  For Oracle CQL data cartridge types, use the fully qualified type name. For more information, see Chapter 14, "Introduction to Data Cartridges".

Example 2–15 Adding the source and target Elements

```
<?xml version="1.0" encoding="UTF-8"?>
<n1:config xmlns:n1="http://www.bea.com/ns/wlevs/config/application">
  <processor>
    <name>processor1</name>
    <rules>
      <query id="q1">
       <![CDATA[
         select str(msg) from cqlInStream [rows 2];
       ]]>
     </query>
    </rules>
    <aliases>
        <type-alias>
            <source>str</source>
            <target>java.lang.String</target>
       </type-alias>
    </aliases>
  </processor>
</n1:config>
```

5. Use the alias in the queries and views you define for this processor.

   You can use the alias in exactly the same way you would use the data type it refers to. As Example 2–16 shows, you can access methods and fields of the aliased type.

Example 2–16 Accessing the Methods and Fields of an Aliased Type

```
<?xml version="1.0" encoding="UTF-8"?>
<n1:config xmlns:n1="http://www.bea.com/ns/wlevs/config/application">
  <processor>
    <name>processor1</name>
    <rules>
      <query id="q1">
       <![CDATA[
         select str(msg).length() from cqlInStream [rows 2];
       ]]>
     </query>
    </rules>
    <aliases>
        <type-alias>
            <source>str</source>
            <target>java.lang.String</target>
       </type-alias>
    </aliases>
  </processor>
</n1:config>
```

Schema Object Names and Qualifiers

Some schema objects are made up of parts that you can or must name, such as the stream elements in a stream or view, integrity constraints, streams, views, and user-defined functions. This section provides:

- Section , "Schema Object Naming Rules"
- Section , "Schema Object Naming Guidelines"
- Section , "Schema Object Naming Examples"

For more information, see Section , "Lexical Conventions".

Schema Object Naming Rules

Every Oracle Event Processing object has a name. In a Oracle CQL statement, you represent the name of an object with an **nonquoted identifier**, meaning an identifier that is not surrounded by any punctuation.

You must use nonquoted identifiers to name an Oracle Event Processing object.

The following list of rules applies to identifiers:

- Identifiers cannot be Oracle Event Processing reserved words.

  Depending on the Oracle product you plan to use to access an Oracle Event Processing object, names might be further restricted by other product-specific reserved words.

  The Oracle CQL language contains other words that have special meanings. These words are not reserved. However, Oracle uses them internally in specific ways. Therefore, if you use these words as names for objects and object parts, then your Oracle CQL statements may be more difficult to read and may lead to unpredictable results.

  For more information, see

  - *identifier::=* on page 7-17
  - "unreserved_keyword" on page 7-18
  - "reserved_keyword" on page 7-18
- Oracle recommends that you use ASCII characters in schema object names because ASCII characters provide optimal compatibility across different platforms and operating systems.
- Identifiers must begin with an alphabetic character (a letter) from your database character set.
- Identifiers can contain only alphanumeric characters from your Java locale's character set and the underscore (_). In particular, space, dot and slash are not permitted.

  For more information, see:

  - *const_string::=* on page 7-13
  - *identifier::=* on page 7-17
- In general, you should choose names that are unique across an application for the following objects:
  - Streams

   - Queries
   - Views
   - User-defined functions

  Specifically, a query and view cannot have the same name.

- Identifier names are case sensitive.
- Stream elements in the same stream or view cannot have the same name. However, stream elements in different streams or views can have the same name.
- Functions can have the same name, if their arguments are not of the same number and datatypes (that is, if they have distinct signatures). Creating multiple functions with the same name with different arguments is called **overloading** the function.

  If you register or create a user-defined function with the same name and signature as a built-in function, your function replaces that signature of the built-in function. Creating a function with the same name and signature as that of a built-in function is called **overriding** the function.

  Built-in functions are public where as user-defined functions belong to a particular schema.

  For more information, see:

   - Chapter 13, "User-Defined Functions"

Schema Object Naming Guidelines

Here are several guidelines for naming objects and their parts:

- Use full, descriptive, pronounceable names (or well-known abbreviations).
- Use consistent naming rules.
- Use the same name to describe the same entity or attribute across streams, views, and queries.

When naming objects, balance the goal of keeping names short and easy to use with the goal of making names as descriptive as possible. When in doubt, choose the more descriptive name, because the objects in Oracle Event Processing may be used by many people over a period of time. Your counterpart ten years from now may have difficulty understanding a stream element with a name like `pmdd` instead of `payment_due_date`.

Using consistent naming rules helps users understand the part that each stream plays in your application. One such rule might be to begin the names of all streams belonging to the `FINANCE` application with `fin_`.

Use the same names to describe the same things across streams. For example, the department number stream element of the `employees` and `departments` streams are both named `department_id`.

Schema Object Naming Examples

The following examples are valid schema object names:

```
last_name
horse
a_very_long_and_valid_name
```

All of these examples adhere to the rules listed in Section , "Schema Object Naming Rules".

3

Pseudocolumns

This chapter provides a reference for Oracle Continuous Query Language (Oracle CQL) pseudocolumns, which you can query for but which are not part of the data from which an event was created.

This chapter includes the following sections:

- Introduction to Pseudocolumns
- ELEMENT_TIME Pseudocolumn

Introduction to Pseudocolumns

You can select from pseudocolumns, but you cannot modify their values. A pseudocolumn is also similar to a function without arguments (see Section , "Functions").

Oracle CQL supports the following pseudocolumns:

- Section , "ELEMENT_TIME Pseudocolumn"

ELEMENT_TIME Pseudocolumn

Every stream element of a base stream or derived stream (a view that evaluates to a stream) has an associated element time.The `ELEMENT_TIME` pseudo column returns this time as an Oracle CQL native type `bigint`.

Note: `ELEMENT_TIME` is not supported on members of an Oracle CQL relation. For more information, see Section , "Streams and Relations".

This section describes:

- Section , "Understanding the Value of the ELEMENT_TIME Pseudocolumn"
- Section , "Using the ELEMENT_TIME Pseudocolumn in Oracle CQL Queries"

For more information, see:

- *pseudo_column::=* on page 7-5
- "to_timestamp" on page 8-21

Understanding the Value of the ELEMENT_TIME Pseudocolumn

The value of the `ELEMENT_TIME` pseudocolumn depends on whether or not you configure the stream element's channel as system- or application-timestamped.

ELEMENT_TIME for a System-Timestamped Stream

In this case, the element time for a stream element is assigned by the Oracle Event Processing system in such a way that subtracting two values of system-assigned time will give a duration that roughly matches the elapsed wall clock time.

For more information, see "System-Timestamped Channels" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

ELEMENT_TIME for an Application-Timestamped Stream

In this case, the associated element time is assigned by the application using the application assembly file `wlevs:expression` element to specify a derived timestamp expression.

Oracle Event Processing processes the result of this expression as follows:

- Section , "Dervied Timestamp Expression Evalutes to int or bigint"
- Section , "Dervied Timestamp Expression Evalutes to timestamp"

For more information, see "Application-Timestamped Channels" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

Dervied Timestamp Expression Evalutes to int or bigint If the dervied timestamp expression evaluates to an Oracle CQL native type of `int`, then it is cast to and returned as a corresponding `bigint` value. If the expression evaluates to an Oracle CQL native type of `bigint`, that value is returned as is.

Dervied Timestamp Expression Evalutes to timestamp If the derived timestamp expression evaluates to an Oracle CQL native type of `timestamp`, it is converted to a `long` value by expressing this time value as the number of milliseconds since the standard base time known as "the epoch", namely January 1, 1970, 00:00:00 GMT.

Using the ELEMENT_TIME Pseudocolumn in Oracle CQL Queries

This section describes how to use `ELEMENT_TIME` in various queries, including:

- Section , "Using ELEMENT_TIME With SELECT"
- Section , "Using ELEMENT_TIME With GROUP BY"
- Section , "Using ELEMENT_TIME With PATTERN"

Using ELEMENT_TIME With SELECT

Example 3–1 shows how you can use the `ELEMENT_TIME` pseudocolumn in a select statement. Stream `S1` has schema `(c1 integer)`. Given the input stream that Example 3–2 shows, this query returns the results that Example 3–3 shows. Note that the function `to_timestamp` is used to convert the `Long` values to timestamp values.

Example 3–1 ELEMENT_TIME Pseudocolumn in a Select Statement

```
<query id="q4"><![CDATA[
    select
        c1,
        to_timestamp(element_time)
    from
        S1[range 10000000 nanoseconds slide 10000000 nanoseconds]
]]></query>
```

Example 3–2 Input Stream

```
Timestamp   Tuple
 8000       80
 9000       90
13000       130
15000       150
23000       230
25000       250
```

Example 3–3 Output Relation

```
Timestamp   Tuple Kind  Tuple
 8000       +           80,12/31/1969 17:00:08
 8010       -           80,12/31/1969 17:00:08
 9000       +           90,12/31/1969 17:00:09
 9010       -           90,12/31/1969 17:00:09
13000       +           130,12/31/1969 17:00:13
13010       -           130,12/31/1969 17:00:13
15000       +           150,12/31/1969 17:00:15
15010       -           150,12/31/1969 17:00:15
23000       +           230,12/31/1969 17:00:23
23010       -           230,12/31/1969 17:00:23
25000       +           250,12/31/1969 17:00:25
25010       -           250,12/31/1969 17:00:25
```

If your query includes a `GROUP BY` clause, you cannot use the `ELEMENT_TIME` pseudocolumn in the `SELECT` statement directly. Instead, use a view as Section , "Using ELEMENT_TIME With GROUP BY" describes.

Using ELEMENT_TIME With GROUP BY

Consider query `Q1` that Example 3–4 shows. You cannot use `ELEMENT_TIME` in the `SELECT` statement of the query because of the `GROUP BY` clause.

Example 3–4 Query With GROUP BY

```
<query id="Q1"><![CDATA[
    SELECT
        R.queryText AS queryText,
        COUNT(*) AS queryCount
    FROM
        queryEventChannel [range 30 seconds] AS R
    GROUP BY
        queryText
]]></query>
```

Instead, create a view as Example 3–5 shows. The derived stream corresponding to `V1` will contain a stream element each time (`queryText`, `queryCount`, `maxTime`) changes for a specific `queryText` group.

Example 3–5 View

```
<view id="V1"><![CDATA[
    ISTREAM (
        SELECT
            R.queryText AS queryText,
            COUNT(*) AS queryCount,
            MAX(R.ELEMENT_TIME) as maxTime
        FROM
            queryEventChannel [range 30 seconds] AS R
        GROUP BY
            queryText
```

```
    )
]]></view>
```

Note that the element time associated with an output element of view V1 need not be the same as the value of the attribute maxTime for that output event. For example, as the window slides and an element from the queryEventChannel input stream expires from the window, the queryCount for that queryText group would change resulting in an output. However, since there was no new event from the input stream queryEventChannel entering the window, the maxTime among all events in the window has not changed, and the value of the maxTime attribute for this output event would be the same as the value of this attribute in the previous output event.

However, the ELEMENT_TIME of the output event corresponds to the instant where the event has expired from the window, which is different than the latest event from the input stream, making this is an example where ELEMENT_TIME of the output event is different from value of "maxTime" attribute of the output event.

To select the ELEMENT_TIME of the output events of view V1, create a query as Example 3–6 shows.

Example 3–6 Query

```
<query id="Q1"><![CDATA[
    SELECT
        queryText,
        queryCount,
        ELEMENT_TIME as eventTime
    FROM
        V1
]]></query>
```

Using ELEMENT_TIME With PATTERN

Example 3–7 shows how the `ELEMENT_TIME` pseudocolumn can be used in a pattern query. Here a tuple or event matches correlation variable `Nth` if the value of `Nth.status` is `>= F.status` and if the difference between the `Nth.ELEMENT_TIME` value of that tuple and the tuple that last matched `F` is less than the given interval as a `java.lang.Math.Bigint(Long)`.

Example 3–7 ELEMENT_TIME Pseudocolumn in a Pattern

```
...
PATTERN (F Nth+? L)
        DEFINE
            Nth AS
                Nth.status >= F.status
                AND
                Nth.ELEMENT_TIME - F.ELEMENT_TIME < 10000000000L,
            L AS
                L.status >= F.status
                AND
                count(Nth.*) = 3
                AND L.ELEMENT_TIME - F.ELEMENT_TIME < 10000000000L
...
```

4

Operators

This chapter provides a reference for operators in Oracle Continuous Query Language (Oracle CQL). An operator manipulates data items and returns a result. Syntactically, an operator appears before or after an operand or between two operands.

This chapter includes the following sections:

Introduction to Operators

Operators manipulate individual data items called **operands** or **arguments**. Operators are represented by special characters or by keywords. For example, the multiplication operator is represented by an asterisk (*).

Oracle CQL provides the following operators:

What You May Need to Know About Unary and Binary Operators

The two general classes of operators are:

- **unary**: A unary operator operates on only one operand. A unary operator typically appears with its operand in this format:

  ```
  operator operand
  ```

- **binary**: A binary operator operates on two operands. A binary operator appears with its operands in this format:

  ```
  operand1 operator operand2
  ```

Other operators with special formats accept more than two operands. If an operator is given a null operand, then the result is always null. The only operator that does not follow this rule is concatenation (||).

What You May Need to Know About Operator Precedence

Precedence is the order in which Oracle Event Processing evaluates different operators in the same expression. When evaluating an expression containing multiple operators, Oracle Event Processing evaluates operators with higher precedence before evaluating those with lower precedence. Oracle Event Processing evaluates operators with equal precedence from left to right within an expression.

Table 4–1 lists the levels of precedence among Oracle CQL operators from high to low. Operators listed on the same line have the same precedence.

Table 4–1 Oracle CQL Operator Precedence

Operator	Operation
`+`, `-` (as unary operators)	Identity, negation
`*`, `/`	Multiplication, division
`+`, `-` (as binary operators), `||`	Addition, subtraction, concatenation
Oracle CQL conditions are evaluated after Oracle CQL operators	See Chapter 6, "Conditions"

Precedence Example In the following expression, multiplication has a higher precedence than addition, so Oracle first multiplies 2 by 3 and then adds the result to 1.

```
1+2*3
```

You can use parentheses in an expression to override operator precedence. Oracle evaluates expressions inside parentheses before evaluating those outside.

Arithmetic Operators

Table 4–2 lists arithmetic operators that Oracle Event Processing supports. You can use an arithmetic operator with one or two arguments to negate, add, subtract, multiply, and divide numeric values. Some of these operators are also used in datetime and interval arithmetic. The arguments to the operator must resolve to numeric datatypes or to any datatype that can be implicitly converted to a numeric datatype.

In certain cases, Oracle Event Processing converts the arguments to the datatype as required by the operation. For example, when an integer and a float are added, the integer argument is converted to a float. The datatype of the resulting expression is a float. For more information, see "Implicit Datatype Conversion" on page 2-5.

Table 4–2 Arithmetic Operators

Operator	Purpose	Example

+ -	When these denote a positive or negative expression, they are unary operators.	<query id="q1"><![CDATA[
		    select * from orderitemsstream
		    where quantity = -1
		]]></query>

+ -	When they add or subtract, they are binary operators.	<query id="q1"><![CDATA[
		    select hire_date
		    from employees
		    where sysdate - hire_date
		  > 365
		]]></query>

* /	Multiply, divide. These are binary operators.	<query id="q1"><![CDATA[
		    select hire_date
		    from employees
		    where bonus > salary * 1.1
		]]></query>

Do not use two consecutive minus signs (--) in arithmetic expressions to indicate double negation or the subtraction of a negative value. You should separate consecutive minus signs with a space or parentheses.

Oracle Event Processing supports arithmetic operations using numeric literals and using datetime and interval literals.

For more information, see:

- "Numeric Literals" on page 2-8
- "Datetime Literals" on page 2-10
- "Interval Literals" on page 2-11

Concatenation Operator

The concatenation operator manipulates character strings. Table 4–3 describes the concatenation operator.

Table 4–3 Concatenation Operator

Operator	Purpose	Example
||	Concatenates character strings.	`<query id="q263"><![CDATA[ select length(c2 || c2) + 1 from S10 where length(c2) = 2 ]]></query>`

The result of concatenating two character strings is another character string. If both character strings are of datatype CHAR, then the result has datatype CHAR and is limited to 2000 characters. Trailing blanks in character strings are preserved by concatenation, regardless of the datatypes of the string.

Although Oracle Event Processing treats zero-length character strings as nulls, concatenating a zero-length character string with another operand always results in the other operand, so null can result only from the concatenation of two null strings. However, this may not continue to be true in future versions of Oracle Event Processing. To concatenate an expression that might be null, use the NVL function to explicitly convert the expression to a zero-length string.

See Also:

- Section , "Datatypes"
- "concat" on page 8-3
- "xmlconcat" on page 8-25
- "nvl" on page 8-8

Example 4–1 shows how to use the concatenation operator to append the String "xyz" to the value of c2 in a select statement.

Example 4–1 Concatenation Operator (||)

```
<query id="q264"><![CDATA[
    select c2 || "xyz" from S10
]]></query>
```

Alternation Operator

The alternation operator allows you to refine the sense of a PATTERN clause. Table 4–4 describes the concatenation operator.

Table 4–4 Alternation Operator

Operator	Purpose	Example

|	Changes the sense of a PATTERN clause to mean one or the other correlation variable rather than one followed by the other correlation variable.	`<query id="q263"><![CDATA[`
		`select T.p1, T.p2, T.p3 from S MATCH_`
		`RECOGNIZE(`
		`    MEASURES`
		`        A.ELEMENT_TIME as p1,`
		`        B.ELEMENT_TIME as p2`
		`        B.c2 as p3`
		`    PATTERN (A+ | B+)`
		`    DEFINE`
		`        A as A.c1 = 10,`
		`        B as B.c1 = 20`
		`) as T`
		`]]></query>`

The alternation operator is applicable only within a PATTERN clause.

Example 4–2 shows how to use the alternation operator to change the sense of the PATTERN clause to mean "A one or more times followed by either B one or more times or C one or more times, whichever comes first".

Example 4–2 Alternation Operator (|)

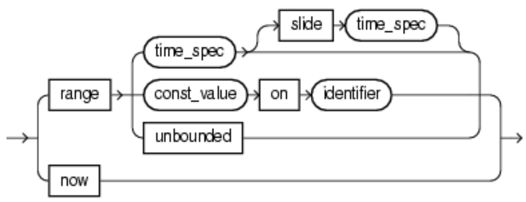
```
<query id="q264"><![CDATA[
select T.p1, T.p2, T.p3 from S MATCH_RECOGNIZE(
    MEASURES
        A.ELEMENT_TIME as p1,
        B.ELEMENT_TIME as p2
        B.c2 as p3
    PATTERN (A+ (B+ | C+))
    DEFINE
        A as A.c1 = 10,
        B as B.c1 = 20
        C as C.c1 = 30
) as T
]]></query>
```

For more information, see Section , "Grouping and Alternation in the PATTERN Clause".

Range-Based Stream-to-Relation Window Operators

Oracle CQL supports the following range-based stream-to-relation window operators:

***window_type_range*::=**



(*time_spec::=* on page 7-30)

- S[now]
- S[range T]
- S[range T1 slide T2]
- S[range unbounded]
- S[range C on E]

For more information, see:

- "Query" on page 22-2
- "Stream-to-Relation Operators (Windows)" on page 1-9
- Section , "Aliases in Window Operators"

S[now]

This time-based range window outputs an instantaneous relation. So at time `t` the output of this `now` window is all the tuples that arrive at that instant `t`. The smallest granularity of time in Oracle Event Processing is nanoseconds and hence all these tuples expire 1 nanosecond later.

For an example, see "S [now] Example" on page 4-7.

Examples

S [now] Example

Consider the query `q1` in Example 4–3 and the data stream `S` in Example 4–4. Timestamps are shown in nanoseconds (`1 sec = 10^9 nanoseconds`). Example 4–5 shows the relation that the query returns at time `5000 ms`. At time `5002 ms`, the query would return an empty relation.

Example 4–3 S [now] Query

```
<query id="q1"><![CDATA[
    SELECT * FROM S [now]
]]></query>
```

Example 4–4 S [now] Stream Input

```
Timestamp          Tuple
  1000000000       10,0.1
  1002000000       15,0.14
  5000000000       33,4.4
  5000000000       23,56.33
 10000000000       34,4.4
200000000000       20,0.2
209000000000       45,23.44
400000000000       30,0.3
h 800000000000
```

Example 4–5 S [now] Relation Output at Time 5000000000 ns

```
Timestamp   Tuple Kind  Tuple
5000000000  +           33,4.4
5000000000  +           23,56.33
5000000001  -           33,4.4
5000000001  -           23,56.33
```

S[range T]

This time-based range window defines its output relation over time by sliding an interval of size `T` time units capturing the latest portion of an ordered stream.

For an example, see "S [range T] Example" on page 4-8.

Examples

S [range T] Example

Consider the query `q1` in Example 4–6. Given the data stream `S` in Example 4–7, the query returns the relation in Example 4–8. By default, the range time unit is `second` (see *time_spec::=* on page 7-30) so `S[range 1]` is equivalent to `S[range 1 second]`. Timestamps are shown in milliseconds (`1 s = 1000 ms`). As many elements as there are in the first `1000 ms` interval enter the window, namely tuple `(10,0.1)`. At time `1002 ms`, tuple `(15,0.14)` enters the window. At time `2000 ms`, any tuples that have been in the window longer than the range interval are subject to deletion from the relation, namely tuple `(10,0.1)`. Tuple `(15,0.14)` is still in the relation at this time. At time `2002 ms`, tuple `(15,0.14)` is subject to deletion because by that time, it has been in the window longer than `1000 ms`.

> **Note:** In stream input examples, lines beginning with `h` (such as `h 3800`) are heartbeat input tuples. These inform Oracle Event Processing that no further input will have a timestamp lesser than the heartbeat value.

Example 4–6 S [range T] Query

```
<query id="q1"><![CDATA[
    SELECT * FROM S [range 1]
]]></query>
```

Example 4–7 S [range T] Stream Input

```
Timestamp   Tuple
     1000   10,0.1
     1002   15,0.14
   200000   20,0.2
   400000   30,0.3
h 800000
100000000   40,4.04
h 200000000
```

Example 4–8 S [range T] Relation Output

```
Timestamp   Tuple Kind  Tuple
     1000:  +           10,0.1
     1002:  +           15,0.14
     2000:  -           10,0.1
     2002:  -           15,0.14
   200000:  +           20,0.2
   201000:  -           20,0.2
   400000:  +           30,0.3
   401000:  -           30,0.3
100000000:  +           40,4.04
100001000:  -           40,4.04
```

S[range T1 slide T2]

This time-based range window allows you to specify the time duration in the past up to which you want to retain the tuples (range) and also how frequently you want to see the output of the tuples (slide).

Suppose a tuple arrives at a time represented by `t`. Assuming a slide value represented by `T2`, the tuple will be visible and sent to output at one of the following timestamps:

- `t` -- If the timestamp `t` is a multiple of slide `T2`
- `Math.ceil(t/T2)*T2` -- If the timestamp is not a multiple of slide `T2`

Assuming a range value represented by `T1`, a tuple that arrives at timestamp `t` will expire at timestamp `t + T1`. However, if a slide is specified and its value is non-zero, then the expired tuple will not necessarily output at timestamp `t + T1`.

The expired tuple (expired timestamp is `t + T1`) will be visible at one of the following timestamps:

- `(t + T1)` -- If the timestamp `(t+T1)` is a multiple of slide `T2`.
- `Math.ceil((t+T1)/T2)*T2` -- If the timestamp `(t+T1)` is not a multiple of slide `T2`.

For an example, see "S [range T1 slide T2] Example" on page 4-9.

Examples

S [range T1 slide T2] Example

Consider the query `q1` in Example 4–9. Given the data stream `S` in Example 4–10, the query returns the relation in Example 4–11. By default, the range time unit is `second` (see *time_spec::=* on page 7-30) so `S[range 10 slide 5]` is equivalent to `S[range 10 seconds slide 5 seconds]`. Timestamps are shown in milliseconds (`1 s = 1000 ms`). Tuples arriving at `1000`, `1002`, and `5000` all enter the window at time `5000` since the slide value is `5 sec` and that means the user is interested in looking at the output after every `5 sec`. Since these tuples enter at `5 sec=5000 ms`, they are expired at `15000 ms` as the range value is `10 sec = 10000 ms`.

Example 4–9 S [range T1 slide T2] Query

```
<query id="q1"><![CDATA[
    SELECT * FROM S [range 10 slide 5]
]]></query>
```

Example 4–10 S [range T1 slide T2] Stream Input

```
Timestamp   Tuple
  1000      10,0.1
  1002      15,0.14
  5000      33,4.4
  8000      23,56.33
 10000     34,4.4
200000      20,0.2
209000      45,23.44
400000      30,0.3
h 800000
```

Example 4–11 S [range T1 slide T2] Relation Output

```
Timestamp   Tuple Kind  Tuple
  5000:     +           10,0.1
```

```
     5000:     +           15,0.14
     5000:     +           33,4.4
    10000:     +           23,56.33
    10000:     +           34,4.4
    15000:     -           10,0.1
    15000:     -           15,0.14
    15000:     -           33,4.4
    20000:     -           23,56.33
    20000:     -           34,44.4
   200000:     +           20,0.2
   210000:     -           20,0.2
   210000:     +           45,23.44
   220000:     -           45,23.44
   400000:     +           30,0.3
   410000:     -           30,0.3
```

S[range unbounded]

This time-based range window defines its output relation such that, when `T = infinity`, the relation at time `t` consists of tuples obtained from all elements of `S` up to `t`. Elements remain in the window indefinitely.

For an example, see "S [range unbounded] Example" on page 4-11.

Examples

S [range unbounded] Example

Consider the query `q1` in Example 4–12 and the data stream `S` in Example 4–13. Timestamps are shown in milliseconds (`1 s = 1000 ms`). Elements are inserted into the relation as they arrive. No elements are subject to deletion. Example 4–14 shows the relation that the query returns at time `5000 ms` and Example 4–15 shows the relation that the query returns at time `205000 ms`.

Example 4–12 S [range unbounded] Query

```
<query id="q1"><![CDATA[
    SELECT * FROM S [range unbounded]
]]></query>
```

Example 4–13 S [range unbounded] Stream Input

```
Timestamp   Tuple
  1000      10,0.1
  1002      15,0.14
  5000      33,4.4
  8000      23,56.33
 10000      34,4.4
200000      20,0.2
209000      45,23.44
400000      30,0.3
h 800000
```

Example 4–14 S [range unbounded] Relation Output at Time 5000 ms

```
Timestamp   Tuple Kind  Tuple
  1000:     +           10,0.1
  1002:     +           15,0.14
  5000:     +           33,4.4
```

Example 4–15 S [range unbounded] Relation Output at Time 205000 ms

```
Timestamp   Tuple Kind  Tuple
  1000:     +           10,0.1
  1002:     +           15,0.14
  5000:     +           33,4.4
  8000:     +           23,56.33
 10000:     +           34,4.4
200000:     +           20,0.2
```

S[range C on E]

This constant value-based range window defines its output relation by capturing the latest portion of a stream that is ordered on the identifier `E` made up of tuples in which the values of stream element `E` differ by less than `C`. A tuple is subject to deletion when the difference between its stream element `E` value and that of any tuple in the relation is greater than or equal to `C`.

For examples, see:

- "S [range C on E] Example: Constant Value" on page 4-12
- "S [range C on E] Example: INTERVAL and TIMESTAMP" on page 4-13

Examples

S [range C on E] Example: Constant Value

Consider the query `tkdata56_q0` in Example 4–16 and the data stream `tkdata56_S0` in Example 4–17. Stream `tkdata56_S0` has schema `(c1 integer, c2 float)`. Example 4–18 shows the relation that the query returns. In this example, at time 200000, the output relation contains the following tuples: `(5,0.1)`, `(8,0.14)`, `(10,0.2)`. The difference between the `c1` value of each of these tuples is less than 10. At time 250000, when tuple `(15,0.2)` is added, tuple `(5,0.1)` is subject to deletion because the difference 15 - 5 = 10, which not less than 10. Tuple `(8,0.14)` remains because 15 - 8 = 7, which is less than 10. Likewise, tuple `(10,0.2)` remains because 15 - 10 = 5, which is less than 10. At time 300000, when tuple `(18,0.22)` is added, tuple `(8,0.14)` is subject to deletion because 18 - 8 = 10, which is not less than 10.

Example 4–16 S [range C on E] Constant Value: Query

```
<query id="tkdata56_q0"><![CDATA[
    select * from tkdata56_S0 [range 10 on c1]
]]></query>
```

Example 4–17 S [range C on E] Constant Value: Stream Input

```
Timestamp    Tuple
 100000      5, 0.1
 150000      8, 0.14
 200000      10, 0.2
 250000      15, 0.2
 300000      18, 0.22
 350000      20, 0.25
 400000      30, 0.3
 600000      40, 0.4
 650000      45, 0.5
 700000      50, 0.6
1000000      58, 4.04
```

Example 4–18 S [range C on E] Constant Value: Relation Output

```
Timestamp    Tuple Kind  Tuple
 100000:     +           5,0.1
 150000:     +           8,0.14
 200000:     +           10,0.2
 250000:     -           5,0.1
 250000:     +           15,0.2
 300000:     -           8,0.14
 300000:     +           18,0.22
 350000:     -           10,0.2
```

```
 350000:    +          20,0.25
 400000:    -          15,0.2
 400000:    -          18,0.22
 400000:    -          20,0.25
 400000:    +          30,0.3
 600000:    -          30,0.3
 600000:    +          40,0.4
 650000:    +          45,0.5
 700000:    -          40,0.4
 700000:    +          50,0.6
1000000:    -          45,0.5
1000000:    +          58,4.04
```

S [range C on E] Example: INTERVAL and TIMESTAMP

Similarly, you can use the `S[range C on ID]` window with `INTERVAL` and `TIMESTAMP`. Consider the query `tkdata56_q2` in Example 4–19 and the data stream `tkdata56_S1` in Example 4–20. Stream `tkdata56_S1` has schema `(c1 timestamp, c2 double)`. Example 4–21 shows the relation that the query returns.

Example 4–19 S [range C on E] INTERVAL Value: Query

```
<query id="tkdata56_q2"><![CDATA[
    select * from tkdata56_S1 [range INTERVAL "530 0:0:0.0" DAY TO SECOND on c1]
]]></query>
```

Example 4–20 S [range C on E] INTERVAL Value: Stream Input

```
Timestamp   Tuple
  10        "08/07/2004 11:13:48", 11.13
2000        "08/07/2005 12:13:48", 12.15
3400        "08/07/2006 10:15:58", 22.25
4700        "08/07/2007 10:10:08", 32.35
```

Example 4–21 S [range C on E] INTERVAL Value: Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           08/07/2004 11:13:48,11.13
2000:       +           08/07/2005 12:13:48,12.15
3400:       -           08/07/2004 11:13:48,11.13
3400:       +           08/07/2006 10:15:58,22.25
4700:       -           08/07/2005 12:13:48,12.15
4700:       +           08/07/2007 10:10:08,32.35
```

Tuple-Based Stream-to-Relation Window Operators

Oracle CQL supports the following tuple-based stream-to-relation window operators:

***window_type_tuple*::=**

- S [rows N]
- S [rows N1 slide N2]

For more information, see:

- "Range-Based Stream-to-Relation Window Operators" on page 4-6
- "Query" on page 22-2
- "Stream-to-Relation Operators (Windows)" on page 1-9
- Section , "Aliases in Window Operators"

S [rows N]

A tuple-based window defines its output relation over time by sliding a window of the last N tuples of an ordered stream.

For the output relation R of S [rows N], the relation at time t consists of the N tuples of S with the largest timestamps <= t (or all tuples if the length of S up to t is <= N).

If more than one tuple has the same timestamp, Oracle Event Processing chooses one tuple in a non-deterministic way to ensure N tuples are returned. For this reason, tuple-based windows may not be appropriate for streams in which timestamps are not unique.

By default, the slide is 1.

For examples, see "S [rows N] Example" on page 4-15.

Examples

S [rows N] Example

Consider the query q1 in Example 4–22 and the data stream S in Example 4–23. Timestamps are shown in milliseconds (1 s = 1000 ms). Elements are inserted into and deleted from the relation as in the case of S [Range 1] (see "S [range T] Example" on page 4-8).

Example 4–24 shows the relation that the query returns at time 1002 ms. Since the length of S at this point is less than or equal to the rows value (3), the query returns all the tuples of S inserted by that time, namely tuples (10,0.1) and (15,0.14).

Example 4–25 shows the relation that the query returns at time 1006 ms. Since the length of S at this point is greater than the rows value (3), the query returns the 3 tuples of S with the largest timestamps less than or equal to 1006 ms, namely tuples (15,0.14), (33,4.4), and (23,56.33).

Example 4–26 shows the relation that the query returns at time 2000 ms. At this time, the query returns the 3 tuples of S with the largest timestamps less than or equal to 2000 ms, namely tuples (45,23.44), (30,0.3), and (17,1.3).

Example 4–22 S [rows N] Query

```
<query id="q1"><![CDATA[
    SELECT * FROM S [rows 3]
]]></query>
```

Example 4–23 S [rows N] Stream Input

```
Timestamp    Tuple
1000         10,0.1
1002         15,0.14
1004         33,4.4
1006         23,56.33
1008         34,4.4
1010         20,0.2
1012         45,23.44
1014         30,0.3
2000         17,1.3
```

Example 4–24 S [rows N] Relation Output at Time 1003 ms

```
Timestamp   Tuple Kind  Tuple
  1000:         +       10,0.1
```

```
  1002:         +       15,0.14
```

Example 4–25   S [rows N] Relation Output at Time 1007 ms

```
Timestamp   Tuple Kind  Tuple
  1000:         +       10,0.1
  1002:         +       15,0.14
  1004:         +       33,4.4
  1006:         -       10,0.1
  1006:         +       23,56.33
```

Example 4–26   S [rows N] Relation Output at Time 2001 ms

```
Timestamp   Tuple Kind  Tuple
  1000          +       10,0.1
  1002          +       15,0.14
  1004          +       33,4.4
  1006          -       10,0.1
  1006          +       23,56.33
  1008          -       15,0.14
  1008          +       34,4.4
  1008          -       33,4.4
  1010          +       20,0.2
  1010          -       23,56.33
  1012          +       45,23.44
  1012          -       34,4.4
  1014          +       30,0.3
  2000          -       20,0.2
  2000          +       17,1.3
```

S [rows N1 slide N2]

A tuple-based window that defines its output relation over time by sliding a window of the last N1 tuples of an ordered stream.

For the output relation R of S [rows N1 slide N2], the relation at time t consists of the N1 tuples of S with the largest timestamps <= t (or all tuples if the length of S up to t is <= N).

If more than one tuple has the same timestamp, Oracle Event Processing chooses one tuple in a non-deterministic way to ensure N tuples are returned. For this reason, tuple-based windows may not be appropriate for streams in which timestamps are not unique.

You can configure the slide N2 as an integer number of stream elements. Oracle Event Processing delays adding stream elements to the relation until it receives N2 number of elements.

For examples, see "S [rows N] Example" on page 4-15.

Examples

S [rows N1 slide N2] Example

Consider the query tkdata55_q0 in Example 4–27 and the data stream tkdata55_S55 in Example 4–28. Stream tkdata55_S55 has schema (c1 integer, c2 float). The output relation is shown in Example 4–29.

As Example 4–29 shows, at time 100000, the output relation is empty because only one tuple (20,0.1) has arrived on the stream. By time 150000, the number of tuples that the slide value specifies (2) have arrived: at that time, the output relation contains tuples (20,0.1) and (15,0.14). By time 250000, another slide number of tuples have arrived and the output relation contains tuples (20,0.1), (15,0.14), (5,0.2), and (8,0.2). By time 350000, another slide number of tuples have arrived. At this time, the oldest tuple (20,0.1) is subject to deletion to meet the constraint that the rows value imposes: namely, that the output relation contain no more than 5 elements. At this time, the output relation contains tuples (15,0.14), (5,0.2), (8,0.2), (10,0.22), and (20,0.25). At time 600000, another slide number of tuples have arrived. At this time, the oldest tuples (15,0.14) and (5,0.2) are subject to deletion to observe the rows value constraint. At this time, the output relation contains tuples (8,0.2), (10,0.22), (20,0.25), (30,0.3), and (40,0.4).

Example 4–27 S [rows N1 slide N2] Query

```
<query id="tkdata55_q0"><![CDATA[
    select * from tkdata55_S55 [rows 5 slide 2 ]
]]></query>
```

Example 4–28 S [rows N1 slide N2] Stream Input

```
Timestamp   Tuple
   100000   20, 0.1
   150000   15, 0.14
   200000    5, 0.2
   250000    8, 0.2
   300000   10, 0.22
   350000   20, 0.25
   400000   30, 0.3
   600000   40, 0.4
   650000   45, 0.5
```

```
   700000   50, 0.6
100000000    8, 4.04
```

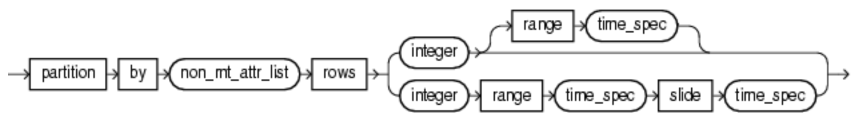
Example 4–29 S [rows N1 slide N2] Relation Output

```
Timestamp   Tuple Kind  Tuple
150000:     +           20,0.1
150000:     +           15,0.14
250000:     +            5,0.2
250000:     +            8,0.2
350000:     -           20,0.1
350000:     +           10,0.22
350000:     +           20,0.25
600000:     -           15,0.14
600000:     -            5,0.2
600000:     +           30,0.3
600000:     +           40,0.4
700000:     -            8,0.2
700000:     -           10,0.22
700000:     +           45,0.5
700000:     +           50,0.6
```

Partitioned Stream-to-Relation Window Operators

Oracle CQL supports the following partitioned stream-to-relation window operators:

***window_type_partition*::=**



(*time_spec::=* on page 7-30, *non_mt_attr_list::=* on page 7-22)

- S [partition by A1,..., Ak rows N]
- S [partition by A1,..., Ak rows N range T]
- S [partition by A1,..., Ak rows N range T1 slide T2]

For more information, see:

- "Tuple-Based Stream-to-Relation Window Operators" on page 4-14
- "Query" on page 22-2
- "Stream-to-Relation Operators (Windows)" on page 1-9
- Section , "Aliases in Window Operators"

S [partition by A1,..., Ak rows N]

This partitioned sliding window on a stream `S` takes a positive integer number of tuples `N` and a subset `{A1,... Ak}` of the stream's attributes as parameters and:

- Logically partitions `S` into different substreams based on equality of attributes `A1,... Ak` (similar to SQL `GROUP BY`).
- Computes a tuple-based sliding window of size `N` independently on each substream.

For an example, see "S[partition by A1, ..., Ak rows N] Example" on page 4-20.

Examples

S[partition by A1, ..., Ak rows N] Example

Consider the query `qPart_row2` in Example 4–30 and the data stream `SP1` in Example 4–31. Stream `SP1` has schema `(c1 integer, name char(10))`. The query returns the relation in Example 4–32. By default, the range (and slide) is 1 second. Timestamps are shown in milliseconds (`1 s = 1000 ms`).

Note: In stream input examples, lines beginning with `h` (such as `h 3800`) are heartbeat input tuples. These inform Oracle Event Processing that no further input will have a timestamp lesser than the heartbeat value.

Example 4–30 S[partition by A1, ..., Ak rows N] Query

```
<query id="qPart_row2"><![CDATA[
    select * from SP1 [partition by c1 rows 2]
]]></query>
```

Example 4–31 S[partition by A1, ..., Ak rows N] Stream Input

```
Timestamp   Tuple
1000        1,abc
1100        2,abc
1200        3,abc
2000        1,def
2100        2,def
2200        3,def
3000        1,ghi
3100        2,ghi
3200        3,ghi
h 3800
4000        1,jkl
4100        2,jkl
4200        3,jkl
5000        1,mno
5100        2,mno
5200        3,mno
h 12000
h 200000000
```

Example 4–32 S[partition by A1, ..., Ak rows N] Relation Output

```
Timestamp   Tuple Kind  Tuple
1000:       +           1,abc
1100:       +           2,abc
1200:       +           3,abc
```

```
2000:       +           1,def
2100:       +           2,def
2200:       +           3,def
3000:       -           1,abc
3000:       +           1,ghi
3100:       -           2,abc
3100:       +           2,ghi
3200:       -           3,abc
3200:       +           3,ghi
4000:       -           1,def
4000:       +           1,jkl
4100:       -           2,def
4100:       +           2,jkl
4200:       -           3,def
4200:       +           3,jkl
5000:       -           1,ghi
5000:       +           1,mno
5100:       -           2,ghi
5100:       +           2,mno
5200:       -           3,ghi
5200:       +           3,mno
```

S [partition by A1,..., Ak rows N range T]

This partitioned sliding window on a stream `S` takes a positive integer number of tuples `N` and a subset `{A1,... Ak}` of the stream's attributes as parameters and:

- Logically partitions `S` into different substreams based on equality of attributes `A1,... Ak` (similar to SQL `GROUP BY`).
- Computes a tuple-based sliding window of size `N` and range `T` independently on each substream.

For an example, see "S[partition by A1, ..., Ak rows N range T] Example" on page 4-22.

Examples

S[partition by A1, ..., Ak rows N range T] Example

Consider the query `qPart_range2` in Example 4–33 and the data stream `SP5` in Example 4–34. Stream `SP5` has schema `(c1 integer, name char(10))`. The query returns the relation in Example 4–35. By default, the range time unit is `second` (see *time_spec::=* on page 7-30) so `range 2` is equivalent to `range 2 seconds`. Timestamps are shown in milliseconds (`1 s = 1000 ms`).

Example 4–33 S[partition by A1, ..., Ak rows N range T] Query

```
<query id="qPart_range2"><![CDATA[
    select * from SP5 [partition by c1 rows 2 range 2]
]]></query>
```

Example 4–34 S[partition by A1, ..., Ak rows N range T] Stream Input

```
Timestamp   Tuple
1000        1,abc
2000        1,abc
3000        1,abc
4000        1,abc
5000        1,def
6000        1,xxx
h 200000000
```

Example 4–35 S[partition by A1, ..., Ak rows N range T] Relation Output

```
Timestamp   Tuple Kind  Tuple
1000:       +           1,abc
2000:       +           1,abc
3000:       -           1,abc
3000:       +           1,abc
4000:       -           1,abc
4000:       +           1,abc
5000:       -           1,abc
5000:       +           1,def
6000:       -           1,abc
6000:       +           1,xxx
7000:       -           1,def
8000:       -           1,xxx
```

S [partition by A1,..., Ak rows N range T1 slide T2]

This partitioned sliding window on a stream `S` takes a positive integer number of tuples `N` and a subset `{A1,... Ak}` of the stream's attributes as parameters and:

- Logically partitions `S` into different substreams based on equality of attributes `A1,... Ak` (similar to SQL `GROUP BY`).
- Computes a tuple-based sliding window of size `N`, range `T1`, and slide `T2` independently on each substream.

For an example, see "S[partition by A1, ..., Ak rows N] Example" on page 4-20.

Examples

S[partition by A1, ..., Ak rows N range T1 slide T2] Example

Consider the query `qPart_rangeslide` in Example 4–36 and the data stream `SP1` in Example 4–37. Stream `SP1` has schema `(c1 integer, name char(10))`. The query returns the relation in Example 4–38. By default, the range and slide time unit is `second` (see *time_spec::=* on page 7-30) so `range 1 slide 1` is equivalent to `range 1 second slide 1 second`. Timestamps are shown in milliseconds (`1 s = 1000 ms`).

Example 4–36 S[partition by A1, ..., Ak rows N range T1 slide T2] Query

```
<query id="qPart_rangeslide"><![CDATA[
    select * from SP1 [partition by c1 rows 1 range 1 slide 1]
]]></query>
```

Example 4–37 S[partition by A1, ..., Ak rows N range T1 slide T2] Stream Input

```
Timestamp   Tuple
1000        1,abc
1100        2,abc
1200        3,abc
2000        1,def
2100        2,def
2200        3,def
3000        1,ghi
3100        2,ghi
3200        3,ghi
h 3800
4000        1,jkl
4100        2,jkl
4200        3,jkl
5000        1,mno
5100        2,mno
5200        3,mno
h 12000
h 200000000
```

Example 4–38 S[partition by A1, ..., Ak rows N range T1 slide T2] Relation Output

```
Timestamp   Tuple Kind  Tuple
1000:       +           1,abc
2000:       +           2,abc
2000:       +           3,abc
2000:       -           1,abc
2000:       +           1,def
3000:       -           2,abc
3000:       +           2,def
3000:       -           3,abc
3000:       +           3,def
```

```
3000:       -           1,def
3000:       +           1,ghi
4000:       -           2,def
4000:       +           2,ghi
4000:       -           3,def
4000:       +           3,ghi
4000:       -           1,ghi
4000:       +           1,jkl
5000:       -           2,ghi
5000:       +           2,jkl
5000:       -           3,ghi
5000:       +           3,jkl
5000:       -           1,jkl
5000:       +           1,mno
6000:       -           2,jkl
6000:       +           2,mno
6000:       -           3,jkl
6000:       +           3,mno
6000:       -           1,mno
7000:       -           2,mno
7000:       -           3,mno
```

IStream Relation-to-Stream Operator

`Istream` (for "Insert stream") applied to a relation `R` contains `(s,t)` whenever tuple `s` is in `R(t) - R(t-1)`, that is, whenever `s` is inserted into `R` at time `t`. If a tuple happens to be both inserted and deleted with the same timestamp then `IStream` does not output the insertion.

In Example 4–39, the select will output a stream of tuples satisfying the filter condition `(viewq3.ACCT_INTRL_ID = ValidLoopCashForeignTxn.ACCT_INTRL_ID)`. The `now` window converts the `viewq3` into a relation, which is kept as a relation by the filter condition. The `IStream` relation-to-stream operator converts the output of the filter back into a stream.

Example 4–39 IStream

```
<query id="q3Txns"><![CDATA[
    Istream(
        select
            TxnId,
            ValidLoopCashForeignTxn.ACCT_INTRL_ID,
            TRXN_BASE_AM,
            ADDR_CNTRY_CD,
            TRXN_LOC_ADDR_SEQ_ID
        from
            viewq3[NOW],
            ValidLoopCashForeignTxn
        where
            viewq3.ACCT_INTRL_ID = ValidLoopCashForeignTxn.ACCT_INTRL_ID
    )
]]></query>
```

You can combine the `Istream` operator with a `DIFFERENCES USING` clause to succinctly detect differences in the `Istream`.

For more information, see:

- *idstream_clause::=* on page 22-6
- Section , "Detecting Differences in Query Results"

DStream Relation-to-Stream Operator

Dstream (for "Delete stream") applied to a relation R contains (s,t) whenever tuple s is in R(t-1) - R(t), that is, whenever s is deleted from R at time t.

In Example 4–40, the query delays the input on stream S by 10 minutes. The range window operator converts the stream S into a relation, whereas the Dstream converts it back to a stream.

Example 4–40 DStream

```
<query id="BBAQuery"><![CDATA[
    Dstream(select * from S[range 10 minutes])
]]></query>
```

Assume that the granularity of time is minutes. Table 4–5 illustrates the contents of the range window operator's relation (S[Range 10 minutes]) and the Dstream stream for the following input stream TradeInputs:

```
Time    Value
05      1,1
25      2,2
50      3,3
```

Table 4–5 DStream Example Output

Input Stream S	Relation Output	Relation Contents	DStream Output
05 1,1	+ 05 1,1	{1, 1}	
	- 15 1,1	{}	+15 1,1
25 2,2	+ 25 2,2	{2,2}	
	- 35 2,2	{}	+35 2,2
50 3,3	+ 50 3,3	{3,3}	
	- 60 3,3	{}	+60 3,3

Note that at time 15, 35, and 60, the relation is empty {} (the empty set).

You can combine the Dstream operator with a DIFFERENCES USING clause to succinctly detect differences in the Dstream.

For more information, see:

- *idstream_clause::=* on page 22-6
- Section , "Detecting Differences in Query Results"

RStream Relation-to-Stream Operator

The `Rstream` operator maintains the entire current state of its input relation and outputs all of the tuples as insertions at each time step.

Since `Rstream` outputs the entire state of the relation at every instant of time, it can be expensive if the relation set is not very small.

In Example 4–41, `Rstream` outputs the entire state of the relation at time `Now` and filtered by the `where` clause.

Example 4–41 RStream

```
<query id="rstreamQuery"><![CDATA[
    Rstream(
        select
            cars.car_id, SegToll.toll
        from
            CarSegEntryStr[now] as cars, SegToll
        where (cars.exp_way = SegToll.exp_way and
               cars.lane = SegToll.lane and
               cars.dir = SegToll.dir and
               cars.seg = SegToll.seg)
    )
]]></query>
```

For more information, see *idstream_clause::=* on page 22-6.

5

Expressions

This chapter provides a reference to expressions in Oracle Continuous Query Language (Oracle CQL). An expression is a combination of one or more values and one or more operations, including a constant having a definite value, a function that evaluates to a value, or an attribute containing a value.

Every expression maps to a datatype. This simple expression evaluates to 4 and has datatype `NUMBER` (the same datatype as its components):

```
2*2
```

The following expression is an example of a more complex expression that uses both functions and operators. The expression adds seven days to the current date, removes the time component from the sum, and converts the result to `CHAR` datatype:

```
TO_CHAR(TRUNC(SYSDATE+7))
```

This chapter includes the following section:

Introduction to Expressions

Oracle Event Processing provides the following expressions:

- Aggregate distinct expressions: "aggr_distinct_expr" on page 5-3.
- Aggregate expressions: "aggr_expr" on page 5-4.
- Arithmetic expressions: "arith_expr" on page 5-6.
- Arithmetic expression list: "arith_expr_list" on page 5-8.
- Case expressions: "case_expr" on page 5-9.
- Decode expressions: "decode" on page 5-13.
- Function expressions: "func_expr" on page 5-15.
- Object expressions: "object_expr" on page 5-19
- Order expressions: "order_expr" on page 5-23.
- XML aggregate expressions: "xml_agg_expr" on page 5-24
- XML column attribute value expressions: "xmlcolattval_expr" on page 5-26.
- XML element expressions: "xmlelement_expr" on page 5-28.
- XML forest expressions: "xmlforest_expr" on page 5-30.
- XML parse expressions: "xml_parse_expr" on page 5-32

You can use expressions in:

- The select list of the `SELECT` statement
- A condition of the `WHERE` clause and `HAVING` clause

Oracle Event Processing does not accept all forms of expressions in all parts of all Oracle CQL statements. Refer to the individual Oracle CQL statements in Chapter 22, "Oracle CQL Statements" for information on restrictions on the expressions in that statement.

You must use appropriate expression notation whenever *`expr`* appears in conditions, Oracle CQL functions, or Oracle CQL statements in other parts of this reference. The sections that follow describe and provide examples of the various forms of expressions.

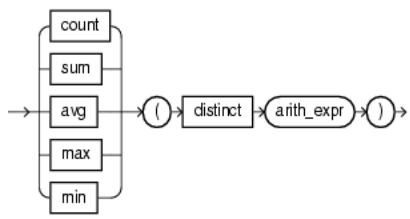
Note: In stream input examples, lines beginning with `h` (such as `h 3800`) are heartbeat input tuples. These inform Oracle Event Processing that no further input will have a timestamp lesser than the heartbeat value.

---

aggr_distinct_expr

Use an *aggr_distinct_expr* aggregate expression when you want to use an aggregate built-in function with distinct. When you want to use an aggregate built-in function without distinct, see "aggr_expr" on page 5-4.

aggr_distinct_expr::=



(*arith_expr::=* on page 5-6)

You can specify an *arith_distinct_expr* as the argument of an aggregate expression.

You can use an *aggr_distinct_expr* in the following Oracle CQL statements:

- *arith_expr::=* on page 5-6

For more information, see Chapter 9, "Built-In Aggregate Functions".

Examples

Example 5–2 shows how to use a COUNT aggregate distinct expression.

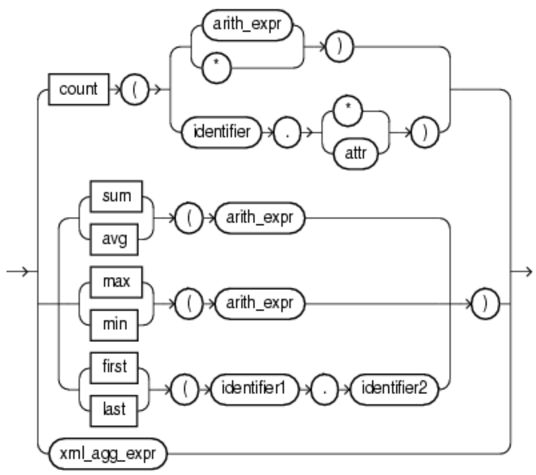
Example 5–1 aggr_distinct_expr for COUNT

```
create view viewq2Cond1(ACCT_INTRL_ID, sumForeign, countForeign) as
        select ACCT_INTRL_ID, sum(TRXN_BASE_AM), count(distinct ADDR_CNTRY_CD)
        from ValidCashForeignTxn[range 48 hours]
        group by ACCT_INTRL_ID
        having ((sum(TRXN_BASE_AM) * 100) >= (1000 * 60) and
                (count(distinct ADDR_CNTRY_CD >= 2)))
```

aggr_expr

Use an *aggr_expr* aggregate expression when you want to use aggregate built-in functions. When you want to use an aggregate built-in function with `distinct`, see "aggr_distinct_expr" on page 5-3

aggr_expr::=



(*arith_expr::=* on page 5-6, *identifier::=* on page 7-17, *attr::=* on page 7-5, *xml_agg_expr::=* on page 5-24)

You can specify an *arith_expr* as the argument of an aggregate expression.

The `count` aggregate built-in function takes a single argument made up of any of the values that Table 5–1 lists and returns the `int` value indicated.

Table 5–1 Return Values for COUNT Aggregate Function

Input Argument	Return Value
arith_expr	The number of tuples where *arith_expr* is not null.
*	The number of all tuples, including duplicates and nulls.
*identifier.**	The number of all tuples that match the correlation variable *identifier*, including duplicates and nulls.
identifier.attr	The number of tuples that match correlation variable *identifier*, where *attr* is not null.

The `first` and `last` aggregate built-in functions take a single argument made up of the following period separated values:

- `identifier1`: the name of a pattern as specified in a `DEFINE` clause.
- `identifier2`: the name of a stream element as specified in a `CREATE STREAM` statement.

You can use an *aggr_expr* in the following Oracle CQL statements:

- *arith_expr::=* on page 5-6

For more information, see:

- Chapter 9, "Built-In Aggregate Functions"

- Section , "Using count With *, identifier.*, and identifier.attr"
- "first" on page 9-7
- "last" on page 9-9

Examples

Example 5–2 shows how to use a COUNT aggregate expression.

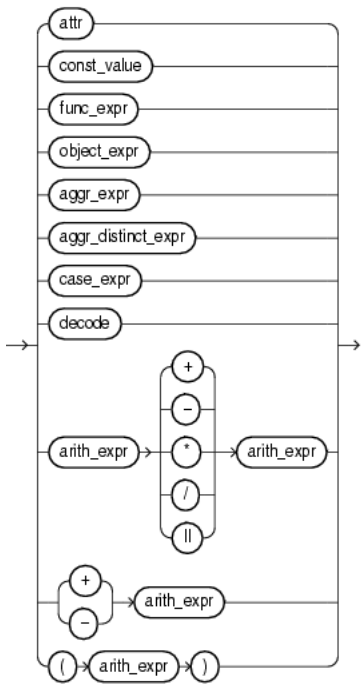
Example 5–2 aggr_expr for COUNT

```
<view id="SegVol" schema="exp_way lane dir seg volume"><![CDATA[
    select
        exp_way,
        lane,
        dir,
        seg,
        count(*) as volume
    from
        CurCarSeg
    group by
        exp_way,
        lane,
        dir,
        seg
    having
        count(*) > 50
]]></view>
```

arith_expr

Use an `arith_expr` arithmetic expression to define an arithmetic expression using any combination of stream element attribute values, constant values, the results of a function expression, aggregate built-in function, case expression, or decode. You can use all of the normal arithmetic operators (`+`,`-`,`*`, and `/`) and the concatenate operator (||).

arith_expr::=



(*attr::=* on page 7-5, *const_value::=* on page 7-14, *func_expr::=* on page 5-15, *aggr_expr::=* on page 5-4, *aggr_distinct_expr::=* on page 5-3, *case_expr::=* on page 5-9, *decode::=* on page 5-13, *arith_expr::=* on page 5-6)

You can use an `arith_expr` in the following Oracle CQL statements:

- *aggr_distinct_expr::=* on page 5-3
- *aggr_expr::=* on page 5-4
- *arith_expr::=* on page 5-6
- *case_expr::=* on page 5-9
- *searched_case::=* on page 5-9
- *simple_case::=* on page 5-9
- *condition::=* on page 6-4
- *between_condition::=* on page 6-8
- *non_mt_arg_list::=* on page 7-21
- *param_list* on page 7-26
- *measure_column::=* on page 21-10

- *projterm::=* on page 22-3

For more information, see "Arithmetic Operators" on page 4-3.

Examples

Example 5–3 shows how to use a `arith_expr` expression.

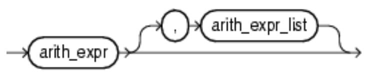
Example 5–3 arith_expr

```
<view id="SegVol" schema="exp_way lane dir seg volume"><![CDATA[
    select
        exp_way,
        speed * 1.5 as adjustedSpeed,
        dir,
        max(seg) as maxSeg,
        count(*) as volume
    from
        CurCarSeg
    having
        adjustedSpeed > 50
]]></view>
```

arith_expr_list

Use an `arith_expr_list` arithmetic expression list to define one or more arithmetic expressions using any combination of stream element attribute values, constant values, the results of a function expression, aggregate built-in function, case expression, or decode. You can use all of the normal arithmetic operators (+,-,*, and /) and the concatenate operator (||).

arith_expr_list::=



(*arith_expr::=* on page 5-6)

You can use an `arith_expr_list` in the following Oracle CQL statements:

- *xmlelement_expr::=* on page 5-28

For more information, see "Arithmetic Operators" on page 4-3.

Examples

Example 5–4 shows how to use a `arith_expr_list` expression.

Example 5–4 arith_expr_list

```
<query id="q1"><![CDATA[
    select
        XMLELEMENT("Emp", XMLELEMENT("Name", e.job_id||' '||e.last_name),
            XMLELEMENT("Hiredate", e.hire_date)
        )
    from
        tkdata51_S0 [range 1] as e
]]></query>
```

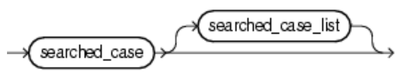
case_expr

Use a *case_expr* case expression to evaluate stream elements against multiple conditions.

case_expr::=

(*searched_case_list::=* on page 5-9, *arith_expr::=* on page 5-6, *simple_case_list::=* on page 5-9)

searched_case_list::=

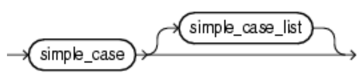


(*searched_case::=* on page 5-9)

searched_case::=

(*non_mt_cond_list::=* on page 7-25, *arith_expr::=* on page 5-6)

simple_case_list::=



(*simple_case::=* on page 5-9)

simple_case::=

(*arith_expr::=* on page 5-6)

The `case_expr` is similar to the `DECODE` clause of an arithmetic expression (see "decode" on page 5-13).

In a `searched_case` clause, when the `non_mt_cond_list` evaluates to true, the `searched_case` clause may return either an arithmetic expression or null.

In a `simple_case` clause, when the arithmetic expression is true, the `simple_case` clause may return either another arithmetic expression or null.

You can use an `case_expr` in the following Oracle CQL statements:

- *arith_expr::=* on page 5-6
- *opt_where_clause::=* on page 22-4
- *select_clause::=* on page 22-3

Examples

This section describes the following case_expr examples:

- "case_expr with SELECT *" on page 5-10
- "case_expr with SELECT" on page 5-10

case_expr with SELECT *

Consider the query q97 in Example 5–5 and the data stream S0 in Example 5–6. Stream S1 has schema (c1 float, c2 integer). The query returns the relation in Example 5–7.

Example 5–5 CASE Expression: SELECT * Query

```
<query id="q97"><![CDATA[
    select * from S0
    where
        case
            when c2 < 25 then c2+5
            when c2 > 25 then c2+10
        end > 25
]]></query>
```

Example 5–6 CASE Expression: SELECT * Stream Input

```
Timestamp   Tuple
     1000    0.1,10
     1002   0.14,15
   200000    0.2,20
   400000    0.3,30
   500000    0.3,35
   600000       ,35
h 800000
100000000   4.04,40
h 200000000
```

Example 5–7 CASE Expression: SELECT * Relation Output

```
Timestamp   Tuple Kind  Tuple
400000:+ 0.3,30
500000:+ 0.3,35
600000:+ ,35
100000000:+ 4.04,40
```

case_expr with SELECT

Consider the query q96 in Example 5–8 and the data streams S0 in Example 5–9 and S1 in Example 5–10. Stream S0 has schema (c1 float, c2 integer) and stream S1 has schema (c1 float, c2 integer). The query returns the relation in Example 5–11.

Example 5–8 CASE Expression: SELECT Query

```
<query id="q96"><![CDATA[
    select
        case to_float(S0.c2+10)
            when (S1.c2*100)+10 then S0.c1+0.5
            when (S1.c2*100)+11 then S0.c1
            else S0.c1+0.3
        end
    from
        S0[rows 100],
        S1[rows 100]
]]></query>
```

Example 5–9 CASE Expression: SELECT Stream S0 Input

```
Timestamp   Tuple
     1000    0.1,10
     1002   0.14,15
   200000    0.2,20
   400000    0.3,30
   500000    0.3,35
   600000       ,35
h 800000
100000000   4.04,40
h 200000000
```

Example 5–10 CASE Expression: SELECT Stream S1 Input

```
Timestamp   Tuple
     1000   10,0.1
     1002   15,0.14
   200000   20,0.2
   300000     ,0.2
   400000   30,0.3
100000000   40,4.04
```

Example 5–11 CASE Expression: SELECT Relation Output

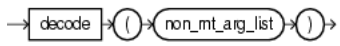
```
Timestamp   Tuple Kind  Tuple
     1000:  +           0.6
     1002:  +           0.44
     1002:  +           0.4
     1002:  +           0.14
   200000:  +           0.5
   200000:  +           0.5
   200000:  +           0.4
   200000:  +           0.44
   200000:  +           0.7
   300000:  +           0.4
   300000:  +           0.44
   300000:  +           0.7
   400000:  +           0.6
   400000:  +           0.6
   400000:  +           0.6
   400000:  +           0.6
   400000:  +           0.4
   400000:  +           0.44
   400000:  +           0.5
   400000:  +           0.8
   500000:  +           0.6
   500000:  +           0.6
   500000:  +           0.6
   500000:  +           0.6
   500000:  +           0.6
   600000:  +
   600000:  +
   600000:  +
   600000:  +
   600000:  +
100000000:  +           4.34
100000000:  +           4.34
100000000:  +           4.34
100000000:  +           4.34
100000000:  +           4.34
100000000:  +           0.4
100000000:  +           0.44
100000000:  +           0.5
100000000:  +           0.6
100000000:  +           0.6
100000000:  +
```

```
100000000:  +           4.34
```

decode

Use a *decode* expression to evaluate stream elements against multiple conditions.

decode::=



(*non_mt_arg_list::=* on page 7-21)

DECODE expects its *non_mt_arg_list* to be of the form:

```
expr, search1, result1, search2, result2, ... , searchN, result N, default
```

DECODE compares *expr* to each *search* value one by one. If *expr* equals a *search* value, the DECODE expressions returns the corresponding *result*. If no match is found, the DECODE expressions returns *default*. If *default* is omitted, the DECODE expressions returns null.

The arguments can be any of the numeric (INTEGER, BIGINT, FLOAT, or DOUBLE) or character (CHAR) datatypes. For more information, see Section , "Datatypes").

The *search*, *result*, and *default* values can be derived from expressions. Oracle Event Processing uses **short-circuit evaluation**. It evaluates each *search* value only before comparing it to *expr*, rather than evaluating all *search* values before comparing any of them with *expr*. Consequently, Oracle Event Processing never evaluates a search i, if a previous search j (0 < j < i) equals *expr*.

Oracle Event Processing automatically converts *expr* and each *search* value to the datatype of the first *search* value before comparing. Oracle Event Processing automatically converts the return value to the same datatype as the first *result*.

In a DECODE expression, Oracle Event Processing considers two nulls to be equivalent. If *expr* is null, then Oracle Event Processing returns the *result* of the first *search* that is also null.

The maximum number of components in the DECODE expression, including *expr*, *searches*, *results*, and *default*, is 255.

The *decode* expression is similar to the case_expr (see *case_expr::=* on page 5-9).

You can use a *decode* expression in the following Oracle CQL statements:

- *arith_expr::=* on page 5-6
- *opt_where_clause::=* on page 22-4
- *select_clause::=* on page 22-3

Examples

Consider the query q in Example 5–12 and the input relation R in Example 5–13. Relation R has schema (c1 float, c2 integer). The query returns the relation in Example 5–14.

Example 5–12 Arithmetic Expression and DECODE Query

```
<query id="q"><![CDATA[
...
    SELECT DECODE (c2, 10, c1+0.5, 20, c1+0.1, 30, c1+0.2, c1+0.3) from R
]]></query>
```

Example 5–13 Arithmetic Expression and DECODE Relation Input

```
Timestamp   Tuple Kind  Tuple
     1000:  +            0.1,10
     1002:  +           0.14,15
     2000:  -            0.1,10
     2002:  -           0.14,15
   200000:  +            0.2,20
   201000:  -            0.2,20
   400000:  +            0.3,30
   401000:  -            0.3,30
   500000:  +            0.3,35
   501000:  -            0.3,35
   600000:  +               ,35
   601000:  -               ,35
100000000:  +           4.04,40
100001000:  -           4.04,40
```

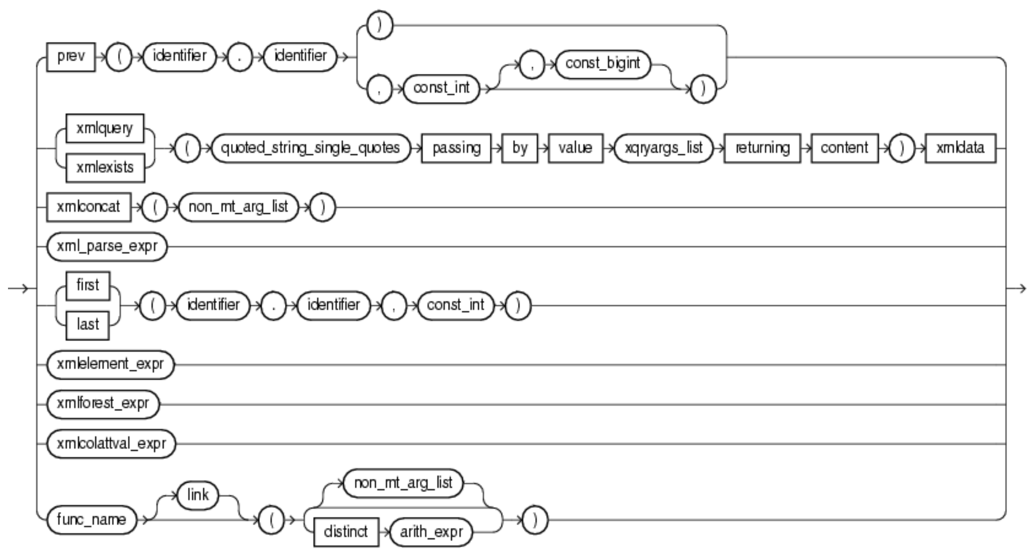
Example 5–14 Arithmetic Expression and DECODE Relation Output

```
Timestamp   Tuple Kind  Tuple
     1000:  +           0.6
     1002:  +           0.44
   200000:  +           0.3
   400000:  +           0.5
   500000:  +           0.6
100000000:  +           4.34
```

func_expr

Use the `func_expr` function expression to define a function invocation using any Oracle CQL built-in, user-defined, or Oracle data cartridge function.

func_expr::=



(*identifier::=* on page 7-17, *const_int::=* on page 7-12, *const_bigint::=* on page 7-11, *const_string::=* on page 7-13, *xqryargs_list::=* on page 7-34, *non_mt_arg_list::=* on page 7-21, *xml_parse_expr::=* on page 5-32, *xmlelement_expr::=* on page 5-28, *xmlforest_expr::=* on page 5-30, *xmlcolattval_expr::=* on page 5-26, *func_name:=* on page 5-15, *link::=* on page 5-19, *arith_expr::=* on page 5-6)

func_name:=

(*identifier::=* on page 7-17)

func_name

You can specify the identifier of a function explicitly:

- with or without a `link`, depending on the type of Oracle data cartridge function.

  For more information, see:

  – *link::=* on page 5-19

  – Chapter 14, "Introduction to Data Cartridges"

- with an empty argument list.
- with an argument list of one or more arguments.
- with a distinct arithmetic expression.

  For more information, see *aggr_distinct_expr* on page 5-3.

PREV

The `PREV` function takes a single argument made up of the following period-separated *identifier* arguments:

- `identifier1`: the name of a pattern as specified in a `DEFINE` clause.
- `identifier2`: the name of a stream element as specified in a `CREATE STREAM` statement.

The `PREV` function also takes the following *const_int* arguments:

- `const_int`: the index of the stream element before the current stream element to compare against. Default: 1.
- `const_bigint`: the timestamp of the stream element before the current stream element to compare against. To obtain the timestamp of a stream element, you can use the `ELEMENT_TIME` pseudocolumn (see Section , "ELEMENT_TIME Pseudocolumn").

For more information, see "prev" on page 8-10. For an example, see "func_expr PREV Function Example" on page 5-18.

XQuery: XMLEXISTS and XMLQUERY

You can specify an XQuery that Oracle Event Processing applies to the XML stream element data that you bind in *xqryargs_list*. For more information, see:

- "xmlexists" on page 8-27
- "xmlquery" on page 8-29

An *xqryargs_list* is a comma separated list of one or more *xqryarg* instances made up of an arithmetic expression involving one or more stream elements from the select list, the `AS` keyword, and a *const_string* that represents the XQuery variable or operator (such as the "." current node operator). For more information, see *xqryargs_list::=* on page 7-34.

For an example, see "func_expr XMLQUERY Function Example" on page 5-18.

For more information, see "SQL/XML (SQLX)" on page 5-16.

XMLCONCAT

The `XMLCONCAT` function returns the concatenation of its comma-delimited `xmltype` arguments as an `xmltype`.

For more information, see:

- "xmlconcat" on page 8-25
- "SQL/XML (SQLX)" on page 5-16

SQL/XML (SQLX)

The SQLX specification extends SQL to support XML data.

Oracle CQL supports event types containing properties of type `SQLX`. In this case, Oracle Event Processing server converts from `SQLX` to `String` when within Oracle CQL, and converts from `String` to `SQLX` on output.

Oracle CQL provides the following expressions (and functions) to manipulate data from an SQLX stream. For example, you can construct XML elements or attributes with SQLX stream elements, combine XML fragments into larger ones, and parse input into XML content or documents.

Note: Oracle CQL does not support external relations with columns of type `XMLTYPE` (for example, a join with a relational database management system). For more information, see Section , "Oracle CQL Built-in Datatypes".

For more information on Oracle CQL SQLX expressions, see:

- "xml_agg_expr" on page 5-24
- "xmlcolattval_expr" on page 5-26
- "xmlelement_expr" on page 5-28
- "xmlforest_expr" on page 5-30
- "xml_parse_expr" on page 5-32

For more information on Oracle CQL SQLX functions, see:

- "XQuery: XMLEXISTS and XMLQUERY" on page 5-16
- "xmlcomment" on page 8-23
- "xmlconcat" on page 8-25
- "xmlagg" on page 9-18

For more information on datatype restrictions when using Oracle CQL with XML, see:

- Section , "Datetime Literals"

For more information on SQLX, see the *Oracle Database SQL Language Reference*.

FIRST and LAST

The `FIRST` and `LAST` functions each take a single argument made up of the following period-separated values:

- `identifier1`: the name of a pattern as specified in a `DEFINE` clause.
- `identifier2`: the name of a stream element as specified in a `CREATE STREAM` statement.

For more information, see:

- "first" on page 9-7
- "last" on page 9-9

You can specify the identifier of a function explicitly with or without a *`non_mt_arg_list`*: a list of arguments appropriate for the built-in or user-defined function being invoked. The list can have single or multiple arguments.

You can use a *`func_expr`* in the following Oracle CQL statements:

- *arith_expr::=* on page 5-6

For more information, see Section , "Functions".

Examples

This section describes the following `func_expr` examples:

- "func_expr PREV Function Example" on page 5-18
- "func_expr XMLQUERY Function Example" on page 5-18

func_expr PREV Function Example

Example 5–15 shows how to compose a *func_expr* to invoke the `PREV` function.

Example 5–15 func_expr for PREV

```
<query id="q36"><![CDATA[
    select T.Ac1 from S15
    MATCH_RECOGNIZE (
        PARTITION BY
            c2
        MEASURES
            A.c1 as Ac1
        PATTERN(A)
        DEFINE
            A as (A.c1 = PREV(A.c1,3,5000) )
    ) as T
]]></query>
```

func_expr XMLQUERY Function Example

Example 5–16 shows how to compose a *func_expr* to invoke the `XMLQUERY` function.

Example 5–16 func_expr for XMLQUERY

```
<query id="q1"><![CDATA[
    select
        xmlexists(
            "for $i in /PDRecord where $i/PDId <= $x return $i/PDName"
            passing by value
                c2 as ".",
                (c1+1) as "x"
            returning content
        ) xmldata
    from
        S1
]]></query>
```

Example 5–17 shows how to compose a *func_expr* to invoke the `SUM` function.

Example 5–17 func_expr for SUM

```
<query id="q3"><![CDATA[
    select sum(c2) from S1[range 5]
]]></query>
```

object_expr

Use the `object_expr` expression to reference the members of a data cartridge complex type.

You can use an `object_expr` anywhere an arithmetic expression can be used. For more information, see "arith_expr" on page 5-6.

object_expr::=

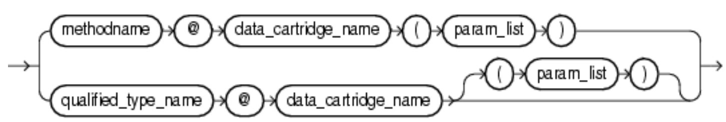
For syntax, see:

- *complex_type::=* on page 7-8
- *array_type::=* on page 7-3

Optionally, you can use a link (`@`) in the `object_expr` to specify the data cartridge name. Use a `link` to specify the location of an Oracle CQL data cartridge complex type class, method, field, or constructor to disambiguate the reference, if necessary. The location must reference a data cartridge by its name. For example, if two data cartridges (`myCartridge` and `yourCartridge`) both define a complex type `com.package.ThisClass`, then you must use the `link` clause to explicitly identify which `com.package.ThisClass` you want to use.

> **Note:** A `link` is not required when using the types that the default Java data cartridge provides.

link::=



(*methodname::=* on page 7-20, *data_cartridge_name::=* on page 5-19, *param_list::=* on page 7-26, *qualified_type_name::=* on page 7-27)

data_cartridge_name::=

(*identifier::=* on page 7-17)

data_cartridge_name

Each Oracle CQL data cartridge implementation is identified by a unique data cartridge name.

Data cartridge names include:

- `java`: identifies the Oracle CQL Java data cartridge.

  This is the default data cartridge name. If you omit a data cartridge name in field or constructor references, Oracle CQL will try to resolve the reference using the `java` data cartridge name. This means the following statements are identical:

  ```
  SELECT java.lang.String@java("foo") …
  SELECT java.lang.String("foo") …
  ```

If you omit a data cartridge name in a method reference, Oracle CQL will try to resolve the reference against its built-in functions (see Section , "Functions").

- `spatial`: identifies the Oracle CQL Oracle Spatial.

For syntax, see *data_cartridge_name::=* on page 5-19 (parent: *link::=* on page 5-19).

Type Declaration

You declare an event property as a complex type using *`qualified_type_name`*`@`*`data_cartridge_name`*.

For examples, see "Type Declaration Example: link" on page 5-21

Field Access

You cannot specify a link when accessing a complex type field because the type of the field already identifies its location. The following is *not* allowed:

```
SELECT java.lang.String("foo").CASE_INSENSITIVE_ORDER@java …
```

For examples, see "Field Access Example: link" on page 5-21.

Method Access

You cannot specify a link when accessing complex type method because the type of the method already identifies its location. The following is *not* allowed:

```
SELECT java.lang.String("foo").substring@java(0,1) …
```

For examples, see "Method Access Example: link" on page 5-21.

Constructor Invocation

You invoke a complex type constructor using *`qualified_type_name`*`@`*`data_cartridge_name`*`(`*`param_list`*`)`.

For examples, see "Constructor Invocation Example: link" on page 5-22.

Examples

The following examples illustrate the various semantics that this statement supports:

- "Object Expression Example" on page 5-20
- "Type Declaration Example: link" on page 5-21
- "Field Access Example: link" on page 5-21
- "Method Access Example: link" on page 5-21
- "Constructor Invocation Example: link" on page 5-22

Object Expression Example

Example 5–18 shows *`object_expr`*:

```
getContainingGeometries@spatial (InputPoints.point)
```

This *`object_expr`* uses a data cartridge `TABLE` clause that invokes the Oracle Spatial method `getContainingGeometries`, passing in one parameter (`InputPoints.point`). The return value of this method, a `Collection` of Oracle Event Processing `IType` records, is aliased as `validGeometries`. The table source itself is aliased as `R2`.

Example 5–18 Data Cartridge TABLE Query

```
<query id="q1"><![CDATA[
RSTREAM (
    SELECT
        R2.validGeometries.shape as containingGeometry,
        R1.point as inputPoint
    FROM
        InputPoints[now] as R1,
        TABLE (getContainingGeometries@spatial (InputPoints.point) as validGeometries) AS R2
)
]]></query>
```

Type Declaration Example: *link*

Example 5–19 shows how to create an event type as a Java class that specifies an event property as an Oracle CQL data cartridge complex type `MyType` defined in package `com.mypackage` that belongs to the Oracle CQL data cartridge `myCartridge`. If a `com.myPackage.MyType` is defined in some other Oracle CQL data cartridge (with data cartridge name `otherCatridge`), specifying the type for the a1 property using a link with the data cartridge name `myCartridge` allows Oracle CQL to reference the correct complex type.

Example 5–19 Type Declaration Using an Oracle CQL Data Cartridge Link

```
package com.myapplication.event;

import java.util.Date;
import // Oracle CQL data cartridge package(s)?

public final class MarketEvent {
    private final String symbol;
    private final Double price;
    private final com.myPackage.MyType@myCartridge a1;

    public MarketEvent(...) {
        ...
        }
    ...
}
```

Field Access Example: *link*

Example 5–20 shows how to instantiate complex type `MyType` and access the static field `MY_FIELD`. The link clause explicitly references the `com.myPackage.MyType` class that belongs to the Oracle CQL data cartridge `myCartridge`.

Example 5–20 Field Access Using an Oracle CQL Data Cartridge Link

```
<query id="q1"><![CDATA[
    SELECT com.myPackage.MyType@myCartridge("foo").MY_FIELD ...
]]></query>
```

Method Access Example: *link*

Example 5–21 shows how to instantiate complex type `MyType` and access the method `myMethod`. The link clause explicitly references the `com.myPackage.MyType` class that belongs to the Oracle CQL data cartridge `myCartridge`.

Example 5–21 Type Declaration Using an Oracle CQL Data Cartridge Link

```
<query id="q1"><![CDATA[
    SELECT com.myPackage.MyType@myCartridge("foo").myMethod("bar") ...
```

```
]]></query>
```

Constructor Invocation Example: *link*

Example 5–22 shows how to instantiate complex type `MyType`. The link clause explicitly references the `com.myPackage.MyType` class that belongs to the Oracle CQL data cartridge `myCartridge`.

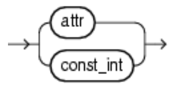
Example 5–22 Type Declaration Using an Oracle CQL Data Cartridge Link

```
<query id="q1"><![CDATA[
    SELECT com.myPackage.MyType@myCartridge(“foo”) ...
]]></query>
```

order_expr

Use the *order_expr* expression to specify the sort order in which Oracle Event Processing returns tuples that a query selects.

order_expr::=



(*attr::=* on page 7-5, *const_int::=* on page 7-12)

You can specify a stream element by *attr* name.

Alternatively, you can specify a stream element by its *const_int* index where the index corresponds to the stream element position you specify at the time you register or create the stream.

You can use an *order_expr* in the following Oracle CQL statements:

- *orderterm::=* on page 22-5

Examples

Consider Example 5–23. Stream S3 has schema (c1 bigint, c2 interval, c3 byte(10), c4 float). This example shows how to order the results of query q210 by c1 and then c2 and how to order the results of query q211 by c2, then by the stream element at index 3 (c3) and then by the stream element at index 4 (c4).

Example 5–23 order_expr

```
<query id="q210"><![CDATA[
    select * from S3 order by c1 desc nulls first, c2 desc nulls last
]]></query>
<query id="q211"><![CDATA[
    select * from S3 order by c2 desc nulls first, 3 desc nulls last, 4 desc
]]></query>
```

xml_agg_expr

Use an *xml_agg_expr* expression to return a collection of XML fragments as an aggregated XML document. Arguments that return null are dropped from the result.

xml_agg_expr::=

(*arith_expr* on page 5-6, *order_by_clause::=* on page 22-5)

You can specify an *xml_agg_expr* as the argument of an aggregate expression.

You can use an *xml_agg_expr* in the following Oracle CQL statements:

- *aggr_expr::=* on page 5-4

For more information, see:

- Chapter 9, "Built-In Aggregate Functions"
- "xmlagg" on page 9-18
- "XMLAGG" in the *Oracle Database SQL Language Reference*

Examples

Consider the query `tkdata67_q1` in Example 5–24 and the input relation `tkdata67_S0` in Example 5–25. Relation `tkdata67_S0` has schema `(c1 integer, c2 float)`. The query returns the relation in Example 5–26.

Example 5–24 xml_agg_expr Query

```
<query id="tkdata67_q1"><![CDATA[
    select
        c1,
        xmlagg(xmlelement("c2",c2))
    from
        tkdata67_S0[rows 10]
    group by c1
]]></query>
```

Example 5–25 xml_agg_expr Relation Input

```
Timestamp           Tuple
 1000               15, 0.1
 1000               20, 0.14
 1000               15, 0.2
 4000               20, 0.3
10000               15, 0.04
h 12000
```

Example 5–26 xml_agg_expr Relation Output

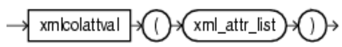
```
Timestamp   Tuple Kind  Tuple
1000:       +           15,<c2>0.1</c2>
                           <c2>0.2</c2>
1000:       +           20,<c2>0.14</c2>
4000:       -           20,<c2>0.14</c2>
4000:       +           20,<c2>0.14</c2>
                           <c2>0.3</c2>
10000:      -           15,<c2>0.1</c2>
```

```
                          <c2>0.2</c2>
10000:      +          15,<c2>0.1</c2>
                          <c2>0.2</c2>
                          <c2>0.04</c2>
```

xmlcolattval_expr

Use an *xmlcolattval_expr* expression to create an XML fragment and then expand the resulting XML so that each XML fragment has the name column with the attribute name.

xmlcolattval_expr::=



(*xml_attr_list::=* on page 7-33)

You can specify an *xmlcolattval_expr* as the argument of a function expression. It is especially useful when processing SQLX streams. For more information, see "SQL/XML (SQLX)" on page 5-16.

You can use an *xmlcolattval_expr* in the following Oracle CQL statements:

- *func_expr::=* on page 5-15

For more information, see "XMLCOLATTVAL" in the *Oracle Database SQL Language Reference*.

Examples

Consider the query `tkdata53_q1` in Example 5–24 and the input relation `tkdata53_S0` in Example 5–25. Relation `tkdata53_S0` has schema `(c1 integer, c2 float)`. The query returns the relation in Example 5–26.

Example 5–27 xmlcolattval_expr Query

```
<query id="tkdata53_q1"><![CDATA[
    select
        XMLELEMENT("tkdata53_S0", XMLCOLATTVAL( tkdata53_S0.c1, tkdata53_S0.c2))
    from
        tkdata53_S0 [range 1]
]]></query>
```

Example 5–28 xmlcolattval_expr Relation Input

```
Timestamp                Tuple
     1000:               10, 0.1
     1002:               15, 0.14
   200000:               20, 0.2
   400000:               30, 0.3
h 800000
100000000:               40, 4.04
h 200000000
```

Example 5–29 xmlcolattval_expr Relation Output

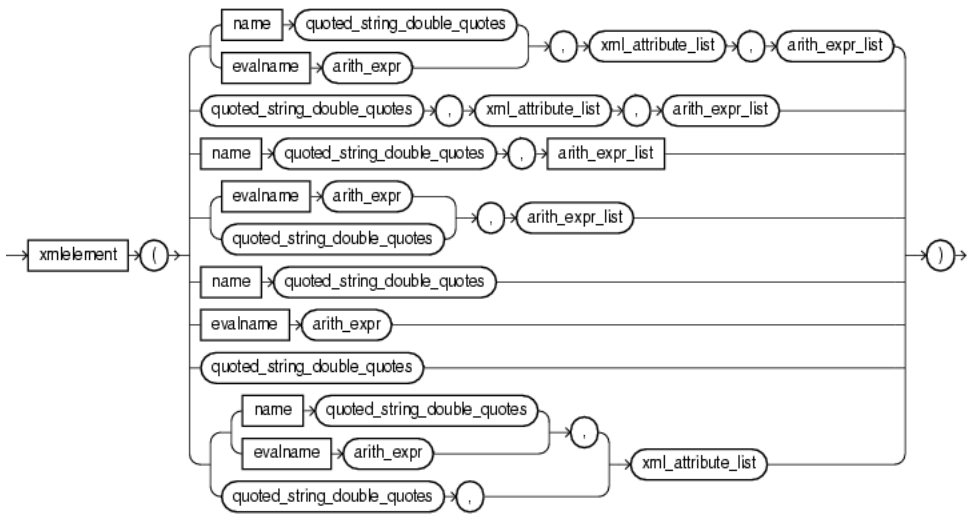
```
Timestamp   Tuple Kind  Tuple
     1000:       +      <tkdata53_S0>
                            <column name="c1">10</column>
                            <column name="c2">0.1</column>
                        </tkdata53_S0>
     1002:       +      <tkdata53_S0>
                            <column name="c1">15</column>
                            <column name="c2">0.14</column>
                        </tkdata53_S0>
     2000:       -      <tkdata53_S0>
```

```
                        <column name="c1">10</column>
                        <column name="c2">0.1</column>
                    </tkdata53_S0>
      2002:      -      <tkdata53_S0>
                        <column name="c1">15</column>
                        <column name="c2">0.14</column>
                    </tkdata53_S0>
    200000:      +      <tkdata53_S0>
                        <column name="c1">20</column>
                        <column name="c2">0.2</column>
                    </tkdata53_S0>
    201000:      -      <tkdata53_S0>
                        <column name="c1">20</column>
                        <column name="c2">0.2</column>
                    </tkdata53_S0>
    400000:      +      <tkdata53_S0>
                        <column name="c1">30</column>
                        <column name="c2">0.3</column>
                    </tkdata53_S0>
    401000:      -      <tkdata53_S0>
                        <column name="c1">30</column>
                        <column name="c2">0.3</column>
                    </tkdata53_S0>
 100000000:      +      <tkdata53_S0>
                        <column name="c1">40</column>
                        <column name="c2">4.04</column>
                    </tkdata53_S0>
 100001000:      -      <tkdata53_S0>
                        <column name="c1">40</column>
                        <column name="c2">4.04</column>
                    </tkdata53_S0>
```

xmlelement_expr

Use an *xmlelement_expr* expression when you want to construct a well-formed XML element from stream elements.

xmlelement_expr::=



(*const_string::=* on page 7-13, *arith_expr::=* on page 5-6, *xml_attribute_list::=* on page 7-32, *arith_expr_list::=* on page 5-8)

You can specify an *xmlelement_expr* as the argument of a function expression. It is especially useful when processing SQLX streams. For more information, see "SQL/XML (SQLX)" on page 5-16.

You can use an *xmlelement_expr* in the following Oracle CQL statements:

- *func_expr::=* on page 5-15

For more information, see "XMLELEMENT" in the *Oracle Database SQL Language Reference*.

Examples

Consider the query `tkdata51_q0` in Example 5–30 and the input relation `tkdata51_S0` in Example 5–31. Relation `tkdata51_S0` has schema `(c1 integer, c2 float)`. The query returns the relation in Example 5–32.

Example 5–30 xmlelement_expr Query

```
<query id="tkdata51_q0"><![CDATA[
    select
        XMLELEMENT(
            NAME "S0",
            XMLELEMENT(NAME "c1", tkdata51_S0.c1),
            XMLELEMENT(NAME "c2", tkdata51_S0.c2)
        )
    from
        tkdata51_S0 [range 1]
]]></query>
```

Example 5–31 xmlelement_expr Relation Input

```
Timestamp                Tuple
     1000:               10, 0.1
     1002:               15, 0.14
   200000:               20, 0.2
   400000:               30, 0.3
h 800000
100000000:               40, 4.04
h 200000000
```

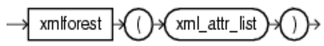
Example 5–32 xmlelement_expr Relation Output

```
Timestamp   Tuple Kind  Tuple
     1000:  +           <S0>
                            <c1>10</c1>
                            <c2>0.1</c2>
                        </S0>
     1002:  +           <S0>
                            <c1>15</c1>
                            <c2>0.14</c2>
                        </S0>
     2000:  -           <S0>
                            <c1>10</c1>
                            <c2>0.1</c2>
                        </S0>
     2002:  -           <S0>
                            <c1>15</c1>
                            <c2>0.14</c2>
                        </S0>
   200000:  +           <S0>
                            <c1>20</c1>
                            <c2>0.2</c2>
                        </S0>
   201000:  -           <S0>
                            <c1>20</c1>
                            <c2>0.2</c2>
                        </S0>
   400000:  +           <S0>
                            <c1>30</c1>
                            <c2>0.3</c2>
                        </S0>
   401000:  -           <S0>
                            <c1>30</c1>
                            <c2>0.3</c2>
                        </S0>
100000000:  +           <S0>
                            <c1>40</c1>
                            <c2>4.04</c2>
                        </S0>
100001000:  -           <S0>
                            <c1>40</c1>
                            <c2>4.04</c2>
                        </S0>
```

xmlforest_expr

Use an *xmlforest_expr* to convert each of its argument parameters to XML, and then return an XML fragment that is the concatenation of these converted arguments.

xmlforest_expr::=



(*xml_attr_list::=* on page 7-33)

You can specify an *xmlforest_expr* as the argument of a function expression. It is especially useful when processing SQLX streams. For more information, see "SQL/XML (SQLX)" on page 5-16.

You can use an *xmlforest_expr* in the following Oracle CQL statements:

- *func_expr::=* on page 5-15

For more information, see "XMLFOREST" in the *Oracle Database SQL Language Reference*.

Examples

Consider the query `tkdata52_q0` in Example 5–33 and the input relation `tkdata52_S0` in Example 5–34. Relation `tkdata52_S0` has schema `(c1 integer, c2 float)`. The query returns the relation in Example 5–35.

Example 5–33 xmlforest_expr Query

```
<query id="tkdata52_q0"><![CDATA[
    select
        XMLFOREST( tkdata52_S0.c1, tkdata52_S0.c2)
    from
        tkdata52_S0 [range 1]
]]></query>
```

Example 5–34 xmlforest_expr Relation Input

```
Timestamp                Tuple
     1000:               10, 0.1
     1002:               15, 0.14
   200000:               20, 0.2
   400000:               30, 0.3
h 800000
100000000:               40, 4.04
h 200000000
```

Example 5–35 xmlforest_expr Relation Output

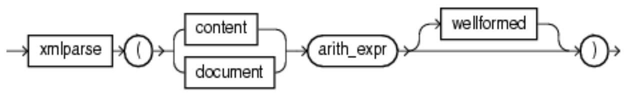
```
Timestamp   Tuple Kind  Tuple
     1000:  +           <c1>10</c1>
                        <c2>0.1</c2>
     1002:  +           <c1>15</c1>
                        <c2>0.14</c2>
     2000:  -           <c1>10</c1>
                        <c2>0.1</c2>
     2002:  -           <c1>15</c1>
                        <c2>0.14</c2>
   200000:  +           <c1>20</c1>
                        <c2>0.2</c2>
```

```
   201000:  -          <c1>20</c1>
                       <c2>0.2</c2>
   400000:  +          <c1>30</c1>
                       <c2>0.3</c2>
   401000:  -          <c1>30</c1>
                       <c2>0.3</c2>
100000000:  +          <c1>40</c1>
                       <c2>4.04</c2>
100001000:  -          <c1>40</c1>
                       <c2>4.04</c2>
```

xml_parse_expr

Use an *xml_parse_expr* expression to parse and generate an XML instance from the evaluated result of *arith_expr*.

xml_parse_expr::=



(*arith_expr::=* on page 5-6)

When using an *xml_parse_expr* expression, note the following:

- If *arith_expr* resolves to null, then the expression returns null.
- If you specify `content`, then *arith_expr* must resolve to a valid XML value. For an example, see "xml_parse_expr Document Example" on page 5-33
- If you specify `document`, then *arith_expr* must resolve to a singly rooted XML document. For an example, see "xml_parse_expr Content Example" on page 5-32.
- When you specify `wellformed`, you are guaranteeing that `value_expr` resolves to a well-formed XML document, so the database does not perform validity checks to ensure that the input is well formed. For an example, see "xml_parse_expr Wellformed Example" on page 5-33.

You can specify an *xml_parse_expr* as the argument of a function expression. It is especially useful when processing SQLX streams. For more information, see "SQL/XML (SQLX)" on page 5-16.

You can use an *xml_parse_expr* in the following Oracle CQL statements:

- *func_expr::=* on page 5-15

For more information, see "XMLPARSE" in the *Oracle Database SQL Language Reference*.

Examples

This section describes the following `xml_parse_expr` examples:

- "xml_parse_expr Content Example" on page 5-32
- "xml_parse_expr Document Example" on page 5-33
- "xml_parse_expr Wellformed Example" on page 5-33

xml_parse_expr Content Example

Consider the query `tkdata62_q3` in Example 5–36 and the input relation `tkdata62_S1` in Example 5–37. Relation `tkdata62_S1` has schema `(c1 char(30))`. The query returns the relation in Example 5–38.

Example 5–36 xml_parse_expr Content: Query

```
<query id="tkdata62_q3"><![CDATA[
    select XMLPARSE(CONTENT c1) from tkdata62_S1
]]></query>
```

Example 5–37 xml_parse_expr Content: Relation Input

```
Timestamp               Tuple
1000                    "<a>3</a>"
1000                    "<e3>blaaaaa</e3>"
1000                    "<r>4</r>"
1000                    "<a></a>
1003                    "<a>s3</a>"
1004                    "<d>b6</d>"
```

Example 5–38 xml_parse_expr Content: Relation Output

```
Timestamp   Tuple Kind  Tuple
1000:       +     <a>3</a>
1000:       +     <e3>blaaaaa</e3>
1000:       +     <r>4</r>
1000:       +     <a/>
1003:       +     <a>s3</a>
1004:       +     <d>b6</d>
```

xml_parse_expr Document Example

Consider the query `tkdata62_q4` in Example 5–39 and the input relation `tkdata62_S1` in Example 5–40. Relation `tkdata62_S1` has schema `(c1 char(30))`. The query returns the relation in Example 5–41.

Example 5–39 xml_parse_expr Document: Query

```
<query id="tkdata62_q4"><![CDATA[
    select XMLPARSE(DOCUMENT c1) from tkdata62_S1
]]></query>
```

Example 5–40 xml_parse_expr Document: Relation Input

```
Timestamp               Tuple
1000                    "<a>3</a>"
1000                    "<e3>blaaaaa</e3>"
1000                    "<r>4</r>"
1000                    "<a></a>
1003                    "<a>s3</a>"
1004                    "<d>b6</d>"
```

Example 5–41 xml_parse_expr Document: Relation Output

```
Timestamp   Tuple Kind  Tuple
1000:       +     <a>3</a>
1000:       +     <e3>blaaaaa</e3>
1000:       +     <r>4</r>
1000:       +     <a/>
1003:       +     <a>s3</a>
1004:       +     <d>b6</d>
```

xml_parse_expr Wellformed Example

Consider the query `tkdata62_q2` in Example 5–42 and the input relation `tkdata62_S` in Example 5–43. Relation `tkdata62_S` has schema `(c char(30))`. The query returns the relation in Example 5–44.

Example 5–42 xml_parse_expr Wellformed: Query

```
<query id="tkdata62_q2"><![CDATA[
    select XMLPARSE(DOCUMENT c WELLFORMED) from tkdata62_S
]]></query>
```

Example 5–43 xml_parse_expr Wellformed: Relation Input

```
Timestamp              Tuple
1000                   "<a>3</a>"
1000                   "<e3>blaaaaa</e3>"
1000                   "<r>4</r>"
1000                   "<a/>"
1003                   "<a>s3</a>"
1004                   "<d>b6</d>"
```

Example 5–44 xml_parse_expr Wellformed: Relation Output

```
Timestamp   Tuple Kind  Tuple
1000:       +           <a>3</a>
1000:       +           <e3>blaaaaa</e3>
1000:       +           <r>4</r>
1000:       +           <a/>
1003:       +           <a>s3</a>
1004:       +           <d>b6</d>
```

6 Conditions

This chapter provides a reference to conditions in Oracle Continuous Query Language (Oracle CQL). A condition specifies a combination of one or more expressions and logical operators and returns a value of `TRUE`, `FALSE`, or `UNKNOWN`.

This chapter includes the following sections:

Introduction to Conditions

You must use appropriate condition syntax whenever *`condition`* appears in Oracle CQL statements.

You can use a condition in the `WHERE` clause of these statements:

- `SELECT`

You can use a condition in any of these clauses of the `SELECT` statement:

- `WHERE`
- `HAVING`

See Also: "Query" on page 22-2

A condition could be said to be of a logical datatype.

The following simple condition always evaluates to `TRUE`:

```
1 = 1
```

The following more complex condition adds the `salary` value to the `commission_pct` value (substituting the value 0 for null using the `nvl` function) and determines whether the sum is greater than the number constant 25000:

```
NVL(salary, 0) + NVL(salary + (salary*commission_pct, 0) > 25000)
```

Logical conditions can combine multiple conditions into a single condition. For example, you can use the AND condition to combine two conditions:

```
(1 = 1) AND (5 < 7)
```

Here are some valid conditions:

```
name = 'SMITH'
S0.department_id = S2.department_id
hire_date > '01-JAN-88'
commission_pct IS NULL AND salary = 2100
```

Condition Precedence

Precedence is the order in which Oracle Event Processing evaluates different conditions in the same expression. When evaluating an expression containing multiple conditions, Oracle Event Processing evaluates conditions with higher precedence before evaluating those with lower precedence. Oracle Event Processing evaluates conditions with equal precedence from left to right within an expression.

Table 6–1 lists the levels of precedence among Oracle CQL condition from high to low. Conditions listed on the same line have the same precedence. As the table indicates, Oracle evaluates operators before conditions.

Table 6–1 Oracle CQL Condition Precedence

Type of Condition	Purpose
Oracle CQL operators are evaluated before Oracle CQL conditions	See Section , "What You May Need to Know About Operator Precedence".
`=, <>, <, >, <=, >=`	comparison
`IS NULL`, `IS NOT NULL`, `LIKE`, `BETWEEN`, `IN`, `NOT IN`	comparison
`NOT`	exponentiation, logical negation
`AND`	conjunction
`OR`	disjunction
`XOR`	disjunction

Comparison Conditions

Comparison conditions compare one expression with another. The result of such a comparison can be TRUE, FALSE, or NULL.

When comparing numeric expressions, Oracle Event Processing uses numeric precedence to determine whether the condition compares INTEGER, FLOAT, or BIGINT values.

Two objects of nonscalar type are comparable if they are of the same named type and there is a one-to-one correspondence between their elements.

A comparison condition specifies a comparison with expressions or view results.

Table 6–2 lists comparison conditions.

Table 6–2 Comparison Conditions

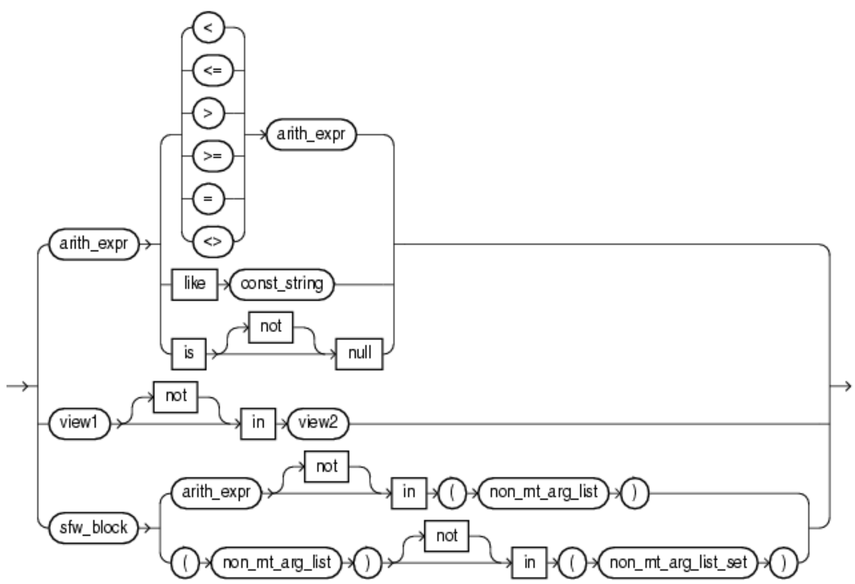
Type of Condition	Purpose	Example
=	Equality test.	<query id="Q1"><![CDATA[SELECT * FROM S0 WHERE salary = 2500]]></query>
<>	Inequality test.	<query id="Q1"><![CDATA[SELECT * FROM S0 WHERE salary <> 2500]]></query>
> <	Greater-than and less-than tests.	<query id="Q1"><![CDATA[SELECT * FROM S0 WHERE salary > 2500]]></query> <query id="Q1"><![CDATA[SELECT * FROM S0 WHERE salary < 2500]]></query>
>= <=	Greater-than-or-equal-to and less-than-or-equal-to tests.	<query id="Q1"><![CDATA[SELECT * FROM S0 WHERE salary >= 2500]]></query> <query id="Q1"><![CDATA[SELECT * FROM S0 WHERE salary <= 2500]]></query>
like	Pattern matching tests on character data. For more information, see Section , "LIKE Condition".	<query id="q291"><![CDATA[select * from SLk1 where first1 like "^Ste(v|ph)en$"]]></query>
is [not] null	Null tests. For more information, see Section , "Null Conditions".	<query id="Q1"><![CDATA[SELECT last_name FROM S0 WHERE commission_pct IS NULL]]></query> <query id="Q2"><![CDATA[SELECT last_name FROM S0 WHERE commission_pct IS NOT NULL]]></query>

Table 6–2 (Cont.) Comparison Conditions

Type of Condition	Purpose	Example

`[not] in`	Set and membership tests.
	For more information, see Section , "IN Condition".
		`<query id="Q1"><![CDATA[`
		`  SELECT * FROM S0`
		`  WHERE job_id NOT IN`
		`  ('PU_CLERK','SH_CLERK')`
		`]]></query>`
		`<view id="V1" schema="salary"><![CDATA[`
		`   SELECT salary`
		`   FROM S0`
		`   WHERE department_id = 30`
		`]]></view>`
		`<view id="V2" schema="salary"><![CDATA[`
		`   SELECT salary`
		`   FROM S0`
		`   WHERE department_id = 20`
		`]]></view>`
		`<query id="Q2"><![CDATA[`
		`  V1 IN V2`
		`]]></query>`

***condition*::=**



(*arith_expr::=* on page 5-6, *const_string::=* on page 7-13, *non_mt_arg_list::=* on page 7-21, *non_mt_arg_list_set::=* on page 6-9, *sfw_block::=* on page 22-3)

Logical Conditions

A logical condition combines the results of two component conditions to produce a single result based on them or to invert the result of a single condition. Table 6–3 lists logical conditions.

Table 6–3 Logical Conditions

Type of Condition	Operation	Examples
NOT	Returns TRUE if the following condition is FALSE. Returns FALSE if it is TRUE. If it is UNKNOWN, then it remains UNKNOWN.	`<query id="Q1"><![CDATA[ SELECT * FROM S0 WHERE NOT (job_id IS NULL) ]]></query>`
AND	Returns TRUE if both component conditions are TRUE. Returns FALSE if either is FALSE. Otherwise returns UNKNOWN.	`<query id="Q1"><![CDATA[ SELECT * FROM S0 WHERE job_id = 'PU_CLERK' AND dept_id = 30 ]]></query>`
OR	Returns TRUE if either component condition is TRUE. Returns FALSE if both are FALSE. Otherwise returns UNKNOWN.	`<query id="Q1"><![CDATA[ SELECT * FROM S0 WHERE job_id = 'PU_CLERK' OR department_id = 10 ]]></query>`
XOR	Returns TRUE if either component condition is TRUE. Returns FALSE if both are FALSE. Otherwise returns UNKNOWN.	`<query id="Q1"><![CDATA[ SELECT * FROM S0 WHERE job_id = 'PU_CLERK' XOR department_id = 10 ]]></query>`

Table 6–4 shows the result of applying the NOT condition to an expression.

Table 6–4 NOT Truth Table

--	TRUE	FALSE	UNKNOWN
NOT	FALSE	TRUE	UNKNOWN

Table 6–5 shows the results of combining the AND condition to two expressions.

Table 6–5 AND Truth Table

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

For example, in the WHERE clause of the following SELECT statement, the AND logical condition returns values only when both product.levelx is BRAND and v1.prodkey equals product.prodkey:

```
<view id="v2" schema="region, dollars, month_"><![CDATA[
    select
        v1.region,
        v1.dollars,
        v1.month_
    from
        v1,
        product
    where
        product.levelx = "BRAND" and v1.prodkey = product.prodkey
]]></view>
```

Table 6–6 shows the results of applying `OR` to two expressions.

Table 6–6 OR Truth Table

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

For example, the following query returns the internal account identifier for `RBK` or `RBR` accounts with a risk of type 2:

```
<view id="ValidAccounts" schema="ACCT_INTRL_ID"><![CDATA[
    select ACCT_INTRL_ID from Acct
    where (
        ((MANTAS_ACCT_BUS_TYPE_CD = "RBK") OR (MANTAS_ACCT_BUS_TYPE_CD = "RBR")) AND
        (ACCT_EFCTV_RISK_NB != 2)
    )
]]></view>
```

Table 6–7 shows the results of applying `XOR` to two expressions.

Table 6–7 XOR Truth Table

XOR	TRUE	FALSE	UNKNOWN
TRUE	FALSE	TRUE	UNKNOWN
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN

For example, the following query returns `c1` and `c2` when `c1` is 15 and `c2` is 0.14 or when `c1` is 20 and `c2` is 100.1, but not both:

```
<query id="q6"><![CDATA[
    select
        S2.c1,
        S3.c2
    from
        S2[range 1000], S3[range 1000]
    where
        (S2.c1 = 15 and S3.c2 = 0.14) xor (S2.c1 = 20 and S3.c2 = 100.1)
]]></query>
```

LIKE Condition

The `LIKE` condition specifies a test involving regular expression pattern matching. Whereas the equality operator (=) exactly matches one character value to another, the `LIKE` conditions match a portion of one character value to another by searching the first value for the regular expression pattern specified by the second. `LIKE` calculates strings using characters as defined by the input character set.

like_condition::=

(*arith_expr::=* on page 5-6, *const_string::=* on page 7-13)

In this syntax:

- *arith_expr* is an arithmetic expression whose value is compared to *const_string*.
- *const_string* is a constant value regular expression to be compared against the *arith_expr*.

If any of *arith_expr* or *const_string* is null, then the result is unknown.

The *const_string* can contain any of the regular expression assertions and quantifiers that java.util.regex supports: that is, a regular expression that is specified in string form in a syntax similar to that used by Perl.

Table 6–8 describes the LIKE conditions.

Table 6–8 LIKE Conditions

Type of Condition	Operation	Example
x LIKE y	TRUE if *x* does match the pattern *y*, FALSE otherwise.	`<query id="q291"><![CDATA[ select * from SLk1 where first1 like "^Ste(v|ph)en$" ]]></query>` `<query id="q292"><![CDATA[ select * from SLk1 where first1 like ".*intl.*" ]]></query>`

See Also: "lk" on page 8-7

For more information on Perl regular expressions, see http://perldoc.perl.org/perlre.html.

Examples

This condition is true for all last_name values beginning with Ma:

```
last_name LIKE '^Ma'
```

All of these last_name values make the condition true:

```
Mallin, Markle, Marlow, Marvins, Marvis, Matos
```

Case is significant, so last_name values beginning with MA, ma, and mA make the condition false.

Consider this condition:

```
last_name LIKE 'SMITH[A-Za-z]'
```

This condition is true for these last_name values:

```
SMITHE, SMITHY, SMITHS
```

This condition is false for SMITH because the [A-Z] must match exactly one character of the last_name value.

Consider this condition:

```
last_name LIKE 'SMITH[A-Z]+'
```

This condition is false for SMITH but true for these last_name values because the [A-Z]+ must match 1 or more such characters at the end of the word.

```
SMITHSTONIAN, SMITHY, SMITHS
```

For more information, see
http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html.

Range Conditions

A range condition tests for inclusion in a range.

between_condition::=

(*arith_expr::=* on page 5-6)

Table 6–9 describes the range conditions.

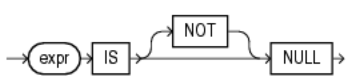
Table 6–9 Range Conditions

Type of Condition	Operation	Example
BETWEEN x AND y	Greater than or equal to *x* and less than or equal to *y*.	<query id="Q1"><![CDATA[SELECT * FROM S0 WHERE salary BETWEEN 2000 AND 3000]]></query>

Null Conditions

A NULL condition tests for nulls. This is the only condition that you should use to test for nulls.

null_conditions::=



(Chapter 5, "Expressions")

Table 6–10 lists the null conditions.

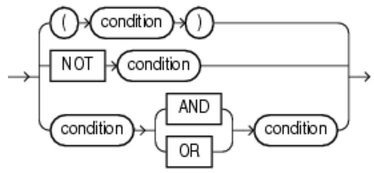
Table 6–10 Null Conditions

Type of Condition	Operation	Example
IS [NOT] NULL	Tests for nulls. **See Also:** Section , "Nulls"	<query id="Q1"><![CDATA[SELECT last_name FROM S0 WHERE commission_pct IS NULL]]></query> <query id="Q2"><![CDATA[SELECT last_name FROM S0 WHERE commission_pct IS NOT NULL]]></query>

Compound Conditions

A compound condition specifies a combination of other conditions.

***compound_conditions*::=**



See Also: Section , "Logical Conditions" for more information about NOT, AND, and OR conditions

IN Condition

You can use the IN and NOT IN condition in the following ways:

- *in_condition_set*: Section , "Using IN and NOT IN as a Set Operation"
- *in_condition_membership*: Section , "Using IN and NOT IN as a Membership Condition"

Note: You cannot combine these two usages.

When using the NOT IN condition, be aware of the effect of null values as Section , "NOT IN and Null Values" describes.

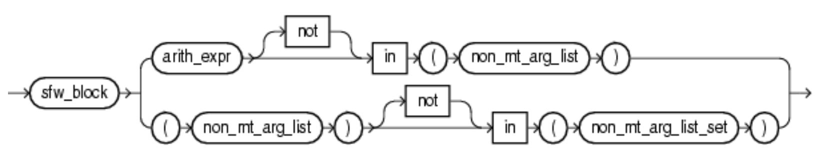
Using IN and NOT IN as a Set Operation

See "BINARY Example: IN and NOT IN" on page 22-19.

Using IN and NOT IN as a Membership Condition

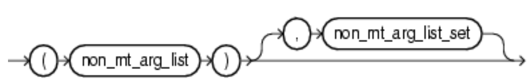
In this usage, the query will be a SELECT-FROM-WHERE query that either tests whether or not one argument is a member of a list of arguments of the same type or tests whether or not a list of arguments is a member of a set of similar lists.

***in_condition_membership*::=**



(*arith_expr::=* on page 5-6, *non_mt_arg_list::=* on page 7-21, *non_mt_arg_list_set::=* on page 6-9)

***non_mt_arg_list_set*::=**



(*non_mt_arg_list::=* on page 7-21)

When you use IN or NOT IN to test whether or not a *non_mt_arg_list* is a member of a set of similar lists, then you must use a *non_mt_arg_list_set*. Each *non_mt_arg_list* in the *non_mt_arg_list_set* must match the *non_mt_arg_list* to the left of the condition in number and type of arguments.

Note: You cannot combine this usage with *in_condition_set* as Section , "Using IN and NOT IN as a Set Operation" describes.

Consider the query `Q1` in Example 6–1 and the data stream `S0` in Example 6–2. Stream `S0` has schema `(c1 integer, c2 integer)`. Example 6–3 shows the relation that the query returns. In Q1, the *non_mt_arg_list_set* is `((50,4),(4,5))`. Note that each *non_mt_arg_list* that it contains matches the number and type of arguments in the *non_mt_arg_list* to the left of the condition, `(c1, c2)`.

Example 6–1 S [range C on E] INTERVAL Value: Query

```
<query id="Q1"><![CDATA[
    select c1,c2 from S0[range 1] where (c1,c2) in ((50,4),(4,5))
]]></query>
```

Example 6–2 S [range C on E] INTERVAL Value: Stream Input

```
Timestamp   Tuple
1000        50, 4
2000        30, 6
3000          , 5
4000        22,
h 200000000
```

Example 6–3 S [range C on E] INTERVAL Value: Relation Output

```
Timestamp   Tuple Kind  Tuple
1000:       +           50,4
2000:       -           50,4
```

NOT IN and Null Values

If any item in the list following a `NOT IN` operation evaluates to null, then all stream elements evaluate to `FALSE` or `UNKNOWN`, and no rows are returned. For example, the following statement returns `c1` and `c2` if `c1` is neither 50 nor 30:

```
<query id="check_notin1"><![CDATA[
    select c1,c2 from S0[range 1]
    where
        c1 not in (50, 30)
]]></query>
```

However, the following statement returns no stream elements:

```
<query id="check_notin1"><![CDATA[
    select c1,c2 from S0[range 1]
    where
        c1 not in (50, 30, NULL)
]]></query>
```

The preceding example returns no stream elements because the `WHERE` clause condition evaluates to:

```
c1 != 50 AND c1 != 30 AND c1 != null
```

Because the third condition compares `c1` with a null, it results in an `UNKNOWN`, so the entire expression results in `FALSE` (for stream elements with `c1` equal to 50 or 30). This behavior can easily be overlooked, especially when the NOT IN operator references a view.

Moreover, if a NOT IN condition references a view that returns no stream elements at all, then all stream elements will be returned, as shown in the following example. Since V1 returns no stream elements at all, Q1 will return

```
<view id="V1" schema="c1"><![CDATA[
    IStream(select * from S1[range 10 slide 10] where 1=2)
]]></view>
<view id="V2" schema="c1"><![CDATA[
    IStream(select * from S1[range 10 slide 10] where c1=2)
]]></view>
<query id="Q1"><![CDATA[
    V1 not in V2
]]></query>
```

7

Common Oracle CQL DDL Clauses

This chapter provides a reference to clauses in the data definition language (DDL) in Oracle Continuous Query Language (Oracle CQL).

This chapter includes the following section:

Introduction to Common Oracle CQL DDL Clauses

Oracle CQL supports the following common DDL clauses:

- "xqryargs_list" on page 7-34

For more information on Oracle CQL statements, see Chapter 22, "Oracle CQL Statements".

array_type

Purpose

Use the *array_type* clause to specify an Oracle CQL data cartridge type composed of a sequence of *complex_type* components, all of the same type.

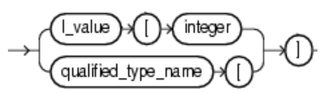
Note: Oracle CQL supports single-dimension arrays only. That is, you can use `java.lang.String[]` but not `java.lang.String[][]`.

Prerequisites

None.

Syntax

***array_type*::=**



(*l-value::=* on page 7-19, *qualified_type_name::=* on page 7-27)

Semantics

Array Declaration

You declare an array type using the *qualified_type_name* of the Oracle CQL data cartridge *complex_type*. Only arrays of *complextype* are supported: you cannot declare an array of Oracle CQL simple types unless there is an equivalent type defined in the Oracle CQL Java data cartridge.

For examples, see:

- "Array Declaration Example: complex_type" on page 7-4
- "Array Declaration Example: Oracle CQL Simple Type" on page 7-4

Array Access

You access a *complex_type* array element by integer index. The index begins at 0 or 1 depending on the data cartridge implementation.

There is no support for the instantiation of new array type instances directly in Oracle CQL at the time you access an array. For example, the following is *not* allowed:

```
SELECT java.lang.String[10] ...
```

For examples, see "Array Access Examples" on page 7-4.

Examples

The following examples illustrate the various semantics that this statement supports:

- "Array Declaration Example: complex_type" on page 7-4
- "Array Declaration Example: Oracle CQL Simple Type" on page 7-4

- "Array Access Examples" on page 7-4

Array Declaration Example: *complex_type*

Example 7–1 shows how to create an event type as a Java class that specifies an event property as an array of Oracle CQL data cartridge complex type `MyClass` defined in package `com.mypackage`.

Example 7–1 Declaring an Oracle CQL Data Cartridge Array in an Event Type

```
package com.myapplication.event;

import java.util.Date;

public final class MarketEvent {
    private final String symbol;
    private final Double price;
    private final com.mypackage.MyClass[] a1;

    public MarketEvent(...) {
        ...
        }
    ...
}
```

Array Declaration Example: Oracle CQL Simple Type

Only arrays of Oracle CQL data cartridge types are supported: you *cannot* declare an array of Oracle CQL simple types.

```
int[] a1
```

However, you can work around this by using the Oracle CQL Java data cartridge and referencing the Java equivalent of the simple type, if one exists:

```
int@java[] a1
```

For more information on the `@` syntax, see *link::=* on page 5-19.

Array Access Examples

Example 7–2 shows how to register the following queries that use Oracle CQL data cartridge complex type array access:

- View `v1` accesses the third element of the array `a1`. This array contains instances of Oracle CQL data cartridge complex type `com.mypackage.MyClass` as defined in Example 7–1.
- Query `q1` accesses the first element of the array `field1`. This array is defined on Oracle CQL data cartridge complex type `a1`.

Example 7–2 Accessing an Oracle CQL Data Cartridge Array in an Oracle CQL Query

```
<view id="v1" schema="symbol price a1"><![CDATA[
    IStream(select symbol, price, a1[3] from S1[range 10 slide 10])
]]></view>
<query id="q1"><![CDATA[
    SELECT a1.field1[1] …
]]></query>
```

attr

Purpose

Use the `attr` clause to specify a stream element or pseudocolumn.

You can use the `attr` clause in the following Oracle CQL statements:

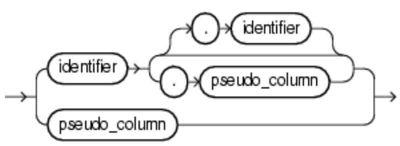
- *arith_expr::=* on page 5-6
- *order_expr::=* on page 5-23
- *non_mt_attr_list::=* on page 7-22

Prerequisites

None.

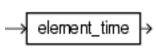
Syntax

attr::=



(*identifier::=* on page 7-17, *pseudo_column::=* on page 7-5)

pseudo_column::=



Semantics

identifier

Specify the identifier of the stream element.

You can specify

- `StreamOrViewName.ElementName`
- `ElementName`
- `CorrelationName.PseudoColumn`
- `PseudoColumn`

For examples, see "Examples" on page 7-6.

For syntax, see *identifier::=* on page 7-17 (parent: *attr::=* on page 7-5).

pseudo_column

Specify the timestamp associated with a specific stream element, all stream elements, or the stream element associated with a correlation name in a `MATCH_RECOGNIZE` clause.

For examples, see:

- "Examples" on page 7-6

- Section , "Using ELEMENT_TIME With SELECT"
- Section , "Using ELEMENT_TIME With GROUP BY"
- Section , "Using ELEMENT_TIME With PATTERN"

For more information, see Chapter 3, "Pseudocolumns".

For syntax, see *pseudo_column::=* on page 7-5 (parent: *attr::=* on page 7-5).

Examples

Given the stream that Example 7–3 shows, valid attribute clauses are:

- `ItemTempStream.temp`
- `temp`
- `B.element_time`
- `element_time`

Example 7–3 attr Clause

```
<view id="ItemTempStream" schema="itemId temp"><![CDATA[
    IStream(select * from ItemTemp)
]]></view>
<query id="detectPerish"><![CDATA[
  select its.itemId
  from ItemTempStream MATCH_RECOGNIZE (
      PARTITION BY itemId
      MEASURES A.itemId as itemId
      PATTERN (A B* C)
      DEFINE
          A  AS  (A.temp >= 25),
          B  AS  ((B.temp >= 25) and (to_timestamp(B.element_time) - to_timestamp(A.element_time) < INTERVAL "0
00:00:05.00" DAY TO SECOND)),
          C  AS  (to_timestamp(C.element_time) - to_timestamp(A.element_time) >= INTERVAL "0 00:00:05.00" DAY TO
SECOND)
  ) as its
]]></query>
```

attrspec

Purpose

Use the `attrspec` clause to define the identifier and datatype of a stream element.

You can use the `attrspec` clause in the following Oracle CQL statements:

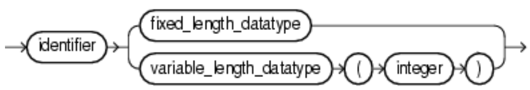
- *non_mt_attrspec_list::=* on page 7-24

Prerequisites

None.

Syntax

attrspec::=



(*identifier::=* on page 7-17, *fixed_length_datatype::=* on page 2-2, *variable_length_datatype::=* on page 2-2)

Semantics

identifier

Specify the identifier of the stream element.

For syntax, see *identifier::=* on page 7-17 (parent: *attrspec::=* on page 7-7).

fixed_length_datatype

Specify the stream element datatype as a fixed-length datatype.

For syntax, see *fixed_length_datatype::=* on page 2-2 (parent: *attrspec::=* on page 7-7).

variable_length_datatype

Specify the stream element datatype as a variable-length datatype.

For syntax, see *variable_length_datatype::=* on page 2-2 (parent: *attrspec::=* on page 7-7).

integer

Specify the length of the variable-length datatype.

For syntax, see *attrspec::=* on page 7-7.

complex_type

Purpose

Use the `complex_type` clause to specify an Oracle CQL data cartridge type that defines:

- member fields (static or instance)
- member methods (static or instance)
- constructors

The type of a field, and the return type and parameter list of a method may be complex types or simple types.

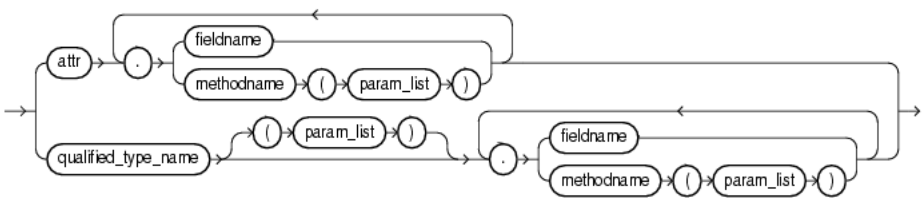
A complex type is identified by its qualified type name (set of identifiers separated by a period ".") and the optional name of the data cartridge to which it belongs (see *link::=* on page 5-19). If you do not specify a link name, then Oracle Event Processing assumes that the complex type is a Java class (that is, Oracle Event Processing assumes that the complex type belongs to the Java data cartridge).

Prerequisites

The Oracle CQL data cartridge that provides the complextype must be loaded by Oracle Event Processing server at runtime.

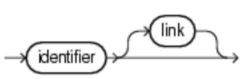
Syntax

complex_type::=



(*attr::=* on page 7-5, *fieldname::=* on page 7-8, *methodname::=* on page 7-20, *param_list::=* on page 7-26, *qualified_type_name::=* on page 7-27)

fieldname::=



(*identifier::=* on page 7-17, *link::=* on page 5-19)

Semantics

fieldname

Use the `fieldname` clause to specify a static field of an Oracle CQL data cartridge complex type.

Syntax: *fieldname::=* on page 7-8 (parent: *complex_type::=* on page 7-8).

Field Access

You cannot use a complex type `l-value` generated in expressions within an `ORDER BY` clause. Currently, only expressions within a `SELECT` clause and a `WHERE` clause may generate a complex type `l-value`.

You may access only a static field using *`qualified_type_name`*. To access a non-static field, you must first instantiate the complex type (see "Constructor Invocation" on page 7-9).

For examples, see "Field Access Examples: complex_type" on page 7-9.

Method Access

Accessing complex type setter methods may cause side effects. Side effects decrease the opportunities for concurrency and sharing. For example, if you invoke a setter method and change the value of a view attribute (such as an event property) shared by different queries that depend on the view, then the query results may change as a side effect of your method invocation.

You may access only a static method using *`qualified_type_name`*. To access a non-static field, you must first instantiate the complex type (see "Constructor Invocation" on page 7-9).

For examples, see "Method Access Examples: complex_type" on page 7-10.

Constructor Invocation

You may access only a static fields and static methods using *`qualified_type_name`*. To access a non-static field or non-static method, you must first instantiate the complex type by invoking one of its constructors.

For examples, see "Constructor Invocation Examples: complex_type" on page 7-10.

Examples

The following examples illustrate the various semantics that this statement supports:

- "Field Access Examples: complex_type" on page 7-9
- "Method Access Examples: complex_type" on page 7-10
- "Constructor Invocation Examples: complex_type" on page 7-10

Field Access Examples: *complex_type*

Example 7–4 shows how to register the following queries that use Oracle CQL data cartridge complex type field access:

- Query `q1` accesses field `myField` from Oracle CQL data cartridge complex type `a1`.
- Query `q2` accesses field `myField` defined on the Oracle CQL data cartridge complex type returned by the method `function-returning-object`. For more information on method access, see "Method Access" on page 7-9.
- Query `q3` accesses field `myNestedField` defined on the Oracle CQL data cartridge complex type `myField` which is defined on Oracle CQL data cartridge complex type `a1`.
- Query `q4` accesses the static field `myStaticField` defined in the class `MyType` in package `com.myPackage`. Note that a link (`@myCartridge`) is necessary in the case of a static field.

Example 7–4 Data Cartridge Field Access

```
<query id="q1"><![CDATA[
    SELECT a1.myField …
]]></query>
<query id="q2"><![CDATA[
    SELECT function-returning-object().myField …
]]></query>
<query id="q3"><![CDATA[
    SELECT a1.myField.myNestedField …
]]></query>
<query id="q4"><![CDATA[
    SELECT com.myPackage.MyType.myStaticField@myCartridge …
]]></query>
```

Method Access Examples: *complex_type*

Example 7–5 shows how to register the following queries that use Oracle CQL data cartridge complex type method access:

- Query `q1` accesses method `myMethod` defined on Oracle CQL data cartridge complex type `a1`. This query accesses the method with an empty parameter list.
- Query `q2` accesses method `myMethod` defined on Oracle CQL data cartridge complex type `a1` with a different signature than in query `q1`. In this case, the query accesses the method with a three-argument parameter list.
- Query `q3` accesses static method `myStaticMethod` defined on Oracle CQL data cartridge complex type `MyType`. This query accesses the method with a single parameter. Note that a link (`@myCartridge`) is necessary in the case of a static method.

Example 7–5 Data Cartridge Method Access

```
<query id="q1"><![CDATA[
    SELECT a1.myMethod() …
]]></query>
<query id="q2"><![CDATA[
    SELECT a1.myMethod(a2, “foo”, 10) …
]]></query>
<query id="q3"><![CDATA[
    SELECT myPackage.MyType.myStaticMethod@myCartridge("foo") …
]]></query>
```

Constructor Invocation Examples: *complex_type*

Example 7–6 shows how to register the following queries that use Oracle CQL data cartridge complex type constructor invocation:

- Query `q1` invokes the constructor `String` defined in package `java.lang`. In this case, the query invokes the constructor with an empty argument list.
- Query `q2` invokes the constructor `String` defined in package `java.lang`. In this case, the query invokes the constructor with a single argument parameter list and invokes the non-static method `substring` defined on the returned `String` instance.

Example 7–6 Data Cartridge Constructor Invocation

```
<query id="q1"><![CDATA[
    SELECT java.lang.String() …
]]></query>
<query id="q2"><![CDATA[
    SELECT java.lang.String(“food”).substring(0,1) …
]]></query>
```

const_bigint

Purpose

Use the `const_bigint` clause to specify a big integer numeric literal.

You can use the `const_bigint` clause in the following Oracle CQL statements:

- *func_expr::=* on page 5-15

For more information, see Section , "Numeric Literals".

Prerequisites

None.

Syntax

***const_bigint*::=**

const_int

Purpose

Use the `const_int` clause to specify an integer numeric literal.

You can use the `const_int` clause in the following Oracle CQL statements:

- *func_expr::=* on page 5-15
- *order_expr::=* on page 5-23

For more information, see Section , "Numeric Literals".

Prerequisites

None.

Syntax

***const_int*::=**

const_string

Purpose

Use the `const_string` clause to specify a constant `String` text literal.

You can use the `const_string` clause in the following Oracle CQL statements:

- *func_expr::=* on page 5-15
- *order_expr::=* on page 5-23
- *condition::=* on page 6-4
- *interval_value::=* on page 7-14
- *xmltable_clause::=* on page 22-6
- *xtbl_col::=* on page 22-7

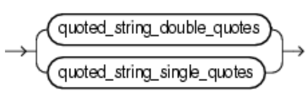
For more information, see Section , "Text Literals".

Prerequisites

None.

Syntax

const_string::=



const_value

Purpose

Use the `const_value` clause to specify a literal value.

You can use the `const_value` clause in the following Oracle CQL statements:

- *arith_expr::=* on page 5-6
- *condition::=* on page 6-4

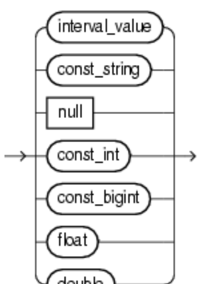
For more information, see Section , "Literals".

Prerequisites

None.

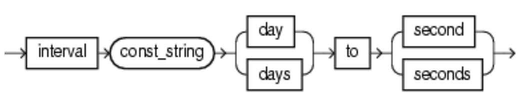
Syntax

const_value::=



(*interval_value::=* on page 7-14, *const_string::=* on page 7-13, *const_int::=* on page 7-12, *const_bigint::=* on page 7-11)

interval_value::=



(*const_string::=* on page 7-13)

Semantics

interval_value

Specify an interval constant value as a quoted string. For example:

```
INTERVAL '4 5:12:10.222' DAY TO SECOND(3)
```

For more information, see Section , "Interval Literals".

For syntax, see *interval_value::=* on page 7-14 (parent: *const_value::=* on page 7-14).

const_string

Specify a quoted `String` constant value.

For more information, see Section , "Text Literals".

For syntax, see *const_string::=* on page 7-13 (parent: *interval_value::=* on page 7-14 and *const_value::=* on page 7-14).

null

Specify a null constant value.

For more information, see Section , "Nulls".

const_int

Specify an `int` constant value.

For more information, see Section , "Numeric Literals".

bigint

Specify a `bigint` constant value.

For more information, see Section , "Numeric Literals".

float

Specify a float constant value.

For more information, see Section , "Numeric Literals".

identifier

Purpose

Use the `identifier` clause to reference an existing Oracle CQL schema object.

You can use the `identifier` clause in the following Oracle CQL statements:

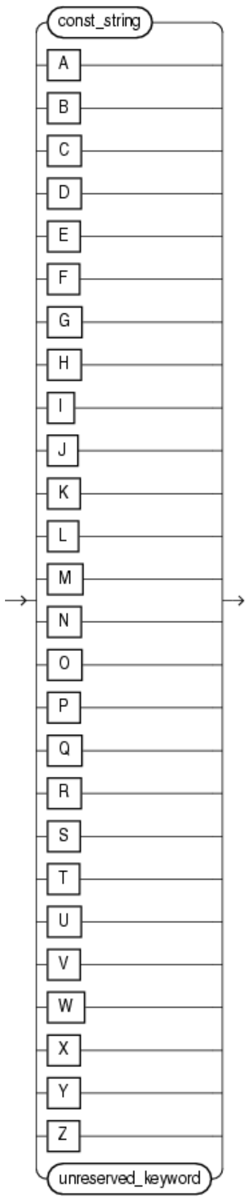
- *binary::=* on page 22-6
- *aggr_expr::=* on page 5-4
- *func_expr::=* on page 5-15
- *attr::=* on page 7-5
- *attrspec::=* on page 7-7
- *query_ref::=* on page 7-29
- *non_mt_attrname_list::=* on page 7-23
- *relation_variable::=* on page 22-4
- *measure_column::=* on page 21-10
- "Query" on page 22-2
- *projterm::=* on page 22-3
- "View" on page 22-29

Prerequisites

The schema object must already exist.

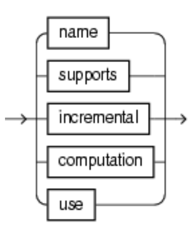
Syntax

***identifier*::=**



(*const_string::=* on page 7-13, *unreserved_keyword::=* on page 7-17)

***unreserved_keyword*::=**



Semantics

const_string

Specify the identifier as a String.

For more information, see Section , "Schema Object Naming Rules".

For syntax, see *identifier::=* on page 7-17.

[A-Z]

Specify the identifier as a single uppercase letter.

For syntax, see *identifier::=* on page 7-17.

unreserved_keyword

These are names that you may use as identifiers.

For more information, see:

- "reserved_keyword" on page 7-18
- Section , "Schema Object Naming Rules"

For syntax, see *unreserved_keyword::=* on page 7-17 (parent: *identifier::=* on page 7-17).

reserved_keyword

These are names that you may not use as identifiers, because they are reserved keywords: `add`, `aggregate`, `all`, `alter`, `and`, `application`, `as`, `asc`, `avg`, `between`, `bigint`, `binding`, `binjoin`, `binstreamjoin`, `boolean`, `by`, `byte`, `callout`, `case`, `char`, `clear`, `columns`, `constraint`, `content`, `count`, `create`, `day`, `days`, `decode`, `define`, `derived`, `desc`, `destination`, `disable`, `distinct`, `document`, `double`, `drop`, `dstream`, `dump`, `duration`, `duration`, `element_time`, `else`, `enable`, `end`, `evalname`, `event`, `events`, `except`, `external`, `false`, `first`, `float`, `from`, `function`, `group`, `groupaggr`, `having`, `heartbeat`, `hour`, `hours`, `identified`, `implement`, `in`, `include`, `index`, `instance`, `int`, `integer`, `intersect`, `interval`, `is`, `istream`, `java`, `key`, `language`, `last`, `level`, `like`, `lineage`, `logging`, `match_recognize`, `matches`, `max`, `measures`, `metadata_query`, `metadata_system`, `metadata_table`, `metadata_userfunc`, `metadata_view`, `metadata_window`, `microsecond`, `microseconds`, `millisecond`, `milliseconds`, `min`, `minus`, `minute`, `minutes`, `monitoring`, `multiples`, `nanosecond`, `nanoseconds`, `not`, `now`, `null`, `nulls`, `object`, `of`, `on`, `operator`, `or`, `order`, `orderbytop`, `output`, `partition`, `partitionwin`, `partnwin`, `passing`, `path`, `pattern`, `patternstrm`, `patternstrmb`, `prev`, `primary`, `project`, `push`, `query`, `queue`, `range`, `rangewin`, `real`, `register`, `relation`, `relsrc`, `remove`, `return`, `returning`, `rows`, `rowwin`, `rstream`, `run`, `run_time`, `sched_name`, `sched_threaded`, `schema`, `second`, `seconds`, `select`, `semantics`, `set`, `silent`, `sink`, `slide`, `source`, `spill`, `start`, `stop`, `storage`, `store`, `stream`, `strmsrc`, `subset`, `sum`, `synopsis`, `system`, `systemstate`, `then`, `time`, `time_slice`, `timeout`, `timer`, `timestamp`, `timestamped`, `to`, `true`, `trusted`, `type`, `unbounded`, `union`, `update`, `using`, `value`, `view`, `viewrelnsrc`, `viewstrmsrc`, `wellformed`, `when`, `where`, `window`, `xmlagg`, `xmlattributes`, `xmlcolattval`, `xmlconcat`, `xmldata`, `xmlelement`, `xmlexists`, `xmlforest`, `xmlparse`, `xmlquery`, `xmltable`, `xmltype`, or `xor`.

l-value

Purpose

Use the `l-value` clause to specify an integer literal.

You can use the `l-value` clause in the following Oracle CQL data cartridge statements:

- *array_type::=* on page 7-3

Prerequisites

None.

Syntax

***l-value*::=**

(*integer::=* on page 2-8)

methodname

Purpose

Use the `methodname` clause to specify a method of an Oracle CQL data cartridge complex type.

You can use the `methodname` clause in the following Oracle CQL data cartridge statements:

- *complex_type::=* on page 7-8

Prerequisites

None.

Syntax

methodname::=

(*identifier::=* on page 7-17, *link::=* on page 5-19)

non_mt_arg_list

Purpose

Use the `non_mt_arg_list` clause to specify one or more arguments as arithmetic expressions involving stream elements. To specify one or more arguments as stream elements directly, see *non_mt_attr_list::=* on page 7-22.

You can use the `non_mt_arg_list` clause in the following Oracle CQL statements:

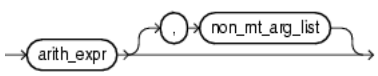
- *decode::=* on page 5-13
- *func_expr::=* on page 5-15
- *condition::=* on page 6-4
- *non_mt_arg_list_set::=* on page 6-9

Prerequisites

If any stream elements are referenced, the stream must already exist.

Syntax

***non_mt_arg_list*::=**



(*arith_expr::=* on page 5-6)

Semantics

arith_expr

Specify the arithmetic expression that resolves to the argument value.

non_mt_attr_list

Purpose

Use the *non_mt_attr_list* clause to specify one or more arguments as stream elements directly. To specify one or more arguments as arithmetic expressions involving stream elements, see *non_mt_arg_list::=* on page 7-21.

You can use the *non_mt_attr_list* clause in the following Oracle CQL statements:

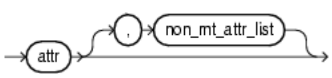
- *pattern_partition_clause::=* on page 21-17
- *window_type::=* on page 22-4
- *opt_group_by_clause::=* on page 22-5

Prerequisites

If any stream elements are referenced, the stream must already exist.

Syntax

non_mt_attr_list::=



(*attr::=* on page 7-5)

Semantics

attr

Specify the argument as a stream element directly.

non_mt_attrname_list

Purpose

Use the `non_mt_attrname_list` clause to one or more stream elements by name.

You can use the `non_mt_attrname_list` clause in the following Oracle CQL statements:

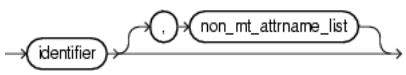
- "View" on page 22-29

Prerequisites

If any stream elements are referenced, the stream must already exist.

Syntax

non_mt_attrname_list::=



(*identifier::=* on page 7-17)

Semantics

identifier

Specify the stream element by name.

non_mt_attrspec_list

Purpose

Use the *non_mt_attrspec_list* clause to specify one or more attribute specifications that define the identifier and datatype of stream elements.

You can use the *non_mt_attrspec_list* clause in the following Oracle CQL statements:

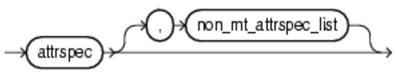
- "View" on page 22-29

Prerequisites

If any stream elements are referenced, the stream must already exist.

Syntax

non_mt_attrspec_list::=



(*attrspec::=* on page 7-7)

Semantics

attrspec

Specify the attribute identifier and datatype.

non_mt_cond_list

Purpose

Use the *non_mt_cond_list* clause to specify one or more conditions using any combination of logical operators AND, OR, XOR and NOT.

You can use the *non_mt_cond_list* clause in the following Oracle CQL statements:

- *correlation_name_definition::=* on page 21-14
- *searched_case::=* on page 5-9
- *opt_where_clause::=* on page 22-4
- *opt_having_clause::=* on page 22-5

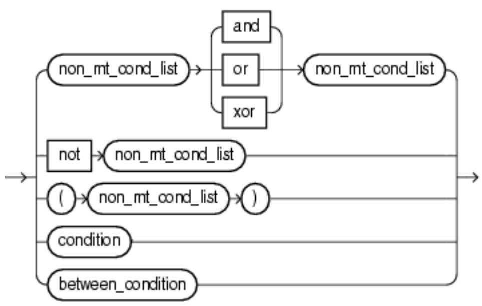
For more information, see Chapter 6, "Conditions".

Prerequisites

None.

Syntax

non_mt_cond_list::=



(*non_mt_cond_list::=* on page 7-25, *condition::=* on page 6-4, *between_condition::=* on page 6-8)

Semantics

condition

Specify a comparison condition.

For more information, see Section , "Comparison Conditions".

For syntax, see *condition::=* on page 6-4 (parent: *non_mt_cond_list::=* on page 7-25).

between_condition

Specify a condition that tests for inclusion in a range.

For more information, see Section , "Range Conditions".

For syntax, see *between_condition::=* on page 6-8 (parent: *non_mt_cond_list::=* on page 7-25).

param_list

Purpose

Use the `param_list` clause to specify a comma-separated list of zero or more parameters, similar to a function parameter list, for an Oracle CQL data cartridge complex type method or constructor.

You can use the `param_list` clause in the following Oracle CQL data cartridge statements:

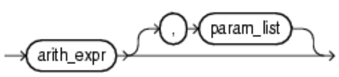
- *complex_type::=* on page 7-8
- *link::=* on page 5-19

Prerequisites

None.

Syntax

param_list::=



(*arith_expr::=* on page 5-6)

qualified_type_name

Purpose

Use the *qualified_type_name* clause to specify a fully specified type name of an Oracle CQL data cartridge complex type, for example `java.lang.String`. Use the *qualified_type_name* when invoking Oracle CQL data cartridge static fields, static methods, or constructors.

There is no default package. For example, using the Java data cartridge, you must specify `java.lang` when referencing the class `String`. To be able to distinguish a reserved word from a qualified type, all qualified types must have at least two identifiers, that is, there must be at least one period (`.`) in a qualified name.

You can use the *qualified_type_name* clause in the following Oracle CQL data cartridge statements:

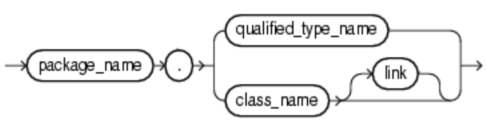
- *complex_type::=* on page 7-8
- *array_type::=* on page 7-3

Prerequisites

None.

Syntax

qualified_type_name::=



(*arith_expr::=* on page 5-6, *package_name::=* on page 7-27, *class_name::=* on page 7-27, *link::=* on page 5-19)

package_name::=

(*identifier::=* on page 7-17)

class_name::=

(*arith_expr::=* on page 5-6)

Semantics

package_name

Use the *package_name* clause to specify the name of an Oracle CQL data cartridge package.

Syntax: *package_name::=* on page 7-27 (parent: *qualified_type_name::=* on page 7-27)

class_name

Use the *class_name* clause to specify the name of an Oracle CQL data cartridge Class.

Syntax: *class_name::=* on page 7-27 (parent: *qualified_type_name::=* on page 7-27)

query_ref

Purpose

Use the `query_ref` clause to reference an existing Oracle CQL query by name.

You can reference a Oracle CQL query in the following Oracle CQL statements:

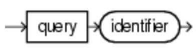
- "View" on page 22-29

Prerequisites

The query must already exist (see "Query" on page 22-2).

Syntax

query_ref::=



(*identifier::=* on page 7-17)

Semantics

identifier

Specify the name of the query. This is the name you use to reference the query in subsequent Oracle CQL statements.

time_spec

Purpose

Use the `time_spec` clause to define a time duration in days, hours, minutes, seconds, milliseconds, or nanoseconds.

Default: if units are not specified, Oracle Event Processing assumes `[second|seconds]`.

You can use the `time_spec` clause in the following Oracle CQL statements:

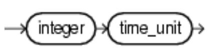
- *window_type::=* on page 22-4
- *duration_clause::=* on page 21-22

Prerequisites

None.

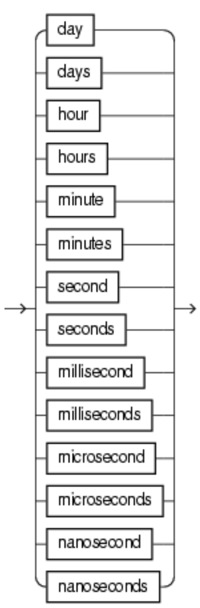
Syntax

time_spec::=



(*time_unit::=* on page 7-30)

time_unit::=



Semantics

integer

Specify the number of time units.

time_unit

Specify the unit of time.

xml_attribute_list

Purpose

Use the `xml_attribute_list` clause to specify one or more XML attributes.

You can use the `xml_attribute_list` clause in the following Oracle CQL statements:

- "xmlelement_expr" on page 5-28

Prerequisites

If any stream elements are referenced, the stream must already exist.

Syntax

xml_attribute_list::=

(*xml_attr_list::=* on page 7-33)

Semantics

xml_attr_list

Specify one or more XML attributes as Example 7–7 shows.

Example 7–7 xml_attr_list

```
<query id="tkdata51_q1"><![CDATA[
    select XMLELEMENT(NAME "S0", XMLATTRIBUTES(tkdata51_S0.c1 as "C1", tkdata51_S0.c2 as
"C2"),
        XMLELEMENT(NAME "c1_plus_c2", c1+c2), XMLELEMENT(NAME "c2_plus_10", c2+10.0)) from
tkdata51_S0 [range 1]
]]>
</query>
```

For syntax, see *xml_attr_list::=* on page 7-33 (parent: *xml_attribute_list::=* on page 7-32).

xml_attr_list

Purpose

Use the `xml_attr_list` clause to specify one or more XML attributes..

You can use the `xml_attr_list` clause in the following Oracle CQL statements:

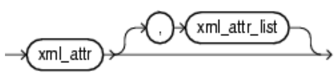
- "xml_attribute_list" on page 7-32
- "xmlforest_expr" on page 5-30
- "xml_agg_expr" on page 5-24

Prerequisites

If any stream elements are referenced, the stream must already exist.

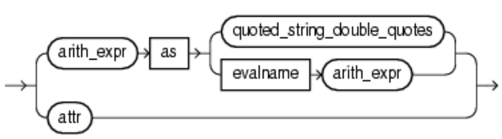
Syntax

xml_attr_list::=



(*xml_attr::=* on page 7-33)

xml_attr::=



(*const_string::=* on page 7-13, *arith_expr::=* on page 5-6, *attr::=* on page 7-5)

Semantics

xml_attr

Specify an XML attribute.

For syntax, see *xml_attr::=* on page 7-33 (parent: *xml_attr_list::=* on page 7-33).

xqryargs_list

Purpose

Use the *xqryargs_list* clause to specify one or more arguments to an XML query.

You can use the *non_mt_arg_list* clause in the following Oracle CQL statements:

- "xmlexists" on page 8-27
- "xmlquery" on page 8-29
- *func_expr::=* on page 5-15
- *xmltable_clause::=* on page 22-6

Prerequisites

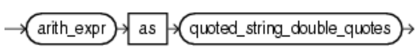
If any stream elements are referenced, the stream must already exist.

Syntax

xqryargs_list::=

(*xqryarg::=* on page 7-34)

xqryarg::=



(*const_string::=* on page 7-13, *arith_expr::=* on page 5-6)

Semantics

xqryarg

A clause that binds a stream element value to an XQuery variable or XPath operator.

You can bind any arithmetic expression that involves one or more stream elements (see *arith_expr::=* on page 5-6) to either a variable in a given XQuery or an XPath operator such as "." as a quoted string.

For syntax, see *xqryarg::=* on page 7-34 (parent: *xqryargs_list::=* on page 7-34).

Part II

Functions

This part contains the following chapters:

8

Built-In Single-Row Functions

This chapter provides a reference to single-row functions in Oracle Continuous Query Language (Oracle CQL). Single-row functions return a single result row for every row of a queried stream or view.

For more information, see Section , "Functions".

This chapter includes the following section:

- Introduction to Oracle CQL Built-In Single-Row Functions

Introduction to Oracle CQL Built-In Single-Row Functions

Table 8–1 lists the built-in single-row functions that Oracle CQL provides.

Table 8–1 Oracle CQL Built-in Single-Row Functions

Type	Function
Character (returning character values)	■ concat
Character (returning numeric values)	■ length
Datetime	■ systimestamp
Conversion	■ to_bigint ■ to_boolean ■ to_char ■ to_double ■ to_float ■ to_timestamp
XML and SQLX	■ xmlcomment ■ xmlconcat ■ xmlexists ■ xmlquery
Encoding and Decoding	■ hextoraw ■ rawtohex
Null-related	■ nvl
Pattern Matching	■ lk ■ prev

Note: Built-in function names are case sensitive and you must use them in the case shown (in lower case).

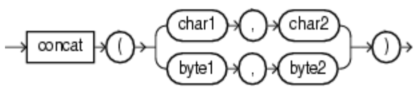
Note: In stream input examples, lines beginning with `h` (such as `h 3800`) are heartbeat input tuples. These inform Oracle Event Processing that no further input will have a timestamp lesser than the heartbeat value.

For more information, see:

- Section , "Functions"
- Section , "Datatypes"

concat

Syntax



Purpose

`concat` returns *`char1`* concatenated with *`char2`* as a `char[]` or *`byte1`* concatenated with *`byte2`* as a `byte[]`. The `char` returned is in the same character set as *`char1`*. Its datatype depends on the datatypes of the arguments.

Using `concat`, you can concatenate any combination of character, byte, and numeric datatypes. The `concat` performs automatic numeric to string conversion.

This function is equivalent to the concatenation operator (||). For more information, see "Concatenation Operator" on page 4-4.

To concatenate `xmltype` arguments, use `xmlconcat`. For more information, see "xmlconcat" on page 8-25.

Examples

concat Function

Consider the query `chr_concat` in Example 8–1 and data stream `S4` in Example 8–2. Stream `S4` has schema `(c1 char(10))`. The query returns the relation in Example 8–3.

Example 8–1 concat Function Query

```
<query id="chr_concat"><![CDATA[
    select
        concat(c1,c1),
        concat("abc",c1),
        concat(c1,"abc")
    from
        S4[range 5]
]]></query>
```

Example 8–2 concat Function Stream Input

```
Timestamp   Tuple
 1000
 2000       hi
 8000       hi1
 9000
15000       xyz
h 200000000
```

Example 8–3 concat Function Relation Output

```
Timestamp   Tuple Kind  Tuple
 1000:      +           ,abc,abc
 2000:      +           hihi,abchi,hiabc
 6000:      -           ,abc,abc
 7000:      -           hihi,abchi,hiabc
 8000:      +           hi1hi1,abchi1,hi1abc
 9000:      +           ,abc,abc
13000:      -           hi1hi1,abchi1,hi1abc
```

```
14000:      -           ,abc,abc
15000:      +           xyzxyz,abcxyz,xyzabc
20000:      -           xyzxyz,abcxyz,xyzabc
```

Concatenation Operator (||)

Consider the query `q264` in Example 8–4 and the data stream `S10` in Example 8–5. Stream `S10` has schema `(c1 integer, c2 char(10))`. The query returns the relation in Example 8–6.

Example 8–4 Concatenation Operator (||) Query

```
<query id="q264"><![CDATA[
    select
        c2 || "xyz"
    from
        S10
]]></query>
```

Example 8–5 Concatenation Operator (||) Stream Input

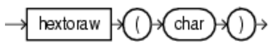
```
Timestamp   Tuple
1           1,abc
2           2,ab
3           3,abc
4           4,a
h 200000000
```

Example 8–6 Concatenation Operator (||) Relation Output

```
Timestamp   Tuple Kind  Tuple
1:          +           abcxyz
2:          +           abxyz
3:          +           abcxyz
4:          +           axyz
```

hextoraw

Syntax



Purpose

`hextoraw` converts *`char`* containing hexadecimal digits in the `char` character set to a raw value.

> **See Also:** "rawtohex" on page 8-14

Examples

Consider the query `q6` in Example 8–7 and the data stream `SinpByte1` in Example 8–8. Stream `SinpByte1` has schema `(c1 byte(10), c2 integer)`. The query returns the relation in Example 8–9.

Example 8–7 hextoraw Function Query

```
<query id="q6"><![CDATA[
    select * from SByt[range 2]
    where
        bytTest(c2) between hextoraw("5200") and hextoraw("5600")
]]></query>
```

Example 8–8 hextoraw Function Stream Input

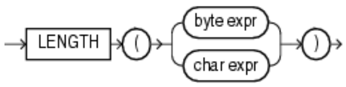
```
Timestamp   Tuple
1000        1,"51c1"
2000        2,"52"
3000        3,"53aa"
4000        4,"5"
5000         ,"55ef"
6000        6,
h 8000
h 200000000
```

Example 8–9 hextoraw Function Relation Output

```
Timestamp   Tuple Kind  Tuple
2000        +           2,"52"
3000        +           3,"53aa"
4000        -           2,"52"
5000        -           3,"53aa"
5000        +            ,"55ef"
7000        -            ,"55ef"
```

length

Syntax



Purpose

The `length` function returns the length of its *`char`* or *`byte`* expression as an `int`. `length` calculates length using characters as defined by the input character set.

For a *`char`* expression, the length includes all trailing blanks. If the expression is null, this function returns null.

Examples

Consider the query `chr_len` in Example 8–10 and the data stream `S2` in Example 8–11. Stream `S2` has schema `(c1 integer, c2 integer)`. The query returns the relation that Example 8–12.

Example 8–10 length Function Query

```
<query id="chr_len"><![CDATA[
    select length(c1) from S4[range 5]
]]></query>
```

Example 8–11 length Function Stream Input

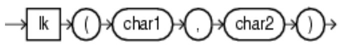
```
Timestamp   Tuple
 1000
 2000       hi
 8000       hi1
 9000
15000       xyz
h 200000000
```

Example 8–12 length Function Relation Output

```
Timestamp   Tuple Kind  Tuple
 1000:      +           0
 2000:      +           2
 6000:      -           0
 7000:      -           2
 8000:      +           3
 9000:      +           0
13000:      -           3
14000:      -           0
15000:      +           3
20000:      -           3
```

lk

Syntax



Purpose

`lk` boolean `true` if *`char1`* matches the regular expression *`char2`*, otherwise it returns `false`.

This function is equivalent to the `LIKE` condition. For more information, see Section , "LIKE Condition".

Examples

Consider the query `q291` in Example 8–13 and the data stream `SLk1` in Example 8–14. Stream `SLk1` has schema `(first1 char(20), last1 char(20))`. The query returns the relation in Example 8–15.

Example 8–13 lk Function Query

```
<query id="q291"><![CDATA[
    select * from SLk1
    where
        lk(first1,"^Ste(v|ph)en$") = true
]]></query>
```

Example 8–14 lk Function Stream Input

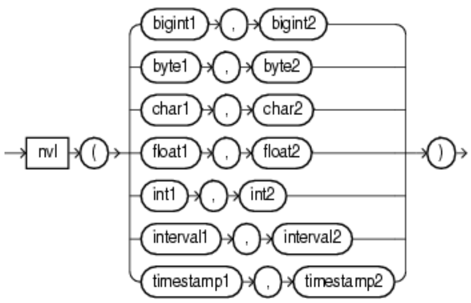
```
Timestamp   Tuple
1           Steven,King
2           Sten,Harley
3           Stephen,Stiles
4           Steven,Markles
h 200000000
```

Example 8–15 lk Function Relation Output

```
Timestamp   Tuple Kind  Tuple
1:          +           Steven,King
3:          +           Stephen,Stiles
4:          +           Steven,Markles
```

nvl

Syntax



Purpose

`nvl` lets you replace null (returned as a blank) with a string in the results of a query. If *`expr1`* is null, then `NVL` returns *`expr2`*. If *`expr1`* is not null, then `NVL` returns *`expr1`*.

The arguments *`expr1`* and *`expr2`* can have any datatype. If their datatypes are different, then Oracle Event Processing implicitly converts one to the other. If they cannot be converted implicitly, Oracle Event Processing returns an error. The implicit conversion is implemented as follows:

- If *`expr1`* is character data, then Oracle Event Processing converts *`expr2`* to character data before comparing them and returns `VARCHAR2` in the character set of *`expr1`*.
- If *`expr1`* is numeric, then Oracle Event Processing determines which argument has the highest numeric precedence, implicitly converts the other argument to that datatype, and returns that datatype.

Examples

Consider the query `q281` in Example 8–16 and the data stream `SNVL` in Example 8–17. Stream `SNVL` has schema `(c1 char(20), c2 integer)`. The query returns the relation in Example 8–18.

Example 8–16 nvl Function Query

```
<query id="q281"><![CDATA[
    select nvl(c1,"abcd") from SNVL
]]></query>
```

Example 8–17 nvl Function Stream Input

```
Timestamp   Tuple
1              ,1
2            ab,2
3           abc,3
4              ,4
h 200000000
```

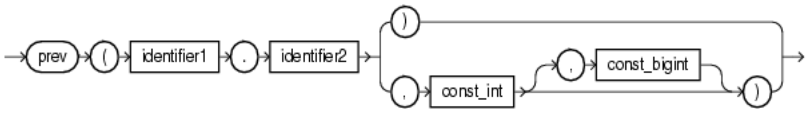
Example 8–18 nvl Function Relation Output

```
Timestamp   Tuple Kind  Tuple
1:          +           abcd
2:          +           ab
3:          +           abc
```

```
4:          +          abcd
```

prev

Syntax



Purpose

`prev` returns the value of the stream attribute (function argument `identifier2`) of the event that occurred previous to the current event and which belongs to the partition to which the current event belongs. It evaluates to `NULL` if there is no such previous event.

The type of the specified stream element may be any of:

- `integer`
- `bigint`
- `float`
- `double`
- `byte`
- `char`
- `interval`
- `timestamp`

The return type of this function depends on the type of the specified stream attribute (function argument `identifier2`).

This function takes the following arguments:

- prev(identifier1.identifier2)
- prev(identifier1.identifier2, const_int)
- prev(identifier1.identifier2, const_int, const_bigint)

Where:

- `identifier1.identifier2` : `identifier1` is the name of a correlation variable used in the `PATTERN` clause and defined in the `DEFINE` clause and `identifier2` is the name of a stream attribute whose value in the previous event should be returned by `prev`.
- `const_int`: if this argument has a value *n*, then it specifies the *n*th previous event in the partition to which the current event belongs. The value of the attribute (specified in argument `identifier2`) in the *n*th previous event will be returned if such an event exists, `NULL` otherwise.
- `const_bigint`: specifies a time range duration in nanoseconds and should be used if you are interested in previous events that occurred only within a certain range of time before the current event.

If the query uses `PARTITION BY` with the `prev` function and input data will include many different partition key values (meaning many partitions), then total memory consumed for storing the previous event(s) per partition could be large. In such cases, consider using the time range duration (the third argument, possibly with a large range value) so that this memory can be reclaimed wherever possible.

See Also:

- "first" on page 9-7
- "last" on page 9-9
- "func_expr" on page 5-15
- *pattern_recognition_clause::=* on page 21-2

Examples

prev(identifier1.identifier2)

Consider query `q2` in Example 8–19 and the data stream `S1` in Example 8–20. Stream `S1` has schema `(c1 integer)`. This example defines pattern `A` as `A.c1 = prev(A.c1)`. In other words, pattern `A` matches when the value of `c1` in the current stream element matches the value of `c1` in the stream element immediately before the current stream element. The query returns the stream in Example 8–21.

Example 8–19 prev(identifier1.identifier2) Function Query

```
<query id="q2"><![CDATA[
    select
        T.Ac1,
        T.Cc1
    from
        S1
    MATCH_RECOGNIZE (
        MEASURES
            A.c1 as Ac1,
            C.c1 as Cc1
        PATTERN(A B+ C)
        DEFINE
            A as A.c1 = prev(A.c1),
            B as B.c1 = 10,
            C as C.c1 = 7
    ) as T
]]></query>
```

Example 8–20 prev(identifier1.identifier2) Function Stream Input

```
Timestamp   Tuple
1000        35
3000        35
4000        10
5000         7
```

Example 8–21 prev(identifier1.identifier2) Function Stream Output

```
Timestamp   Tuple Kind  Tuple
5000:       +           35,7
```

prev(identifier1.identifier2, const_int)

Consider query `q35` in Example 8–22 and the data stream `S15` in Example 8–23. Stream `S15` has schema `(c1 integer, c2 integer)`. This example defines pattern `A` as `A.c1 = prev(A.c1,3)`. In other words, pattern `A` matches when the value of `c1` in the current stream element matches the value of `c1` in the third stream element before the current stream element. The query returns the stream in Example 8–24.

Example 8–22 prev(identifier1.identifier2, const_int) Function Query

```
<query id="q35"><![CDATA[
    select T.Ac1 from S15
    MATCH_RECOGNIZE (
        MEASURES
            A.c1 as Ac1
        PATTERN(A)
        DEFINE
            A as (A.c1 = prev(A.c1,3) )
    ) as T
]]></query>
```

Example 8–23 prev(identifier1.identifier2, const_int) Function Stream Input

```
Timestamp   Tuple
 1000       45,20
 2000       45,30
 3000       45,30
 4000       45,30
 5000       45,30
 6000       45,20
 7000       45,20
 8000       45,20
 9000       43,40
10000       52,10
11000       52,30
12000       43,40
13000       52,50
14000       43,40
15000       43,40
```

Example 8–24 prev(identifier1.identifier2, const_int) Function Stream Output

```
Timestamp   Tuple Kind  Tuple
 4000:       +          45
 5000:       +          45
 6000:       +          45
 7000:       +          45
 8000:       +          45
12000:       +          43
13000:       +          52
15000:       +          43
```

prev(identifier1.identifier2, const_int, const_bigint)

Consider query `q36` in Example 8–26 and the data stream `S15` in Example 8–27. Stream `S15` has schema `(c1 integer, c2 integer)`. This example defines pattern `A` as `A.c1 = prev(A.c1,3,5000000000L)`. In other words, pattern `A` matches when:

- the value of `c1` in the current event equals the value of `c1` in the third previous event of the partition to which the current event belongs, and
- the difference between the timestamp of the current event and that third previous event is less than or equal to `5000000000L` nanoseconds.

The query returns the output stream that Example 8–28 shows. Notice that in the output stream, there is no output at `8000`. Example 8–25 shows the contents of the partition (partitioned by the value of the `c2` attribute) to which the event at `8000` belongs.

Example 8–25 Partition Containing the Event at 8000

```
Timestamp   Tuple
1000        45,20
6000        45,20
```

```
7000        45,20
8000        45,20
```

As Example 8–25 shows, even though the value of c1 in the third previous event (the event at 1000) is the same as the value c1 in the current event (the event at 8000), the range condition is not satisfied. This is because the difference in the timestamps of these two events is more than 5000000000 nanoseconds. So it is treated as if there is no previous tuple and prev returns NULL so the condition fails to match.

Example 8–26   prev(identifier1.identifier2, const_int, const_bigint) Function Query

```
<query id="q36"><![CDATA[
    select T.Ac1 from S15
    MATCH_RECOGNIZE (
        PARTITION BY
            c2
        MEASURES
            A.c1 as Ac1
        PATTERN(A)
        DEFINE
            A as (A.c1 = prev(A.c1,3,5000000000L) )
    ) as T
]]></query>
```

Example 8–27   prev(identifier1.identifier2, const_int, const_bigint) Function Stream Input

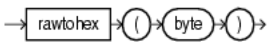
```
Timestamp   Tuple
 1000       45,20
 2000       45,30
 3000       45,30
 4000       45,30
 5000       45,30
 6000       45,20
 7000       45,20
 8000       45,20
 9000       43,40
10000       52,10
11000       52,30
12000       43,40
13000       52,50
14000       43,40
15000       43,40
```

Example 8–28   prev(identifier1.identifier2, const_int, const_bigint) Function Stream Output

```
Timestamp   Tuple Kind  Tuple
5000:       +           45
```

rawtohex

Syntax



Purpose

rawtohex converts *byte* containing a raw value to hexadecimal digits in the CHAR character set.

See Also: "hextoraw" on page 8-5

Examples

Consider the query byte_to_hex in Example 8–29 and the data stream S5 in Example 8–30. Stream S5 has schema (c1 integer, c2 byte(10)). This query uses the rawtohex function to convert a ten byte raw value to the equivalent ten hexidecimal digits in the character set of your current locale. The query returns the relation in Example 8–31.

Example 8–29 rawtohex Function Query

```
<query id="byte_to_hex"><![CDATA[
    select rawtohex(c2) from S5[range 4]
]]></query>
```

Example 8–30 rawtohex Function Stream Input

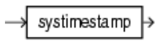
```
Timestamp   Tuple
1000        1,"51c1"
2000        2,"52"
2500        7,"axc"
3000        3,"53aa"
4000        4,"5"
5000         ,"55ef"
6000        6,
h 8000
h 200000000
```

Example 8–31 rawtohex Function Relation Output

```
Timestamp   Tuple Kind  Tuple
 1000:      +           51c1
 2000:      +           52
 3000:      +           53aa
 4000:      +           05
 5000:      -           51c1
 5000:      +           55ef
 6000:      -           52
 6000:      +
 7000:      -           53aa
 8000:      -           05
 9000:      -           55ef
10000:      -
```

systimestamp

Syntax



Purpose

`systimestamp` returns the system date, including fractional seconds and time zone, of the system on which the Oracle Event Processing server resides. The return type is `TIMESTAMP WITH TIME ZONE`.

Examples

Consider the query `q106` in Example 8–32 and the data stream `S0` in Example 8–33. Stream `S0` has schema `(c1 float, c2 integer)`. The query returns the relation in Example 8–34. For more information about `case`, see "case_expr" on page 5-9.

Example 8–32 systimestamp Function Query

```
<query id="q106"><![CDATA[
    select * from S0
    where
        case c2
            when 10 then null
            when 20 then null
            else systimestamp()
        end > "07/06/2007 14:13:33"
]]></query>
```

Example 8–33 systimestamp Function Stream Input

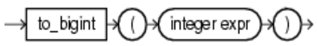
```
Timestamp   Tuple
     1000   0.1 ,10
     1002   0.14,15
   200000   0.2 ,20
   400000   0.3 ,30
   500000   0.3 ,35
   600000       ,35
h 800000
100000000   4.04,40
h 200000000
```

Example 8–34 systimestamp Function Relation Output

```
Timestamp   Tuple Kind  Tuple
     1002:  +           0.14,15
   400000:  +           0.3 ,30
   500000:  +           0.3 ,35
   600000:  +               ,35
100000000:  +           4.04,40
```

to_bigint

Syntax



Purpose

`to_bigint` returns a `bigint` number equivalent of its `integer` argument.

For more information, see:

- *arith_expr::=* on page 5-6
- Section , "Explicit Datatype Conversion"

Examples

Consider the query `q282` in Example 8–35 and the data stream `S11` in Example 8–36. Stream `S11` has schema `(c1 integer, name char(10))`. The query returns the relation in Example 8–37.

Example 8–35 to_bigint Function Query

```
<query id="q282"><![CDATA[
    select nvl(to_bigint(c1), 5.2) from S11
]]></query>
```

Example 8–36 to_bigint Function Stream Input

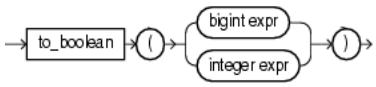
```
Timestamp   Tuple
  10        1,abc
2000         ,ab
3400        3,abc
4700         ,a
h 8000
h 200000000
```

Example 8–37 to_bigint Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1
2000:       +           5.2
3400:       +           3
4700:       +           5.2
```

to_boolean

Syntax



Purpose

`to_boolean` returns a value of `true` or `false` for its `bigint` or `integer` expression argument.

For more information, see:

- *arith_expr::=* on page 5-6
- Section , "Explicit Datatype Conversion"

Examples

Consider the query `q282` in Example 8–35 and the data stream `S11` in Example 8–36. Stream `S11` has schema `(c1 integer, name char(10))`. The query returns the relation in Example 8–37.

Example 8–38 to_boolean Function Query

```
<view id="v2" schema="c1 c2" ><![CDATA[
    select to_boolean(c1), c1 from tkboolean_S3 [now] where c2 = 0.1
]]></view><query id="q1"><![CDATA[
    select * from v2
]]></query>
```

Example 8–39 to_boolean Function Stream Input

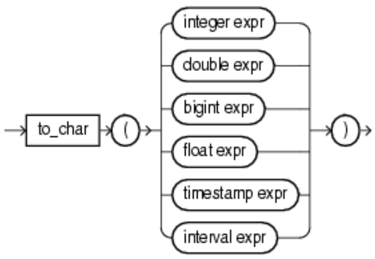
```
Timestamp   Tuple
1000        -2147483648, 0.1
2000        2147483647, 0.2
3000        12345678901, 0.3
4000        -12345678901, 0.1
5000        9223372036854775799, 0.2
6000        -9223372036854775799, 0.3
7000        , 0.1
8000        10000000000, 0.2
9000        60000000000, 0.3
h 200000000
```

Example 8–40 to_boolean Function Relation Output

```
Timestamp   Tuple Kind  Tuple
1000       +           true,-2147483648
1000       -           true,-2147483648
4000       +           true,-12345678901
4000       -           true,-12345678901
7000       +           ,
7000       -           ,
```

to_char

Syntax



Purpose

`to_char` returns a `char` value for its `integer`, `double`, `bigint`, `float`, `timestamp`, or `interval` expression argument. If the `bigint` argument exceeds the `char` precision, Oracle Event Processing returns an error.

For more information, see:

- *arith_expr::=* on page 5-6
- Section , "Explicit Datatype Conversion"

Examples

Consider the query `q282` in Example 8–35 and the data stream `S11` in Example 8–36. Stream `S11` has schema `(c1 integer, name char(10))`. The query returns the relation in Example 8–37.

Example 8–41 to_char Function Query

```
<query id="q1"><![CDATA[
    select to_char(c1), to_char(c2), to_char(c3), to_char(c4), to_char(c5), to_char(c6)
    from S1
]]></query>
```

Example 8–42 to_char Function Stream Input

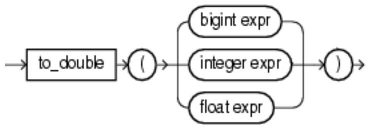
```
Timestamp   Tuple
1000        99,99999, 99.9, 99.9999, "4 1:13:48.10", "08/07/2004 11:13:48", cep
```

Example 8–43 to_char Function Relation Output

```
Timestamp   Tuple Kind  Tuple
1000:       +           99,99999,99.9,99.9999,4 1:13:48.10,08/07/2004 11:13:48
```

to_double

Syntax



Purpose

to_double returns a double value for its bigint, integer, or float expression argument. If the bigint argument exceeds the double precision, Oracle Event Processing returns an error.

For more information, see:

- *arith_expr::=* on page 5-6
- Section , "Explicit Datatype Conversion"

Examples

Consider the query q282 in Example 8–35 and the data stream S11 in Example 8–36. Stream S11 has schema (c1 integer, name char(10)). The query returns the relation in Example 8–37.

Example 8–44 to_double Function Query

```
<query id="q282"><![CDATA[
    select nvl(to_double(c1), 5.2) from S11
]]></query>
```

Example 8–45 to_double Function Stream Input

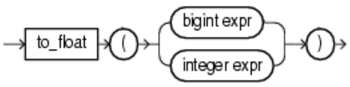
```
Timestamp   Tuple
  10        1,abc
2000         ,ab
3400        3,abc
4700         ,a
h 8000
h 200000000
```

Example 8–46 to_double Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1
2000:       +           5.2
3400:       +           3
4700:       +           5.2
```

to_float

Syntax



Purpose

`to_float` returns a `float` number equivalent of its `bigint` or `integer` argument. If the `bigint` argument exceeds the `float` precision, Oracle Event Processing returns an error.

For more information, see:

- *arith_expr::=* on page 5-6
- Section , "Explicit Datatype Conversion"

Examples

Consider the query `q1` in Example 8–47 and the data stream `S11` in Example 8–48. Stream `S1` has schema `(c1 integer, name char(10))`. The query returns the relation in Example 8–49.

Example 8–47 to_float Function Query

```
<query id="q1"><![CDATA[
    select nvl(to_float(c1), 5.2) from S11
]]></query>
```

Example 8–48 to_float Function Stream Input

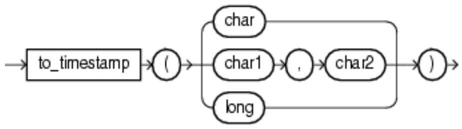
```
Timestamp   Tuple
  10        1, abc
2000         , ab
3400        3, abc
4700         , a
h 8000
h 200000000
```

Example 8–49 to_float Function Relation Output

```
Timestamp   Tuple Kind  Tuple
10:+ 1.02000:+ 5.23400:+ 3.04700:+ 5.2
```

to_timestamp

Syntax



Purpose

`to_timestamp` converts `char` literals that conform to `java.text.SimpleDateFormat` format models to `timestamp` datatypes. There are two forms of the `to_timestamp` function distinguished by the number of arguments:

- `char`: this form of the `to_timestamp` function converts a single `char` argument that contains a `char` literal that conforms to the default `java.text.SimpleDateFormat` format model (`MM/dd/yyyy HH:mm:ss`) into the corresponding `timestamp` datatype.
- `char1, char2`: this form of the `to_timestamp` function converts the `char1` argument that contains a `char` literal that conforms to the `java.text.SimpleDateFormat` format model specified in the second `char2` argument into the corresponding `timestamp` datatype.
- `long`: this form of the `to_timestamp` function converts a single `long` argument that represents the number of nanoseconds since the standard base time known as "the epoch", namely January 1, 1970, 00:00:00 GMT, into the corresponding `timestamp` datatype represented as a number in milliseconds since "the epoch" with a date format that conforms to the default `java.text.SimpleDateFormat` format model (`MM/dd/yyyy HH:mm:ss`).

For more information, see:

- Section , "Oracle CQL Built-in Datatypes"
- Section , "Datetime Literals"
- Section , "Datetime Format Models"

Examples

Consider the query `q277` in Example 8–50 and the data stream `STs2` in Example 8–51. Stream `STs2` has schema `(c1 integer, c2 char(20))`. The query returns the relation that Example 8–52.

Example 8–50 to_timestamp Function Query

```
<query id="q277"><![CDATA[
    select * from STs2
    where
        to_timestamp(c2,"yyMMddHHmmss") = to_timestamp("09/07/2005 10:13:48")
]]></query>
```

Example 8–51 to_timestamp Function Stream Input

```
Timestamp   Tuple
1           1,"040807111348"
2           2,"050907101348"
3           3,"041007111348"
4           4,"060806111248"
```

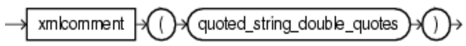
```
h 200000000
```

Example 8–52 to_timestamp Function Relation Output

```
Timestamp   Tuple Kind  Tuple
2:          +           2,050907101348
```

xmlcomment

Syntax



Purpose

`xmlcomment` returns its double-quote delimited constant `String` argument as an `xmltype`.

Using `xmlcomment`, you can add a well-formed XML comment to your query results.

This function takes the following arguments:

- `quoted_string_double_quotes`: a double-quote delimited `String` constant.

The return type of this function is `xmltype`. The exact schema depends on that of the input stream of XML data.

> **See Also:**
>
> - *const_string::=* on page 7-13
> - "SQL/XML (SQLX)" on page 5-16

Examples

Consider the query `tkdata64_q1` in Example 8–53 and data stream `tkdata64_S` in Example 8–54. Stream `tkdata64_S` has schema `(c1 char(30))`. The query returns the relation in Example 8–55.

Example 8–53 xmlcomment Function Query

```
<query id="tkdata64_q1"><![CDATA[
    xmlconcat(xmlelement("parent", c1), xmlcomment("this is a comment"))
from tkdata64_S
]]></query>
```

Example 8–54 xmlcomment Function Stream Input

```
Timestamp   Tuple
c 30
1000        "san jose"
1000        "mountain view"
1000
1000        "sunnyvale"
1003
1004        "belmont"
```

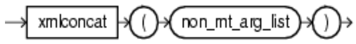
Example 8–55 xmlcomment Function Relation Output

```
Timestamp   Tuple Kind  Tuple
1000:       +           <parent>san jose</parent>
                        <!--this is a comment-->
1000:       +           <parent>mountain view</parent>
                        <!--this is a comment-->
1000:       +           <parent/>
                        <!--this is a comment-->
1000:       +           <parent>sunnyvale</parent>
                        <!--this is a comment-->
1003:       +           <parent/>
```

```
                        <!--this is a comment-->
1004:       +           <parent>belmont</parent>
                        <!--this is a comment-->
```

xmlconcat

Syntax



Purpose

`xmlconcat` returns the concatenation of its comma-delimited `xmltype` arguments as an `xmltype`.

Using `xmlconcat`, you can concatenate any combination of `xmltype` arguments.

This function takes the following arguments:

- `non_mt_arg_list`: a comma-delimited list of `xmltype` arguments. For more information, see *non_mt_arg_list::=* on page 7-21.

The return type of this function is `xmltype`. The exact schema depends on that of the input stream of XML data.

This function is especially useful when processing SQLX streams. For more information, see "SQL/XML (SQLX)" on page 5-16.

To concatenate datatypes other than `xmltype`, use `CONCAT`. For more information, see "concat" on page 8-3.

> **See Also:** "SQL/XML (SQLX)" on page 5-16

Examples

Consider the query `tkdata64_q1` in Example 8–53 and the data stream `tkdata64_S` in Example 8–54. Stream `tkdata64_S` has schema `(c1 char(30))`. The query returns the relation in Example 8–55.

Example 8–56 xmlconcat Function Query

```
<query id="tkdata64_q1"><![CDATA[
    select
        xmlconcat(xmlelement("parent", c1), xmlcomment("this is a comment"))
    from tkdata64_S
]]></query>
```

Example 8–57 xmlconcat Function Stream Input

```
Timestamp   Tuple
c 30
1000        "san jose"
1000        "mountain view"
1000
1000        "sunnyvale"
1003
1004        "belmont"
```

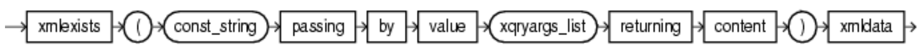
Example 8–58 xmlconcat Function Relation Output

```
Timestamp   Tuple Kind  Tuple
1000:       +           <parent>san jose</parent>
                        <!--this is a comment-->
1000:       +           <parent>mountain view</parent>
                        <!--this is a comment-->
1000:       +           <parent/>
```

```
                    <!--this is a comment-->
1000:     +         <parent>sunnyvale</parent>
                    <!--this is a comment-->
1003:     +         <parent/>
                    <!--this is a comment-->
1004:     +         <parent>belmont</parent>
                    <!--this is a comment-->
```

xmlexists

Syntax



Purpose

`xmlexists` creates a continuous query against a stream of XML data to return a `boolean` that indicates whether or not the XML data satisfies the XQuery you specify.

This function takes the following arguments:

- `const_string`: An XQuery that Oracle Event Processing applies to the XML stream element data that you bind in `xqryargs_list`. For more information, see *const_string::=* on page 7-13.
- `xqryargs_list`: A list of one or more bindings between stream elements and XQuery variables or XPath operators. For more information, see *xqryargs_list::=* on page 7-34.

The return type of this function is `boolean`: `true` if the XQuery is satisfied; `false` otherwise.

This function is especially useful when processing SQLX streams. For more information, see "SQL/XML (SQLX)" on page 5-16.

See Also:

- "xmlquery" on page 8-29
- *func_expr::=* on page 5-15
- *xmltable_clause::=* on page 22-6
- "SQL/XML (SQLX)" on page 5-16

Examples

Consider the query `q1` in Example 8–59 and the XML data stream `S` in Example 8–60. Stream `S` has schema `(c1 integer, c2 xmltype)`. In this example, the value of stream element `c2` is bound to the current node (`"."`) and the value of stream element `c1 + 1` is bound to XQuery variable `x`. The query returns the relation in Example 8–61.

Example 8–59 xmlexists Function Query

```
<query id="q1"><![CDATA[
    SELECT
        xmlexists(
            "for $i in /PDRecord WHERE $i/PDId <= $x RETURN $i/PDName"
            PASSING BY VALUE
                c2 as ".",
               (c1+1) AS "x"
            RETURNING CONTENT
        ) XMLData
    FROM
        S
]]></query>
```

Example 8–60 xmlexists Function Stream Input

```
Timestamp   Tuple
```

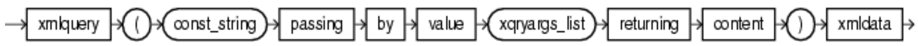
```
3          1, "<PDRecord><PDName>hello</PDName></PDRecord>"
4          2, "<PDRecord><PDName>hello</PDName><PDName>hello1</PDName></PDRecord>"
5          3, "<PDRecord><PDId>6</PDId><PDName>hello1</PDName></PDRecord>"
6          4, "<PDRecord><PDId>46</PDId><PDName>hello2</PDName></PDRecord>"
```

Example 8–61   xmlexists Function Relation Output

```
Timestamp   Tuple Kind  Tuple
3:          +           false
4:          +           false
5:          +           true
6:          +           false
```

xmlquery

Syntax



Purpose

`xmlquery` creates a continuous query against a stream of XML data to return the XML data that satisfies the XQuery you specify.

This function takes the following arguments:

- `const_string`: An XQuery that Oracle Event Processing applies to the XML stream element data that you bind in `xqryargs_list`. For more information, see *const_string::=* on page 7-13.
- `xqryargs_list`: A list of one or more bindings between stream elements and XQuery variables or XPath operators. For more information, see *xqryargs_list::=* on page 7-34.

The return type of this function is `xmltype`. The exact schema depends on that of the input stream of XML data.

This function is especially useful when processing SQLX streams. For more information, see "SQL/XML (SQLX)" on page 5-16.

See Also:

- "xmlexists" on page 8-27
- *func_expr::=* on page 5-15
- *xmltable_clause::=* on page 22-6
- "SQL/XML (SQLX)" on page 5-16

Examples

Consider the query `q1` in Example 8–62 and the XML data stream `S` in Example 8–63. Stream `S` has schema `(c1 integer, c2 xmltype)`. In this example, the value of stream element c2 is bound to the current node (`"."`) and the value of stream element `c1 + 1` is bound to XQuery variable `x`. The query returns the relation in Example 8–64.

Example 8–62 xmlquery Function Query

```
<query id="q1"><![CDATA[
    SELECT
        xmlquery(
            "for $i in /PDRecord WHERE $i/PDId <= $x RETURN $i/PDName"
            PASSING BY VALUE
                c2 as ".",
               (c1+1) AS "x"
            RETURNING CONTENT
        ) XMLData
    FROM
        S
]]></query>
```

Example 8–63 xmlquery Function Stream Input

```
Timestamp   Tuple
```

```
3           1, "<PDRecord><PDName>hello</PDName></PDRecord>"
4           2, "<PDRecord><PDName>hello</PDName><PDName>hello1</PDName></PDRecord>"
5           3, "<PDRecord><PDId>6</PDId><PDName>hello1</PDName></PDRecord>"
6           4, "<PDRecord><PDId>46</PDId><PDName>hello2</PDName></PDRecord>"
```

Example 8–64 xmlquery Function Relation Output

```
Timestamp   Tuple Kind  Tuple
3:          +
4:          +
5:          +           "<PDName>hello1</PDName>"
6:          +           "<PDName>hello2</PDName>"
```

9

Built-In Aggregate Functions

This chapter provides a reference to built-in aggregate functions included in Oracle Continuous Query Language (Oracle CQL). Built-in aggregate functions perform a summary operation on all the values that a query returns.

For more information, see Section , "Functions".

This chapter includes the following section:

- Introduction to Oracle CQL Built-In Aggregate Functions

Introduction to Oracle CQL Built-In Aggregate Functions

Table 9–1 lists the built-in aggregate functions that Oracle CQL provides:

Table 9–1 Oracle CQL Built-in Aggregate Functions

Type	Function
Aggregate	▪ `listagg` ▪ `max` ▪ `min` ▪ `xmlagg`
Aggregate (incremental computation)	▪ `avg` ▪ `count` ▪ `sum`
Extended aggregate	▪ `first` ▪ `last`

Specify `distinct` if you want Oracle Event Processing to return only one copy of each set of duplicate tuples selected. Duplicate tuples are those with matching values for each expression in the select list. For more information, see *aggr_distinct_expr* on page 5-3.

Oracle Event Processing does not support nested aggregations.

Note: Built-in function names are case sensitive and you must use them in the case shown (in lower case).

Note: In stream input examples, lines beginning with `h` (such as `h 3800`) are heartbeat input tuples. These inform Oracle Event Processing that no further input will have a timestamp lesser than the heartbeat value.

For more information, see:

- Section , "Built-In Aggregate Functions and the Where, Group By, and Having Clauses"
- Section , "Relation-to-Relation Operators"
- Section , "Functions"
- Section , "Datatypes"
- *select_clause::=* on page 22-3

Built-In Aggregate Functions and the Where, Group By, and Having Clauses

In Oracle CQL, the `where` clause is applied before the `group by` and `having` clauses. This means the Oracle CQL statement in Example 9–1 is invalid:

Example 9–1 Invalid Use of count

```
<query id="q1"><![CDATA[
    select * from InputChanel[rows 4 slide 4] as ic where count(*) = 4
]]></query>
```

Instead, you must use the Oracle CQL statement that Example 9–2 shows:

Example 9–2 Valid Use of count

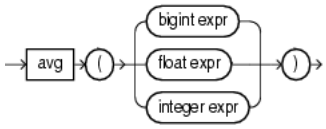
```
<query id="q1"><![CDATA[
    select * from InputChanel[rows 4 slide 4] as ic, count(*) as myCount having
myCount = 4
]]></query>
```

For more information, see:

- "opt_where_clause::=" on page 22-4
- "opt_group_by_clause::=" on page 22-5
- "opt_having_clause::=" on page 22-5

avg

Syntax



Purpose

avg returns average value of *expr*.

This function takes as an argument any bigint, float, or int datatype. The function returns a float regardless of the numeric datatype of the argument.

Examples

Consider the query float_avg in Example 9–3 and the data stream S3 in Example 9–4. Stream S3 has schema (c1 float). The query returns the relation in Example 9–5. Note that the avg function returns a result of NaN if the average value is not a number. For more information, see Section , "Numeric Literals".

Example 9–3 avg Function Query

```
<query id="float_avg"><![CDATA[
    select avg(c1) from S3[range 5]
]]></query>
```

Example 9–4 avg Function Stream Input

```
Timestamp   Tuple
 1000
 2000        5.5
 8000        4.4
 9000
15000       44.2
h 200000000
```

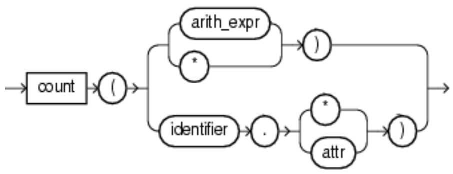
Example 9–5 avg Function Relation Output

```
Timestamp   Tuple Kind  Tuple
    0:      +
 1000:      -
 1000:      +            0.0
 2000:      -            0.0
 2000:      +            5.5
 6000:      -            5.5
 6000:      +            5.5
 7000:      -            5.5
 7000:      +
 8000:      -
 8000:      +            4.4
 9000:      -            4.4
 9000:      +            4.4
13000:      -            4.4
13000:      +           NaN
14000:      -           NaN
14000:      +
15000:      -
15000:      +           44.2
```

```
20000:      -           44.2
20000:      +
```

count

Syntax



(*arith_expr::=* on page 5-6, *attr::=* on page 7-5, *identifier::=* on page 7-17)

Purpose

`count` returns the number of tuples returned by the query as an `int` value.

The return value depends on the argument as Table 9–2 shows.

Table 9–2 Return Values for COUNT Aggregate Function

Input Argument	Return Value
arith_expr	The number of tuples where *arith_expr* is not null.
*	The number of all tuples, including duplicates and nulls.
identifier.*	The number of all tuples that match the correlation variable *identifier*, including duplicates and nulls.
identifier.*attr*	The number of tuples that match correlation variable *identifier*, where *attr* is not null.

`count` never returns null.

Example

Consider the query `q2` in Example 9–6 and the data stream `S2` in Example 9–7. Stream `S2` has schema `(c1 integer, c2 integer)`. The query returns the relation in Example 9–8.

Example 9–6 count Function Query

```
<query id="q2"><![CDATA[
    SELECT COUNT(c2), COUNT(*) FROM  S [RANGE 10]
]]></query>
```

Example 9–7 count Function Stream Input

```
Timestamp   Tuple
1000        1,2
2000        1,
3000        1,4
6000        1,6
```

Example 9–8 count Function Relation Output

```
Timestamp                Tuple Kind  Tuple
-9223372036854775808:    +           0,0
1000:                    -           0,0
1000:                    +           1,1
2000:                    -           1,1
2000:                    +           1,2
```

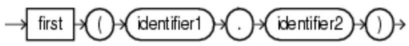
```
3000:              -         1,2
3000:              +         2,3
6000:              -         2,3
```

For more information, see:

- Section , "Using count With *, identifier.*, and identifier.attr"
- "Range-Based Stream-to-Relation Window Operators" on page 4-6

first

Syntax



Purpose

`first` returns the value of the specified stream element the first time the specified pattern is matched.

The type of the specified stream element may be any of:

- `bigint`
- `integer`
- `byte`
- `char`
- `float`
- `interval`
- `timestamp`

The return type of this function depends on the type of the specified stream element.

This function takes a single argument made up of the following period-separated values:

- `identifier1`: the name of a pattern as specified in a `DEFINE` clause.
- `identifier2`: the name of a stream element as specified in a `CREATE STREAM` statement.

See Also:

- "last" on page 9-9
- "prev" on page 8-10
- *pattern_recognition_clause::=* on page 21-2

Examples

Consider the query `q9` in Example 9–9 and the data stream `S0` in Example 9–10. Stream `S0` has schema `(c1 integer, c2 float)`. This example defines pattern `C` as `C.c1 = 7`. It defines `firstc` as `first(C.c2)`. In other words, `firstc` will equal the value of `c2` the first time `c1 = 7`. The query returns the relation in Example 9–11.

Example 9–9 first Function Query

```
<query id="q9"><![CDATA[
    select
        T.firstc,
        T.lastc,
        T.Ac1,
        T.Bc1,
        T.avgCc1,
        T.Dc1
    from
        S0
```

```
    MATCH_RECOGNIZE (
        MEASURES
            first(C.c2) as firstc,
            last(C.c2) as lastc,
            avg(C.c1) as avgCc1,
            A.c1 as Ac1,
            B.c1 as Bc1,
            D.c1 as Dc1
        PATTERN(A B C* D)
        DEFINE
            A as A.c1 = 30,
            B as B.c2 = 10.0,
            C as C.c1 = 7,
            D as D.c1 = 40
    ) as T
]]></query>
```

Example 9–10 first Function Stream Input

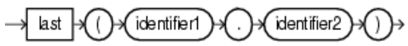
```
Timestamp   Tuple
 1000       33,0.9
 3000       44,0.4
 4000       30,0.3
 5000       10,10.0
 6000        7,0.9
 7000        7,2.3
 9000        7,8.7
11000       40,6.6
15000       19,8.8
17000       30,5.5
20000        5,10.0
23000       40,6.6
25000        3,5.5
30000       30,2.2
35000        2,10.0
40000        7,5.5
44000       40,8.9
```

Example 9–11 first Function Relation Output

```
Timestamp   Tuple Kind  Tuple
11000:      +           0.9,8.7,30,10,7.0,40
23000:      +           ,,30,5,,40
44000:      +           5.5,5.5,30,2,7.0,40
```

last

Syntax



Purpose

`last` returns the value of the specified stream element the last time the specified pattern is matched.

The type of the specified stream element may be any of:

- `bigint`
- `integer`
- `byte`
- `char`
- `float`
- `interval`
- `timestamp`

The return type of this function depends on the type of the specified stream element.

This function takes a single argument made up of the following period-separated values:

- `identifier1`: the name of a pattern as specified in a `DEFINE` clause.
- `identifier2`: the name of a stream element as specified in a `CREATE STREAM` statement.

See Also:

- "first" on page 9-7
- "prev" on page 8-10
- *pattern_recognition_clause::=* on page 21-2

Examples

Consider the query `q9` in Example 9–12 and the data stream `S0` in Example 9–13. Stream `S1` has schema `(c1 integer, c2 float)`. This example defines pattern `C` as `C.c1 = 7`. It defines `lastc` as `last(C.c2)`. In other words, `lastc` will equal the value of `c2` the last time `c1 = 7`. The query returns the relation in Example 9–14.

Example 9–12 last Function Query

```
<query id="q9"><![CDATA[
    select
        T.firstc,
        T.lastc,
        T.Ac1,
        T.Bc1,
        T.avgCc1,
        T.Dc1
    from
        S0
```

```
    MATCH_RECOGNIZE (
        MEASURES
            first(C.c2) as firstc,
            last(C.c2) as lastc,
            avg(C.c1) as avgCc1,
            A.c1 as Ac1,
            B.c1 as Bc1,
            D.c1 as Dc1
        PATTERN(A B C* D)
        DEFINE
            A as A.c1 = 30,
            B as B.c2 = 10.0,
            C as C.c1 = 7,
            D as D.c1 = 40
    ) as T
]]></query>
```

Example 9–13   last Function Stream Input

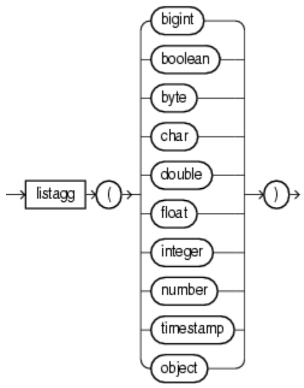
```
Timestamp   Tuple
 1000       33,0.9
 3000       44,0.4
 4000       30,0.3
 5000       10,10.0
 6000        7,0.9
 7000        7,2.3
 9000        7,8.7
11000       40,6.6
15000       19,8.8
17000       30,5.5
20000        5,10.0
23000       40,6.6
25000        3,5.5
30000       30,2.2
35000        2,10.0
40000        7,5.5
44000       40,8.9
```

Example 9–14   last Function Relation Output

```
Timestamp   Tuple Kind  Tuple
11000:      +           0.9,8.7,30,10,7.0,40
23000:      +           ,,30,5,,40
44000:      +           5.5,5.5,30,2,7.0,40
```

listagg

Syntax



Purpose

`listagg` returns a `java.util.List` containing the Java equivalent of the function's argument.

Note that when a user-defined class is used as the function argument, the class must implement the `equals` method.

Examples

In Example 9–15, "listagg Query", view v1 aggregates the values from the third column of Example 9–16, "listagg Example Input" into a `java.util.List` (the default return type) and a `java.util.LinkedHashSet`. Query q1 then selects the size of each to generate the output in Example 9–17, "listagg Relation Output".

Example 9–15 listagg Query

```
<view id="v1"><![CDATA[
    ISTREAM(select c1, listAgg(c3) as l1,
        java.util.LinkedHashSet(listAgg(c3)) as set1
    from S1
    group by c1)
]]></view>

<query id="q1"><![CDATA[
    select v1.l1.size(), v1.set1.size()
    from v1
]]></query>
```

Example 9–16 listagg Example Input

```
Timestamp     Tuple
1000          rcl, 1, 15, 400
1000          msft, 1, 15, 400
2000          orcl, 2, 20, 300
2000          msft, 2, 20, 300
5000          orcl, 4, 5, 200
5000          msft, 4, 5, 200
7000          orcl, 3, 10, 100
```

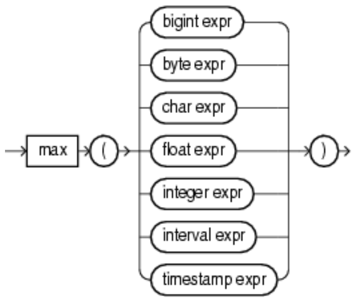
```
7000          msft, 3, 20, 100
h 20000000
```

Example 9–17   listagg Relation Output

```
Timestamp   Tuple Kind   Tuple
 1000:          +        1,1
 1000:          +        1,1
 2000:          +        2,2
 2000:          +        2,2
 5000:          +        3,3
 5000:          +        3,3
 7000:          +        4,4
 7000:          +        4,3
```

max

Syntax



Purpose

`max` returns maximum value of *expr*. Its datatype depends on the datatype of the argument.

Examples

Consider the query `test_max_timestamp` in Example 9–18 and the data stream `S15` in Example 9–19. Stream `S15` has schema `(c1 int, c2 timestamp)`. The query returns the relation in Example 9–20.

Example 9–18 max Function Query

```
<query id="test_max_timestamp"><![CDATA[
    select max(c2) from S15[range 2]
]]></query>
```

Example 9–19 max Function Stream Input

```
Timestamp   Tuple
  10        1,"08/07/2004 11:13:48"
2000         ,"08/07/2005 11:13:48"
3400        3,"08/07/2006 11:13:48"
4700         ,"08/07/2007 11:13:48"
h 8000
h 200000000
```

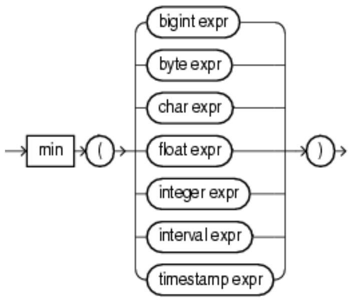
Example 9–20 max Function Relation Output

```
Timestamp   Tuple Kind  Tuple
   0:       +
  10:       -
  10:       +           08/07/2004 11:13:48
2000:       -           08/07/2004 11:13:48
2000:       +           08/07/2005 11:13:48
2010:       -           08/07/2005 11:13:48
2010:       +           08/07/2005 11:13:48
3400:       -           08/07/2005 11:13:48
3400:       +           08/07/2006 11:13:48
4000:       -           08/07/2006 11:13:48
4000:       +           08/07/2006 11:13:48
4700:       -           08/07/2006 11:13:48
4700:       +           08/07/2007 11:13:48
5400:       -           08/07/2007 11:13:48
5400:       +           08/07/2007 11:13:48
```

```
6700:       -          08/07/2007 11:13:48
6700:       +
```

min

Syntax



Purpose

`min` returns minimum value of *expr*. Its datatype depends on the datatype of its argument.

Examples

Consider the query `test_min_timestamp` in Example 9–21 and the data stream `S15` in Example 9–22. Stream `S15` has schema `(c1 int, c2 timestamp)`. The query returns the relation in Example 9–23.

Example 9–21 min Function Query

```
<query id="test_min_timestamp"><![CDATA[
    select min(c2) from S15[range 2]
]]></query>
```

Example 9–22 min Function Stream Input

```
Timestamp   Tuple
  10        1,"08/07/2004 11:13:48"
2000         ,"08/07/2005 11:13:48"
3400        3,"08/07/2006 11:13:48"
4700         ,"08/07/2007 11:13:48"
h 8000
h 200000000
```

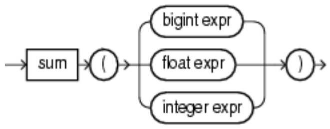
Example 9–23 min Function Relation Output

```
Timestamp   Tuple Kind  Tuple
   0:       +
  10:       -
  10:       +           08/07/2004 11:13:48
2000:       -           08/07/2004 11:13:48
2000:       +           08/07/2004 11:13:48
2010:       -           08/07/2004 11:13:48
2010:       +           08/07/2005 11:13:48
3400:       -           08/07/2005 11:13:48
3400:       +           08/07/2005 11:13:48
4000:       -           08/07/2005 11:13:48
4000:       +           08/07/2006 11:13:48
4700:       -           08/07/2006 11:13:48
4700:       +           08/07/2006 11:13:48
5400:       -           08/07/2006 11:13:48
5400:       +           08/07/2007 11:13:48
```

```
6700:       -           08/07/2007 11:13:48
6700:       +
```

sum

Syntax



Purpose

`sum` returns the sum of values of *`expr`*. This function takes as an argument any `bigint`, `float`, or `integer` expression. The function returns the same datatype as the numeric datatype of the argument.

Examples

Consider the query `q3` in Example 9–24 and the data stream `S1` in Example 9–25. Stream `S1` has schema `(c1 integer, c2 bigint)`. The query returns the relation in Example 9–26. For more information on `range`, see "Range-Based Stream-to-Relation Window Operators" on page 4-6.

Example 9–24 sum Query

```
<query id="q3"><![CDATA[
    select sum(c2) from S1[range 5]
]]></query>
```

Example 9–25 sum Stream Input

```
Timestamp   Tuple
1000         5,
1000        10,5
2000          ,4
3000        30,6
5000        45,44
7000        55,3
h 200000000
```

Example 9–26 sum Relation Output

```
Timestamp   Tuple Kind  Tuple
    0:      +
 1000:      -
 1000:      +           5
 2000:      -           5
 2000:      +           9
 3000:      -           9
 3000:      +           15
 5000:      -           15
 5000:      +           59
 6000:      -           59
 6000:      +           54
 7000:      -           54
 7000:      +           53
 8000:      -           53
 8000:      +           47
10000:      -           47
10000:      +           3
12000:      -           3
12000:      +
```

xmlagg

Syntax

Purpose

`xmlagg` returns a collection of XML fragments as an aggregated XML document. Arguments that return null are dropped from the result.

You can control the order of fragments using an `ORDER BY` clause. For more information, see Section , "Sorting Query Results".

Examples

This section describes the following `xmlagg` examples:

- "xmlagg Function and the xmlelement Function" on page 9-18
- "xmlagg Function and the ORDER BY Clause" on page 9-19

xmlagg Function and the xmlelement Function

Consider the query `tkdata67_q1` in Example 9–27 and the input relation in Example 9–28. Stream `tkdata67_S0` has schema `(c1 integer, c2 float)`. This query uses `xmlelement` to create XML fragments from stream elements and then uses `xmlagg` to aggregate these XML fragments into an XML document. The query returns the relation in Example 9–29.

For more information about `xmlelement`, see "xmlelement_expr" on page 5-28.

Example 9–27 xmlagg Query

```
<query id="tkdata67_q1"><![CDATA[
    select
        c1,
        xmlagg(xmlelement("c2",c2))
    from
        tkdata67_S0[rows 10]
    group by c1
]]></query>
```

Example 9–28 xmlagg Relation Input

```
Timestamp           Tuple
 1000               15, 0.1
 1000               20, 0.14
 1000               15, 0.2
 4000               20, 0.3
10000               15, 0.04
h 12000
```

Example 9–29 xmlagg Relation Output

```
Timestamp   Tuple Kind  Tuple
1000:       +           15,<c2>0.1</c2>
                           <c2>0.2</c2>
1000:       +           20,<c2>0.14</c2>
4000:       -           20,<c2>0.14</c2>
```

```
4000:       +          20,<c2>0.14</c2>
                          <c2>0.3</c2>
10000:      -          15,<c2>0.1</c2>
                          <c2>0.2</c2>
10000:      +          15,<c2>0.1</c2>
                          <c2>0.2</c2>
                          <c2>0.04</c2>
```

xmlagg Function and the ORDER BY Clause

Consider the query `tkxmlAgg_q5` in Example 9–30 and the input relation in Example 9–31. Stream `tkxmlAgg_S1` has schema `(c1 int, c2 xmltype)`. These query selects `xmltype` stream elements and uses XMLAGG to aggregate them into an XML document. This query uses an `ORDER BY` clause to order XML fragments. The query returns the relation in Example 9–32.

Example 9–30 xmlagg and ORDER BY Query

```
<query id="tkxmlAgg_q5"><![CDATA[
    select
        xmlagg(c2),
        xmlagg(c2 order by c1)
    from
        tkxmlAgg_S1[range 2]
]]></query>
```

Example 9–31 xmlagg and ORDER BY Relation Input

```
Timestamp           Tuple
1000                 1, "<a>hello</a>"
2000                10, "<b>hello1</b>"
3000                15, "<PDRecord><PDName>hello</PDName></PDRecord>"
4000                 5, "<PDRecord><PDName>hello</PDName><PDName>hello1</PDName></PDRecord>"
5000                51, "<PDRecord><PDId>6</PDId><PDName>hello1</PDName></PDRecord>"
6000                15, "<PDRecord><PDId>46</PDId><PDName>hello2</PDName></PDRecord>"
7000                55,
"<PDRecord><PDId>6</PDId><PDName>hello2</PDName><PDName>hello3</PDName></PDRecord>"
```

Example 9–32 xmlagg and ORDER BY Relation Output

```
Timestamp   Tuple Kind  Tuple
    0:      +
1000:       -
1000:       +           <a>hello</a>
                       ,<a>hello</a>
2000:       -           <a>hello</a>
                       ,<a>hello</a>
2000:       +           <a>hello</a>
                        <b>hello1</b>
                       ,<a>hello</a>
                        <b>hello1</b>
3000:       -           <a>hello</a>
                        <b>hello1</b>
                       ,<a>hello</a>
                        <b>hello1</b>
3000:       +           <b>hello1</b>
                        <PDRecord>
                          <PDName>hello</PDName>
                        </PDRecord>
                       ,<b>hello1</b>
                        <PDRecord>
                          <PDName>hello</PDName>
                        </PDRecord>
4000:       -           <b>hello1</b>
                        <PDRecord>
```

```
                                  <PDName>hello</PDName>
                                </PDRecord>
                               ,<b>hello1</b>
                                <PDRecord>
                                  <PDName>hello</PDName>
                                </PDRecord>
4000:        +                  <PDRecord>
                                  <PDName>hello</PDName>
                                </PDRecord>
                                <PDRecord>
                                  <PDName>hello</PDName>
                                  <PDName>hello1</PDName>
                                </PDRecord>
                               ,<PDRecord>
                                  <PDName>hello</PDName>
                                  <PDName>hello1</PDName>
                                </PDRecord>
                                <PDRecord>
                                  <PDName>hello</PDName>
                                </PDRecord>
5000:        -                  <PDRecord>
                                  <PDName>hello</PDName>
                                </PDRecord>
                                <PDRecord>
                                  <PDName>hello</PDName>
                                  <PDName>hello1</PDName>
                                </PDRecord>
                               ,<PDRecord>
                                  <PDName>hello</PDName>
                                  <PDName>hello1</PDName>
                                </PDRecord>
                                <PDRecord>
                                  <PDName>hello</PDName>
                                </PDRecord>
5000:        +                  <PDRecord>
                                  <PDName>hello</PDName>
                                  <PDName>hello1</PDName>
                                </PDRecord>
                                <PDRecord>
                                  <PDId>6</PDId>
                                  <PDName>hello1</PDName>
                                </PDRecord>
                               ,<PDRecord>
                                  <PDName>hello</PDName>
                                  <PDName>hello1</PDName>
                                </PDRecord>
                                <PDRecord>
                                  <PDId>6</PDId>
                                  <PDName>hello1</PDName>
                                </PDRecord>
6000:        -                  <PDRecord>
                                  <PDName>hello</PDName>
                                  <PDName>hello1</PDName>
                                </PDRecord>
                                <PDRecord>
                                  <PDId>6</PDId>
                                  <PDName>hello1</PDName>
                                </PDRecord>
                               ,<PDRecord>
                                  <PDName>hello</PDName>
                                  <PDName>hello1</PDName>
                                </PDRecord>
                                <PDRecord>
                                  <PDId>6</PDId>
                                  <PDName>hello1</PDName>
                                </PDRecord>
```

```
6000:       +           <PDRecord>
                          <PDId>6</PDId>
                          <PDName>hello1</PDName>
                        </PDRecord>
                        <PDRecord>
                          <PDId>46</PDId>
                          <PDName>hello2</PDName>
                        </PDRecord>
                       ,<PDRecord>
                          <PDId>46</PDId>
                          <PDName>hello2</PDName>
                        </PDRecord>
                        <PDRecord>
                          <PDId>6</PDId>
                          <PDName>hello1</PDName>
                        </PDRecord>
7000:       -           <PDRecord>
                          <PDId>6</PDId>
                          <PDName>hello1</PDName>
                        </PDRecord>
                        <PDRecord>
                          <PDId>46</PDId>
                          <PDName>hello2</PDName>
                        </PDRecord>
                       ,<PDRecord>
                          <PDId>46</PDId>
                          <PDName>hello2</PDName>
                        </PDRecord>
                        <PDRecord>
                          <PDId>6</PDId>
                          <PDName>hello1</PDName>
                        </PDRecord>
```

10

Colt Single-Row Functions

This chapter provides a reference to Colt single-row functions included in Oracle Continuous Query Language (Oracle CQL). Colt single-row functions are based on the Colt open source libraries for high performance scientific and technical computing.

For more information, see Section , "Functions".

This chapter includes the following section:

- Introduction to Oracle CQLBuilt-In Single-Row Colt Functions

Introduction to Oracle CQLBuilt-In Single-Row Colt Functions

Table 10–1 lists the built-in single-row Colt functions that Oracle CQL provides.

Table 10–1 Oracle CQL Built-in Single-Row Colt-Based Functions

Colt Package	Function
`cern.jet.math.Arithmetic` A set of basic polynomials, rounding, and calculus functions.	■ `binomial` ■ `binomial1` ■ `ceil` ■ `factorial` ■ `floor` ■ `log` ■ `log2` ■ `log10` ■ `logFactorial` ■ `longFactorial` ■ `stirlingCorrection`

Table 10–1 (Cont.) Oracle CQL Built-in Single-Row Colt-Based Functions

Colt Package	Function
`cern.jet.math.Bessel` A set of Bessel functions.	■ i0 ■ i0e ■ i1 ■ i1e ■ j0 ■ j1 ■ jn ■ k0 ■ k0e ■ k1 ■ k1e ■ kn ■ y0 ■ y1 ■ yn
`cern.jet.random.engine.RandomSeedTable` A table with good seeds for pseudo-random number generators. Each sequence in this table has a period of 10**9 numbers.	■ getSeedAtRowColumn
`cern.jet.stat.Gamma` A set of Gamma and Beta functions.	■ beta ■ gamma ■ incompleteBeta ■ incompleteGamma ■ incompleteGammaComplement ■ logGamma
`cern.jet.stat.Probability` A set of probability distributions.	■ beta1 ■ betaComplemented ■ binomial2 ■ binomialComplemented ■ chiSquare ■ chiSquareComplemented ■ errorFunction ■ errorFunctionComplemented ■ gamma1 ■ gammaComplemented ■ negativeBinomial ■ negativeBinomialComplemented ■ normal ■ normal1 ■ normalInverse ■ poisson ■ poissonComplemented ■ studentT ■ studentTInverse
`cern.colt.bitvector.QuickBitVector` A set of non polymorphic, non bounds checking, low level bit-vector functions.	■ bitMaskWithBitsSetFromTo ■ leastSignificantBit ■ mostSignificantBit

Table 10–1 (Cont.) Oracle CQL Built-in Single-Row Colt-Based Functions

Colt Package	Function
cern.colt.map.HashFunctions A set of hash functions.	■ hash ■ hash1 ■ hash2 ■ hash3

Note: Built-in function names are case sensitive and you must use them in the case shown (in lower case).

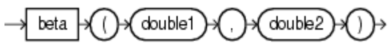
Note: In stream input examples, lines beginning with `h` (such as `h 3800`) are heartbeat input tuples. These inform Oracle Event Processing that no further input will have a timestamp lesser than the heartbeat value.

For more information, see:

- Section , "Functions"
- Section , "Datatypes"
- `http://dst.lbl.gov/ACSSoftware/colt/`

beta

Syntax



Purpose

`beta` is based on `cern.jet.stat.Gamma`. It returns the beta function (see Figure 10–1) of the input arguments as a `double`.

Figure 10–1 cern.jet.stat.Gamma beta

$$B(x, y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x + y)}$$

This function takes the following arguments:

- `double1`: the `x` value.
- `double2`: the `y` value.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Gamma.html#beta(double,%20double)`.

Examples

Consider the query `qColt28` in Example 10–1. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–2, the query returns the relation in Example 10–3.

Example 10–1 beta Function Query

```
<query id="qColt28"><![CDATA[
    select beta(c2,c2) from SColtFunc
]]></query>
```

Example 10–2 beta Function Stream Input

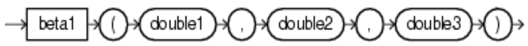
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–3 beta Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           3.1415927
1000:       +           1.899038
1200:       +           1.251922
2000:       +           4.226169
```

beta1

Syntax



Purpose

`beta1` is based on `cern.jet.stat.Probability`. It returns the area `P(x)` from `0` to `x` under the beta density function (see Figure 10–2) as a `double`.

Figure 10–2 cern.jet.stat.Probability beta1

$$P(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \int_0^x t^{a-1} (1-t)^{b-1} dt$$

This function takes the following arguments:

- `double1`: the alpha parameter of the beta distribution `a`.
- `double2`: the beta parameter of the beta distribution `b`.
- `double3`: the integration end point `x`.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#beta(double,%20double,%20double)`.

Examples

Consider the query `qColt35` in Example 10–4. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–5, the query returns the relation in Example 10–6.

Example 10–4 beta1 Function Query

```
<query id="qColt35"><![CDATA[
    select beta1(c2,c2,c2) from SColtFunc
]]></query>
```

Example 10–5 beta1 Function Stream Input

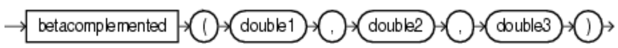
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–6 beta1 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.5
1000:       +           0.66235894
1200:       +           0.873397
2000:       +           0.44519535
```

betaComplemented

Syntax



Purpose

`betaComplemented` is based on `cern.jet.stat.Probability`. It returns the area under the right hand tail (from `x` to infinity) of the beta density function (see Figure 10–2) as a `double`.

This function takes the following arguments:

- `double1`: the alpha parameter of the beta distribution `a`.
- `double2`: the beta parameter of the beta distribution `b`.
- `double3`: the integration end point `x`.

For more information, see:

- "incompleteBeta" on page 10-32
- `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#betaComplemented(double,%20double,%20double)`

Examples

Consider the query `qColt37` in Example 10–7. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–8, the query returns the relation in Example 10–9.

Example 10–7 betaComplemented Function Query

```
<query id="qColt37"><![CDATA[
    select betaComplemented(c2,c2,c2) from SColtFunc
]]></query>
```

Example 10–8 betaComplemented Function Stream Input

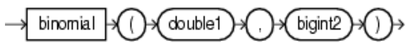
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–9 betaComplemented Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.5
1000:       +           0.66235894
1200:       +           0.873397
2000:       +           0.44519535
```

binomial

Syntax



Purpose

`binomial` is based on `cern.jet.math.Arithmetic`. It returns the binomial coefficient `n` over `k` (see Figure 10–3) as a `double`.

Figure 10–3 Definition of binomial coefficient

$$\frac{(n * n-1 \ldots * n-k+1)}{(1 * 2 * \ldots * k)}$$

This function takes the following arguments:

- `double1`: the `n` value.
- `long2`: the `k` value.

Table 10–2 lists the `binomial` function return values for various values of `k`.

Table 10–2 cern.jet.math.Arithmetic binomial Return Values

Arguments	Return Value
`k` < 0	0
`k` = 0	1
`k` = 1	`n`
Any other value of `k`	Computed binomial coefficient as given in Figure 10–3.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#binomial(double,%20long)`.

Examples

Consider the query `qColt6` in Example 10–10. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 long)` in Example 10–11, the query returns the relation in Example 10–12.

Example 10–10 binomial Function Query

```
<query id="qColt6"><![CDATA[
    select binomial(c2,c3) from SColtFunc
]]></query>
```

Example 10–11 binomial Function Stream Input

```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–12 binomial Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           -0.013092041
1000:       +           -0.012374863
1200:       +           -0.0010145549
2000:       +           -0.0416
```

binomial1

Syntax

Purpose

`binomial1` is based on `cern.jet.math.Arithmetic`. It returns the binomial coefficient `n` over `k` (see Figure 10–3) as a `double`.

This function takes the following arguments:

- `long1`: the `n` value.
- `long2`: the `k` value.

Table 10–3 lists the `BINOMIAL` function return values for various values of `k`.

Table 10–3 cern.jet.math.Arithmetic Binomial1 Return Values

<table>
<tr><th>Arguments</th><th>Return Value</th></tr>
<tr><td>k < 0</td><td>0</td></tr>
<tr><td>k = 0 || k = n</td><td>1</td></tr>
<tr><td>k = 1 || k = n-1</td><td>n</td></tr>
<tr><td>Any other value of k</td><td>Computed binomial coefficient as given in Figure 10–3.</td></tr>
</table>

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#binomial(long,%20long)`.

Examples

Consider the query `qColt7` in Example 10–13. Given the data stream `SColtFunc` with schema `(c1 integer, c2 float, c3 long)` in Example 10–14, the query returns the relation in Example 10–15.

Example 10–13 binomial1 Function Query

```
<query id="qColt7"><![CDATA[
    select binomial1(c3,c3) from SColtFunc
]]></query>
```

Example 10–14 binomial1 Function Stream Input

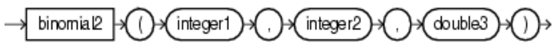
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–15 binomial1 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1.0
1000:       +           1.0
1200:       +           1.0
2000:       +           1.0
```

binomial2

Syntax



Purpose

`binomial2` is based on `cern.jet.stat.Probability`. It returns the sum of the terms 0 through `k` of the binomial probability density (see Figure 10–4) as a `double`.

Figure 10–4 cern.jet.stat.Probability binomial2

$$\sum_{j=0}^{k} \binom{n}{j} p^{j} (1-p)^{n-j}$$

This function takes the following arguments (all arguments must be positive):

- `integer1`: the end term `k`.
- `integer2`: the number of trials `n`.
- `double3`: the probability of success `p` in (0.0, 1.0)

For more information, see
`http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#binomial(int,%20int,%20double)`.

Examples

Consider the query `qColt34` in Example 10–16. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–17, the query returns the relation in Example 10–18.

Example 10–16 binomial2 Function Query

```
<query id="qColt34"><![CDATA[
    select binomial2(c1,c1,c2) from SColtFunc
]]></query>
```

Example 10–17 binomial2 Function Stream Input

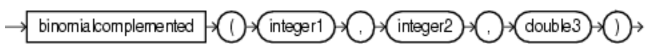
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–18 binomial2 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1.0
1000:       +           1.0
1200:       +           1.0
2000:       +           1.0
```

binomialComplemented

Syntax



Purpose

`binomialComplemented` is based on `cern.jet.stat.Probability`. It returns the sum of the terms `k+1` through `n` of the binomial probability density (see Figure 10–5) as a `double`.

Figure 10–5 cern.jet.stat.Probability binomialComplemented

$$\sum_{j=k+1}^{n} \binom{n}{j} p^{j} (1-p)^{n-j}$$

This function takes the following arguments (all arguments must be positive):

- `integer1`: the end term `k`.
- `integer2`: the number of trials `n`.
- `double3`: the probability of success `p` in (0.0, 1.0)

For more information, see http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#binomialComplemented(int,%20int,%20double).

Examples

Consider the query `qColt38` in Example 10–19. Given the data stream `SColtFunc` with schema `(integer, c2 double, c3 bigint)` in Example 10–20, the query returns the relation in Example 10–21.

Example 10–19 binomialComplemented Function Query

```
<query id="qColt38"><![CDATA[
    select binomialComplemented(c1,c1,c2) from SColtFunc
]]></query>
```

Example 10–20 binomialComplemented Function Stream Input

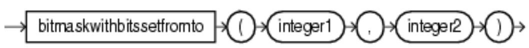
```
Timestamp    Tuple
  10         1,0.5,8
1000         4,0.7,6
1200         3,0.89,12
2000         8,0.4,4
```

Example 10–21 binomialComplemented Function Relation Output

```
Timestamp    Tuple Kind  Tuple
  10:        +           0.0
1000:        +           0.0
1200:        +           0.0
2000:        +           0.0
```

bitMaskWithBitsSetFromTo

Syntax



Purpose

`bitMaskWithBitsSetFromTo` is based on `cern.colt.bitvector.QuickBitVector`. It returns a 64-bit wide bit mask as a `long` with bits in the specified range set to 1 and all other bits set to 0.

This function takes the following arguments:

- `integer1`: the `from` value; index of the start bit (inclusive).
- `integer2`: the `to` value; index of the end bit (inclusive).

Precondition (not checked): `to` - `from` + 1 >= 0 && `to` - `from` + 1 <= 64.

If `to` - `from` + 1 = 0 then returns a bit mask with all bits set to 0.

For more information, see:

- http://dst.lbl.gov/ACSSoftware/colt/api/cern/colt/bitvector/QuickBitVector.html#bitMaskWithBitsSetFromTo(int,%20int)
- "leastSignificantBit" on page 10-43
- "mostSignificantBit" on page 10-50

Examples

Consider the query `qColt53` in Example 10–22. Given the data stream `SColtFunc` with schema `(c1 integer, c2 float, c3 bigint)` in Example 10–23, the query returns the relation in Example 10–24.

Example 10–22 bitMaskWithBitsSetFromTo Function Query

```
<query id="qColt53"><![CDATA[
    select bitMaskWithBitsSetFromTo(c1,c1) from SColtFunc
]]></query>
```

Example 10–23 bitMaskWithBitsSetFromTo Function Stream Input

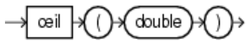
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–24 bitMaskWithBitsSetFromTo Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           2
1000:       +           16
1200:       +           8
2000:       +           256
```

ceil

Syntax



Purpose

`ceil` is based on `cern.jet.math.Arithmetic`. It returns the smallest `long` greater than or equal to its `double` argument.

This method is safer than using `(float) java.lang.Math.ceil(long)` because of possible rounding error.

For more information, see:

- `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#ceil(double)`
- "ceil1" on page 12-12

Examples

Consider the query `qColt1` in Example 10–25. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–26, the query returns the relation in Example 10–27.

Example 10–25 ceil Function Query

```
<query id="qColt1"><![CDATA[
    select ceil(c2) from SColtFunc
]]></query>
```

Example 10–26 ceil Function Stream Input

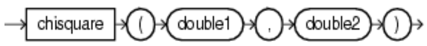
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–27 ceil Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1
1000:       +           1
1200:       +           1
2000:       +           1
```

chiSquare

Syntax



Purpose

`chiSquare` is based on `cern.jet.stat.Probability`. It returns the area under the left hand tail (from 0 to `x`) of the Chi square probability density function with `v` degrees of freedom (see Figure 10–6) as a `double`.

Figure 10–6 cern.jet.stat.Probability chiSquare

$$P(x \mid v) = \frac{1}{2^{\frac{v}{2}}\Gamma(\frac{v}{2})} \int_{x}^{\infty} t^{(\frac{v}{2})-1} e^{\frac{-t}{2}} dt$$

This function takes the following arguments (all arguments must be positive):

- `double1`: the degrees of freedom `v`.
- `double2`: the integration end point `x`.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#chiSquare(double,%20double)`.

Examples

Consider the query `qColt39` in Example 10–28. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–29, the query returns the relation in Example 10–30.

Example 10–28 chiSquare Function Query

```
<query id="qColt39"><![CDATA[
    select chiSquare(c2,c2) from SColtFunc
]]></query>
```

Example 10–29 chiSquare Function Stream Input

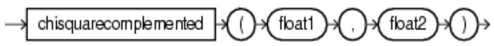
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–30 chiSquare Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.0
1000:       +           0.0
1200:       +           0.0
2000:       +           0.0
```

chiSquareComplemented

Syntax



Purpose

`chiSquareComplemented` is based on `cern.jet.stat.Probability`. It returns the area under the right hand tail (from `x` to infinity) of the Chi square probability density function with `v` degrees of freedom (see Figure 10–6) as a `double`.

This function takes the following arguments (all arguments must be positive):

- `double1`: the degrees of freedom `v`.
- `double2`: the integration end point `x`.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#chiSquareComplemented(double,%20double)`.

Examples

Consider the query `qColt40` in Example 10–31. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–32, the query returns the relation in Example 10–33.

Example 10–31 chiSquareComplemented Function Query

```
<query id="qColt40"><![CDATA[
    select chiSquareComplemented(c2,c2) from SColtFunc
]]></query>
```

Example 10–32 chiSquareComplemented Function Stream Input

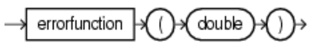
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–33 chiSquareComplemented Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.0
1000:       +           0.0
1200:       +           0.0
2000:       +           0.0
```

errorFunction

Syntax



Purpose

`errorFunction` is based on `cern.jet.stat.Probability`. It returns the error function of the normal distribution of the `double` argument as a `double`, using the integral that Figure 10–7 shows.

Figure 10–7 cern.jet.stat.Probability errorFunction

$$f(x) = \frac{2}{\sqrt{\pi}} \int_{0}^{x} \exp(-t^2) dt$$

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#errorFunction(double)`.

Examples

Consider the query `qColt41` in Example 10–34. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–35, the query returns the relation in Example 10–36.

Example 10–34 errorFunction Function Query

```
<query id="qColt41"><![CDATA[
    select errorFunction(c2) from SColtFunc
]]></query>
```

Example 10–35 errorFunction Function Stream Input

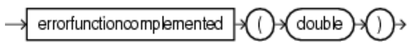
```
Timestamp    Tuple
  10         1,0.5,8
1000         4,0.7,6
1200         3,0.89,12
2000         8,0.4,4
```

Example 10–36 errorFunction Function Relation Output

```
Timestamp    Tuple Kind  Tuple
  10:        +           0.5204999
1000:        +           0.6778012
1200:        +           0.79184324
2000:        +           0.42839235
```

errorFunctionComplemented

Syntax



Purpose

`errorFunctionComplemented` is based on `cern.jet.stat.Probability`. It returns the complementary error function of the normal distribution of the `double` argument as a `double`, using the integral that Figure 10–8 shows.

Figure 10–8 cern.jet.stat.Probability errorfunctioncompelemented

$$f(x) = \frac{2}{\sqrt{\pi}} \int_{x}^{\infty} \exp(-t^2)\,dt$$

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#errorFunctionComplemented(double)`.

Examples

Consider the query `qColt42` in Example 10–37. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–38, the query returns the relation in Example 10–39.

Example 10–37 errorFunctionComplemented Function Query

```
<query id="qColt42"><![CDATA[
    select errorFunctionComplemented(c2) from SColtFunc
]]></query>
```

Example 10–38 errorFunctionComplemented Function Stream Input

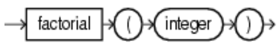
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–39 errorFunctionComplemented Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.47950011
1000:       +           0.3221988
1200:       +           0.20815676
2000:       +           0.57160765
```

factorial

Syntax



Purpose

`factorial` is based on `cern.jet.math.Arithmetic`. It returns the factorial of the positive `integer` argument as a `double`.

For more information, see http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#factorial(int).

Examples

Consider the query `qColt8` in Example 10–40. Given the data stream `SColtFunc` with schema `(c1 integer, c2 float, c3 bigint)` in Example 10–41, the query returns the relation in Example 10–42.

Example 10–40 factorial Function Query

```
<query id="qColt8"><![CDATA[
    select factorial(c1) from SColtFunc
]]></query>
```

Example 10–41 factorial Function Stream Input

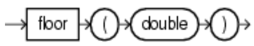
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–42 factorial Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +               1.0
1000:       +              24.0
1200:       +               6.0
2000:       +           40320.0
```

floor

Syntax



Purpose

`floor` is based on `cern.jet.math.Arithmetic`. It returns the largest `long` value less than or equal to the `double` argument.

This method is safer than using `(double) java.lang.Math.floor(double)` because of possible rounding error.

For more information, see:

- http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#floor(double)
- "floor1" on page 12-17

Examples

Consider the query `qColt2` in Example 10–43. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–44, the query returns the relation in Example 10–45.

Example 10–43 floor Function Query

```
<query id="qColt2"><![CDATA[
    select floor(c2) from SColtFunc
]]></query>
```

Example 10–44 floor Function Stream Input

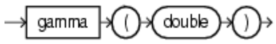
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–45 floor Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0
1000:       +           0
1200:       +           0
2000:       +           0
```

gamma

Syntax



Purpose

`gamma` is based on `cern.jet.stat.Gamma`. It returns the Gamma function of the `double` argument as a `double`.

For more information, see http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Gamma.html#gamma(double).

Examples

Consider the query `qColt29` in Example 10–46. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–47, the query returns the relation in Example 10–48.

Example 10–46 gamma Function Query

```
<query id="qColt29"><![CDATA[
    select gamma(c2) from SColtFunc
]]></query>
```

Example 10–47 gamma Function Stream Input

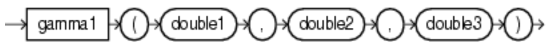
```
Timestamp    Tuple
  10         1,0.5,8
1000         4,0.7,6
1200         3,0.89,12
2000         8,0.4,4
```

Example 10–48 gamma Function Relation Output

```
Timestamp    Tuple Kind  Tuple
  10:        +           1.7724539
1000:        +           1.2980554
1200:        +           1.0768307
2000:        +           2.2181594
```

gamma1

Syntax



Purpose

gamma1 is based on cern.jet.stat.Probability. It returns the integral from zero to x of the gamma probability density function (see Figure 10–9) as a double.

Figure 10–9 cern.jet.stat.Probability gamma1

$$y = \frac{a^b}{\Gamma(b)} \int_0^x t^{b-1} e^{-at} \, dt$$

This function takes the following arguments:

- double1: the gamma distribution alpha value a
- double2: the gamma distribution beta or lambda value b
- double3: the integration end point x

For more information, see http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#gamma(double,%20double,%20double).

Examples

Consider the query qColt36 in Example 10–49. Given the data stream SColtFunc with schema (c1 integer, c2 double, c3 bigint) in Example 10–50, the query returns the relation in Example 10–51.

Example 10–49 gamma1 Function Query

```
<query id="qColt36"><![CDATA[
    select gamma1(c2,c2,c2) from SColtFunc
]]></query>
```

Example 10–50 gamma1 Function Stream Input

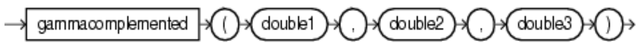
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–51 gamma1 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.5204999
1000:       +           0.55171627
1200:       +           0.59975785
2000:       +           0.51785487
```

gammaComplemented

Syntax



Purpose

`gammaComplemented` is based on `cern.jet.stat.Probability`. It returns the integral from `x` to infinity of the gamma probability density function (see Figure 10–10) as a `double`.

Figure 10–10 cern.jet.stat.Probability gammaComplemented

$$y = \frac{a^b}{\Gamma(b)} \int_x^{\infty} t^{b-1} e^{-at} dt$$

This function takes the following arguments:

- `double1`: the gamma distribution alpha value `a`
- `double2`: the gamma distribution beta or lambda value `b`
- `double3`: the integration end point `x`

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#gammaComplemented(double,%20double,%20double)`.

Examples

Consider the query `qColt43` in Example 10–52. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–53, the query returns the relation in Example 10–54.

Example 10–52 gammaComplemented Function Query

```
<query id="qColt43"><![CDATA[
    select gammaComplemented(c2,c2,c2) from SColtFunc
]]></query>
```

Example 10–53 gammaComplemented Function Stream Input

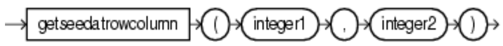
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–54 gammaComplemented Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.47950011
1000:       +           0.44828376
1200:       +           0.40024218
2000:       +           0.48214513
```

getSeedAtRowColumn

Syntax



Purpose

`getSeedAtRowColumn` is based on `cern.jet.random.engine.RandomSeedTable`. It returns a deterministic seed as an `integer` from a (seemingly gigantic) matrix of predefined seeds.

This function takes the following arguments:

- `integer1`: the `row` value; should (but need not) be in `[0,Integer.MAX_VALUE]`.
- `integer2`: the `column` value; should (but need not) be in `[0,1]`.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/random/engine/RandomSeedTable.html#getSeedAtRowColumn(int,%20int)`.

Examples

Consider the query `qColt27` in Example 10–55. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–56, the query returns the relation in Example 10–57.

Example 10–55 getSeedAtRowColumn Function Query

```
<query id="qColt27"><![CDATA[
    select getSeedAtRowColumn(c1,c1) from SColtFunc
]]></query>
```

Example 10–56 getSeedAtRowColumn Function Stream Input

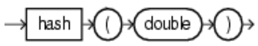
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–57 getSeedAtRowColumn Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           253987020
1000:       +           1289741558
1200:       +           417696270
2000:       +           350557787
```

hash

Syntax



Purpose

hash is based on cern.colt.map.HashFunctions. It returns an integer hashcode for the specified double value.

For more information, see http://dst.lbl.gov/ACSSoftware/colt/api/cern/colt/map/HashFunctions.html#hash(double).

Examples

Consider the query qColt56 in Example 10–58. Given the data stream SColtFunc with schema (c1 integer, c2 double, c3 bigint) in Example 10–59, the query returns the relation in Example 10–60.

Example 10–58 hash Function Query

```
<query id="qColt56"><![CDATA[
    select hash(c2) from SColtFunc
]]></query>
```

Example 10–59 hash Function Stream Input

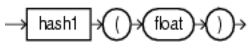
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–60 hash Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +            1071644672
1000:       +            1608935014
1200:       +            2146204385
2000:       +           -1613129319
```

hash1

Syntax



Purpose

`hash1` is based on `cern.colt.map.HashFunctions`. It returns an `integer` hashcode for the specified `float` value.

For more information, see http://dst.lbl.gov/ACSSoftware/colt/api/cern/colt/map/HashFunctions.html#hash(float).

Examples

Consider the query `qColt57` in Example 10–61. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–62, the query returns the relation in Example 10–63.

Example 10–61 hash1 Function Query

```
<query id="qColt57"><![CDATA[
    select hash1(c2) from SColtFunc
]]></query>
```

Example 10–62 hash1 Function Stream Input

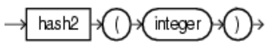
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–63 hash1 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1302214522
1000:       +           1306362078
1200:       +           1309462552
2000:       +           1300047248
```

hash2

Syntax



Purpose

`hash2` is based on `cern.colt.map.HashFunctions`. It returns an `integer` hashcode for the specified `integer` value.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/colt/map/HashFunctions.html#hash(int)`.

Examples

Consider the query `qColt58` in Example 10–64. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–65, the query returns the relation in Example 10–66.

Example 10–64 hash2 Function Query

```
<query id="qColt58"><![CDATA[
    select hash2(c1) from SColtFunc
]]></query>
```

Example 10–65 hash2 Function Stream Input

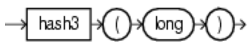
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–66 hash2 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1
1000:       +           4
1200:       +           3
2000:       +           8
```

hash3

Syntax



Purpose

hash3 is based on cern.colt.map.HashFunctions. It returns an integer hashcode for the specified long value.

For more information, see http://dst.lbl.gov/ACSSoftware/colt/api/cern/colt/map/HashFunctions.html#hash(long).

Examples

Consider the query qColt59 in Example 10–67. Given the data stream SColtFunc with schema (c1 integer, c2 double, c3 bigint) in Example 10–68, the query returns the relation in Example 10–69.

Example 10–67 hash3 Function Query

```
<query id="qColt59"><![CDATA[
    select hash3(c3) from SColtFunc
]]></query>
```

Example 10–68 hash3 Function Stream Input

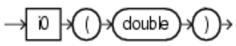
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–69 hash3 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           8
1000:       +           6
1200:       +           12
2000:       +           4
```

i0

Syntax



Purpose

`i0` is based on `cern.jet.math.Bessel`. It returns the modified Bessel function of order 0 of the `double` argument as a `double`.

The function is defined as `i0(x) = j0(ix)`.

The range is partitioned into the two intervals `[0,8]` and `(8,infinity)`.

For more information, see:

- http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#i0(double)
- "j0" on page 10-35

Examples

Consider the query `qColt12` in Example 10–70. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–71, the query returns the relation in Example 10–72.

Example 10–70 i0 Function Query

```
<query id="qColt12"><![CDATA[
    select i0(c2) from SColtFunc
]]></query>
```

Example 10–71 i0 Function Stream Input

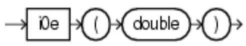
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–72 i0 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1.0634834
1000:       +           1.126303
1200:       +           1.2080469
2000:       +           1.0404018
```

i0e

Syntax



Purpose

`i0e` is based on `cern.jet.math.Bessel`. It returns the exponentially scaled modified Bessel function of order 0 of the `double` argument as a `double`.

The function is defined as: `i0e(x) = exp(-|x|) j0(ix)`.

For more information, see:

- http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#i0e(double)
- "j0" on page 10-35

Examples

Consider the query `qColt13` in Example 10–73. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–74, the query returns the relation in Example 10–75.

Example 10–73 i0e Function Query

```
<query id="qColt13"><![CDATA[
    select i0e(c2) from SColtFunc
]]></query>
```

Example 10–74 i0e Function Stream Input

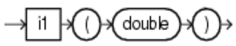
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–75 i0e Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.64503527
1000:       +           0.55930555
1200:       +           0.4960914
2000:       +           0.6974022
```

i1

Syntax



Purpose

`i1` is based on `cern.jet.math.Bessel`. It returns the modified Bessel function of order 1 of the `double` argument as a `double`.

The function is defined as: `i1(x) = -i j1(ix)`.

The range is partitioned into the two intervals `[0,8]` and `(8,infinity)`. Chebyshev polynomial expansions are employed in each interval.

For more information, see:

- `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#i1(double)`
- "j1" on page 10-36

Examples

Consider the query `qColt14` in Example 10–76. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–77, the query returns the relation in Example 10–78.

Example 10–76 i1 Function Query

```
<query id="qColt14"><![CDATA[
    select i1(c2) from SColtFunc
]]></query>
```

Example 10–77 i1 Function Stream Input

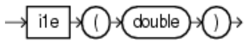
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–78 i1 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.2578943
1000:       +           0.37187967
1200:       +           0.49053898
2000:       +           0.20402676
```

i1e

Syntax



Purpose

i1e is based on cern.jet.math.Bessel. It returns the exponentially scaled modified Bessel function of order 1 of the double argument as a double.

The function is defined as i1(x) = -i exp(-|x|) j1(ix).

For more information, see

- http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#i1e(double)
- "j1" on page 10-36

Examples

Consider the query qColt15 in Example 10–79. Given the data stream SColtFunc with schema (c1 integer, c2 double, c3 bigint) in Example 10–80, the query returns the relation in Example 10–81.

Example 10–79 i1e Function Query

```
<query id="qColt15"><![CDATA[
    select i1e(c2) from SColtFunc
]]></query>
```

Example 10–80 i1e Function Stream Input

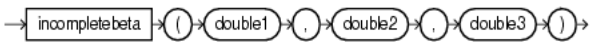
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–81 i1e Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.1564208
1000:       +           0.18466999
1200:       +           0.20144266
2000:       +           0.13676323
```

incompleteBeta

Syntax



Purpose

incompleteBeta is based on cern.jet.stat.Gamma. It returns the Incomplete Beta Function evaluated from zero to x as a double.

This function takes the following arguments:

- double1: the beta distribution alpha value a
- double2: the beta distribution beta value b
- double3: the integration end point x

For more information, see http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Gamma.html#incompleteBeta(double,%20double,%20double).

Examples

Consider the query qColt30 in Example 10–82. Given the data stream SColtFunc with schema (c1 integer, c2 double, c3 bigint) in Example 10–83, the query returns the relation in Example 10–84.

Example 10–82 incompleteBeta Function Query

```
<query id="qColt30"><![CDATA[
    select incompleteBeta(c2,c2,c2) from SColtFunc
]]></query>
```

Example 10–83 incompleteBeta Function Stream Input

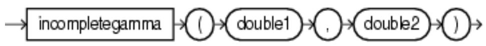
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–84 incompleteBeta Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.5
1000:       +           0.66235894
1200:       +           0.873397
2000:       +           0.44519535
```

incompleteGamma

Syntax



Purpose

`incompleteGamma` is based on `cern.jet.stat.Gamma`. It returns the Incomplete Gamma function of the arguments as a `double`.

This function takes the following arguments:

- `double1`: the gamma distribution alpha value `a`.
- `double2`: the integration end point `x`.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Gamma.html#incompleteGamma(double,%20double)`.

Examples

Consider the query `qColt31` in Example 10–85. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–86, the query returns the relation in Example 10–87.

Example 10–85 incompleteGamma Function Query

```
<query id="qColt31"><![CDATA[
    select incompleteGamma(c2,c2) from SColtFunc
]]></query>
```

Example 10–86 incompleteGamma Function Stream Input

```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–87 incompleteGamma Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.6826895
1000:       +           0.6565891
1200:       +           0.6397422
2000:       +           0.7014413
```

incompleteGammaComplement

Syntax

Purpose

`incompleteGammaComplement` is based on `cern.jet.stat.Gamma`. It returns the Complemented Incomplete Gamma function of the arguments as a `double`.

This function takes the following arguments:

- `double1`: the gamma distribution alpha value `a`.
- `double2`: the integration start point `x`.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Gamma.html#incompleteGammaComplement(double,%20double)`.

Examples

Consider the query `qColt32` in Example 10–88. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–89, the query returns the relation in Example 10–90.

Example 10–88 incompleteGammaComplement Function Query

```
<query id="qColt32"><![CDATA[
    select incompleteGammaComplement(c2,c2) from SColtFunc
]]></query>
```

Example 10–89 incompleteGammaComplement Function Stream Input

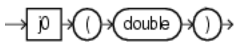
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–90 incompleteGammaComplement Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.3173105
1000:       +           0.34341094
1200:       +           0.3602578
2000:       +           0.29855874
```

j0

Syntax



Purpose

`j0` is based on `cern.jet.math.Bessel`. It returns the Bessel function of the first kind of order 0 of the `double` argument as a `double`.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#j0(double)`.

Examples

Consider the query `qColt16` in Example 10–91. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–92, the query returns the relation in Example 10–93.

Example 10–91   j0 Function Query

```
<query id="qColt16"><![CDATA[
    select j0(c2) from SColtFunc
]]></query>
```

Example 10–92   j0 Function Stream Input

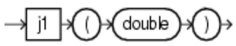
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–93   j0 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.9384698
1000:       +           0.8812009
1200:       +           0.8115654
2000:       +           0.9603982
```

j1

Syntax



Purpose

`j1` is based on `cern.jet.math.Bessel`. It returns the Bessel function of the first kind of order 1 of the `double` argument as a `double`.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#j1(double)`.

Examples

Consider the query `qColt17` in Example 10–94. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–95, the query returns the relation in Example 10–96.

Example 10–94 j1 Function Query

```
<query id="qColt17"><![CDATA[
    select j1(c2) from SColtFunc
]]></query>
```

Example 10–95 j1 Function Stream Input

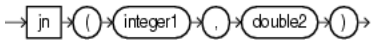
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–96 j1 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.24226846
1000:       +           0.32899573
1200:       +           0.40236986
2000:       +           0.19602658
```

jn

Syntax



Purpose

`jn` is based on `cern.jet.math.Bessel`. It returns the Bessel function of the first kind of order `n` of the argument as a `double`.

This function takes the following arguments:

- `integer1`: the order of the Bessel function `n`.
- `double2`: the value to compute the bessel function of `x`.

For more information, see
http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#jn(int,%20double).

Examples

Consider the query `qColt18` in Example 10–97. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–98, the query returns the relation in Example 10–99.

Example 10–97 jn Function Query

```
<query id="qColt18"><![CDATA[
    select jn(c1,c2) from SColtFunc
]]></query>
```

Example 10–98 jn Function Stream Input

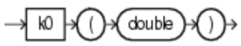
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–99 jn Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.24226846
1000:       +           6.1009696E-4
1200:       +           0.0139740035
2000:       +           6.321045E-11
```

k0

Syntax



Purpose

`k0` is based on `cern.jet.math.Bessel`. It returns the modified Bessel function of the third kind of order 0 of the `double` argument as a `double`.

The range is partitioned into the two intervals `[0,8]` and `(8, infinity)`. Chebyshev polynomial expansions are employed in each interval.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#k0(double)`.

Examples

Consider the query `qColt19` in Example 10–100. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–101, the query returns the relation in Example 10–102.

Example 10–100 k0 Function Query

```
<query id="qColt19"><![CDATA[
    select k0(c2) from SColtFunc
]]></query>
```

Example 10–101 k0 Function Stream Input

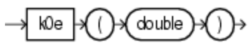
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–102 k0 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.92441905
1000:       +           0.6605199
1200:       +           0.49396032
2000:       +           1.1145291
```

k0e

Syntax



Purpose

`k0e` is based on `cern.jet.math.Bessel`. It returns the exponentially scaled modified Bessel function of the third kind of order 0 of the `double` argument as a `double`.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#k0e(double)`.

Examples

Consider the query `qColt20` in Example 10–103. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–104, the query returns the relation in Example 10–105.

Example 10–103 k0e Function Query

```
<query id="qColt20"><![CDATA[
    select k0e(c2) from SColtFunc
]]></query>
```

Example 10–104 k0e Function Stream Input

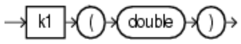
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–105 k0e Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1.5241094
1000:       +           1.3301237
1200:       +           1.2028574
2000:       +           1.662682
```

k1

Syntax



Purpose

`k1` is based on `cern.jet.math.Bessel`. It returns the modified Bessel function of the third kind of order 1 of the `double` argument as a `double`.

The range is partitioned into the two intervals `[0,2]` and `(2, infinity)`. Chebyshev polynomial expansions are employed in each interval.

For more information, see
`http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#k1(double)`.

Examples

Consider the query `qColt21` in Example 10–106. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–107, the query returns the relation in Example 10–108.

Example 10–106 k1 Function Query

```
<query id="qColt21"><![CDATA[
    select k1(c2) from SColtFunc
]]></query>
```

Example 10–107 k1 Function Stream Input

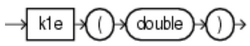
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–108 k1 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1.6564411
1000:       +           1.0502836
1200:       +           0.7295154
2000:       +           2.1843543
```

k1e

Syntax



Purpose

`k1e` is based on `cern.jet.math.Bessel`. It returns the exponentially scaled modified Bessel function of the third kind of order 1 of the `double` argument as a `double`.

The function is defined as: `k1e(x) = exp(x) * k1(x)`.

For more information, see:

- http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#k1e(double)
- "k1" on page 10-40

Examples

Consider the query `qColt22` in Example 10–109. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–110, the query returns the relation in Example 10–111.

Example 10–109 k1e Function Query

```
<query id="qColt22"><![CDATA[
    select k1e(c2) from SColtFunc
]]></query>
```

Example 10–110 k1e Function Stream Input

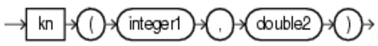
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–111 k1e Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           2.7310097
1000:       +           2.1150115
1200:       +           1.7764645
2000:       +           3.258674
```

kn

Syntax



Purpose

`kn` is based on `cern.jet.math.Bessel`. It returns the modified Bessel function of the third kind of order `n` of the argument as a `double`.

This function takes the following arguments:

- `integer1`: the `n` value order of the Bessel function.
- `double2`: the `x` value to compute the bessel function of.

The range is partitioned into the two intervals `[0,9.55]` and `(9.55, infinity)`. An ascending power series is used in the low range, and an asymptotic expansion in the high range.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#kn(int,%20double)`.

Examples

Consider the query `qColt23` in Example 10–112. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–113, the query returns the relation in Example 10–114.

Example 10–112 kn Function Query

```
<query id="qColt23"><![CDATA[
    select kn(c1,c2) from SColtFunc
]]></query>
```

Example 10–113 kn Function Stream Input

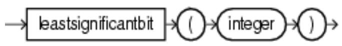
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–114 kn Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +             1.6564411
1000:       +           191.99422
1200:       +            10.317473
2000:       +             9.7876858E8
```

leastSignificantBit

Syntax



Purpose

`leastSignificantBit` is based on `cern.colt.bitvector.QuickBitVector`. It returns the index (as an `integer`) of the least significant bit in state `true` of the `integer` argument. Returns 32 if no bit is in state `true`.

For more information, see:

- `http://dst.lbl.gov/ACSSoftware/colt/api/cern/colt/bitvector/QuickBitVector.html#leastSignificantBit(int)`
- "bitMaskWithBitsSetFromTo" on page 10-12
- "mostSignificantBit" on page 10-50

Examples

Consider the query `qColt54` in Example 10–115. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–116, the query returns the relation in Example 10–117.

Example 10–115 leastSignificantBit Function Query

```
<query id="qColt54"><![CDATA[
    select leastSignificantBit(c1) from SColtFunc
]]></query>
```

Example 10–116 leastSignificantBit Function Stream Input

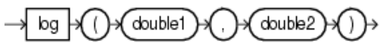
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–117 leastSignificantBit Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0
1000:       +           2
1200:       +           0
2000:       +           3
```

log

Syntax



Purpose

`log` is based on `cern.jet.math.Arithmetic`. It returns the computation that Figure 10–11 shows as a `double`.

Figure 10–11 cern.jet.math.Arithmetic log

$$\log_{base} value$$

This function takes the following arguments:

- `double1`: the `base`.
- `double2`: the `value`.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#log(double,%20double)`.

Examples

Consider the query `qColt3` in Example 10–118. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–119, the query returns the relation in Example 10–120.

Example 10–118 log Function Query

```
<query id="qColt3"><![CDATA[
    select log(c2,c2) from SColtFunc
]]></query>
```

Example 10–119 log Function Stream Input

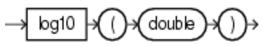
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–120 log Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1.0
1000:       +           1.0
1200:       +           1.0
2000:       +           1.0
```

log10

Syntax



Purpose

`log10` is based on `cern.jet.math.Arithmetic`. It returns the base 10 logarithm of a `double` value as a `double`.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#log10(double)`.

Examples

Consider the query `qColt4` in Example 10–121. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–122, the query returns the relation in Example 10–123.

Example 10–121 log10 Function Query

```
<query id="qColt4"><![CDATA[
    select log10(c2) from SColtFunc
]]></query>
```

Example 10–122 log10 Function Stream Input

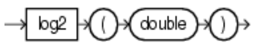
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–123 log10 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           -0.30103
1000:       +           -0.15490197
1200:       +           -0.050610002
2000:       +           -0.39794
```

log2

Syntax



Purpose

`log2` is based on `cern.jet.math.Arithmetic`. It returns the base 2 logarithm of a `double` value as a `double`.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#log2(double)`.

Examples

Consider the query `qColt9` in Example 10–124. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–125, the query returns the relation in Example 10–126.

Example 10–124 log2 Function Query

```
<query id="qColt9"><![CDATA[
    select log2(c2) from SColtFunc
]]></query>
```

Example 10–125 log2 Function Stream Input

```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–126 log2 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           -1.0
1000:       +           -0.5145732
1200:       +           -0.16812278
2000:       +           -1.321928
```

logFactorial

Syntax

Purpose

`logFactorial` is based on `cern.jet.math.Arithmetic`. It returns the natural logarithm (base *e*) of the factorial of its `integer` argument as a `double`

For argument values `k<30`, the function looks up the result in a table in `O(1)`. For argument values `k>=30`, the function uses Stirlings approximation.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#logFactorial(int)`.

Examples

Consider the query `qColt10` in Example 10–127. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–128, the query returns the relation in Example 10–129.

Example 10–127 logFactorial Function Query

```
<query id="qColt10"><![CDATA[
    select logFactorial(c1) from SColtFunc
]]></query>
```

Example 10–128 logFactorial Function Stream Input

```
Timestamp    Tuple
  10         1,0.5,8
1000         4,0.7,6
1200         3,0.89,12
2000         8,0.4,4
```

Example 10–129 logFactorial Function Relation Output

```
Timestamp    Tuple Kind  Tuple
  10:        +            0.0
1000:        +            3.1780539
1200:        +            1.7917595
2000:        +           10.604603
```

logGamma

Syntax

Purpose

`logGamma` is based on `cern.jet.stat.Gamma`. It returns the natural logarithm (base *e*) of the gamma function of the `double` argument as a `double`.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Gamma.html#logGamma(double)`.

Examples

Consider the query `qColt33` in Example 10–130. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–131, the query returns the relation in Example 10–132.

Example 10–130 logGamma Function Query

```
<query id="qColt33"><![CDATA[
    select logGamma(c2) from SColtFunc
]]></query>
```

Example 10–131 logGamma Function Stream Input

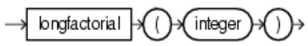
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–132 logGamma Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.5723649
1000:       +           0.26086727
1200:       +           0.07402218
2000:       +           0.7966778
```

longFactorial

Syntax



Purpose

`longFactorial` is based on `cern.jet.math.Arithmetic`. It returns the factorial of its `integer` argument (in the range `k >= 0 && k < 21`) as a `long`.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#longFactorial(int)`.

Examples

Consider the query `qColt11` in Example 10–133. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–134, the query returns the relation in Example 10–135.

Example 10–133 longFactorial Function Query

```
<query id="qColt11"><![CDATA[
    select longFactorial(c1) from SColtFunc
]]></query>
```

Example 10–134 longFactorial Function Stream Input

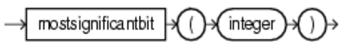
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–135 longFactorial Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1
1000:       +           24
1200:       +           6
2000:       +           40320
```

mostSignificantBit

Syntax



Purpose

`mostSignificantBit` is based on `cern.colt.bitvector.QuickBitVector`. It returns the index (as an `integer`) of the most significant bit in state `true` of the `integer` argument. Returns `-1` if no bit is in state `true`

For more information, see:

- `http://dst.lbl.gov/ACSSoftware/colt/api/cern/colt/bitvector/QuickBitVector.html#mostSignificantBit(int)`
- "bitMaskWithBitsSetFromTo" on page 10-12
- "leastSignificantBit" on page 10-43

Examples

Consider the query `qColt55` in Example 10–136. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–137, the query returns the relation in Example 10–138.

Example 10–136 mostSignificantBit Function Query

```
<query id="qColt55"><![CDATA[
    select mostSignificantBit(c1) from SColtFunc
]]></view>
```

Example 10–137 mostSignificantBit Function Stream Input

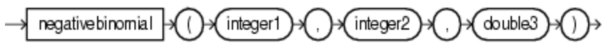
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–138 mostSignificantBit Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0
1000:       +           2
1200:       +           1
2000:       +           3
```

negativeBinomial

Syntax



Purpose

`negativeBinomial` is based on `cern.jet.stat.Probability`. It returns the sum of the terms 0 through `k` of the Negative Binomial Distribution (see Figure 10–12) as a `double`.

Figure 10–12 cern.jet.stat.Probability negativeBinomial

$$\sum_{j=0}^{k} \binom{n+j-1}{j} p^n (1-p)^j$$

This function takes the following arguments:

- `integer1`: the end term `k`.
- `integer2`: the number of trials `n`.
- `double3`: the probability of success `p` in (0.0,1.0).

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#negativeBinomial(int,%20int,%20double)`.

Examples

Consider the query `qColt44` in Example 10–139. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–140, the query returns the relation in Example 10–141.

Example 10–139 negativeBinomial Function Query

```
<query id="qColt44"><![CDATA[
    select negativeBinomial(c1,c1,c2) from SColtFunc
]]></query>
```

Example 10–140 negativeBinomial Function Stream Input

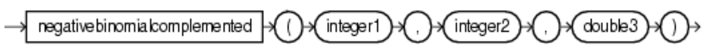
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–141 negativeBinomial Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.75
1000:       +           0.94203234
1200:       +           0.99817264
2000:       +           0.28393665
```

negativeBinomialComplemented

Syntax



Purpose

`negativeBinomialComplemented` is based on `cern.jet.stat.Probability`. It returns the sum of the terms `k+1` to infinity of the Negative Binomial distribution (see Figure 10–13) as a `double`.

Figure 10–13 cern.jet.stat.Probability negativeBinomialComplemented

$$\sum_{j=k+1}^{\infty} \binom{n+j-1}{j} p^n (1-p)^j$$

This function takes the following arguments:

- `integer1`: the end term `k`.
- `integer2`: the number of trials `n`.
- `double3`: the probability of success `p` in (0.0,1.0).

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#negativeBinomialComplemented(int,%20int,%20double)`.

Examples

Consider the query `qColt45` in Example 10–142. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–143, the query returns the relation in Example 10–144.

Example 10–142 negativeBinomialComplemented Function Query

```
<query id="qColt45"><![CDATA[
    select negativeBinomialComplemented(c1,c1,c2) from SColtFunc
]]></query>
```

Example 10–143 negativeBinomialComplemented Function Stream Input

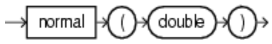
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–144 negativeBinomialComplemented Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.25
1000:       +           0.05796766
1200:       +           0.0018273441
2000:       +           0.7160633
```

normal

Syntax



Purpose

`normal` is based on `cern.jet.stat.Probability`. It returns the area under the Normal (Gaussian) probability density function, integrated from minus infinity to the `double` argument `x` (see Figure 10–14) as a `double`.

Figure 10–14 cern.jet.stat.Probability normal

$$f(x) = \frac{1}{\sqrt{(2\pi)}} \int_{-\infty}^{x} \exp(-\frac{t^2}{2})dt$$

For more information, see
`http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#normal(double)`.

Examples

Consider the query `qColt46` in Example 10–145. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–146, the query returns the relation in Example 10–147.

Example 10–145 normal Function Query

```
<query id="qColt46"><![CDATA[
    select normal(c2) from SColtFunc
]]></query>
```

Example 10–146 normal Function Stream Input

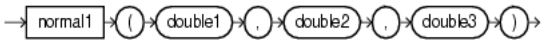
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–147 normal Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.69146246
1000:       +           0.7580363
1200:       +           0.81326705
2000:       +           0.65542173
```

normal1

Syntax



Purpose

`normal1` is based on `cern.jet.stat.Probability`. It returns the area under the Normal (Gaussian) probability density function, integrated from minus infinity to `x` (see Figure 10–15) as a `double`.

Figure 10–15 cern.jet.stat.Probability normal1

$$f(x) = \frac{1}{\sqrt{(2\pi * v)}} \int_{-\infty}^{x} \exp\left(-\frac{(t - mean)^2}{2v}\right) dt$$

This function takes the following arguments:

- `double1`: the normal distribution `mean`.
- `double2`: the variance of the normal distribution `v`.
- `double3`: the integration limit `x`.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#normal(double,%20double,%20double)`.

Examples

Consider the query `qColt47` in Example 10–148. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–149, the query returns the relation in Example 10–150.

Example 10–148 normal1 Function Query

```
<query id="qColt47"><![CDATA[
    select normal1(c2,c2,c2) from SColtFunc
]]></query>
```

Example 10–149 normal1 Function Stream Input

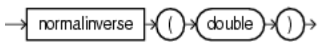
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–150 normal1 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.5
1000:       +           0.5
1200:       +           0.5
2000:       +           0.5
```

normalInverse

Syntax



Purpose

`normalInverse` is based on `cern.jet.stat.Probability`. It returns the `double` value, `x`, for which the area under the Normal (Gaussian) probability density function (integrated from minus infinity to `x`) equals the `double` argument `y` (assumes mean is zero and variance is one).

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#normalInverse(double)`.

Examples

Consider the query `qColt48` in Example 10–151. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–152, the query returns the relation in Example 10–153.

Example 10–151 normalInverse Function Query

```
<query id="qColt48"><![CDATA[
    select normalInverse(c2) from SColtFunc
]]></view>
```

Example 10–152 normalInverse Function Stream Input

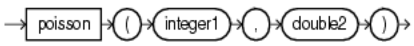
Timestamp	Tuple
10	1,0.5,8
1000	4,0.7,6
1200	3,0.89,12
2000	8,0.4,4

Example 10–153 normalInverse Function Relation Output

Timestamp	Tuple Kind	Tuple
10:	+	0.0
1000:	+	0.5244005
1200:	+	1.226528
2000:	+	0.2533471

poisson

Syntax



Purpose

`poisson` is based on `cern.jet.stat.Probability`. It returns the sum of the first `k` terms of the Poisson distribution (see Figure 10–16) as a `double`.

Figure 10–16 cern.jet.stat.Probability poisson

$$\sum_{j=0}^{k} e^{-m} \frac{m^j}{j!}$$

This function takes the following arguments:

- `integer1`: the number of terms `k`.
- `double2`: the mean of the Poisson distribution `m`.

For more information, see
`http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#poisson(int,%20double)`.

Examples

Consider the query `qColt49` in Example 10–154. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–155, the query returns the relation in Example 10–156.

Example 10–154 poisson Function Query

```
<query id="qColt49"><![CDATA[
    select poisson(c1,c2) from SColtFunc
]]></query>
```

Example 10–155 poisson Function Stream Input

```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–156 poisson Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.909796
1000:       +           0.9992145
1200:       +           0.9870295
2000:       +           1.0
```

poissonComplemented

Syntax

Purpose

`poissonComplemented` is based on `cern.jet.stat.Probability`. It returns the sum of the terms `k+1` to Infinity of the Poisson distribution (see Figure 10–17) as a `double`.

Figure 10–17 cern.jet.stat.Probability poissonComplemented

$$\sum_{j=k+1}^{\infty} e^{-m} \frac{m^j}{j!}$$

This function takes the following arguments:

- `integer1`: the start term `k`.
- `double2`: the mean of the Poisson distribution `m`.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#poissonComplemented(int,%20double)`.

Examples

Consider the query `qColt50` in Example 10–157. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–158, the query returns the relation in Example 10–159.

Example 10–157 poissonComplemented Function Query

```
<query id="qColt50"><![CDATA[
    select poissonComplemented(c1,c2) from SColtFunc
]]></query>
```

Example 10–158 poissonComplemented Function Stream Input

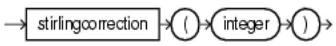
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–159 poissonComplemented Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.09020401
1000:       +           7.855354E-4
1200:       +           0.012970487
2000:       +           5.043364E-10
```

stirlingCorrection

Syntax



Purpose

`stirlingCorrection` is based on `cern.jet.math.Arithmetic`. It returns the correction term of the Stirling approximation of the natural logarithm (base *e*) of the factorial of the `integer` argument (see Figure 10–18) as a `double`.

Figure 10–18 cern.jet.math.Arithmetic stirlingCorrection

$$\log k! = (k + \frac{1}{2})\log(k+1) - (k+1) + (\frac{1}{2})\log(2\pi) + STIRLINGCORRECTION(k+1)$$

$$\log k! = (k + \frac{1}{2})\log(k) - k + (\frac{1}{2})\log(2\pi) + STIRLINGCORRECTION(k)$$

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#stirlingCorrection(int)`.

Examples

Consider the query `qColt5` in Example 10–160. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–161, the query returns the relation in Example 10–162.

Example 10–160 stirlingCorrection Function Query

```
<query id="qColt5"><![CDATA[
    select stirlingCorrection(c1) from SColtFunc
]]></query>
```

Example 10–161 stirlingCorrection Function Stream Input

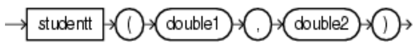
```
Timestamp    Tuple
  10         1,0.5,8
1000         4,0.7,6
1200         3,0.89,12
2000         8,0.4,4
```

Example 10–162 stirlingCorrection Function Relation Output

```
Timestamp    Tuple Kind  Tuple
  10:        +           0.08106147
1000:        +           0.020790672
1200:        +           0.027677925
2000:        +           0.010411265
```

studentT

Syntax



Purpose

`studentT` is based on `cern.jet.stat.Probability`. It returns the integral from minus infinity to `t` of the Student-t distribution with `k` > 0 degrees of freedom (see Figure 10–19) as a `double`.

Figure 10–19 cern.jet.stat.Probability studentT

$$\frac{\Gamma(\frac{k+1}{2})}{\sqrt{k\pi}\,\Gamma(\frac{k}{2})}\int_{-\infty}^{t}\left(1+\frac{x^2}{k}\right)^{\frac{-(K+1)}{2}}dx$$

This function takes the following arguments:

- `double1`: the degrees of freedom `k`.
- `double2`: the integration end point `t`.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#studentT(double,%20double)`.

Examples

Consider the query `qColt51` in Example 10–163. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–164, the query returns the relation in Example 10–165.

Example 10–163 studentT Function Query

```
<query id="qColt51"><![CDATA[
    select studentT(c2,c2) from SColtFunc
]]></query>
```

Example 10–164 studentT Function Stream Input

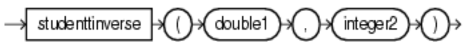
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–165 studentT Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.621341
1000:       +           0.67624015
1200:       +           0.7243568
2000:       +           0.5930112
```

studentTInverse

Syntax



Purpose

`studentTInverse` is based on `cern.jet.stat.Probability`. It returns the `double` value, `t`, for which the area under the Student-t probability density function (integrated from minus infinity to `t`) equals `1-alpha/2`. The value returned corresponds to the usual Student t-distribution lookup table for `talpha[size]`. This function uses the `studentt` function to determine the return value iteratively.

This function takes the following arguments:

- `double1`: the probability `alpha`.
- `integer2`: the data set size.

For more information, see:

- `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#studentTInverse(double,%20int)`
- "studentT" on page 10-59

Examples

Consider the query `qColt52` in Example 10–166. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–167, the query returns the relation in Example 10–168.

Example 10–166 studentTInverse Function Query

```
<query id="qColt52"><![CDATA[
    select studentTInverse(c2,c1) from SColtFunc
]]></query>
```

Example 10–167 studentTInverse Function Stream Input

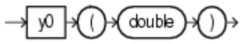
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–168 studentTInverse Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1.0
1000:       +           0.4141633
1200:       +           0.15038916
2000:       +           0.8888911
```

y0

Syntax



Purpose

y0 is based on cern.jet.math.Bessel. It returns the Bessel function of the second kind of order 0 of the double argument as a double.

For more information, see http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#y0(double).

Examples

Consider the query qColt24 in Example 10–169. Given the data stream SColtFunc with schema (c1 integer, c2 double, c3 bigint) in Example 10–170, the query returns the relation in Example 10–171.

Example 10–169 y0 Function Query

```
<query id="qColt24"><![CDATA[
    select y0(c2) from SColtFunc
]]></query>
```

Example 10–170 y0 Function Stream Input

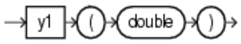
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–171 y0 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           -0.44451874
1000:       +           -0.19066493
1200:       +           -0.0031519707
2000:       +           -0.60602456
```

y1

Syntax



Purpose

`y1` is based on `cern.jet.math.Bessel`. It returns the Bessel function of the second kind of order 1 of the float argument as a `double`.

For more information, see `http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#y1(double)`.

Examples

Consider the query `qColt25` in Example 10–172. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 10–173, the query returns the relation in Example 10–174.

Example 10–172 y1 Function Query

```
<query id="qColt25"><![CDATA[
    select y1(c2) from SColtFunc
]]></query>
```

Example 10–173 y1 Function Stream Input

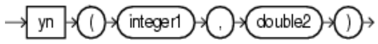
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–174 y1 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           -1.4714724
1000:       +           -1.1032499
1200:       +           -0.88294965
2000:       +           -1.780872
```

yn

Syntax



Purpose

yn is based on cern.jet.math.Bessel. It returns the Bessel function of the second kind of order n of the double argument as a double.

This function takes the following arguments:

- integer1: the n value order of the Bessel function.
- double2: the x value to compute the Bessel function of.

For more information, see http://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#yn(int,%20double).

Examples

Consider the query qColt26 in Example 10–175. Given the data stream SColtFunc with schema (c1 integer, c2 double, c3 bigint) in Example 10–176, the query returns the relation in Example 10–177.

Example 10–175 yn Function Query

```
<query id="qColt26"><![CDATA[
    select yn(c1,c2) from SColtFunc
]]></query>
```

Example 10–176 yn Function Stream Input

```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 10–177 yn Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +             -1.4714724
1000:       +           -132.63406
1200:       +             -8.020442
2000:       +             -6.3026547E8
```

11

Colt Aggregate Functions

This chapter provides a reference to Colt aggregate functions provided in Oracle Continuous Query Language (Oracle CQL). Colt aggregate functions are based on the Colt open source libraries for high performance scientific and technical computing.

For more information, see Section , "Functions".

This chapter includes the following section:

- Introduction to Oracle CQL Built-In Aggregate Colt Functions

Introduction to Oracle CQL Built-In Aggregate Colt Functions

Table 11–1 lists the built-in aggregate Colt functions that Oracle CQL provides.

Table 11–1 Oracle CQL Built-in Aggregate Colt-Based Functions

Colt Package	Function
`cern.jet.stat.Descriptive` A set of basic descriptive statistics functions.	▪ `autoCorrelation` ▪ `correlation` ▪ `covariance` ▪ `geometricMean` ▪ `geometricMean1` ▪ `harmonicMean` ▪ `kurtosis` ▪ `lag1` ▪ `mean` ▪ `meanDeviation` ▪ `median` ▪ `moment` ▪ `pooledMean` ▪ `pooledVariance` ▪ `product` ▪ `quantile` ▪ `quantileInverse` ▪ `rankInterpolated` ▪ `rms` ▪ `sampleKurtosis` ▪ `sampleKurtosisStandardError` ▪ `sampleSkew` ▪ `sampleSkewStandardError` ▪ `sampleVariance` ▪ `skew` ▪ `standardDeviation` ▪ `standardError` ▪ `sumOfInversions` ▪ `sumOfLogarithms` ▪ `sumOfPowerDeviations` ▪ `sumOfPowers` ▪ `sumOfSquaredDeviations` ▪ `sumOfSquares` ▪ `trimmedMean` ▪ `variance` ▪ `weightedMean` ▪ `winsorizedMean`

Note: Built-in function names are case sensitive and you must use them in the case shown (in lower case).

Note: In stream input examples, lines beginning with `h` (such as `h 3800`) are heartbeat input tuples. These inform Oracle Event Processing that no further input will have a timestamp lesser than the heartbeat value.

In relation output examples, the first tuple output is:

```
-9223372036854775808:+
```

This value is `-Long.MIN_VALUE()` and represents the largest negative timestamp possible.

For more information, see:

- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"
- Section , "Colt Aggregate Functions and the Where, Group By, and Having Clauses"
- Section , "Functions"
- Section , "Datatypes"
- `http://dst.lbl.gov/ACSSoftware/colt/`

Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments

Note that the signatures of the Oracle CQL Colt aggregate functions do not match the signatures of the corresponding Colt aggregate functions.

Consider the following Colt aggregate function:

```
double autoCorrelation(DoubleArrayList data, int lag, double mean, double variance)
```

In this signature, `data` is the `Collection` over which aggregates will be calculated and `mean` and `variance` are the other two parameter aggregates which are required to calculate `autoCorrelation` (where `mean` and `variance` aggregates are calculated on `data`).

In Oracle Event Processing, `data` will never come in the form of a `Collection`. The Oracle CQL function receives input data in a stream of tuples.

So suppose our stream is defined as `S:(double val, integer lag)`. On each input tuple, the Oracle CQL `autoCorrelation` function will compute two intermediate aggregates, `mean` and `variance`, and one final aggregate, `autoCorrelation`.

Since the function expects a stream of tuples having a `double data` value and an `integer lag` value only, the signature of the Oracle CQL `autoCorrelation` function is:

```
double autoCorrelation (double data, int lag)
```

Colt Aggregate Functions and the Where, Group By, and Having Clauses

In Oracle CQL, the `where` clause is applied before the `group by` and `having` clauses. This means the Oracle CQL statement in Example 11–1 is invalid:

Example 11–1 Invalid Use of count

```
<query id="q1"><![CDATA[
    select * from InputChannel[rows 4 slide 4] as ic where geometricMean(c3) > 4
]]></query>
```

Instead, you must use the Oracle CQL statement that Example 11–2 shows:

Example 11–2 Valid Use of count

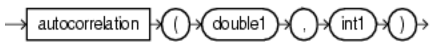
```
<query id="q1"><![CDATA[
    select * from InputChannel[rows 4 slide 4] as ic, myGeoMean =
geometricMean(c3) where myGeoMean > 4
]]></query>
```

For more information, see:

- "opt_where_clause::=" on page 22-4
- "opt_group_by_clause::=" on page 22-5
- "opt_having_clause::=" on page 22-5

autoCorrelation

Syntax



Purpose

autoCorrelation is based on cern.jet.stat.Descriptive.autoCorrelation(DoubleArrayList data, int lag, double mean, double variance). It returns the auto-correlation of a data sequence of the input arguments as a double.

> **Note:** This function has semantics different from "lag1" on page 11-18

This function takes the following tuple arguments:

- double1: data value.
- int1: lag.

For more information, see

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#autoCorrelation(cern.colt.list.DoubleArrayList,%20int,%20double,%20double)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query qColtAggr1 in Example 11–3. Given the data stream SColtAggrFunc with schema (c1 integer, c2 float, c3 double, c4 bigint) in Example 11–4, the query returns the relation in Example 11–5.

Example 11–3 autoCorrelation Function Query

```
<query id="qColtAggr1"><![CDATA[
     select autoCorrelation(c3, 0) from SColtAggrFunc
]]></query>
```

Example 11–4 autoCorrelation Function Stream Input

```
Timestamp   Tuple
  10        5.441341838866902
1000        6.1593756700951054
1200        3.7269733222923676
1400        4.625160266213489
1600        3.490061774090248
1800        3.6354484064421917
2000        5.635401664977703
2200        5.006087562207967
2400        3.632574304861612
2600        7.618087248962962
h 8000
h 200000000
```

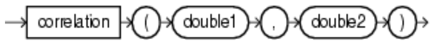
Example 11–5 autoCorrelation Function Relation Output

```
Timestamp   Tuple Kind  Tuple
```

```
-9223372036854775808:+
  10:       -
  10:       +          NaN
1000:       -          NaN
1000:       +          1.0
1200:       -          1.0
1200:       +          1.0
1400:       -          1.0
1400:       +          1.0
1600:       -          1.0
1600:       +          1.000000000000002
1800:       -          1.000000000000002
1800:       +          1.0
2000:       -          1.0
2000:       +          0.9999999999999989
2200:       -          0.9999999999999989
2200:       +          0.999999999999999
2400:       -          0.999999999999999
2400:       +          0.9999999999999991
2600:       -          0.9999999999999991
2600:       +          1.0000000000000013
```

correlation

Syntax



Purpose

`correlation` is based on `cern.jet.stat.Descriptive.correlation(DoubleArrayList data1, double standardDev1, DoubleArrayList data2, double standardDev2)` . It returns the correlation of two data sequences of the input arguments as a `double`.

This function takes the following tuple arguments:

- `double1`: data value 1.
- `double2`: data value 2.

For more information, see

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#correlation(cern.colt.list.DoubleArrayList,%20double,%20cern.colt.list.DoubleArrayList,%20double)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr2` in Example 11–6. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–7, the query returns the relation in Example 11–8.

Example 11–6 correlation Function Query

```
<query id="qColtAggr2"><![CDATA[
     select correlation(c3, c3) from SColtAggrFunc
]]></query>
```

Example 11–7 correlation Function Stream Input

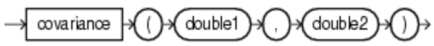
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 11–8 correlation Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           NaN
1000:       -           NaN
1000:       +           2.0
1200:       -           2.0
1200:       +           1.5
2000:       -           1.5
2000:       +           1.3333333333333333
```

covariance

Syntax



Purpose

`covariance` is based on `cern.jet.stat.Descriptive.covariance(DoubleArrayList data1, DoubleArrayList data2)`. It returns the correlation of two data sequences (see Figure 11–1) of the input arguments as a `double`.

Figure 11–1 cern.jet.stat.Descriptive.covariance

$$\mathrm{cov}(x, y) = (\frac{1}{size() - 1}) * Sum(x[i] - mean(x)) * (y[i] - mean(y))$$

This function takes the following tuple arguments:

- `double1`: data value 1.
- `double2`: data value 2.

For more information, see:

- `http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#covariance(cern.colt.list.DoubleArrayList,%20cern.colt.list.DoubleArrayList)`
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr3` in Example 11–9. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–10, the query returns the relation in Example 11–11.

Example 11–9 covariance Function Query

```
<query id="qColtAggr3"><![CDATA[
     select covariance(c3, c3) from SColtAggrFunc
]]></query>
```

Example 11–10 covariance Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

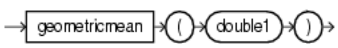
Example 11–11 covariance Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           NaN
1000:       -           NaN
1000:       +           50.0
```

```
1200:       -           50.0
1200:       +           100.0
2000:       -           100.0
2000:       +           166.66666666666666
```

geometricMean

Syntax



Purpose

geometricMean is based on cern.jet.stat.Descriptive.geometricMean(DoubleArrayList data). It returns the geometric mean of a data sequence (see Figure 11–2) of the input argument as a double.

Figure 11–2 cern.jet.stat.Descriptive.geometricMean(DoubleArrayList data)

$$pow(product(data[i]), \frac{1}{data.size()})$$

This function takes the following tuple arguments:

- double1: data value.

Note that for a geometric mean to be meaningful, the minimum of the data values must not be less than or equal to zero.

For more information, see:

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#geometricMean(cern.colt.list.DoubleArrayList)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query qColtAggr6 in Example 11–12. Given the data stream SColtAggrFunc with schema (c1 integer, c2 float, c3 double, c4 bigint) in Example 11–13, the query returns the relation in Example 11–14.

Example 11–12 geometricMean Function Query

```
<query id="qColtAggr6"><![CDATA[
    select geometricMean(c3) from SColtAggrFunc
]]></query>
```

Example 11–13 geometricMean Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

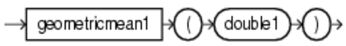
Example 11–14 geometricMean Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
```

```
  10:       +          40.0
1000:       -          40.0
1000:       +          34.64101615137755
1200:       -          34.64101615137755
1200:       +          28.844991406148168
2000:       -          28.844991406148168
2000:       +          22.133638394006436
```

geometricMean1

Syntax



Purpose

`geometricMean1` is based on `cern.jet.stat.Descriptive.geometricMean(double sumOfLogarithms)`. It returns the geometric mean of a data sequence (see Figure 11–3) of the input arguments as a `double`.

Figure 11–3 cern.jet.stat.Descriptive.geometricMean1(int size, double sumOfLogarithms)

$$pow(product(data[i]), \frac{1}{size})$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#geometricMean(cern.colt.list.DoubleArrayList)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr7` in Example 11–15. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–16, the query returns the relation in Example 11–17.

Example 11–15 geometricMean1 Function Query

```
<query id="qColtAggr7"><![CDATA[
    select geometricMean1(c3) from SColtAggrFunc
]]></query>
```

Example 11–16 geometricMean1 Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

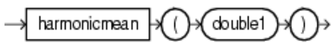
Example 11–17 geometricMean1 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           Infinity
1000:       -           Infinity
1000:       +           Infinity
```

```
1200:       -           Infinity
1200:       +           Infinity
2000:       -           Infinity
2000:       +           Infinity
```

harmonicMean

Syntax



Purpose

`harmonicMean` is based on `cern.jet.stat.Descriptive.harmonicMean(int size, double sumOfInversions)`. It returns the harmonic mean of a data sequence as a `double`.

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#harmonicMean(int,%20double)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr8` in Example 11–18. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–19, the query returns the relation in Example 11–20.

Example 11–18 harmonicMean Function Query

```
<query id="qColtAggr8"><![CDATA[
    select harmonicMean(c3) from SColtAggrFunc
]]></query>
```

Example 11–19 harmonicMean Function Stream Input

```
Timestamp   Tuple
  10        5.441341838866902
1000        6.1593756700951054
1200        3.7269733222923676
1400        4.625160266213489
1600        3.490061774090248
1800        3.6354484064421917
2000        5.635401664977703
2200        5.006087562207967
2400        3.632574304861612
2600        7.618087248962962
h 8000
h 200000000
```

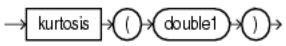
Example 11–20 harmonicMean Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           5.441341876983643
1000:       -           5.441341876983643
1000:       +           5.778137193205395
1200:       -           5.778137193205395
1200:       +           4.882442561720335
1400:       -           4.882442561720335
```

```
1400:       +          4.815475325819701
1600:       -          4.815475325819701
1600:       +          4.475541862878903
1800:       -          4.475541862878903
1800:       +          4.309563447664887
2000:       -          4.309563447664887
2000:       +          4.45944509362759
2200:       -          4.45944509362759
2200:       +          4.5211563834502515
2400:       -          4.5211563834502515
2400:       +          4.401525382790638
2600:       -          4.401525382790638
2600:       +          4.595562422157167
```

kurtosis

Syntax



Purpose

`kurtosis` is based on `cern.jet.stat.Descriptive.kurtosis(DoubleArrayList data, double mean, double standardDeviation)`. It returns the kurtosis or excess (see Figure 11–4) of a data sequence as a `double`.

Figure 11–4 cern.jet.stat.Descriptive.kurtosis(DoubleArrayList data, double mean, double standardDeviation)

$$-3+\frac{moment(data,4,mean)}{\text{StandardDeviation}^4}$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#kurtosis(cern.colt.list.DoubleArrayList,%20double,%20double)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr12` in Example 11–21. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–22, the query returns the relation in Example 11–23.

Example 11–21 kurtosis Function Query

```
<query id="qColtAggr12"><![CDATA[
    select kurtosis(c3) from SColtAggrFunc
]]></query>
```

Example 11–22 kurtosis Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

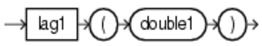
Example 11–23 kurtosis Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           NaN
1000:       -           NaN
1000:       +           -2.0
```

```
1200:       -           -2.0
1200:       +           -1.5000000000000002
2000:       -           -1.5000000000000002
2000:       +           -1.3600000000000003
```

lag1

Syntax



Purpose

`lag1` is based on `cern.jet.stat.Descriptive.lag1(DoubleArrayList data, double mean)`. It returns the `lag - 1` auto-correlation of a dataset as a `double`.

Note: This function has semantics different from "autoCorrelation" on page 11-5.

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#lag1(cern.colt.list.DoubleArrayList,%20double)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr14` in Example 11–24. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–25, the query returns the relation in Example 11–26.

Example 11–24 lag1 Function Query

```
<query id="qColtAggr14"><![CDATA[
    select lag1(c3) from SColtAggrFunc
]]></query>
```

Example 11–25 lag1 Function Stream Input

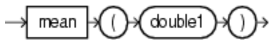
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 11–26 lag1 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           NaN
1000:       -           NaN
1000:       +           -0.5
1200:       -           -0.5
1200:       +           0.0
2000:       -           0.0
2000:       +           0.25
```

mean

Syntax



Purpose

`mean` is based on `cern.jet.stat.Descriptive.mean(DoubleArrayList data)`. It returns the arithmetic mean of a data sequence (see Figure 11–5) as a `double`.

Figure 11–5   cern.jet.stat.Descriptive.mean(DoubleArrayList data)

$$\frac{sum(data[i])}{data.size()}$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- `http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#mean(cern.colt.list.DoubleArrayList)`
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr16` in Example 11–27. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–28, the query returns the relation in Example 11–29.

Example 11–27   mean Function Query

```
<query id="qColtAggr16"><![CDATA[
    select mean(c3) from SColtAggrFunc
]]></query>
```

Example 11–28   mean Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

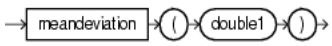
Example 11–29   mean Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           40.0
1000:       -           40.0
1000:       +           35.0
1200:       -           35.0
1200:       +           30.0
```

```
2000:       -           30.0
2000:       +           25.0
```

meanDeviation

Syntax



Purpose

`meanDeviation` is based on `cern.jet.stat.Descriptive.meanDeviation(DoubleArrayList data, double mean)`. It returns the mean deviation of a dataset (see Figure 11–6) as a `double`.

Figure 11–6 cern.jet.stat.Descriptive.meanDeviation(DoubleArrayList data, double mean)

$$\frac{sum(Math.abs(data[i]-mean))}{data.size()}$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#meanDeviation(cern.colt.list.DoubleArrayList,%20double)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr17` in Example 11–30. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–31, the query returns the relation in Example 11–32.

Example 11–30 meanDeviation Function Query

```
<query id="qColtAggr17"><![CDATA[
    select meanDeviation(c3) from SColtAggrFunc
]]></query>
```

Example 11–31 meanDeviation Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

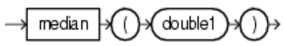
Example 11–32 meanDeviation Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           0.0
1000:       -           0.0
1000:       +           5.0
1200:       -           5.0
```

```
1200:       +          6.666666666666667
2000:       -          6.666666666666667
2000:       +          10.0
```

median

Syntax



Purpose

`median` is based on `cern.jet.stat.Descriptive.median(DoubleArrayList sortedData)`. It returns the median of a sorted data sequence as a `double`.

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- `http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#median(cern.colt.list.DoubleArrayList)`
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr18` in Example 11–33. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–34, the query returns the relation in Example 11–35.

Example 11–33 median Function Query

```
<query id="qColtAggr18"><![CDATA[
    select median(c3) from SColtAggrFunc
]]></query>
```

Example 11–34 median Function Stream Input

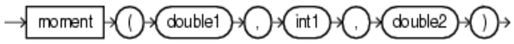
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 11–35 median Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           40.0
1000:       -           40.0
1000:       +           35.0
1200:       -           35.0
1200:       +           30.0
2000:       -           30.0
2000:       +           25.0
```

moment

Syntax



Purpose

`moment` is based on `cern.jet.stat.Descriptive.moment(DoubleArrayList data, int k, double c)`. It returns the moment of the `k`-th order with constant `c` of a data sequence (see Figure 11–7) as a `double`.

Figure 11–7 cern.jet.stat.Descriptive.moment(DoubleArrayList data, int k, double c)

$$\frac{sum((data[i]-c)^{k})}{data.size()}$$

This function takes the following tuple arguments:

- `double1`: data value.
- `int1`: `k`.
- `double2`: `c`.

For more information, see:

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#moment(cern.colt.list.DoubleArrayList,%20int,%20double)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr21` in Example 11–36. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–37, the query returns the relation in Example 11–38.

Example 11–36 moment Function Query

```
<query id="qColtAggr21"><![CDATA[
    select moment(c3, c1, c3) from SColtAggrFunc
]]></query>
```

Example 11–37 moment Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

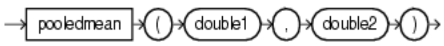
Example 11–38 moment Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           0.0
```

```
1000:      -         0.0
1000:      +         5000.0
1200:      -         5000.0
1200:      +         3000.0
2000:      -         3000.0
2000:      +         1.7045E11
```

pooledMean

Syntax



Purpose

`pooledMean` is based on `cern.jet.stat.Descriptive.pooledMean(int size1, double mean1, int size2, double mean2)`. It returns the pooled mean of two data sequences (see Figure 11–8) as a `double`.

Figure 11–8 cern.jet.stat.Descriptive.pooledMean(int size1, double mean1, int size2, double mean2)

$$\frac{(size1 * mean1 + size2 * mean2)}{(size1 + size2)}$$

This function takes the following tuple arguments:

- `double1`: mean 1.
- `double2`: mean 2.

For more information, see

- `http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#pooledMean(int,%20double,%20int,%20double)`
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr22` in Example 11–39. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–40, the query returns the relation in Example 11–41.

Example 11–39 pooledMean Function Query

```
<query id="qColtAggr22"><![CDATA[
    select pooledMean(c3, c3) from SColtAggrFunc
]]></query>
```

Example 11–40 pooledMean Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

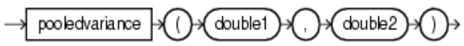
Example 11–41 pooledMean Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           40.0
```

```
1000:       -           40.0
1000:       +           35.0
1200:       -           35.0
1200:       +           30.0
2000:       -           30.0
2000:       +           25.0
```

pooledVariance

Syntax



Purpose

`pooledVariance` is based on `cern.jet.stat.Descriptive.pooledVariance(int size1, double variance1, int size2, double variance2)`. It returns the pooled variance of two data sequences (see Figure 11–9) as a `double`.

Figure 11–9 cern.jet.stat.Descriptive.pooledVariance(int size1, double variance1, int size2, double variance2)

$$\frac{(size1 * \text{variance1} + size2 * \text{variance2})}{(size1 + size2)}$$

This function takes the following tuple arguments:

- `double1`: variance 1.
- `double2`: variance 2.

For more information, see

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#pooledVariance(int,%20double,%20int,%20double)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr23` in Example 11–42. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–43, the query returns the relation in Example 11–44.

Example 11–42 pooledVariance Function Query

```
<query id="qColtAggr23"><![CDATA[
    select pooledVariance(c3, c3) from SColtAggrFunc
]]></query>
```

Example 11–43 pooledVariance Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

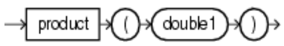
Example 11–44 pooledVariance Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           0.0
```

```
1000:       -           0.0
1000:       +           25.0
1200:       -           25.0
1200:       +           66.66666666666667
2000:       -           66.66666666666667
2000:       +           125.0
```

product

Syntax



Purpose

`product` is based on `cern.jet.stat.Descriptive.product(DoubleArrayList data)`. It returns the product of a data sequence (see Figure 11–10) as a `double`.

Figure 11–10 cern.jet.stat.Descriptive.product(DoubleArrayList data)

$$data[0] * data[1] * \ldots * data[data.size() - 1]$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- `http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#product(cern.colt.list.DoubleArrayList)`
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr24` in Example 11–45. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–46, the query returns the relation in Example 11–47.

Example 11–45 product Function Query

```
<query id="qColtAggr24"><![CDATA[
    select product(c3) from SColtAggrFunc
]]></query>
```

Example 11–46 product Function Stream Input

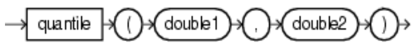
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 11–47 product Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           40.0
1000:       -           40.0
1000:       +           1200.0
1200:       -           1200.0
1200:       +           24000.0
2000:       -           24000.0
2000:       +           240000.0
```

quantile

Syntax



Purpose

quantile is based on cern.jet.stat.Descriptive.quantile(DoubleArrayList sortedData, double phi). It returns the phi-quantile as a double; that is, an element elem for which holds that phi percent of data elements are less than elem.

This function takes the following tuple arguments:

- double1: data value.
- double2: phi; the percentage; must satisfy 0 <= phi <= 1.

For more information, see:

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#quantile(cern.colt.list.DoubleArrayList,%20double)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query qColtAggr26 in Example 11–48. Given the data stream SColtAggrFunc with schema (c1 integer, c2 float, c3 double, c4 bigint) in Example 11–49, the query returns the relation in Example 11–50.

Example 11–48 quantile Function Query

```
<query id="qColtAggr26"><![CDATA[
    select quantile(c3, c2) from SColtAggrFunc
]]></query>
```

Example 11–49 quantile Function Stream Input

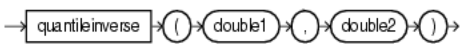
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 11–50 quantile Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:         -
  10:         +         40.0
1000:         -         40.0
1000:         +         36.99999988079071
1200:         -         36.99999988079071
1200:         +         37.799999713897705
2000:         -         37.799999713897705
2000:         +         22.000000178813934
```

quantileInverse

Syntax



Purpose

`quantileInverse` is based on `cern.jet.stat.Descriptive.quantileInverse(DoubleArrayList sortedList, double element)`. It returns the percentage phi of elements `<= element` (`0.0 <= phi <= 1.0`) as a `double`. This function does linear interpolation if the `element` is not contained but lies in between two contained elements.

This function takes the following tuple arguments:

- `double1`: data.
- `double2`: `element`.

For more information, see:

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#quantileInverse(cern.colt.list.DoubleArrayList,%20double)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr27` in Example 11–51. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–52, the query returns the relation in Example 11–53.

Example 11–51 quantileInverse Function Query

```
<query id="qColtAggr27"><![CDATA[
    select quantileInverse(c3, c3) from SColtAggrFunc
]]></query>
```

Example 11–52 quantileInverse Function Stream Input

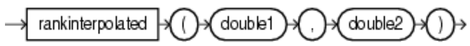
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 11–53 quantileInverse Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           1.0
1000:       -           1.0
1000:       +           0.5
1200:       -           0.5
1200:       +           0.3333333333333333
2000:       -           0.3333333333333333
2000:       +           0.25
```

rankInterpolated

Syntax



Purpose

`rankInterpolated` is based on `cern.jet.stat.Descriptive.rankInterpolated(DoubleArrayList sortedList, double element)`. It returns the linearly interpolated number of elements in a list less or equal to a given `element` as a double.

The rank is the number of elements `<= element`. Ranks are of the form `{0, 1, 2,..., sortedList.size()}`. If no element is `<= element`, then the rank is zero. If the element lies in between two contained elements, then linear interpolation is used and a non-integer value is returned.

This function takes the following tuple arguments:

- `double1`: data value.
- `double2`: `element`.

For more information, see:

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#rankInterpolated(cern.colt.list.DoubleArrayList,%20double)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr29` in Example 11–54. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–55, the query returns the relation in Example 11–56.

Example 11–54 rankInterpolated Function Query

```
<query id="qColtAggr29"><![CDATA[
    select rankInterpolated(c3, c3) from SColtAggrFunc
]]></query>
```

Example 11–55 rankInterpolated Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

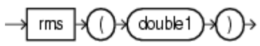
Example 11–56 rankInterpolated Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           1.0
1000:       -           1.0
1000:       +           1.0
```

```
1200:      -          1.0
1200:      +          1.0
2000:      -          1.0
2000:      +          1.0
```

rms

Syntax



Purpose

`rms` is based on `cern.jet.stat.Descriptive.rms(int size, double sumOfSquares)`. It returns the Root-Mean-Square (RMS) of a data sequence (see Figure 11–11) as a `double`.

Figure 11–11 cern.jet.stat.Descriptive.rms(int size, double sumOfSquares)

$$Math.sqrt(\frac{Sum(data[i] * data[i])}{data.size()})$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see

- `http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#rms(int,%20double)`
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr30` in Example 11–57. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–58, the query returns the relation in Example 11–59.

Example 11–57 rms Function Query

```
<query id="qColtAggr30"><![CDATA[
    select rms(c3) from SColtAggrFunc
]]></query>
```

Example 11–58 rms Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

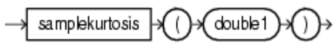
Example 11–59 rms Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:        -
  10:        +          40.0
1000:        -          40.0
1000:        +          35.35533905932738
1200:        -          35.35533905932738
```

```
1200:       +          31.09126351029605
2000:       -          31.09126351029605
2000:       +          27.386127875258307
```

sampleKurtosis

Syntax



Purpose

`sampleKurtosis` is based on `cern.jet.stat.Descriptive.sampleKurtosis(DoubleArrayList data, double mean, double sampleVariance)`. It returns the sample kurtosis (excess) of a data sequence as a `double`.

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- `http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#sampleKurtosis(cern.colt.list.DoubleArrayList,%20double,%20double)`
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr31` in Example 11–60. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–61, the query returns the relation in Example 11–62.

Example 11–60 sampleKurtosis Function Query

```
<query id="qColtAggr31"><![CDATA[
     select sampleKurtosis(c3) from SColtAggrFunc
]]></query>
```

Example 11–61 sampleKurtosis Function Stream Input

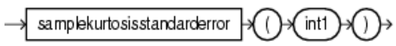
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 11–62 sampleKurtosis Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           NaN
1000:       -           NaN
1000:       +           NaN
1200:       -           NaN
1200:       +           NaN
2000:       -           NaN
2000:       +           -1.1999999999999993
```

sampleKurtosisStandardError

Syntax



Purpose

`sampleKurtosisStandardError` is based on `cern.jet.stat.Descriptive.sampleKurtosisStandardError(int size)`. It returns the standard error of the sample Kurtosis as a `double`.

This function takes the following tuple arguments:

- `int1`: data value.

For more information, see:

- `http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#sampleKurtosisStandardError(int)`
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr33` in Example 11–63. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–64, the query returns the relation in Example 11–65.

Example 11–63 sampleKurtosisStandardError Function Query

```
<query id="qColtAggr33"><![CDATA[
     select sampleKurtosisStandardError(c1) from SColtAggrFunc
]]></query>
```

Example 11–64 sampleKurtosisStandardError Function Stream Input

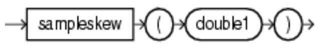
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 11–65 sampleKurtosisStandardError Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           0.0
1000:       -           0.0
1000:       +           Infinity
1200:       -           Infinity
1200:       +           Infinity
2000:       -           Infinity
2000:       +           2.6186146828319083
```

sampleSkew

Syntax



Purpose

`sampleSkew` is based on `cern.jet.stat.Descriptive.sampleSkew(DoubleArrayList data, double mean, double sampleVariance)`. It returns the sample skew of a data sequence as a `double`.

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- `http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#sampleSkew(cern.colt.list.DoubleArrayList,%20double,%20double)`
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr34` in Example 11–66. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–67, the query returns the relation in Example 11–68.

Example 11–66 sampleSkew Function Query

```
<query id="qColtAggr34"><![CDATA[
    select sampleSkew(c3) from SColtAggrFunc
]]></query>
```

Example 11–67 sampleSkew Function Stream Input

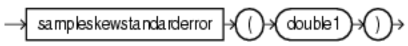
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 11–68 sampleSkew Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           NaN
1000:       -           NaN
1000:       +           NaN
1200:       -           NaN
1200:       +           0.0
2000:       -           0.0
2000:       +           0.0
```

sampleSkewStandardError

Syntax



Purpose

`sampleSkewStandardError` is based on `cern.jet.stat.Descriptive.sampleSkewStandardError(int size)`. It returns the standard error of the sample skew as a `double`.

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#sampleSkewStandardError(int)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr36` in Example 11–69. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–70, the query returns the relation in Example 11–71.

Example 11–69 sampleSkewStandardError Function Query

```
<query id="qColtAggr36"><![CDATA[
    select sampleSkewStandardError(c1) from SColtAggrFunc
]]></query>
```

Example 11–70 sampleSkewStandardError Function Stream Input

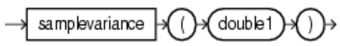
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 11–71 sampleSkewStandardError Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           -0.0
1000:       -           -0.0
1000:       +           Infinity
1200:       -           Infinity
1200:       +           1.224744871391589
2000:       -           1.224744871391589
2000:       +           1.01418510567422
```

sampleVariance

Syntax



Purpose

`sampleVariance` is based on `cern.jet.stat.Descriptive.sampleVariance(DoubleArrayList data, double mean)`. It returns the sample variance of a data sequence (see Figure 11–12) as a `double`.

Figure 11–12 cern.jet.stat.Descriptive.sampleVariance(DoubleArrayList data, double mean)

$$\frac{Sum((data[i]-mean)^2)}{(data.size()-1)}$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#sampleVariance(cern.colt.list.DoubleArrayList,%20double)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr38` in Example 11–72. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–73, the query returns the relation in Example 11–74.

Example 11–72 sampleVariance Function Query

```
<query id="qColtAggr38"><![CDATA[
    select sampleVariance(c3) from SColtAggrFunc
]]></query>
```

Example 11–73 sampleVariance Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

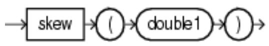
Example 11–74 sampleVariance Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           NaN
1000:       -           NaN
1000:       +           50.0
```

```
1200:       -           50.0
1200:       +           100.0
2000:       -           100.0
2000:       +           166.66666666666666
```

skew

Syntax



Purpose

`skew` is based on `cern.jet.stat.Descriptive.skew(DoubleArrayList data, double mean, double standardDeviation)`. It returns the skew of a data sequence of a data sequence (see Figure 11–13) as a `double`.

Figure 11–13 cern.jet.stat.Descriptive.skew(DoubleArrayList data, double mean, double standardDeviation)

$$\frac{moment(data, 3, mean)}{\mathrm{standardDeviation}^3}$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- `http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#skew(cern.colt.list.DoubleArrayList,%20double,%20double)`
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr41` in Example 11–75. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–76, the query returns the relation in Example 11–77.

Example 11–75 skew Function Query

```
<query id="qColtAggr41"><![CDATA[
    select skew(c3) from SColtAggrFunc
]]></query>
```

Example 11–76 skew Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

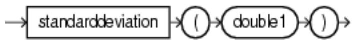
Example 11–77 skew Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           NaN
1000:       -           NaN
1000:       +           0.0
```

```
1200:      -         0.0
1200:      +         0.0
2000:      -         0.0
2000:      +         0.0
```

standardDeviation

Syntax



Purpose

standardDeviation is based on cern.jet.stat.Descriptive.standardDeviation(double variance). It returns the standard deviation from a variance as a double.

This function takes the following tuple arguments:

- double1: data value.

For more information, see

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#standardDeviation(double)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query qColtAggr44 in Example 11–78. Given the data stream SColtAggrFunc with schema (c1 integer, c2 float, c3 double, c4 bigint) in Example 11–79, the query returns the relation in Example 11–80.

Example 11–78 standardDeviation Function Query

```
<query id="qColtAggr44"><![CDATA[
    select standardDeviation(c3) from SColtAggrFunc
]]></query>
```

Example 11–79 standardDeviation Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 11–80 standardDeviation Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           0.0
1000:       -           0.0
1000:       +           5.0
1200:       -           5.0
1200:       +           8.16496580927726
2000:       -           8.16496580927726
2000:       +           11.180339887498949
```

standardError

Syntax

Purpose

`standardError` is based on `cern.jet.stat.Descriptive.standardError(int size, double variance)`. It returns the standard error of a data sequence (see Figure 11–14) as a `double`.

Figure 11–14 cern.jet.stat.Descriptive.cern.jet.stat.Descriptive.standardError(int size, double variance)

$$Math.sqrt(\frac{\text{variance}}{size})$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see

- `http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#standardError(int,%20double)`
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr45` in Example 11–81. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–82, the query returns the relation in Example 11–83.

Example 11–81 standardError Function Query

```
<query id="qColtAggr45"><![CDATA[
     select standardError(c3) from SColtAggrFunc
]]></query>
```

Example 11–82 standardError Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

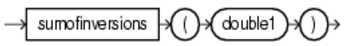
Example 11–83 standardError Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           0.0
1000:       -           0.0
1000:       +           3.5355339059327378
1200:       -           3.5355339059327378
```

```
1200:       +          4.714045207910317
2000:       -          4.714045207910317
2000:       +          5.5901699437494745
```

sumOfInversions

Syntax



Purpose

`sumOfInversions` is based on `cern.jet.stat.Descriptive.sumOfInversions(DoubleArrayList data, int from, int to)`. It returns the sum of inversions of a data sequence (see Figure 11–15) as a `double`.

Figure 11–15 cern.jet.stat.Descriptive.sumOfInversions(DoubleArrayList data, int from, int to)

$$Sum(\frac{1.0}{data[i]})$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#sumOfInversions(cern.colt.list.DoubleArrayList,%20int,%20int)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr48` in Example 11–84. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–85, the query returns the relation in Example 11–86.

Example 11–84 sumOfInversions Function Query

```
<query id="qColtAggr48"><![CDATA[
     select sumOfInversions(c3) from SColtAggrFunc
]]></query>
```

Example 11–85 sumOfInversions Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

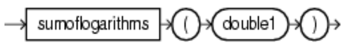
Example 11–86 sumOfInversions Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           Infinity
1000:       -           Infinity
```

```
1000:       +           Infinity
1200:       -           Infinity
1200:       +           Infinity
2000:       -           Infinity
2000:       +           Infinity
```

sumOfLogarithms

Syntax



Purpose

`sumOfLogarithms` is based on `cern.jet.stat.Descriptive.sumOfLogarithms(DoubleArrayList data, int from, int to)`. It returns the sum of logarithms of a data sequence (see Figure 11–16) as a `double`.

Figure 11–16 cern.jet.stat.Descriptive.sumOfLogarithms(DoubleArrayList data, int from, int to)

$$Sum(Log(data[i]))$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- `http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#sumOfLogarithms(cern.colt.list.DoubleArrayList,%20int,%20int)`
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr49` in Example 11–87. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–88, the query returns the relation in Example 11–89.

Example 11–87 sumOfLogarithms Function Query

```
<query id="qColtAggr49"><![CDATA[
    select sumOfLogarithms(c3) from SColtAggrFunc
]]></query>
```

Example 11–88 sumOfLogarithms Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

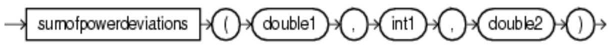
Example 11–89 sumOfLogarithms Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           -Infinity
1000:       -           -Infinity
1000:       +           -Infinity
1200:       -           -Infinity
```

```
1200:       +           -Infinity
2000:       -           -Infinity
2000:       +           -Infinity
```

sumOfPowerDeviations

Syntax



Purpose

`sumOfPowerDeviations` is based on `cern.jet.stat.Descriptive.sumOfPowerDeviations(DoubleArrayList data, int k, double c)`. It returns sum of power deviations of a data sequence (see Figure 11–17) as a `double`.

Figure 11–17 cern.jet.stat.Descriptive.sumOfPowerDeviations(DoubleArrayList data, int k, double c)

$$Sum((data[i]-c)^k)$$

This function is optimized for common parameters like `c == 0.0`, `k == -2 .. 4`, or both.

This function takes the following tuple arguments:

- `double1`: data value.
- `int1`: `k`.
- `double2`: `c`.

For more information, see:

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#sumOfPowerDeviations(cern.colt.list.DoubleArrayList,%20int,%20double)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr50` in Example 11–90. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–91, the query returns the relation in Example 11–92.

Example 11–90 sumOfPowerDeviations Function Query

```
<query id="qColtAggr50"><![CDATA[
    select sumOfPowerDeviations(c3, c1, c3) from SColtAggrFunc
]]></query>
```

Example 11–91 sumOfPowerDeviations Function Stream Input

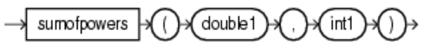
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 11–92 sumOfPowerDeviations Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           0.0
1000:       -           0.0
1000:       +           10000.0
1200:       -           10000.0
1200:       +           9000.0
2000:       -           9000.0
2000:       +           6.818E11
```

sumOfPowers

Syntax



Purpose

`sumOfPowers` is based on `cern.jet.stat.Descriptive.sumOfPowers(DoubleArrayList data, int k)`. It returns the sum of powers of a data sequence (see Figure 11–18) as a `double`.

Figure 11–18 cern.jet.stat.Descriptive.sumOfPowers(DoubleArrayList data, int k)

$$Sum(data[i]^k)$$

This function takes the following tuple arguments:

- `double1`: data value.
- `int1`: k.

For more information, see:

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#sumOfPowers(cern.colt.list.DoubleArrayList,%20int)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr52` in Example 11–93. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–94, the query returns the relation in Example 11–95.

Example 11–93 sumOfPowers Function Query

```
<query id="qColtAggr52"><![CDATA[
    select sumOfPowers(c3, c1) from SColtAggrFunc
]]></query>
```

Example 11–94 sumOfPowers Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

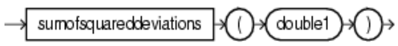
Example 11–95 sumOfPowers Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           40.0
1000:       -           40.0
1000:       +           3370000.0
1200:       -           3370000.0
```

```
1200:      +          99000.0
2000:      -          99000.0
2000:      +          7.2354E12
```

sumOfSquaredDeviations

Syntax



Purpose

`sumOfSquaredDeviations` is based on `cern.jet.stat.Descriptive.sumOfSquaredDeviations(int size, double variance)`. It returns the sum of squared mean deviation of a data sequence (see Figure 11–19) as a `double`.

Figure 11–19 cern.jet.stat.Descriptive.sumOfSquaredDeviations(int size, double variance)

$$variance * (size - 1) == Sum((data[i] - mean)^2)$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see

- `http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#sumOfSquaredDeviations(int,%20double)`
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr53` in Example 11–96. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–97, the query returns the relation in Example 11–98.

Example 11–96 sumOfSquaredDeviations Function Query

```
<query id="qColtAggr53"><![CDATA[
    select sumOfSquaredDeviations(c3) from SColtAggrFunc
]]></query>
```

Example 11–97 sumOfSquaredDeviations Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

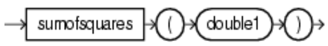
Example 11–98 sumOfSquaredDeviations Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           0.0
1000:       -           0.0
1000:       +           25.0
1200:       -           25.0
```

```
1200:       +           133.33333333333334
2000:       -           133.33333333333334
2000:       +           375.0
```

sumOfSquares

Syntax



Purpose

`sumOfSquares` is based on `cern.jet.stat.Descriptive.sumOfSquares(DoubleArrayList data)`. It returns the sum of squares of a data sequence (see Figure 11–20) as a `double`.

Figure 11–20 cern.jet.stat.Descriptive.sumOfSquares(DoubleArrayList data)

$$Sum(data[i] * data[i])$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- `http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#sumOfSquares(cern.colt.list.DoubleArrayList)`
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr54` in Example 11–99. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–100, the query returns the relation in Example 11–101.

Example 11–99 sumOfSquares Function Query

```
<query id="qColtAggr54"><![CDATA[
    select sumOfSquares(c3) from SColtAggrFunc
]]></query>
```

Example 11–100 sumOfSquares Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

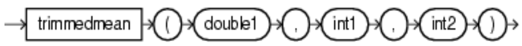
Example 11–101 sumOfSquares Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           1600.0
1000:       -           1600.0
1000:       +           2500.0
1200:       -           2500.0
1200:       +           2900.0
2000:       -           2900.0
```

```
2000:       +           3000.0
```

trimmedMean

Syntax



Purpose

`trimmedMean` is based on `cern.jet.stat.Descriptive.trimmedMean(DoubleArrayList sortedData, double mean, int left, int right)`. It returns the trimmed mean of an ascending sorted data sequence as a `double`.

This function takes the following tuple arguments:

- `double1`: data value.
- `int1`: `left`.
- `int2`: `right`.

For more information, see:

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#trimmedMean(cern.colt.list.DoubleArrayList,%20double,%20int,%20int)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr55` in Example 11–102. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–103, the query returns the relation in Example 11–104.

Example 11–102 trimmedMean Function Query

```
<query id="qColtAggr55"><![CDATA[
    select trimmedMean(c3, c1, c1) from SColtAggrFunc
]]></query>
```

Example 11–103 trimmedMean Function Stream Input

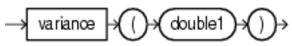
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 11–104 trimmedMean Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
```

variance

Syntax



Purpose

`variance` is based on `cern.jet.stat.Descriptive.variance(int size, double sum, double sumOfSquares)`. It returns the variance of a data sequence (see Figure 11–21) as a `double`.

Figure 11–21 cern.jet.stat.Descriptive.variance(int size, double sum, double sumOfSquares)

$$\frac{(SumofSquares - mean * sum)}{\text{size with mean}} = \frac{sum}{size}$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- `http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#variance(int,%20double,%20double)`
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr57` in Example 11–105. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–106, the query returns the relation in Example 11–107.

Example 11–105 variance Function Query

```
<query id="qColtAggr57"><![CDATA[
    select variance(c3) from SColtAggrFunc
]]></query>
```

Example 11–106 variance Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

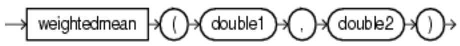
Example 11–107 variance Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           0.0
1000:       -           0.0
1000:       +           25.0
```

```
1200:       -           25.0
1200:       +           66.66666666666667
2000:       -           66.66666666666667
2000:       +           125.0
```

weightedMean

Syntax



Purpose

`weightedMean` is based on `cern.jet.stat.Descriptive.weightedMean(DoubleArrayList data, DoubleArrayList weights)`. It returns the weighted mean of a data sequence (see Figure 11–22) as a `double`.

Figure 11–22 cern.jet.stat.Descriptive.weightedMean(DoubleArrayList data, DoubleArrayList weights)

$$\frac{Sum(data[i] * weights[i])}{Sum(weights[i])}$$

This function takes the following tuple arguments:

- `double1`: data value.
- `double2`: weight value.

For more information, see:

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#weightedMean(cern.colt.list.DoubleArrayList,%20cern.colt.list.DoubleArrayList)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr58` in Example 11–108. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–109, the query returns the relation in Example 11–110.

Example 11–108 weightedMean Function Query

```
<query id="qColtAggr58"><![CDATA[
    select weightedMean(c3, c3) from SColtAggrFunc
]]></query>
```

Example 11–109 weightedMean Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

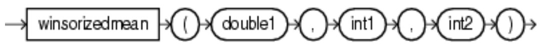
Example 11–110 weightedMean Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
```

```
  10:       -
  10:       +           40.0
1000:       -           40.0
1000:       +           35.714285714285715
1200:       -           35.714285714285715
1200:       +           32.22222222222222
2000:       -           32.22222222222222
2000:       +           30.0
```

winsorizedMean

Syntax



Purpose

`winsorizedMean` is based on `cern.jet.stat.Descriptive.winsorizedMean(DoubleArrayList sortedData, double mean, int left, int right)`. It returns the winsorized mean of a sorted data sequence as a `double`.

This function takes the following tuple arguments:

- `double1`: data value.
- `int1`: `left`.
- `int2`: `right`.

For more information, see:

- http://acs.lbl.gov/~hoschek/colt/api/cern/jet/stat/Descriptive.html#winsorizedMean(cern.colt.list.DoubleArrayList,%20double,%20int,%20int)
- Section , "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr60` in Example 11–111. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 11–112, the query returns the relation in Example 11–113.

Example 11–111 winsorizedMean Function Query

```
<query id="qColtAggr60"><![CDATA[
    select winsorizedMean(c3, c1, c1) from SColtAggrFunc
]]></query>
```

Example 11–112 winsorizedMean Function Stream Input

```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 11–113 winsorizedMean Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
```

12

java.lang.Math Functions

This chapter provides a reference to the `java.lang.Math` functions provided in Oracle Continuous Query Language (Oracle CQL).

For more information, see Section , "Functions".

This chapter includes the following section:

- Introduction to Oracle CQL Built-In java.lang.Math Functions

Introduction to Oracle CQL Built-In java.lang.Math Functions

Table 12–1 lists the built-in `java.lang.Math` functions that Oracle CQL provides.

Table 12–1 Oracle CQL Built-in java.lang.Math Functions

Type	Function
Trigonometric	▪ `sin` ▪ `cos` ▪ `tan` ▪ `asin` ▪ `acos` ▪ `atan` ▪ `atan2` ▪ `cosh` ▪ `sinh` ▪ `tanh`
Logarithmic	▪ `log1` ▪ `log101` ▪ `log1p`
Euler's Number	▪ `exp` ▪ `expm1`
Roots	▪ `cbrt` ▪ `sqrt` ▪ `hypot`
Signum Function	▪ `signum` ▪ `signum1`
Unit of Least Precision	▪ `ulp` ▪ `ulp1`

Table 12–1 (Cont.) Oracle CQL Built-in java.lang.Math Functions

Type	Function
Other	■ abs ■ abs1 ■ abs2 ■ abs3 ■ ceil1 ■ floor1 ■ IEEEremainder ■ pow ■ rint ■ round ■ round1 ■ todegrees ■ toradians

Note: Built-in function names are case sensitive and you must use them in the case shown (in lower case).

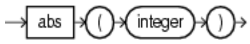
Note: In stream input examples, lines beginning with `h` (such as `h 3800`) are heartbeat input tuples. These inform Oracle Event Processing that no further input will have a timestamp lesser than the heartbeat value.

For more information, see:

- Section , "Functions"
- `http://java.sun.com/javase/6/docs/api/java/lang/Math.html`

abs

Syntax



Purpose

abs returns the absolute value of the input integer argument as an integer.

For more information, see http://java.sun.com/javase/6/docs/api/java/lang/Math.html#abs(int).

Examples

Consider the query q66 in Example 12–1. Given the data stream SFunc with schema (c1 integer, c2 double, c3 bigint) in Example 12–2, the query returns the stream in Example 12–3.

Example 12–1 abs Function Query

```
<query id="q66"><![CDATA[
    select abs(c1) from SFunc
]]></query>
```

Example 12–2 abs Function Stream Input

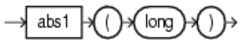
```
Timestamp   Tuple
  10         1,0.5,8
1000        -4,0.7,6
1200        -3,0.89,12
2000         8,0.4,4
```

Example 12–3 abs Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1
1000:       +           4
1200:       +           3
2000:       +           8
```

abs1

Syntax



Purpose

abs1 returns the absolute value of the input long argument as a long.

For more information, see
http://java.sun.com/javase/6/docs/api/java/lang/Math.html#abs(long).

Examples

Consider the query q67 in Example 12–4. Given the data stream SFunc with schema (c1 integer, c2 float, c3 long) in Example 12–5, the query returns the stream in Example 12–6.

Example 12–4 abs1 Function Query

```
<query id="q67"><![CDATA[
    select abs1(c3) from SFunc
]]></query>
```

Example 12–5 abs1 Function Stream Input

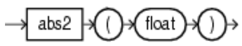
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,-6
1200        3,0.89,-12
2000        8,0.4,4
```

Example 12–6 abs1 Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           8
1000:       +           6
1200:       +           12
2000:       +           4
```

abs2

Syntax



Purpose

`abs2` returns the absolute value of the input `float` argument as a `float`.

For more information, see
`http://java.sun.com/javase/6/docs/api/java/lang/Math.html#abs(float)`.

Examples

Consider the query `q68` in Example 12–7. Given the data stream `SFunc` with schema `(c1 integer, c2 float, c3 bigint)` in Example 12–8, the query returns the stream in Example 12–9.

Example 12–7 abs2 Function Query

```
<query id="q68"><![CDATA[
    select abs2(c2) from SFunc
]]></query>
```

Example 12–8 abs2 Function Stream Input

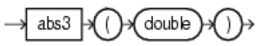
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,-0.7,6
1200        3,-0.89,12
2000        8,0.4,4
```

Example 12–9 abs2 Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.5
1000:       +           0.7
1200:       +           0.89
2000:       +           0.4
```

abs3

Syntax



Purpose

`abs3` returns the absolute value of the input `double` argument as a `double`.

For more information, see
`http://java.sun.com/javase/6/docs/api/java/lang/Math.html#abs(double)`.

Examples

Consider the query `q69` in Example 12–10. Given the data stream `SFunc` with schema `(c1 integer, c2 float, c3 bigint, c4 double)` in Example 12–11, the query returns the stream in Example 12–12.

Example 12–10   abs3 Function Query

```
<query id="q69"><![CDATA[
    select abs3(c4) from SFunc
]]></query>
```

Example 12–11   abs3 Function Stream Input

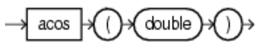
```
Timestamp   Tuple
  10        1,0.5,8,0.25334
1000        4,0.7,6,-4.64322
1200        3,0.89,12,-1.4672272
2000        8,0.4,4,2.66777
```

Example 12–12   abs3 Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.25334
1000:       +           4.64322
1200:       +           1.4672272
2000:       +           2.66777
```

acos

Syntax



Purpose

acos returns the arc cosine of a double angle, in the range of 0.0 through pi, as a double.

For more information, see http://java.sun.com/javase/6/docs/api/java/lang/Math.html#acos(double).

Examples

Consider the query q73 in Example 12–13. Given the data stream SFunc with schema (c1 integer, c2 double, c3 bigint) in Example 12–14, the query returns the stream in Example 12–15.

Example 12–13   acos Function Query

```
<query id="q73"><![CDATA[
    select acos(c2) from SFunc
]]></query>
```

Example 12–14   acos Function Stream Input

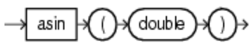
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–15   acos Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1.0471976
1000:       +           0.79539883
1200:       +           0.4734512
2000:       +           1.1592795
```

asin

Syntax



Purpose

`asin` returns the arc sine of a `double` angle, in the range of `-pi/2` through `pi/2`, as a `double`.

For more information, see `http://java.sun.com/javase/6/docs/api/java/lang/Math.html#asin(double)`.

Examples

Consider the query `q74` in Example 12–16. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–17, the query returns the stream in Example 12–18.

Example 12–16 asin Function Query

```
<query id="q74"><![CDATA[
    select asin(c2) from SFunc
]]></query>
```

Example 12–17 asin Function Stream Input

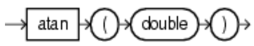
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–18 asin Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.5235988
1000:       +           0.7753975
1200:       +           1.0973451
2000:       +           0.41151685
```

atan

Syntax



Purpose

`atan` returns the arc tangent of a `double` angle, in the range of `-pi/2` through `pi/2`, as a `double`.

For more information, see
`http://java.sun.com/javase/6/docs/api/java/lang/Math.html#atan(double)`.

Examples

Consider the query `q75` in Example 12–19. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–20, the query returns the stream in Example 12–21.

Example 12–19 atan Function Query

```
<query id="q75"><![CDATA[
    select atan(c2) from SFunc
]]></query>
```

Example 12–20 atan Function Stream Input

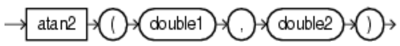
```
Timestamp    Tuple
  10         1,0.5,8
1000         4,0.7,6
1200         3,0.89,12
2000         8,0.4,4
```

Example 12–21 atan Function Stream Output

```
Timestamp    Tuple Kind  Tuple
  10:        +           0.4636476
1000:        +           0.61072594
1200:        +           0.7272627
2000:        +           0.3805064
```

atan2

Syntax



Purpose

`atan2` converts rectangular coordinates (`x,y`) to polar (`r,theta`) coordinates.

This function takes the following arguments:

- `double1`: the ordinate coordinate.
- `double2`: the abscissa coordinate.

This function returns the theta component of the point (`r,theta`) in polar coordinates that corresponds to the point (`x,y`) in Cartesian coordinates as a `double`.

For more information, see `http://java.sun.com/javase/6/docs/api/java/lang/Math.html#atan2(double,%20 double)`.

Examples

Consider the query `q63` in Example 12–22. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–23, the query returns the stream in Example 12–24.

Example 12–22 atan2 Function Query

```
<query id="q63"><![CDATA[
    select atan2(c2,c2) from SFunc
]]></query>
```

Example 12–23 atan2 Function Stream Input

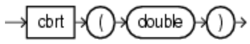
```
Timestamp    Tuple
  10         1,0.5,8
1000         4,0.7,6
1200         3,0.89,12
2000         8,0.4,4
```

Example 12–24 atan2 Function Stream Output

```
Timestamp    Tuple Kind  Tuple
  10:        +           0.7853982
1000:        +           0.7853982
1200:        +           0.7853982
2000:        +           0.7853982
```

cbrt

Syntax



Purpose

`cbrt` returns the cube root of the `double` argument as a `double`.

For positive finite `a`, `cbrt(-a) == -cbrt(a)`; that is, the cube root of a negative value is the negative of the cube root of that value's magnitude.

For more information, see
`http://java.sun.com/javase/6/docs/api/java/lang/Math.html#cbrt(double)`.

Examples

Consider the query `q76` in Example 12–25. Given the data stream `SFunc` with schema `(c1 integer, c2 float, c3 bigint)` in Example 12–26, the query returns the stream in Example 12–27.

Example 12–25 cbrt Function Query

```
<query id="q76"><![CDATA[
    select cbrt(c2) from SFunc
]]></query>
```

Example 12–26 cbrt Function Stream Input

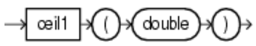
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–27 cbrt Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.7937005
1000:       +           0.887904
1200:       +           0.9619002
2000:       +           0.73680633
```

ceil1

Syntax



Purpose

`ceil1` returns the smallest (closest to negative infinity) `double` value that is greater than or equal to the `double` argument and equals a mathematical integer.

To avoid possible rounding error, consider using `(long) cern.jet.math.Arithmetic.ceil(double)`.

For more information, see:

- `http://java.sun.com/javase/6/docs/api/java/lang/Math.html#ceil(double)`
- "ceil" on page 10-13

Examples

Consider the query `q77` in Example 12–28. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–29, the query returns the stream in Example 12–30.

Example 12–28 ceil1 Function Query

```
<query id="q77"><![CDATA[
    select ceil1(c2) from SFunc
]]></query>
```

Example 12–29 ceil1 Function Stream Input

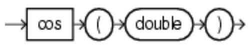
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–30 ceil1 Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1.0
1000:       +           1.0
1200:       +           1.0
2000:       +           1.0
```

cos

Syntax



Purpose

`cos` returns the trigonometric cosine of a `double` angle as a `double`.

For more information, see
`http://java.sun.com/javase/6/docs/api/java/lang/Math.html#cos(double)`.

Examples

Consider the query `q61` in Example 12–31. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–32, the query returns the stream in Example 12–33.

Example 12–31 cos Function Query

```
<query id="q61"><![CDATA[
    select cos(c2) from SFunc
]]></query>
```

Example 12–32 cos Function Stream Input

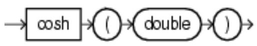
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–33 cos Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.87758255
1000:       +           0.7648422
1200:       +           0.62941206
2000:       +           0.921061
```

cosh

Syntax



Purpose

`cosh` returns the hyperbolic cosine of a `double` value as a `double`.

For more information, see
http://java.sun.com/javase/6/docs/api/java/lang/Math.html#cosh(double).

Examples

Consider the query `q78` in Example 12–34. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–35, the query returns the stream in Example 12–36.

Example 12–34 cosh Function Query

```
<query id="q78"><![CDATA[
    select cosh(c2) from SFunc
]]></query>
```

Example 12–35 cosh Function Stream Input

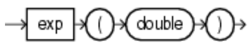
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–36 cosh Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1.127626
1000:       +           1.255169
1200:       +           1.4228927
2000:       +           1.0810723
```

exp

Syntax



Purpose

exp returns Euler's number *e* raised to the power of the double argument as a double.

Note that for values of x near 0, the exact sum of expm1(x) + 1 is much closer to the true result of Euler's number *e* raised to the power of x than EXP(x).

For more information, see:

- http://java.sun.com/javase/6/docs/api/java/lang/Math.html#exp(double)
- "expm1" on page 12-16

Examples

Consider the query q79 in Example 12–37. Given the data stream SFunc with schema (c1 integer, c2 double, c3 bigint) in Example 12–38, the query returns the stream in Example 12–39.

Example 12–37 exp Function Query

```
<query id="q79"><![CDATA[
    select exp(c2) from SFunc
]]></query>
```

Example 12–38 exp Function Stream Input

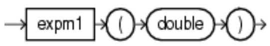
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–39 exp Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1.6487212
1000:       +           2.0137527
1200:       +           2.4351296
2000:       +           1.4918247
```

expm1

Syntax



Purpose

expm1 returns the computation that Figure 12–1 shows as a double, where x is the double argument and *e* is Euler's number.

Figure 12–1 java.lang.Math Expm1

$$e^x - 1$$

Note that for values of x near 0, the exact sum of expm1(x) + 1 is much closer to the true result of Euler's number *e* raised to the power of x than exp(x).

For more information, see:

- http://java.sun.com/javase/6/docs/api/java/lang/Math.html#expm1(double)
- "exp" on page 12-15

Examples

Consider the query q80 in Example 12–40. Given the data stream SFunc with schema (c1 integer, c2 double, c3 bigint) in Example 12–41, the query returns the stream in Example 12–42.

Example 12–40 expm1 Function Query

```
<query id="q80"><![CDATA[
    select expm1(c2) from SFunc
]]></query>
```

Example 12–41 expm1 Function Stream Input

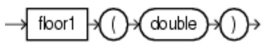
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–42 expm1 Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.6487213
1000:       +           1.0137527
1200:       +           1.4351296
2000:       +           0.49182472
```

floor1

Syntax



Purpose

`floor1` returns the largest (closest to positive infinity) `double` value that is less than or equal to the `double` argument and equals a mathematical integer.

To avoid possible rounding error, consider using `(long) cern.jet.math.Arithmetic.floor(double)`.

For more information, see:

- `http://java.sun.com/javase/6/docs/api/java/lang/Math.html#floor(double)`
- "floor" on page 10-19

Examples

Consider the query `q81` in Example 12–43. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–44, the query returns the stream in Example 12–45.

Example 12–43 floor1 Function Query

```
<query id="q81"><![CDATA[
    select floor1(c2) from SFunc
]]></query>
```

Example 12–44 floor1 Function Stream Input

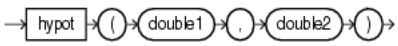
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–45 floor1 Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.0
1000:       +           0.0
1200:       +           0.0
2000:       +           0.0
```

hypot

Syntax



Purpose

`hypot` returns the hypotenuse (see Figure 12–2) of the `double` arguments as a `double`.

Figure 12–2 java.lang.Math hypot

$$\sqrt{(x^2 + y^2)}$$

This function takes the following arguments:

- `double1`: the `x` value.
- `double2`: the `y` value.

The hypotenuse is computed without intermediate overflow or underflow.

For more information, see `http://java.sun.com/javase/6/docs/api/java/lang/Math.html#hypot(double,%20double)`.

Examples

Consider the query `q82` in Example 12–46. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–47, the query returns the stream in Example 12–48.

Example 12–46 hypot Function Query

```
<query id="q82"><![CDATA[
    select hypot(c2,c2) from SFunc
]]></query>
```

Example 12–47 hypot Function Stream Input

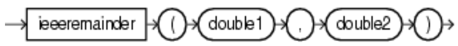
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–48 hypot Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.70710677
1000:       +           0.98994946
1200:       +           1.2586501
2000:       +           0.56568545
```

IEEEremainder

Syntax



Purpose

`IEEEremainder` computes the remainder operation on two `double` arguments as prescribed by the IEEE 754 standard and returns the result as a `double`.

This function takes the following arguments:

- `double1`: the dividend.
- `double2`: the divisor.

The remainder value is mathematically equal to `f1 - f2 × n`, where `n` is the mathematical integer closest to the exact mathematical value of the quotient `f1/f2`, and if two mathematical integers are equally close to `f1/f2`, then `n` is the integer that is even. If the remainder is zero, its sign is the same as the sign of the first argument.

For more information, see `http://java.sun.com/javase/6/docs/api/java/lang/Math.html#IEEEremainder(double,%20double)`.

Examples

Consider the query `q72` in Example 12–49. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–50, the query returns the stream in Example 12–51.

Example 12–49 IEEEremainder Function Query

```
<query id="q72"><![CDATA[
    select IEEEremainder(c2,c2) from SFunc
]]></query>
```

Example 12–50 IEEEremainder Function Stream Input

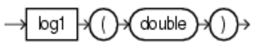
```
Timestamp    Tuple
  10         1,0.5,8
1000         4,0.7,6
1200         3,0.89,12
2000         8,0.4,4
```

Example 12–51 IEEEremainder Function Stream Output

```
Timestamp    Tuple Kind  Tuple
  10:        +           0.0
1000:        +           0.0
1200:        +           0.0
2000:        +           0.0
```

log1

Syntax



Purpose

`log1` returns the natural logarithm (base *e*) of a `double` value as a `double`.

Note that for small values `x`, the result of `log1p(x)` is much closer to the true result of `ln(1 + x)` than the floating-point evaluation of `log(1.0+x)`.

For more information, see:

- `http://java.sun.com/javase/6/docs/api/java/lang/Math.html#log(double)`
- "log1p" on page 12-22

Examples

Consider the query `q83` in Example 12–52. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–53, the query returns the stream in Example 12–54.

Example 12–52 log1 Function Query

```
<query id="q83"><![CDATA[
    select log1(c2) from SFunc
]]></query>
```

Example 12–53 log1 Function Stream Input

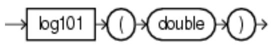
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–54 log1 Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           -0.6931472
1000:       +           -0.35667497
1200:       +           -0.11653383
2000:       +           -0.9162907
```

log101

Syntax



Purpose

`log101` returns the base 10 logarithm of a `double` value as a `double`.

For more information, see
`http://java.sun.com/javase/6/docs/api/java/lang/Math.html#log10(double)`.

Examples

Consider the query `q84` in Example 12–55. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–56, the query returns the stream in Example 12–57.

Example 12–55 log101 Function Query

```
<query id="q84"><![CDATA[
    select log101(c2) from SFunc
]]></query>
```

Example 12–56 log101 Function Stream Input

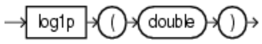
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–57 log101 Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           -0.30103
1000:       +           -0.15490197
1200:       +           -0.050610002
2000:       +           -0.39794
```

log1p

Syntax



Purpose

`log1p` returns the natural logarithm of the sum of the `double` argument and 1 as a `double`.

Note that for small values `x`, the result of `log1p(x)` is much closer to the true result of `ln(1 + x)` than the floating-point evaluation of `log(1.0+x)`.

For more information, see:

- `http://java.sun.com/javase/6/docs/api/java/lang/Math.html#log1p(double)`
- "log1" on page 12-20

Examples

Consider the query `q85` in Example 12–58. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–59, the query returns the stream in Example 12–60.

Example 12–58 log1p Function Query

```
<query id="q85"><![CDATA[
    select log1p(c2) from SFunc
]]></query>
```

Example 12–59 log1p Function Stream Input

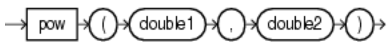
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–60 log1p Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.4054651
1000:       +           0.53062826
1200:       +           0.63657683
2000:       +           0.33647224
```

pow

Syntax



Purpose

pow returns the value of the first double argument (the base) raised to the power of the second double argument (the exponent) as a double.

This function takes the following arguments:

- double1: the base.
- double2: the exponent.

For more information, see http://java.sun.com/javase/6/docs/api/java/lang/Math.html#pow(double,%20double).

Examples

Consider the query q65 in Example 12–61. Given the data stream SFunc with schema (c1 integer, c2 double, c3 bigint) in Example 12–62, the query returns the stream in Example 12–63.

Example 12–61 pow Function Query

```
<query id="q65"><![CDATA[
    select pow(c2,c2) from SFunc
]]></query>
```

Example 12–62 pow Function Stream Input

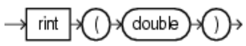
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–63 pow Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.70710677
1000:       +           0.7790559
1200:       +           0.9014821
2000:       +           0.69314486
```

rint

Syntax



Purpose

`rint` returns the `double` value that is closest in value to the `double` argument and equals a mathematical integer. If two `double` values that are mathematical integers are equally close, the result is the integer value that is even.

For more information, see
`http://java.sun.com/javase/6/docs/api/java/lang/Math.html#rint(double)`.

Examples

Consider the query `q86` in Example 12–64. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–65, the query returns the stream in Example 12–66.

Example 12–64 rint Function Query

```
<query id="q86"><![CDATA[
    select rint(c2) from SFunc
]]></query>
```

Example 12–65 rint Function Stream Input

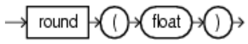
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–66 rint Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.0
1000:       +           1.0
1200:       +           1.0
2000:       +           0.0
```

round

Syntax



Purpose

round returns the closest integer to the float argument.

For more information, see
http://java.sun.com/javase/6/docs/api/java/lang/Math.html#round(float).

Examples

Consider the query q87 in Example 12–67. Given the data stream SFunc with schema (c1 integer, c2 double, c3 bigint) in Example 12–68, the query returns the stream in Example 12–69.

Example 12–67 round Function Query

```
<query id="q87"><![CDATA[
    select round(c2) from SFunc
]]></query>
```

Example 12–68 round Function Stream Input

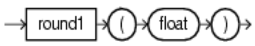
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–69 round Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1
1000:       +           1
1200:       +           1
2000:       +           0
```

round1

Syntax



Purpose

round1 returns the closest integer to the float argument.

For more information, see
http://java.sun.com/javase/6/docs/api/java/lang/Math.html#round(float).

Examples

Consider the query q88 in Example 12–70. Given the data stream SFunc with schema (c1 integer, c2 double, c3 bigint) in Example 12–71, the query returns the stream in Example 12–72.

Example 12–70 round1 Function Query

```
<query id="q88"><![CDATA[
    select round1(c2) from SFunc
]]></query>
```

Example 12–71 round1 Function Stream Input

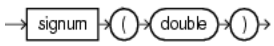
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–72 round1 Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1
1000:       +           1
1200:       +           1
2000:       +           0
```

signum

Syntax



Purpose

`signum` returns the signum function of the `double` argument as a `double`:

- zero if the argument is zero
- 1.0 if the argument is greater than zero
- -1.0 if the argument is less than zero

For more information, see
`http://java.sun.com/javase/6/docs/api/java/lang/Math.html#signum(double)`.

Examples

Consider the query `q70` in Example 12–73. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–74, the query returns the stream in Example 12–75.

Example 12–73 signum Function Query

```
<query id="q70"><![CDATA[
    select signum(c2) from SFunc
]]></query>
```

Example 12–74 signum Function Stream Input

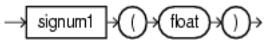
```
Timestamp    Tuple
  10         1,0.5,8
1000         4,-0.7,6
1200         3,-0.89,12
2000         8,0.4,4
```

Example 12–75 signum Function Stream Output

```
Timestamp    Tuple Kind  Tuple
  10:        +           1.0
1000:        +           -1.0
1200:        +           -1.0
2000:        +           1.0
```

signum1

Syntax



Purpose

`signum1` returns the signum function of the `float` argument as a `float`:

- zero if the argument is zero
- 1.0 if the argument is greater than zero
- -1.0 if the argument is less than zero

For more information, see
`http://java.sun.com/javase/6/docs/api/java/lang/Math.html#signum(float)`.

Examples

Consider the query `q71` in Example 12–76. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–77, the query returns the relation in Example 12–78.

Example 12–76 signum1 Function Query

```
<query id="q71"><![CDATA[
    select signum1(c2) from SFunc
]]></query>
```

Example 12–77 signum1 Function Stream Input

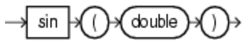
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,-0.7,6
1200        3,-0.89,12
2000        8,0.4,4
```

Example 12–78 signum1 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1.0
1000:       +           -1.0
1200:       +           -1.0
2000:       +           1.0
```

sin

Syntax



Purpose

`sin` returns the trigonometric sine of a `double` angle as a `double`.

For more information, see
`http://java.sun.com/javase/6/docs/api/java/lang/Math.html#sin(double)`.

Examples

Consider the query `q60` in Example 12–79. Given the data stream `SFunc` with schema (`c1 integer, c2 float, c3 bigint`) in Example 12–80, the query returns the stream in Example 12–81.

Example 12–79 sin Function Query

```
<query id="q60"><![CDATA[
    select sin(c2) from SFunc
]]></query>
```

Example 12–80 sin Function Stream Input

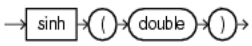
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–81 sin Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.47942555
1000:       +           0.64421767
1200:       +           0.7770717
2000:       +           0.38941833
```

sinh

Syntax



Purpose

`sinh` returns the hyperbolic sine of a `double` value as a `double`.

For more information, see
`http://java.sun.com/javase/6/docs/api/java/lang/Math.html#sinh(double)`.

Examples

Consider the query `q89` in Example 12–82. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–83, the query returns the stream in Example 12–84.

Example 12–82 sinh Function Query

```
<query id="q89"><![CDATA[
    select sinh(c2) from SFunc
]]></query>
```

Example 12–83 sinh Function Stream Input

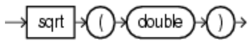
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–84 sinh Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.5210953
1000:       +           0.75858366
1200:       +           1.012237
2000:       +           0.41075233
```

sqrt

Syntax



Purpose

`sqrt` returns the correctly rounded positive square root of a `double` value as a `double`.

For more information, see
`http://java.sun.com/javase/6/docs/api/java/lang/Math.html#sqrt(double)`.

Examples

Consider the query `q64` in Example 12–85. Given the data stream `SFunc` with schema (`c1 integer, c2 float, c3 bigint`) in Example 12–86, the query returns the stream in Example 12–87.

Example 12–85 sqrt Function Query

```
<query id="q64"><![CDATA[
    select sqrt(c2) from SFunc
]]></query>
```

Example 12–86 sqrt Function Stream Input

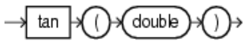
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–87 sqrt Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.70710677
1000:       +           0.83666
1200:       +           0.9433981
2000:       +           0.6324555
```

tan

Syntax



Purpose

`tan` returns the trigonometric tangent of a `double` angle as a `double`.

For more information, see
`http://java.sun.com/javase/6/docs/api/java/lang/Math.html#tan(double)`.

Examples

Consider the query `q62` in Example 12–88. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–89, the query returns the stream in Example 12–90.

Example 12–88 tan Function Query

```
<query id="q62"><![CDATA[
    select tan(c2) from SFunc
]]></query>
```

Example 12–89 tan Function Stream Input

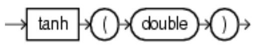
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–90 tan Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.5463025
1000:       +           0.8422884
1200:       +           1.2345995
2000:       +           0.42279324
```

tanh

Syntax



Purpose

`tanh` returns the hyperbolic tangent of a `double` value as a `double`.

For more information, see
`http://java.sun.com/javase/6/docs/api/java/lang/Math.html#tanh(double)`.

Examples

Consider the query `q90` in Example 12–91. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–92, the query returns the stream in Example 12–93.

Example 12–91 tanh Function Query

```
<query id="q90"><![CDATA[
    select tanh(c2) from SFunc
]]></query>
```

Example 12–92 tanh Function Stream Input

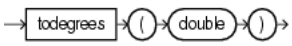
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–93 tanh Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.46211717
1000:       +           0.6043678
1200:       +           0.7113937
2000:       +           0.37994897
```

todegrees

Syntax



Purpose

`todegrees` converts a `double` angle measured in radians to an approximately equivalent angle measured in degrees as a `double`.

The conversion from radians to degrees is generally inexact; do not expect `COS(TORADIANS(90.0))` to exactly equal `0.0`.

For more information, see:

- `http://java.sun.com/javase/6/docs/api/java/lang/Math.html#toDegrees(double)`
- "toradians" on page 12-35
- "cos" on page 12-13

Examples

Consider the query `q91` in Example 12–94. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–95, the query returns the stream in Example 12–96.

Example 12–94 todegrees Function Query

```
<query id="q91"><![CDATA[
    select todegrees(c2) from SFunc
]]></query>
```

Example 12–95 todegrees Function Stream Input

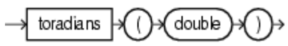
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–96 todegrees Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           28.64789
1000:       +           40.107044
1200:       +           50.993244
2000:       +           22.918312
```

toradians

Syntax



Purpose

`toradians` converts a `double` angle measured in degrees to an approximately equivalent angle measured in radians as a `double`.

For more information, see:

- `http://java.sun.com/javase/6/docs/api/java/lang/Math.html#toRadians(double)`
- "todegrees" on page 12-34
- "cos" on page 12-13

Examples

Consider the query `q92` in Example 12–97. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–98, the query returns the stream in Example 12–99.

Example 12–97 toradians Function Query

```
<query id="q92"><![CDATA[
    select toradians(c2) from SFunc
]]></query>
```

Example 12–98 toradians Function Stream Input

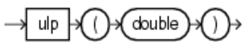
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–99 toradians Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.008726646
1000:       +           0.012217305
1200:       +           0.0155334305
2000:       +           0.006981317
```

ulp

Syntax



Purpose

ulp returns the size of an ulp of the double argument as a double. In this case, an ulp of the argument value is the positive distance between this floating-point value and the double value next larger in magnitude.

For more information, see http://java.sun.com/javase/6/docs/api/java/lang/Math.html#ulp(double).

Examples

Consider the query q93 in Example 12–100. Given the data stream SFunc with schema (c1 integer, c2 double, c3 bigint) in Example 12–101, the query returns the stream in Example 12–102.

Example 12–100 ulp Function Query

```
<query id="q93"><![CDATA[
    select ulp(c2) from SFunc
]]></query>
```

Example 12–101 ulp Function Stream Input

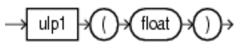
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–102 ulp Function Stream Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           1.110223E-16
1000:       +           1.110223E-16
1200:       +           1.110223E-16
2000:       +           5.551115E-17
```

ulp1

Syntax



Purpose

`ulp1` returns the size of an ulp of the `float` argument as a `float`. An ulp of a float value is the positive distance between this floating-point value and the float value next larger in magnitude.

For more information, see `http://java.sun.com/javase/6/docs/api/java/lang/Math.html#ulp(float)`.

Examples

Consider the query `q94` in Example 12–103. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 12–104, the query returns the relation in Example 12–105.

Example 12–103 ulp1 Function Query

```
<query id="q94"><![CDATA[
    select ulp1(c2) from SFunc
]]></query>
```

Example 12–104 ulp1 Function Stream Input

```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 12–105 ulp1 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           5.9604645E-8
1000:       +           5.9604645E-8
1200:       +           5.9604645E-8
2000:       +           2.9802322E-8
```

13

User-Defined Functions

This chapter describes how you can write user-defined functions for use in Oracle Continuous Query Language (Oracle CQL) to perform more advanced or application-specific operations on stream data than is possible using built-in functions.

For more information, see Section , "Functions".

This chapter includes the following sections:

- Introduction to Oracle CQL User-Defined Functions
- Implementing a User-Defined Function

Introduction to Oracle CQL User-Defined Functions

You can write user-defined functions in Java to provide functionality that is not available in Oracle CQL or Oracle CQL built-in functions. You can create a user-defined function that returns an aggregate value or a single (non-aggregate) value.

For example, you can use user-defined functions in the following:

- The select list of a `SELECT` statement
- The condition of a `WHERE` clause

To make your user-defined function available for use in Oracle CQL queries, the JAR file that contains the user-defined function implementation class must be in the Oracle Event Processing server classpath or the Oracle Event Processing server classpath must be modified to include the JAR file.

For more information, see:

- Section , "Types of User-Defined Functions"
- Section , "User-Defined Function Datatypes"
- Section , "User-Defined Functions and the Oracle Event Processing Server Cache"
- Section , "Implementing a User-Defined Function"
- Section , "Functions"

Types of User-Defined Functions

Using the classes in the `oracle.cep.extensibility.functions` package you can create the following types of user-defined functions:

- Section , "User-Defined Single-Row Functions"

- Section , "User-Defined Aggregate Functions"

You can create overloaded functions and you can override built-in functions.

User-Defined Single-Row Functions

A user-defined single-row function is a function that returns a single result row for every row of a queried stream or view (for example, like the `concat` built-in function does).

For more information, see "How to Implement a User-Defined Single-Row Function" on page 13-3.

User-Defined Aggregate Functions

A user-defined aggregate is a function that implements `com.bea.wlevs.processor.AggregationFunctionFactory` and returns a single aggregate result based on group of tuples, rather than on a single tuple (for example, like the `sum` built-in function does).

Consider implementing your aggregate function so that it performs incremental processing, if possible. This will improve scalability and performance because the cost of (re)computation on arrival of new events will be proportional to the number of new events as opposed to the total number of events seen thus far.

For more information, see "How to Implement a User-Defined Aggregate Function" on page 13-4.

User-Defined Function Datatypes

User-defined functions support any of the built-in Oracle CQL datatypes listed in Section , "Oracle CQL Built-in Datatypes". See the table in that section for a list of Oracle CQL datatypes and their Java equivalents.

The Oracle CQL datatypes shown there lists the datatypes you can specify in the Oracle CQL statement you use to register your user-defined function. The Java equivalents are the Java datatypes you can use in your user-defined function implementation.

At run time, Oracle Event Processing maps between the Oracle CQL datatype and the Java datatype. If your user-defined function returns a datatype that is not in this list, Oracle Event Processing will throw a `ClassCastException`.

For more information about data conversion, see Section , "Datatype Conversion".

User-Defined Functions and the Oracle Event Processing Server Cache

You can access an Oracle Event Processing cache from an Oracle CQL statement or user-defined function.

For more information, see:

- "Configuring Oracle Event Processing Caching" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*
- "Accessing a Cache From an Oracle CQL Statement" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*
- "Accessing a Cache From an Oracle CQL User-Defined Function" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*

Implementing a User-Defined Function

This section describes:

- Section , "How to Implement a User-Defined Single-Row Function"
- Section , "How to Implement a User-Defined Aggregate Function"

For more information, see Section , "Introduction to Oracle CQL User-Defined Functions".

How to Implement a User-Defined Single-Row Function

You implement a user-defined single-row function by implementing a Java class that provides a public constructor and a public method that is invoked to execute the function.

To implement a user-defined single-row function:

1. Implement a Java class as Example 13–1 shows.

   Ensure that the data type of the return value corresponds to a supported data type as Section , "User-Defined Function Datatypes" describes.

   For more information on accessing the Oracle Event Processing cache from a user-defined function, see Section , "User-Defined Functions and the Oracle Event Processing Server Cache".

Example 13–1 MyMod.java User-Defined Single-Row Function

```
package com.bea.wlevs.example.function;

public class MyMod {
    public Object execute(int arg0, int arg1) {
        return new Integer(arg0 % arg1);
    }
}
```

2. Compile the user-defined function Java implementation class and register the class in your Oracle Event Processing application assembly file as Example 13–2 shows.

Example 13–2 Single-Row User Defined Function for an Oracle CQL Processor

```
<wlevs:processor id="testProcessor">
    <wlevs:listener ref="providerCache"/>
    <wlevs:listener ref="outputCache"/>
    <wlevs:cache-source ref="testCache"/>
    <wlevs:function function-name="mymod" exec-method="execute" />
        <bean class="com.bea.wlevs.example.function.MyMod"/>
    </wlevs:function>
</wlevs:processor>
```

   Specify the method that is invoked to execute the function using the `wlevs:function` element `exec-method` attribute. This method must be public and must be uniquely identifiable by its name (that is, the method cannot have been overridden).

   For more information, see "wlevs:function" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

3. Invoke your user-defined function in the select list of a `SELECT` statement or the condition of a `WHERE` clause as Example 13–3 shows.

Example 13–3 Accessing a User-Defined Single-Row Function in Oracle CQL

```
...
<view id="v1" schema="c1 c2 c3 c4"><![CDATA[
    select
        mymod(c1, 100), c2, c3, c4
    from
        S1
]]></view>
...
<query id="q1"><![CDATA[
    select * from v1 [partition by c1 rows 1] where c4 - c3 = 2.3
]]></query>
...
```

How to Implement a User-Defined Aggregate Function

You implement a user-defined aggregate function by implementing a Java class that implements the `com.bea.wlevs.processor.AggregationFunctionFactory` interface.

To implement a user-defined aggregate function:

1. Implement a Java class as Example 13–4 shows.

   Consider implementing your aggregate function so that it performs incremental processing, if possible. This will improve scalability and performance because the cost of (re)computation on arrival of new events will be proportional to the number of new events as opposed to the total number of events seen thus far. The user-defined aggregate function in Example 13–4 supports incremental processing.

   Ensure that the data type of the return value corresponds to a supported data type as Section , "User-Defined Function Datatypes" describes.

   For more information on accessing the Oracle Event Processing cache from a user-defined function, see Section , "User-Defined Functions and the Oracle Event Processing Server Cache".

Example 13–4 Variance.java User-Defined Aggregate Function

```
package com.bea.wlevs.test.functions;

import com.bea.wlevs.processor.AggregationFunction;
import com.bea.wlevs.processor.AggregationFunctionFactory;

public class Variance implements AggregationFunctionFactory, AggregationFunction {

    private int count;
    private float sum;
    private float sumSquare;

    public Class<?>[] getArgumentTypes() {
        return new Class<?>[] {Integer.class};
    }

    public Class<?> getReturnType() {
        return Float.class;
    }

    public AggregationFunction newAggregationFunction() {
        return new Variance();
    }

    public void releaseAggregationFunction(AggregationFunction function) {
    }
```

```
    public Object handleMinus(Object[] params) {
        if (params != null && params.length == 1) {
            Integer param = (Integer) params[0];
            count--;
            sum -= param;
            sumSquare -= (param * param);
        }

        if (count == 0) {
            return null;
        } else {
            return getVariance();
        }
    }

    public Object handlePlus(Object[] params) {
        if (params != null && params.length == 1) {
            Integer param = (Integer) params[0];
            count++;
            sum += param;
            sumSquare += (param * param);
        }

        if (count == 0) {
            return null;
        } else {
            return getVariance();
        }
    }

    public Float getVariance() {
        float avg = sum / (float) count;
        float avgSqr = avg * avg;
        float var = sumSquare / (float)count - avgSqr;
        return var;
    }

    public void initialize() {
        count = 0;
        sum = 0.0F;
        sumSquare = 0.0F;
    }

}
```

2. Compile the user-defined function Java implementation class and register the class in your Oracle Event Processing application assembly file as Example 13–5 shows.

Example 13–5 Aggregate User Defined Function for an Oracle CQL Processor

```
<wlevs:processor id="testProcessor">
   <wlevs:listener ref="providerCache"/>
   <wlevs:listener ref="outputCache"/>
   <wlevs:cache-source ref="testCache"/>
   <wlevs:function function-name="var">
     <bean class="com.bea.wlevs.test.functions.Variance"/>
   </wlevs:function>
 </wlevs:processor>
```

For more information, see "wlevs:function" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

3. Invoke your user-defined function in the select list of a `SELECT` statement or the condition of a `WHERE` clause as Example 13–6 shows.

Example 13–6 Accessing a User-Defined Aggregate Function in Oracle CQL

```
...
<query id="uda6"><![CDATA[
    select var(c2) from S4[range 3]
]]></query>
...
```

At run-time, when the user-defined aggregate is executed, and a new event becomes active in the window of interest, the aggregations will have to be recomputed (since the set over which the aggregations are defined has a new member). To do so, Oracle Event Processing passes only the new event (rather than the entire active set) to the appropriate handler context by invoking the appropriate `handlePlus*` method in Example 13–4. This state can now be updated to include the new event. Thus, the aggregations have been recomputed in an incremental fashion.

Similarly, when an event expires from the window of interest, the aggregations will have to be recomputed (since the set over which the aggregations are defined has lost a member). To do so, Oracle Event Processing passes only the expired event (rather than the entire active set) to the appropriate handler context by invoking the appropriate `handleMinus` method in Example 13–4. As before, the state in the handler context can be incrementally updated to accommodate expiry of the event in an incremental fashion.

Part III

Data Cartridges

This part contains the following chapters:

14

Introduction to Data Cartridges

This chapter introduces data cartridges in Oracle Event Processing. Data cartridges extend Oracle Continuous Query Language (Oracle CQL) to support domain-specific abstract data types of the following forms: simple types, complex types, array types, and domain-specific functions.

This chapter includes the following sections:

Understanding Data Cartridges

The Oracle CQL data cartridge framework allows you to tightly integrate arbitrary domain data types and functions with the Oracle CQL language, allowing the usage of these extensions within Oracle CQL queries in the same way you use Oracle CQL native types and built-in functions.

With regards to data types, the framework supports both simple and complex types, the latter allowing the usage of object-oriented programming.

Using Oracle CQL data cartridges, you can extend the Oracle CQL engine with domain-specific types that augment and interoperate with native Oracle CQL built-in types.

Data Cartridge Name

Each data cartridge is identified by a unique data cartridge name that defines a name space for the data cartridge implementation. You use the data cartridge name to disambiguate references to types, methods, fields, and constructors, if necessary (see *link::=* on page 5-19).

Data Cartridge Application Context

Depending on the data cartridge implementation, you may be able to define an application context that the Oracle Event Processing server propagates to the functions and types that an instance of the data cartridge provides.

For example, you might be able to configure an Oracle Event Processing server resource or a default data cartridge option and associate this application context information with a particular data cartridge instance.

For more information, see "Understanding Data Cartridge Application Context" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

Oracle CQL Data Cartridge Types

How you access data cartridge types, methods, fields, and constructors using Oracle CQL is the same for all data cartridge implementations.

You may reference a data-cartridge function by using the `func_expr`, which may optionally take a link name.

To reference the members of a complex type, Oracle CQL provides the `object_expr` production.

For more information, see:

- *func_expr::=* on page 5-15
- *object_expr::=* on page 5-19

What you access is, of course, unique to each data cartridge implementation. For more information, see:

- Chapter 15, "Oracle Java Data Cartridge"
- Chapter 16, "Oracle Spatial"
- Chapter 17, "Oracle Event Processing JDBC Data Cartridge"
- Chapter 19, "Oracle Event Processing NoSQL Database Data Cartridge"
- Chapter 18, "Oracle Event Processing Hadoop Data Cartridge"

Note: To simplify Oracle data cartridge type names, you can use aliases as Section , "Defining Aliases Using the Aliases Element" describes.

15

Oracle Java Data Cartridge

This chapter describes how to use the Oracle Java Data Cartridge, an extension of Oracle Continuous Query Language (Oracle CQL) with which you can write CQL code that seamlessly interacts with Java classes in your Oracle Event Processing application.

This chapter describes the types, methods, fields, and constructors that the Oracle Java data cartridge exposes. You can use these types, methods, fields, and constructors in Oracle CQL queries and views as you would Oracle CQL native types.

This chapter includes the following sections:

- Understanding the Oracle Java Data Cartridge
- Using the Oracle Java Data Cartridge

For more information, see:

- Section , "Understanding Data Cartridges"
- Section , "Oracle CQL Data Cartridge Types"

Understanding the Oracle Java Data Cartridge

The Oracle Java data cartridge is a built-in Java cartridge which allows you to write Oracle CQL queries and views that seamlessly interact with the Java classes in your Oracle Event Processing application.

This section describes:

- Section , "Data Cartridge Name"
- Section , "Class Loading"
- Section , "Method Resolution"
- Section , "Datatype Mapping"
- Section , "Oracle CQL Query Support for the Oracle Java Data Cartridge"

Data Cartridge Name

The Oracle Java data cartridge uses the cartridge ID `com.oracle.cep.cartrdiges.java`.

The Oracle Java data cartridge is the default Oracle Event Processing data cartridge.

For types under the default Java package name or types under the system package of `java.lang`, you may reference the Java type in an Oracle CQL query unqualified by package or data cartridge name:

```
<query id="q1"><![CDATA[
    select String("foo") …
]]></query>
```

Note: To simplify Oracle Java data cartridge type names, you can use aliases as Section , "Defining Aliases Using the Aliases Element" describes.

For more information, see:

- Section , "Class Loading"
- "View" on page 22-29
- Chapter 13, "User-Defined Functions"

Class Loading

The Oracle Java data cartridge supports the following policies for loading the Java classes that your Oracle CQL queries reference:

- Section , "Application Class Space Policy"
- Section , "No Automatic Import Class Space Policy"
- Section , "Server Class Space Policy"

For more information, see:

- Section , "Class Loading Example"
- Section , "Method Resolution"
- "How to Export a Package" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

Application Class Space Policy

This is the default class loading policy.

In this mode, the Oracle Java data cartridge uses the class-space of the application in scope when searching for a Java class.

This is only applicable when a type is specified only by its local name, that is, there is a single identifier, and no other identifiers are being used for its package. That is:

```
select String("foo") …
```

And not:

```
select java.lang.String("foo") …
```

In this case the procedure is as follows:

- Attempt to load the class defined by the single identifier (call it `ID1`) using the application's class-space as usual; if this fails then:
- Verify if the application defines any class within its bundle's internal class-path whose name matches `ID1`, independent of the package; if this fails then:
- Verify if application specifies an `Import-Package` `MANIFEST` header statement which in conjunction with `ID1` can be used to load a Java class.

For an example, see Section , "Class Loading Example".

No Automatic Import Class Space Policy

This is an optional class loading policy. To use this policy, you must include the following MANIFEST header entry in your Oracle Event Processing application:

```
OCEP_JAVA_CARTRIDGE_CLASS_SPACE: APPLICATION_NO_AUTO_IMPORT_CLASS_SPACE
```

This mode is similar to the application class space policy except that Oracle Event Processing will not attempt to automatically import a package when a package is not specified.

For more information, see Section , "Application Class Space Policy".

Server Class Space Policy

This is an optional class loading policy. To use this policy, you must include the following MANIFEST header entry in your Oracle Event Processing application:

```
OCEP_JAVA_CARTRIDGE_CLASS_SPACE: SERVER_CLASS_SPACE
```

An Oracle CQL query can reference any exported Java class, regardless of the application or module that is exporting it.

The query can also access all classes visible to the OSGi framework's parent class-loader, which includes the runtime JDK classes.

This means that an Oracle CQL application may contain an Oracle CQL query that references classes defined by other Oracle Event Processing applications, as long as they are exported. This behavior facilitates the creation of Java-based cartridges whose sole purpose is to provide new Java libraries.

Note: You may only reference a Java class that is part of the internal class-path of an Oracle Event Processing application if it is exported, even if a processor within this application defines the Oracle CQL query.

For an example, see Section , "Class Loading Example".

Class Loading Example

Consider the example that Figure 15–1 shows: application B1 imports package mypackage3 that application B2 exports.

Figure 15–1 Example Oracle Event Processing Applications

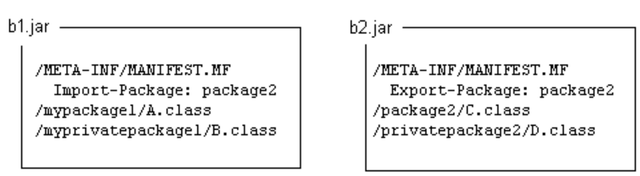


Table 15–1 summarizes which classes these two different applications can access depending on whether they are running in the application class space or server class space.

Table 15–1 Class Accessibility by Class Loading Policy

Class Loading Policy	Application B1	Application B2
Application Class Space	■ mypackage1.A ■ myprivatepackage1.B ■ package2.C	■ package2.C ■ privatepackage2.D
Server Class Space	■ package2.C	■ package2.C

In application B1, you can use any of the Java classes A, B, and C in your Oracle CQL queries:

```
select A …
select B …
select C …
```

However, in application B2, you cannot use Java classes A and B in your Oracle CQL queries. You can only use Java classes C and D:

```
select C …
select D …
```

Method Resolution

An Oracle CQL expression that accesses a Java method uses the following algorithm to resolve the method:

1. All parameter types are converted to Java types as Section , "Datatype Mapping" describes.

   For example, an Oracle CQL `INTEGER` is converted to a Java primitive `int`.

2. Standard Java method resolution rules are applied as the Java Language Specification, Third Edition, Section 15.12, "Method Invocation Expressions" describes.

Note: Variable arity methods are not supported. For more information, see the Java Language Specification, Third Edition, Section 12.12.2.4.

As an example, consider the following Oracle CQL expression:

```
attribute.methodA(10)
```

Where `attribute` is of type `mypackage.MyType` which defines the following overloaded methods:

- `methodA(int)`
- `methodA(Integer)`
- `methodA(Object)`
- `methodA(long)`

As the literal 10 is of the primitive type `int`, the order of precedence is:

- `methodA(int)`
- `methodA(long)`
- `methodA(Integer)`

- `methodA(Object)`

For more information, see Section , "Class Loading".

Datatype Mapping

The Oracle Java data cartridge applies a fixed, asymmetrical mapping between Oracle CQL native datatypes and Java datatypes.

- Table 15–2 lists the mappings between Oracle CQL native datatypes and Java datatypes.
- Table 15–3 lists the mappings between Java datatypes and Oracle CQL native datatypes.

Table 15–2 Oracle Java Data Cartridge: Oracle CQL to Java Datatype Mapping

Oracle CQL Native Datatype	Java Datatype
BIGINT	long[1]
BOOLEAN	boolean[1]
BYTE	byte[][1]
CHAR	java.lang.String
DOUBLE	double[1]
FLOAT	float[1]
INTEGER	int[1]
INTERVAL	long[1]
XMLTYPE	java.lang.String

[1] primitive Java datatype

Table 15–3 Oracle Java Data Cartridge: Java Datatype to Oracle CQL Mapping

Java Datatype	Oracle CQL Native Datatype
long[1]	BIGINT
boolean[1]	BOOLEAN
byte[][1]	BYTE
java.lang.String	CHAR
double[1]	DOUBLE
float[1]	FLOAT
int[1]	INTEGER
java.sql.Date java.sql.Timestamp	INTERVAL
java.sql.SQLXML	XMLTYPE

[1] primitive Java datatype

All other Java classes are mapped as a complex type.

For more information on these datatype mappings:

- Section , "Java Datatype String and Oracle CQL Datatype CHAR"
- Section , "Literals"
- Section , "Arrays"
- Section , "Collections"

For more information on Oracle CQL native datatypes and their implicit and explicit datatype conversion, see Section , "Datatypes".

Java Datatype String and Oracle CQL Datatype CHAR

Oracle CQL datatype `CHAR` is mapped to `java.lang.String` and `java.lang.String` is mapped to Oracle CQL datatype `CHAR`. This means you can access `java.lang.String` member fields and methods for an attribute defined as Oracle CQL `CHAR`. For example, if `a1` is declared as type Oracle CQL `CHAR`, then you can write a query like this:

```
<query id="q1"><![CDATA[
    select a1.substring(1,2)
]]></query>
```

Literals

You cannot access member fields and methods on literals, even Oracle CQL `CHAR` literals. For example, the following query is *not* allowed:

```
<query id="q1-forbidden"><![CDATA[
    select "hello".substring(1,2)
]]></query>
```

Arrays

Java arrays are converted to Oracle CQL data cartridge arrays, and Oracle CQL data cartridge arrays are converted to Java arrays. This applies to both complex types and simple types.

You can use the data cartridge `TABLE` clause to access the multiple rows returned by a data cartridge function in the `FROM` clause of an Oracle CQL query.

For more information, see:

- "array_type" on page 7-3
- Section , "Function TABLE Query"
- Section , "Collections"

Collections

Typically, the Oracle Java data cartridge converts an instance that implements the `java.util.Collection` interface to an Oracle CQL complex type.

An Oracle CQL query can iterate through the members of the `java.util.Collection`.

You can use the data cartridge `TABLE` clause to access the multiple rows returned by a data cartridge function in the `FROM` clause of an Oracle CQL query.

For more information, see:

- "complex_type" on page 7-8
- Section , "Function TABLE Query"
- Section , "Arrays"

Oracle CQL Query Support for the Oracle Java Data Cartridge

You may use Oracle Java data cartridge types in expressions within a `SELECT` clause and `WHERE` clause.

You may not use Oracle Java data cartridge types in expressions within an `ORDER BY` clause.

For more information, see:

- Section , "Using the Oracle Java Data Cartridge"
- Section , "Introduction to Oracle CQL Statements"

Using the Oracle Java Data Cartridge

This section describes common use-cases that highlight how you can use the Oracle Java data cartridge in your Oracle Event Processing applications, including:

- Section , "How to Query Using the Java API"
- Section , "How to Query Using Exported Java Classes"

For more information, see:

- Section , "Oracle CQL Query Support for the Oracle Java Data Cartridge"
- Section , "Introduction to Oracle CQL Statements"

How to Query Using the Java API

This procedure describes how to use the Oracle Java data cartridge in an Oracle Event Processing application that uses one event type defined as a tuple (`Student`) that has an event property type defined as a Java class (`Address.java`).

To query with Java classes:

1. Implement the `Address.java` class as Example 15–1 shows.

Example 15–1 Address.java Class

```
package test;

class Address {
    String street;
    String state;
    String city;
    String [] phones;
}
```

In this example, assume that the `Address.java` class belongs to this application.

If the `Address.java` class belonged to another Oracle Event Processing application, it must be exported in its parent application. For more information, see Section , "How to Query Using Exported Java Classes".

2. Define the event type repository as Example 15–2 shows.

Example 15–2 Event Type Repository

```
<event-type-repository>
    <event-type name="Student">
        <properties>
```

```
            <property name="name" type="char"/>
            <property name="address" type="Address"/>
        </properties>
    </event-type>

    <event-type name="Address">
        <class-name>test.Address</class-name>
    </event-type>
<event-type-repository>
```

Because the `test.Address` class belongs to this application, it can be declared in the event type repository. This automatically makes the class globally accessible within this application; its package does not need to be exported.

3. Assume that an adapter is providing `Student` events to channel `StudentStream` as Example 15–3 shows

Example 15–3 Channel

```
<channel id="StudentStream" event-type="Student"/>
```

4. Assume that the `StudentStream` is connected to a processor with the Oracle CQL query `q1` that Example 15–4 shows.

Example 15–4 Oracle CQL Query

```
<processor>
    <rules>
        <query id="q1"><![CDATA[
            select
                name,
                address.street as street,
                address.phones[0] as primary_phone
            from
                StudentStream
        ]]></query>
    </rules>
</processor>
```

The Oracle Java data cartridge allows you to access the `address` event property from within the Oracle CQL query using normal Java API.

How to Query Using Exported Java Classes

This procedure describes how to use the Oracle Java data cartridge in an Oracle Event Processing application that uses one event type defined as a tuple (`Student`) that has an event property type defined as a Java class (`Address.java`). In this procedure, the Address.java class belongs to a separate Oracle Event Processing application. It is exported in its parent application to make it accessible to other Oracle Event Processing applications deployed to the same Oracle Event Processing server.

To query with Java classes:

1. Implement the `Address.java` class as Example 15–1 shows.

Example 15–5 Address.java Class

```
package test;

class Address {
```

```
    String street;
    String state;
    String city;
    String [] phones;
}
```

2. Export the `test` package that contains the `Address.java` class.

   For more information, see "How to Export a Package" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

   The `test` package may be part of this Oracle Event Processing application or it may be part of some other Oracle Event Processing application deployed to the same Oracle Event Processing server as this application.

3. Define the event type repository as Example 15–2 shows.

Example 15–6 Event Type Repository

```
<event-type-repository>
    <event-type name="Student">
        <property name="name" type="char"/>
        <property name="address" type="Address"/>
    </event-type>
<event-type-repository>
```

4. Assume that an adapter is providing `Student` events to channel `StudentStream` as Example 15–3 shows

Example 15–7 Channel

```
<channel id="StudentStream" event-type="Student"/>
```

5. Assume that the `StudentStream` is connected to a processor with the Oracle CQL query `q1` that Example 15–4 shows.

Example 15–8 Oracle CQL Query

```
<processor>
    <rules>
       <query id="q1"><![CDATA[
            select
                name,
                address.street as street,
                address.phones[0] as primary_phone
            from
                StudentStream
       ]]></query>
    </rules>
</processor>
```

   The Oracle Java data cartridge allows you to access the `address` event property from within the Oracle CQL query using normal Java API.

16

Oracle Spatial

This chapter provides a reference and guide to using the Oracle Spatial cartridge, which extends Oracle Continuous Query Language (Oracle CQL) to provide advanced spatial features for location-enabled applications.

You can use Oracle Spatial types, methods, fields, and constructors in Oracle CQL queries and views as you would Oracle CQL native types when building Oracle Event Processing applications.

This chapter includes the following sections:

- Understanding Oracle Spatial
- Using Oracle Spatial

For more information, see:

- Section , "Understanding Data Cartridges"
- Section , "Oracle CQL Data Cartridge Types"

Understanding Oracle Spatial

Oracle Spatial is an option for Oracle Database that provides advanced spatial features to support high-end geographic information systems (GIS) and location-enabled business intelligence solutions (LBS).

Oracle Spatial is an optional data cartridge which allows you to write Oracle CQL queries and views that seamlessly interact with Oracle Spatial classes in your Oracle Event Processing application.

Using Oracle Spatial, you can configure Oracle CQL queries that perform the most important geographic domain operations such as storing spatial data, performing proximity and overlap comparisons on spatial data, and integrating spatial data with the Oracle Event Processing server by providing the ability to index on spatial data.

To use Oracle Spatial, you require a working knowledge of the Oracle Spatial API. For more information about Oracle Spatial, see:

- Product overview:
  `http://www.oracle.com/technology/products/spatial/index.html`
- Oracle Spatial documentation:
  `http://www.oracle.com/pls/db112/portal.portal_db?selected=7&frame=#oracle_spatial_and_location_information`
- Oracle Spatial Java API reference:
  `http://download.oracle.com/docs/cd/E11882_01/appdev.112/e11829/toc.htm`

This section describes:

- Section , "Data Cartridge Name"
- Section , "Scope"
- Section , "Datatype Mapping"
- Section , "Oracle Spatial Application Context"

Data Cartridge Name

Oracle Spatial uses the cartridge ID `com.oracle.cep.cartrdiges.spatial` and registers the server-scoped reserved link name `spatial`.

Use the `spatial` link name to associate an Oracle Spatial method call with the Oracle Spatial application context.

For more information, see:

- Section , "Oracle Spatial Application Context"
- Section , "Geometry API"

Scope

Oracle Spatial is based on the Oracle Spatial Java API. Oracle Spatial exposes Oracle Spatial functionality in the `com.oracle.cep.cartridge.spatial.Geometry` class. Oracle Spatial functionality that is not in the Oracle Spatial Java API is not accessible from Oracle Spatial.

Using Oracle Spatial, your Oracle CQL queries may access the Oracle Spatial functionality that Table 16–1 describes.

Table 16–1 Oracle Spatial Scope

Oracle Spatial Feature	Scope
Geometry Types	The following geometry types from the Oracle Spatial Java API: ■ 2D points ■ 2D simple polygons ■ 2D rectangles The following geometry operations: ■ Creating geometry types ■ Accessing geometry type public member functions and public fields For more information, see: ■ Section , "Geometry Types" ■ Section , "Element Info Array"
Coordinate Systems	■ Cartesian and WGS84 geodetic coordinates (default) ■ Specifying the default coordinate system through SRID ■ Using other geodetic coordinates For more information, see Section , "Ordinates and Coordinate Systems and the SDO_SRID".
Geometric Index	■ R-Tree For more information, see Section , "Geometric Index".

Table 16–1 (Cont.) Oracle Spatial Scope

Oracle Spatial Feature	Scope
Geometric Relation Operators	■ ANYINTERACT ■ CONTAIN ■ INSIDE ■ WITHINDISTANCE For more information, see Section , "Geometric Relation Operators".
Geometric Filter Operators	■ FILTER ■ NN For more information, see Section , "Geometric Filter Operators".
Geometry API	For a complete list of the methods that `com.oracle.cep.cartridge.spatial.Geometry` provides, see Section , "Geometry API".

For more information on how to access these Oracle Spatial features using Oracle Spatial, see Section , "Using Oracle Spatial".

Geometry Types

The Oracle Spatial data model consists of geometries. A geometry is an ordered sequence of vertices. The semantics of the geometry are determined by its type.

Oracle Spatial allows you to access the following Oracle Spatial types directly in Oracle CQL queries and views:

- `SDO_GTYPES`: Oracle Spatial supports the following geometry types:
  - 2D points
  - 2D simple polygons
  - 2D rectangles

  Table 16–2 describes the geometry types from the `com.oracle.cep.cartridge.spatial.Geometry` class that you can use.

Table 16–2 Oracle Spatial Geometry Types

Geometry Type	Description
`GTYPE_POINT`	Point geometry type that contains one point.
`GTYPE_POLYGON`	Polygon geometry type that contains one polygon.

- `SDO_ELEMENT_INFO`: You can create the Element Info array using:
  - `com.oracle.cep.cartridge.spatial.Geometry.createElemInfo` static method
  - `einfogenerator` function

  For more information, see Section , "Element Info Array".
- `ORDINATES`: You can create the ordinates using the Oracle Spatial `ordsgenerator` function.

  For more information, see Section , "Ordinates and Coordinate Systems and the SDO_SRID".

For more information, see:

- Section , "How to Access the Geometry Types That the Oracle Spatial Java API Supports"

- Section , "How to Create a Geometry"
- Section , "How to Access Geometry Type Public Methods and Fields"

Element Info Array

The Element Info attribute is defined using a varying length array of numbers. This attribute specifies how to interpret the ordinates stored in the Ordinates attribute.

Oracle Spatial provides the following helper function for generating Element Info attribute values:

```
com.oracle.cep.cartridge.spatial.Geometry.createElemInfo(int SDO_STARTING_OFFSET,
int SDO_ETYPE , int SDO_INTERPRETATION)
```

You can also use the `einfogenerator` function.

For more information, see:

- "createElemInfo" on page 16-20
- "einfogenerator" on page 16-27
- "SDO_ELEM_INFO" in the *Oracle Spatial Developer's Guide*.

Ordinates and Coordinate Systems and the SDO_SRID

Table 16–3 lists the coordinate systems that Oracle Spatial supports by default and the `SDO_SRID` value that identifies each coordinate system.

Table 16–3 Oracle Spatial Coordinate Systems

Coordinate System	SDO_SRID	Description
Cartesian	0	Cartesian coordinates are coordinates that measure the position of a point from a defined origin along axes that are perpendicular in the represented space.
Geodetic (WGS84)	8307	Geodetic coordinates (sometimes called geographic coordinates) are angular coordinates (longitude and latitude), closely related to spherical polar coordinates, and are defined relative to a particular Earth geodetic datum. This is the default coordinate system in Oracle Spatial.

You can specify the `SDO_SRID` value as an argument to each Oracle Spatial method and constructor you call or you can configure the `SDO_SRID` in the Oracle Spatial application context once and use `com.oracle.cep.cartridge.spatial.Geometry` methods without having to set the `SDO_SRID` as an argument each time. Using the application context, you can also specify any coordinate system that Oracle Spatial supports.

Note: If you use a `com.oracle.cep.cartridge.spatial.Geometry` method that does not take an `SDO_SRID` value, then you must use the Oracle Spatial application context. For example, the following method call will cause a runtime exception:

```
com.oracle.cep.cartridge.spatial.Geometry.createPoint(lng, lat)
```

Instead, you must use the `spatial` link name to associate the method call with the Oracle Spatial application context:

```
com.oracle.cep.cartridge.spatial.Geometry.createPoint@spatial(lng,
lat)
```

If you use a `Geometry` method that takes an `SDO_SRID` value, then the use of the `spatial` link name is optional. For example, both the following method calls are valid:

```
com.oracle.cep.cartridge.spatial.Geometry.createPoint(8307, lng,
lat)
com.oracle.cep.cartridge.spatial.Geometry.createPoint@spatial(lng,
lat)
```

For more information, see Section , "Oracle Spatial Application Context".

Ordinates define the array of coordinates for a geometry using a double array. Oracle Spatial provides the `ordsgenerator` helper function for generating the array of coordinates. For syntax, see "ordsgenerator" on page 16-34.

For more information, see:

- "SDO_SRID" in the *Oracle Spatial Developer's Guide*
- "Coordinate Systems (Spatial Reference Systems)" in the *Oracle Spatial Developer's Guide*
- Section , "How to Use the Default Geodetic Coordinates"
- Section , "How to Use Other Geodetic Coordinates"

Geometric Index

Oracle Spatial uses a spatial index to implement the primary filter. The purpose of the spatial index is to quickly create a subset of the data and reduce the processing burden on the secondary filter.

A spatial index, like any other index, provides a mechanism to limit searches, but in this case the mechanism is based on spatial criteria such as intersection and containment.

Oracle Spatial uses R-Tree indexing for the default indexing mechanism. A spatial R-tree index can index spatial data of up to four dimensions. An R-tree index approximates each geometry by a single rectangle that minimally encloses the geometry (called the Minimum Bounding Rectangle, or MBR)

For more information, see:

- "Indexing of Spatial Data" in the *Oracle Spatial Developer's Guide*
- Section , "Geometric Filter Operators"

Geometric Relation Operators

Oracle Spatial supports the following Oracle Spatial geometric relation operators:

- `ANYINTERACT`
- `CONTAIN`
- `INSIDE`
- `WITHINDISTANCE`

You can use any of these operators in either the Oracle CQL query projection clause or where clause.

When you use a geometric relation operator in the where clause of an Oracle CQL query, Oracle Spatial enables Rtree indexing on the relation specified in the where clause.

Oracle Spatial supports only geometric relations between point and other geometry types.

For more information, see Section , "How to Use Geometry Relation Operators".

Geometric Filter Operators

Oracle Spatial supports the following Oracle Spatial geometric filter operators:

- `FILTER`
- `NN`

These filter operators perform primary filtering and so they may only appear in an Oracle CQL query where clause.

These filter operators use the spatial index to identify the set of spatial objects that are likely to interact spatially with the given object.

For more information, see:

- Section , "Geometric Index"
- Section , "How to Use Geometry Filter Operators".

Geometry API

Oracle Spatial is based on the Oracle Spatial Java API. Oracle Spatial exposes Oracle Spatial functionality in the `com.oracle.cep.cartridge.spatial.Geometry` class. This `Geometry` class also extends `oracle.spatial.geometry.J3D_Geometry`.

Although Oracle Spatial supports only 2D geometries, for efficiency, the `Geometry` class uses some `J3D_Geometry` methods. The `Geometry` class automatically zero-pads the Z coordinates for `J3D_Geometry` methods.

Oracle Spatial functionality inaccessible from the `Geometry` class (or not conforming to the scope and geometry types that Oracle Spatial supports) is inaccessible from Oracle Spatial.

This section describes:

- Section , "com.oracle.cep.cartridge.spatial.Geometry Methods"
- Section , "oracle.spatial.geometry.JGeometry Methods"

For more information, see:

- Section , "Scope"

- "ordsgenerator" on page 16-34

Note: To simplify Oracle Spatial type names, you can use aliases as Section , "Defining Aliases Using the Aliases Element" describes.

---

com.oracle.cep.cartridge.spatial.Geometry Methods Table 16–4 lists the public methods that the `Geometry` class provides.

Table 16–4 Oracle Spatial Geometry Methods

Type	Method
Buffers	■ `bufferPolygon`
Distance	■ `distance`
Element information	■ `createElemInfo`
Geometries	■ `createGeometry`
Linear polygons	■ `createLinearPolygon`
Minimum Bounding Rectangle (MBR)	■ `get2dMbr`
Points	■ `createPoint`
Rectangles	■ `createRectangle`
Type and type conversion	■ `to_Geometry` ■ `to_JGeometry`

Note: `Geometry` class methods are case sensitive and you must use them in the case shown.

Note: If you use a `com.oracle.cep.cartridge.spatial.Geometry` method that does not take an `SDO_SRID` value, then you must use the Oracle Spatial application context. For example, the following method call will cause a runtime exception:

```
com.oracle.cep.cartridge.spatial.Geometry.createPoint(lng, lat)
```

Instead, you must use the `spatial` link name to associate the method call with the Oracle Spatial application context:

```
com.oracle.cep.cartridge.spatial.Geometry.createPoint@spatial(lng,
lat)
```

If you use a `Geometry` method that takes an `SDO_SRID` value, then the use of the `spatial` link name is optional. For example, both the following method calls are valid:

```
com.oracle.cep.cartridge.spatial.Geometry.createPoint(8307, lng,
lat)
com.oracle.cep.cartridge.spatial.Geometry.createPoint@spatial(lng,
lat)
```

For more information, see Section , "Oracle Spatial Application Context".

---

oracle.spatial.geometry.JGeometry Methods The following `JGeometry` public methods are applicable to Oracle Spatial:

- `double area(double tolerance)`: returns the total planar surface area of a 2D geometry.
- `double length(double tolerance)`: returns the perimeter of a 2D geometry. All edge lengths are added.
- `double[] getMBR()`: returns the Minimum Bounding Rectangle (MBR) of this geometry. It returns a double array containing the `minX`, `minY`, `maxX`, and `maxY` value of the MBR for 2D.

For more information, see:

- http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28401/oracle/spatial/geometry/JGeometry.html

Datatype Mapping

The Oracle Spatial cartridge supports one data type: `com.oracle.cep.cartridge.spatial.Geometry`.

For a complete list of the methods that `com.oracle.cep.cartridge.spatial.Geometry` provides, see Section , "Geometry API".

Oracle Spatial Application Context

You can define an application context for an instance of Oracle Spatial and propagate this application context at runtime. This allows you to associate specific Oracle Spatial application defaults (such as an `SDO_SRID`) with a particular Oracle Spatial instance.

Before you can define an Oracle Spatial application context, edit your Oracle Event Processing application EPN assembly file to add the required namespace and schema location entries as Example 16–1 shows:

Example 16–1 EPN Assembly File: Oracle Spatial Namespace and Schema Location

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:osgi="http://www.springframework.org/schema/osgi"
       xmlns:wlevs="http://www.bea.com/ns/wlevs/spring"
       xmlns:spatial="http://www.oracle.com/ns/ocep/spatial"
       xsi:schemaLocation="
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/osgi
  http://www.springframework.org/schema/osgi/spring-osgi.xsd
  http://www.bea.com/ns/wlevs/spring
  http://www.bea.com/ns/wlevs/spring/spring-wlevs-v11_1_1_6.xsd"
  http://www.oracle.com/ns/ocep/spatial
  http://www.oracle.com/ns/ocep/spatial/ocep-spatial.xsd">
```

Example 16–2 shows how to create a spatial context named `SpatialGRS80` in an EPN assembly file using the Geodetic Reference System 1980 (GRS80) coordinate system.

Example 16–2 spatial:context Element in EPN Assembly File

```
<spatial:context id="SpatialGRS80" srid="4269" sma="6378137" rof="298.25722101" />
```

Example 16–3 shows how to reference a `spatial:context` in an Oracle CQL query. In this case, the query uses link name `SpatialGRS80` (defined in Example 16–2) to propagate this application context to Oracle Spatial. The `spatial:context` attribute settings of `SpatialGRS80` are applied to the `createPoint` method call.

Example 16–3 Referencing spatial:context in an Oracle CQL Query

```
<view  id="createPoint">
    select com.oracle.cep.cartridge.spatial.Geometry.createPoint@SpatialGRS80(
        lng, lat)
    from CustomerPos[NOW]
</view>
```

For more information (including a complete list of all `spatial:context` attributes), see "How to Configure Oracle Spatial Application Context" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

Note: If you use a `com.oracle.cep.cartridge.spatial.Geometry` method that does not take an `SDO_SRID` value, then you must use the Oracle Spatial application context. For example, the following method call will cause a runtime exception:

```
com.oracle.cep.cartridge.spatial.Geometry.createPoint(lng, lat)
```

Instead, you must use the `spatial` link name to associate the method call with the Oracle Spatial application context:

```
com.oracle.cep.cartridge.spatial.Geometry.createPoint@spatial(lng,
lat)
```

If you use a `Geometry` method that takes an `SDO_SRID` value, then the use of the `spatial` link name is optional. For example, both the following method calls are valid:

```
com.oracle.cep.cartridge.spatial.Geometry.createPoint(8307, lng,
lat)
com.oracle.cep.cartridge.spatial.Geometry.createPoint@spatial(lng,
lat)
```

For more information, see Section , "Geometry API".

Using Oracle Spatial

This section describes common use-cases that highlight how you can use Oracle Spatial in your Oracle Event Processing applications, including:

- Section , "How to Access the Geometry Types That the Oracle Spatial Java API Supports"
- Section , "How to Create a Geometry"
- Section , "How to Access Geometry Type Public Methods and Fields"
- Section , "How to Use Geometry Relation Operators"
- Section , "How to Use Geometry Filter Operators"
- Section , "How to Use the Default Geodetic Coordinates"
- Section , "How to Use Other Geodetic Coordinates"

For more information, see Section , "Geometry API".

How to Access the Geometry Types That the Oracle Spatial Java API Supports

This procedure describes how to access Oracle Spatial geometry types SDO_GTYPE, SDO_ELEMENT_INFO, and ORDINATES using Oracle Spatial in an Oracle CQL query.

To access the geometry types that the Oracle Spatial Java API supports:

1. Import the package com.oracle.cep.cartridge.spatial into your Oracle Event Processing application's MANIFEST.MF file.

   For more information, see "How to Import a Package" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

2. Define your Oracle Event Processing application event type using the appropriate Oracle Spatial data types.

   Example 16–4 shows how to define event type MySpatialEvent with two event properties x and y of type com.oracle.cep.cartridge.spatial.Geometry.

Example 16–4 Oracle Event Processing Event Using Oracle Spatial Types

```
<wlevs:event-type-repository>
    <wlevs:event-type type-name="MySpatialEvent">
        <wlevs:properties>
            <wlevs:property name="x" type="com.oracle.cep.cartridge.spatial.Geometry"/>
            <wlevs:property name="y" type="com.oracle.cep.cartridge.spatial.Geometry"/>
        </wlevs:properties>
    </wlevs:event-type>
</wlevs:event-type-repository>
```

   You can use these event properties in an Oracle CQL query like this:

   ```
   CONTAIN@spatial(x, y, 20.0d)
   ```

   For more information, see "Defining and Using Event Types" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

3. Choose an SDO_GTYPE, for example, GTYPE_POLYGON.

   For more information, see Section , "Geometry Types".

4. Choose the Element Info appropriate for your ordinates.

   For more information, see Section , "Element Info Array"

5. Define your coordinate values.

   For more information, see Section , "Ordinates and Coordinate Systems and the SDO_SRID".

6. Create your Oracle CQL query as Example 16–5 shows.

Example 16–5 Oracle CQL Query Using Oracle Spatial Geometry Types

```
view id="ShopGeom">
    select  com.oracle.cep.cartridge.spatial.Geometry.createGeometry@spatial(
        com.oracle.cep.cartridge.spatial.Geometry.GTYPE_POLYGON,
        com.oracle.cep.cartridge.spatial.Geometry.createElemInfo(1, 1003, 1),
        ordsgenerator@spatial(
            lng1, lat1, lng2, lat2, lng3, lat3,
            lng4, lat4, lng5, lat5, lng6, lat6
        )
    ) as geom
```

```
    from ShopDesc
</view>
```

How to Create a Geometry

You can use Oracle Spatial to create a geometry in an Oracle CQL query by invoking:

- static methods in `com.oracle.cartridge.spatial.Geometry`
- methods in `oracle.spatial.geometry.JGeometry` that conform to the scope and geometry types that Oracle Spatial supports.

For more information, see Section , "Geometry API".

Using a Static Method in the Oracle Spatial Geometry Class

Example 16–6 shows how to create a point geometry using a static method in `com.oracle.cartridge.spatial.Geometry`. In this case, you must use a link (`@spatial`) to identify the data cartridge that provides this class. The advantage of using this approach is that the Oracle Spatial application context is applied to set the SRID and other Oracle Spatial options, either by default or based on an application context you configure (see Section , "Oracle Spatial Application Context").

Example 16–6 Creating a Point Geometry Using a Geometry Static Method

```
<view  id="CustomerPosGeom">
    select com.oracle.cep.cartridge.spatial.Geometry.createPoint@spatial(
        lng, lat) as geom
    from CustomerPos[NOW]
</view>
```

For more information, see Section , "Geometry Types".

How to Access Geometry Type Public Methods and Fields

Using Oracle Spatial, you can access the public member functions and public member fields of Oracle Spatial classes directly in Oracle CQL.

Oracle Spatial functionality inaccessible from the `Geometry` class (or not conforming to the scope and geometry types that Oracle Spatial supports) is inaccessible from Oracle Spatial.

In Example 16–7, the view `ShopGeom` creates an Oracle Spatial geometry called `geom`. The view `shopMBR` calls `JGeometry` static method `getMBR` which returns a `double[]` as stream element `mbr`. The query `qshopMBR` accesses this `double[]` using regular Java API.

Example 16–7 Accessing Geometry Type Public Methods and Fields

```
<view id="ShopGeom">
    select  com.oracle.cep.cartridge.spatial.Geometry.createGeometry@spatial(
        com.oracle.cep.cartridge.spatial.Geometry.GTYPE_POLYGON,
        com.oracle.cep.cartridge.spatial.Geometry.createElemInfo(1, 1003, 1),
        ordsgenerator@spatial(
            lng1, lat1, lng2, lat2, lng3, lat3,
            lng4, lat4, lng5, lat5, lng6, lat6
        )
    ) as geom
    from ShopDesc
</view>
```

```
<view  id="shopMBR">
    select geom.getMBR() as mbr
    from ShopGeom
</view>
<query  id="qshopMBR">
    select mbr[0], mbr[1], mbr[2], mbr[3]
    from shopMBR
</query>
```

For more information, see:

- Section , "Geometry Types"
- Chapter 15, "Oracle Java Data Cartridge".

How to Use Geometry Relation Operators

Using Oracle Spatial, you can access the following Oracle Spatial geometry relation operators in either the WHERE or SELECT clause of an Oracle CQL query:

- ANYINTERACT
- CONTAIN
- INSIDE
- WITHINDISTANCE

In Example 16–8, the view op_in_where uses the CONTAIN geometry relation operator in the WHERE clause: in this case, Oracle Spatial uses R-Tree indexing. The view op_in_proj uses CONTAIN in the SELECT clause.

Example 16–8 Using Geometry Relation Operators

```
<view id="op_in_where">
    RStream(
        select
            loc.customerId,
            shop.shopId
        from
            LocGeomStream[NOW] as loc,
            ShopGeomRelation as shop
        where
            CONTAIN@spatial(shop.geom, loc.curLoc, 5.0d) = true
    )
</view>
<view id="op_in_proj">
    RStream(
        select
            loc.customerId,
            shop.shopId,
            CONTAIN@spatial(shop.geom, loc.curLoc, 5.0d)
        from
            LocGeomStream[NOW] as loc,
            ShopGeomRelation as shop
    )
</view>
```

For more information, see Section , "Geometric Relation Operators".

How to Use Geometry Filter Operators

Using Oracle Spatial, you can access the following Oracle Spatial geometry filter operators in the WHERE clause of an Oracle CQL query:

- FILTER
- NN

In Example 16–9, the view filter uses the FILTER geometry filter operator in the WHERE clause.

Example 16–9 Using Geometry Filter Operators

```
<view id="filter">
    RStream(
        select loc.customerId, shop.shopId
        from LocGeomStream[NOW] as loc, ShopGeomRelation as shop
        where FILTER@spatial(shop.geom, loc.curLoc, 5.0d) = true
    )
</view>
```

For more information, see Section , "Geometric Filter Operators".

How to Use the Default Geodetic Coordinates

When you create an Oracle CQL query using the default Oracle Spatial application context, the default SRID will be set to CARTESIAN.

As Example 16–10 shows, the createPoint method call uses the default link (@spatial). This guarantees that the default Oracle Spatial application context is applied.

Example 16–10 Using the Default Geodetic Coordinates in an Oracle CQL Query

```
<view  id="createPoint">
    select com.oracle.cep.cartridge.spatial.Geometry.createPoint@spatial(
        lng, lat)
    from CustomerPos[NOW]
</view>
```

For more information, see:

- Section , "Oracle Spatial Application Context"
- Section , "Ordinates and Coordinate Systems and the SDO_SRID"

How to Use Other Geodetic Coordinates

This procedure describes how to use the Oracle Spatial application context to specify a geodetic coordinate system other than the default Cartesian geodetic coordinate system in an Oracle CQL query:

For more information, see:

- Section , "Oracle Spatial Application Context"
- Section , "Ordinates and Coordinate Systems and the SDO_SRID"

To use other geodetic coordinates:

1. Create an Oracle Spatial application context and define the srid attribute for the geodetic coordinate system you want to use.

Example 16–11 shows how to create a spatial context named SpatialGRS80 in an EPN assembly file using the Geodetic Reference System 1980 (GRS80) coordinate system.

Example 16–11 spatial:context Element in EPN Assembly File

```
<spatial:context id="SpatialGRS80" srid="4269" sma="6378137" rof="298.25722101" />
```

2. In your Oracle CQL query, use the id of this spatial:context in your links.

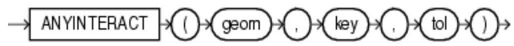
   Example 16–12 shows how to reference a spatial:context in an Oracle CQL query. In this case, the query uses link name SpatialGRS80 (defined in Example 16–11) to propagate this application context to Oracle Spatial. The spatial:context attribute settings of SpatialGRS80 are applied to the createPoint method call.

Example 16–12 Referencing spatial:context in an Oracle CQL Query

```
<view  id="createPoint">
    select com.oracle.cep.cartridge.spatial.Geometry.createPoint@SpatialGRS80(
        lng, lat)
    from CustomerPos[NOW]
</view>
```

ANYINTERACT

Syntax



Note: This is an Oracle Spatial geometric relation operator and not a method of the `com.oracle.cep.cartridge.spatial.Geometry` class so you invoke this operator as Example 16–13 shows, without a package prefix:

```
ANYINTERACT@spatial
```

Purpose

This operator returns `true` if the `GTYPE_POINT` interacts with the geometry, and `false` otherwise.

This operator takes the following arguments:

- `geom`: any supported geometry type.
- `key`: a `GTYPE_POINT` geometry type.

  The geometry type of this geometry must be `GTYPE_POINT` or a `RUNTIME_EXCEPTION` will be thrown.
- `tol`: the tolerance as a `double` value.

For more information, see "SDO_ANYINTERACT" in the *Oracle Spatial Developer's Guide*.

Examples

Example 16–27 shows how to use the `ANYINTERACT` Oracle Spatial geometric relation operator in an Oracle CQL query.

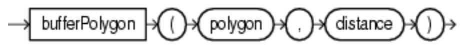
Example 16–13 Oracle CQL Query Using Geometric Relation Operator ANYINTERACT

```
<view id="op_in_where">
    RStream(
        select
            loc.customerId,
            shop.shopId
        from
            LocGeomStream[NOW] as loc,
            ShopGeomRelation as shop
        where
            ANYINTERACT@spatial(shop.geom, loc.curLoc, 5.0d) = true
    )
</view>
<view id="op_in_proj">
    RStream(
        select
            loc.customerId,
            shop.shopId,
            ANYINTERACT@spatial(shop.geom, loc.curLoc, 5.0d)
        from
```

```
                LocGeomStream[NOW] as loc,
                ShopGeomRelation as shop
    )
</view>
```

bufferPolygon

Syntax



Purpose

This `com.oracle.cep.cartridge.spatial.Geometry` method returns a new `com.oracle.cep.cartridge.spatial.Geometry` object which is the buffered version of the input `oracle.spatial.geometry.JGeometry` polygon.

This method takes the following arguments:

- `polygon`: an `oracle.spatial.geometry.JGeometry` polygon.
- `distance`: the distance value used for this buffer as a `double`.

  This value is assumed to be in the same unit as the Unit of Projection for projected geometry. If the geometry is geodetic, this buffer width should be in meters.

This method obtains parameters from the Oracle Spatial application context. Consequently, you must use the `spatial` link name to associate the method call with the Oracle Spatial application context:

```
com.oracle.cep.cartridge.spatial.Geometry.bufferPolygon@spatial(geom, 1300)
```

For more information, see Section , "Oracle Spatial Application Context".

Examples

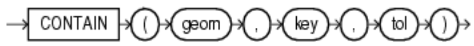
Example 16–14 shows how to use the `bufferPolygon` method. Because this `bufferPolygon` call depends on the Oracle Spatial application context, it uses the `spatial` link name.

Example 16–14 Oracle CQL Query Using Geometry.bufferPolygon

```
<view id="LocGeomStream" schema="customerId curLoc">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.bufferPolygon@spatial(geom, 13)
    from
        CustomerLocStream
</view>
```

CONTAIN

Syntax



Note: This is an Oracle Spatial geometric relation operator and not a method of the `com.oracle.cep.cartridge.spatial.Geometry` class so you invoke this operator as Example 16–13 shows, without a package prefix:

```
CONTAIN@spatial
```

Purpose

This operator returns `true` if the `GTYPE_POINT` is contained by the geometry, and `false` otherwise.

This operator takes the following arguments:

- `geom`: any supported geometry type.
- `key`: a `GTYPE_POINT` geometry type.

  The geometry type of this geometry must be `GTYPE_POINT` or a `RUNTIME_EXCEPTION` will be thrown.
- `tol`: the tolerance as a `double` value.

For more information, see "SDO_CONTAINS" in the *Oracle Spatial Developer's Guide*.

Examples

Example 16–27 shows how to use the `CONTAIN` Oracle Spatial geometric relation operator in an Oracle CQL query.

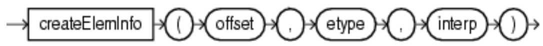
Example 16–15   Oracle CQL Query Using Geometric Relation Operator CONTAIN

```
<view id="op_in_where">
    RStream(
        select
            loc.customerId,
            shop.shopId
        from
            LocGeomStream[NOW] as loc,
            ShopGeomRelation as shop
        where
            CONTAIN@spatial(shop.geom, loc.curLoc, 5.0d) = true
    )
</view>
<view id="op_in_proj">
    RStream(
        select
            loc.customerId,
            shop.shopId,
            CONTAIN@spatial(shop.geom, loc.curLoc, 5.0d)
        from
            LocGeomStream[NOW] as loc,
```

```
            ShopGeomRelation as shop
    )
</view>
```

createElemInfo

Syntax



> **Note:** Alternatively, you can use the function `einfogenerator`. For more information, see "einfogenerator" on page 16-27.

Purpose

This `com.oracle.cep.cartridge.spatial.Geometry` method returns a single element info value as an `int[]` from the given arguments.

This method takes the following arguments:

- `soffset`: the offset, as an `int`, within the ordinates array where the first ordinate for this element is stored.

  `SDO_STARTING_OFFSET` values start at 1 and not at 0. Thus, the first ordinate for the first element will be at `SDO_GEOMETRY.Ordinates(1)`. If there is a second element, its first ordinate will be at `SDO_GEOMETRY.Ordinates(`*n* `* 3 + 2)`, where *n* reflects the position within the `SDO_ORDINATE_ARRAY` definition.

- `etype`: the type of the element as an `int`.

  Oracle Spatial supports `SDO_ETYPE` values 1, 1003, and 2003 are considered simple elements (not compound types). They are defined by a single triplet entry in the element info array. These types are:

  – 1: point.

  – 1003: exterior polygon ring (must be specified in counterclockwise order).

  – 2003: interior polygon ring (must be specified in clockwise order).

  These types are further qualified by the `SDO_INTERPRETATION`.

  > **Note:** You cannot mix 1-digit and 4-digit `SDO_ETYPE` values in a single geometry.

- `interp`: the interpretation as an `int`.

  For an SDO_ETYPE that is a simple element (1, 1003, or 2003) the `SDO_INTERPRETATION` attribute determines how the sequence of ordinates for this element is interpreted. For example, a polygon boundary may be made up of a sequence of connected straight line segments.

  If a geometry consists of more than one element, then the last ordinate for an element is always one less than the starting offset for the next element. The last element in the geometry is described by the ordinates from its starting offset to the end of the ordinates varying length array.

  Table 16–5 describes the relationship between `SDO_ETYPE` and `SDO_INTERPREATION`.

Table 16–5 SDO_ETYPE and SDO_INTERPRETATION

SDO_ETYPE	SDO_INTERPRETATION	Description
0	Any numeric value	Used to model geometry types not supported by Oracle Spatial.
1	1	Point type.
1	0	Orientation for an oriented point.
1003 or 2003	1	Simple polygon whose vertices are connected by straight line segments. You must specify a point for each vertex; and the last point specified must be exactly the same point as the first (within the tolerance value), to close the polygon. For example, for a 4-sided polygon, specify 5 points, with point 5 the same as point 1.

Examples

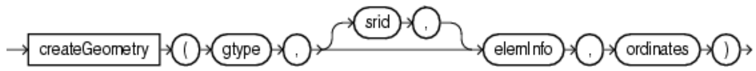
Example 16–16 shows how to use the `createElemInfo` method.

Example 16–16 Oracle CQL Query Using Geometry.createElemInfo

```
<view id="ShopGeom">
    select  com.oracle.cep.cartridge.spatial.Geometry.createGeometry@spatial(
        com.oracle.cep.cartridge.spatial.Geometry.GTYPE_POLYGON,
        com.oracle.cep.cartridge.spatial.Geometry.createElemInfo(1, 1003, 1),
        ordsgenerator@spatial(
            lng1, lat1, lng2, lat2, lng3, lat3,
            lng4, lat4, lng5, lat5, lng6, lat6
        )
    ) as geom
    from ShopDesc
</view>
```

createGeometry

Syntax



Purpose

This `com.oracle.cep.cartridge.spatial.Geometry` method returns a new 2D `oracle.cep.cartridge.spatial.Geometry` object.

This method takes the following arguments:

- `gtype`: the geometry type as an `int`.

  For more information, see Table 16–2.

- `eleminfo`: the geometry element info as an `int[]`.

  For more information, see "createElemInfo" on page 16-20.

- `ordinates`: the geometry ordinates as a `double[]`.
- `srid`: the optional `SDO_SRID` of the geometry as an `int`.

  If you omit the `srid` parameter, then this method obtains parameters from the Oracle Spatial application context. Consequently, you must use the `spatial` link name to associate the method call with the Oracle Spatial application context:

  ```
  com.oracle.cep.cartridge.spatial.Geometry.createGeometry@spatial(gtype,
  eleminfo, ordinates)
  ```

  For more information, see Section , "Oracle Spatial Application Context".

Examples

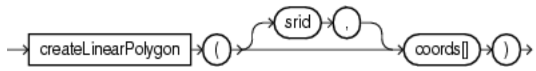
Example 16–17 shows how to use the `createGeometry` method. Because this `createGeometry` call does not include the `srid` argument, it uses the `spatial` link name to associate the method call with the Oracle Spatial application context.

Example 16–17 Oracle CQL Query Using Geometry.createGeometry

```
<view id="ShopGeom">
    select  com.oracle.cep.cartridge.spatial.Geometry.createGeometry@spatial(
        com.oracle.cep.cartridge.spatial.Geometry.GTYPE_POLYGON,
        com.oracle.cep.cartridge.spatial.Geometry.createElemInfo(1, 1003, 1),
        ordsgenerator@spatial(
            lng1, lat1, lng2, lat2, lng3, lat3,
            lng4, lat4, lng5, lat5, lng6, lat6
        )
    ) as geom
    from ShopDesc
</view>
```

createLinearPolygon

Syntax



Purpose

This com.oracle.cep.cartridge.spatial.Geometry method returns a new com.oracle.cep.cartridge.spatial.Geometry object which is a 2D simple linear polygon without holes. If the coordinate array does not close itself (the last coordinate is not the same as the first) then this method copies the first coordinate and appends this coordinate value to the end of the input coordinates array.

To create a simple linear polygon without holes, use the following arguments:

- coords: the coordinates of the linear polygon as a double[].
- srid: the optional SRID of the geometry as an int.

  If you omit the srid parameter, then this method obtains parameters from the Oracle Spatial application context. Consequently, you must use the spatial link name to associate the method call with the Oracle Spatial application context:

  ```
  com.oracle.cep.cartridge.spatial.Geometry.createLinearPolygon@spatial(coords)
  ```

  For more information, see Section , "Oracle Spatial Application Context".

Examples

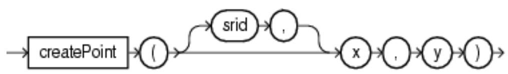
Example 16–18 shows how to use the createLinearPolygon method. Because this createLinearPolygon method call does not include the srid argument, it must use the spatial link name to associate the method call with the Oracle Spatial application context.

Example 16–18 Oracle CQL Query Using Geometry.createLinearPolygon

```
<view id="LocGeomStream" schema="customerId curLoc">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.createLinearPolygon@spatial(coords)
    from
        CustomerLocStream
</view>
```

createPoint

Syntax



Purpose

This `com.oracle.cep.cartridge.spatial.Geometry` method returns a new `com.oracle.cep.cartridge.spatial.Geometry` object which is a 3D point.

This method takes the following arguments:

- `x`: the x coordinate of the lower left as a `double`.
- `y`: the y coordinate of the lower left as a `double`.
- `srid`: the optional `SRID` of the geometry as an `int`.

  If you omit the `srid` parameter, then this method obtains parameters from the Oracle Spatial application context. Consequently, you must use the `spatial` link name to associate the method call with the Oracle Spatial application context:

  ```
  com.oracle.cep.cartridge.spatial.Geometry.createPoint@spatial(x, y)
  ```

  For more information, see Section , "Oracle Spatial Application Context".

Examples

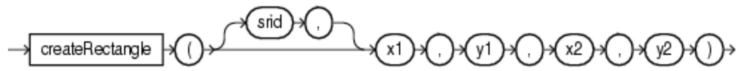
Example 16–19 shows how to use the `createPoint` method. Because this `createPoint` call includes the `srid` argument, it does not need to use the `spatial` link name.

Example 16–19 Oracle CQL Query Using Geometry.createPoint

```
<view id="LocGeomStream" schema="customerId curLoc">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.createPoint@spatial(lng, lat)
    from
        CustomerLocStream
</view>
```

createRectangle

Syntax



Purpose

This `com.oracle.cep.cartridge.spatial.Geometry` method returns a new `com.oracle.cep.cartridge.spatial.Geometry` object which is a 2D rectangle.

This method takes the following arguments:

- `x1`: the x coordinate of the lower left as a `double`.
- `y1`: the y coordinate of the lower left as a `double`.
- `x2`: the x coordinate of the upper right as a `double`.
- `y2`: the y coordinate of the upper right as a `double`.
- `srid`: the optional `SRID` of the geometry as an `int`.

  If you omit the `srid` parameter, then this method obtains parameters from the Oracle Spatial application context. Consequently, you must use the `spatial` link name to associate the method call with the Oracle Spatial application context:

  ```
  com.oracle.cep.cartridge.spatial.Geometry.createRectangle@spatial(x1, y1, x2,
  y2)
  ```

  For more information, see Section , "Oracle Spatial Application Context".

Examples

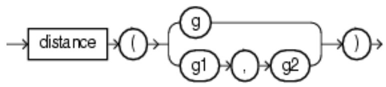
Example 16–20 shows how to use the `createRectangle` method. Because this `createRectangle` method call does not include the `srid` argument, it must use the `spatial` link name to associate the method call with the Oracle Spatial application context.

Example 16–20 Oracle CQL Query Using Geometry.createRectangle

```
<view id="LocGeomStream" schema="customerId curLoc">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.createRectangle@spatial(x1, y1, x2, y2)
    from
        CustomerLocStream
</view>
```

distance

Syntax



Purpose

This `com.oracle.cep.cartridge.spatial.Geometry` method calculates the distance between two geometries as a `double`.

To calculate the distance between a given `com.oracle.cep.cartridge.spatial.Geometry` object and another, use the non-static `distance` method of the current `Geometry` object with the following arguments:

- `g`: the other `com.oracle.cep.cartridge.spatial.Geometry` object.

To calculate the distance between two `com.oracle.cep.cartridge.spatial.Geometry` objects, use the static `distance` method with the following arguments:

- `g1`: the first `com.oracle.cep.cartridge.spatial.Geometry` object.
- `g2`: the second `com.oracle.cep.cartridge.spatial.Geometry` object.

In both cases, this method obtains parameters from the Oracle Spatial application context. Consequently, you must use the `spatial` link name to associate the method call with the Oracle Spatial application context:

```
com.oracle.cep.cartridge.spatial.Geometry.distance@spatial(geom)
com.oracle.cep.cartridge.spatial.Geometry.distance@spatial(geom1, geom2)
```

For more information, see Section , "Oracle Spatial Application Context".

Examples

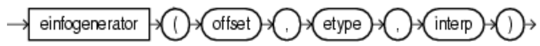
Example 16–21 shows how to use the `distance` method. Because the `distance` method depends on the Oracle Spatial application context, it must use the `spatial` link name.

Example 16–21 Oracle CQL Query Using Geometry.distance

```
<view id="LocGeomStream" schema="customerId curLoc">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.createRectangle(x1, y1, x2, y2, 8307)
    from
        CustomerLocStream
    where
        com.oracle.cep.cartridge.spatial.Geometry.distance@spatial(geom1, geom2) < 5
</view>
```

einfogenerator

Syntax



Note: This is an Oracle CQL function and not a method of the `com.oracle.cep.cartridge.spatial.Geometry` class so you invoke this function as Example 16–22 shows, without a package prefix:

```
einfogenerator@spatial
```

Alternatively, you can use the `Geometry` method `createElemInfo`. For more information, see "createElemInfo" on page 16-20.

Purpose

This function returns a single element info value as an `int[]` from the given arguments.

This function takes the following arguments:

- `offset`: the offset, as an `int`, within the ordinates array where the first ordinate for this element is stored.

  `SDO_STARTING_OFFSET` values start at 1 and not at 0. Thus, the first ordinate for the first element will be at `SDO_GEOMETRY.Ordinates(1)`. If there is a second element, its first ordinate will be at `SDO_GEOMETRY.Ordinates(n * 3 + 2)`, where *n* reflects the position within the `SDO_ORDINATE_ARRAY` definition.

- `etype`: the type of the element as an `int`.

  Oracle Spatial supports `SDO_ETYPE` values 1, 1003, and 2003 are considered simple elements (not compound types). They are defined by a single triplet entry in the element info array. These types are:

  – 1: point.

  – 1003: exterior polygon ring (must be specified in counterclockwise order).

  – 2003: interior polygon ring (must be specified in clockwise order).

  These types are further qualified by the `SDO_INTERPRETATION`.

  Note: You cannot mix 1-digit and 4-digit `SDO_ETYPE` values in a single geometry.

- `interp`: the interpretation as an `int`.

  For an `SDO_ETYPE` that is a simple element (1, 1003, or 2003) the `SDO_INTERPRETATION` attribute determines how the sequence of ordinates for this element is interpreted. For example, a polygon boundary may be made up of a sequence of connected straight line segments.

  If a geometry consists of more than one element, then the last ordinate for an element is always one less than the starting offset for the next element. The last element in the geometry is described by the ordinates from its starting offset to the end of the ordinates varying length array.

Table 16–6 describes the relationship between `SDO_ETYPE` and `SDO_INTERPREATION`.

Table 16–6 SDO_ETYPE and SDO_INTERPRETATION

SDO_ETYPE	SDO_INTERPRETATION	Description
0	Any numeric value	Used to model geometry types not supported by Oracle Spatial.
1	1	Point type.
1	0	Orientation for an oriented point.
1003 or 2003	1	Simple polygon whose vertices are connected by straight line segments. You must specify a point for each vertex; and the last point specified must be exactly the same point as the first (within the tolerance value), to close the polygon. For example, for a 4-sided polygon, specify 5 points, with point 5 the same as point 1.

Examples

Example 16–22 shows how to use the `oeinfogenerator` function to create the element information for a geometry.

Example 16–22 Oracle CQL Query Using Oracle Spatial Geometry Types

```
view id="ShopGeom">
    select  com.oracle.cep.cartridge.spatial.Geometry.createGeometry@spatial(
        com.oracle.cep.cartridge.spatial.Geometry.GTYPE_POLYGON,
        einfogenerator@spatial(1, 1003, 1),
        ordsgenerator@spatial(
            lng1, lat1, lng2, lat2, lng3, lat3,
            lng4, lat4, lng5, lat5, lng6, lat6
        )
    ) as geom
    from ShopDesc
</view>
```

FILTER

Syntax

Note: This is an Oracle Spatial geometric filter operator and not a method of the `com.oracle.cep.cartridge.spatial.Geometry` class so you invoke this operator as Example 16–13 shows, without a package prefix:

```
FILTER@spatial
```

Purpose

This operator returns `true` for object pairs that are non-disjoint, and `false` otherwise.

This operator takes the following arguments:

- `key`: a `GTYPE_POINT` geometry type.

  The geometry type of this geometry must be `GTYPE_POINT` or a `RUNTIME_EXCEPTION` will be thrown.
- `tol`: the tolerance as a `double` value.

For more information, see "SDO_FILTER" in the *Oracle Spatial Developer's Guide*.

Examples

Example 16–27 shows how to use the `FILTER` Oracle Spatial geometric filter operator in an Oracle CQL query.

Example 16–23 Oracle CQL Query Using Geometric Relation Operator FILTER

```
<view id="filter">
    RStream(
        select loc.customerId, shop.shopId
        from LocGeomStream[NOW] as loc, ShopGeomRelation as shop
        where FILTER@spatial(loc.curLoc, 5.0d) = true
    )
</view>
```

get2dMbr

Syntax

Purpose

This `com.oracle.cep.cartridge.spatial.Geometry` method returns the Minimum Bounding Rectangle (MBR) of a given `Geometry` as a `double[][]` that contains the following values:

- `[0][0]: minX`
- `[0][1]: maxX`
- `[1][0]: minY`
- `[1][1]: maxY`

This method takes the following arguments:

- `geom`: the `com.oracle.cep.cartridge.spatial.Geometry` object.

Examples

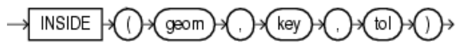
Example 16–24 shows how to use the `get2dMbr` method.

Example 16–24 Oracle CQL Query Using Geometry.get2dMbr

```
<view id="LocGeomStream" schema="customerId mbr">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.get2dMbr(geom)
    from
        CustomerLocStream
    where
        com.oracle.cep.cartridge.spatial.Geometry.distance@spatial(geom1, geom2) < 5
</view>
```

INSIDE

Syntax



Note: This is an Oracle Spatial geometric relation operator and not a method of the `com.oracle.cep.cartridge.spatial.Geometry` class so you invoke this operator as Example 16–13 shows, without a package prefix:

```
INSIDE@spatial
```

Purpose

This operator returns `true` if the `GTYPE_POINT` is inside the geometry, and `false` otherwise.

This operator takes the following arguments:

- `geom`: any supported geometry type.
- `key`: a `GTYPE_POINT` geometry type.

  The geometry type of this geometry must be `GTYPE_POINT` or a `RUNTIME_EXCEPTION` will be thrown.
- `tol`: the tolerance as a `double` value.

For more information, see "SDO_INSIDE" in the *Oracle Spatial Developer's Guide*.

Examples

Example 16–27 shows how to use the `INSIDE` Oracle Spatial geometric relation operator in an Oracle CQL query.

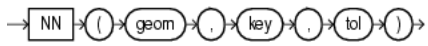
Example 16–25 Oracle CQL Query Using Geometric Relation Operator INSIDE

```
<view id="op_in_where">
    RStream(
        select
            loc.customerId,
            shop.shopId
        from
            LocGeomStream[NOW] as loc,
            ShopGeomRelation as shop
        where
            INSIDE@spatial(shop.geom, loc.curLoc, 5.0d) = true
    )
</view>
<view id="op_in_proj">
    RStream(
        select
            loc.customerId,
            shop.shopId,
            INSIDE@spatial(shop.geom, loc.curLoc, 5.0d)
        from
            LocGeomStream[NOW] as loc,
```

```
                ShopGeomRelation as shop
    )
</view>
```

NN

Syntax



> **Note:** This is an Oracle Spatial geometric filter operator and not a method of the `com.oracle.cep.cartridge.spatial.Geometry` class so you invoke this operator as Example 16–13 shows, without a package prefix:
>
> `NN@spatial`

Purpose

This operator returns the objects (nearest neighbors) from `geom` that are nearest to `key`. In determining how near two geometry objects are, the shortest possible distance between any two points on the surface of each object is used.

This function takes the following arguments:

- `geom`: any supported geometry type.
- `key`: a `GTYPE_POINT` geometry type.

  The geometry type of this geometry must be `GTYPE_POINT` or a `RUNTIME_EXCEPTION` will be thrown.
- `tol`: the tolerance as a `double` value.

For more information, see "SDO_NN" in the *Oracle Spatial Developer's Guide*.

Examples

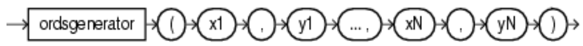
Example 16–27 shows how to use the `NN` Oracle Spatial geometric filter operator in an Oracle CQL query.

Example 16–26 Oracle CQL Query Using Geometric Relation Operator NN

```
<view id="filter">
    RStream(
        select loc.customerId, shop.shopId
        from LocGeomStream[NOW] as loc, ShopGeomRelation as shop
        where NN@spatial(shop.geom, loc.curLoc, 5.0d) = true
    )
</view>
```

ordsgenerator

Syntax



> **Note:** This is an Oracle CQL function and not a method of the `com.oracle.cep.cartridge.spatial.Geometry` class so you invoke this function as Example 16–27 shows, without a package prefix:
>
> `ordsgenerator@spatial`

Purpose

This function returns the `double` array of 2D coordinates that Oracle Spatial requires.

This function takes the following arguments:

- `x1, y1, ... xN, yN`: a comma-separated list of `double` coordinate values.

Examples

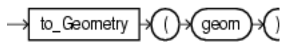
Example 16–27 shows how to use the `ordsgenerator` function to create an Oracle Spatial double array out of six `double` coordinate values.

Example 16–27 Oracle CQL Query Using Oracle Spatial Geometry Types

```
view id="ShopGeom">
    select  com.oracle.cep.cartridge.spatial.Geometry.createGeometry@spatial(
        com.oracle.cep.cartridge.spatial.Geometry.GTYPE_POLYGON,
        com.oracle.cep.cartridge.spatial.Geometry.createElemInfo(1, 1003, 1),
        ordsgenerator@spatial(
            lng1, lat1, lng2, lat2, lng3, lat3,
            lng4, lat4, lng5, lat5, lng6, lat6
        )
    ) as geom
    from ShopDesc
</view>
```

to_Geometry

Syntax



Purpose

This `com.oracle.cep.cartridge.spatial.Geometry` method converts an `oracle.spatial.geometry.JGeometry` type to a 3D `com.oracle.cep.cartridge.spatial.Geometry` type. If the given geometry is already a `Geometry` type and a 3D geometry, then no conversion is done. If the given geometry is a 2D geometry, then the given geometry is converted to 3D by padding z coordinates.

This method takes the following arguments:

- `geom`: the `oracle.spatial.geometry.JGeometry` object to convert.

Examples

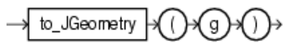
Example 16–28 shows how to use the `to_Geometry` method.

Example 16–28 Oracle CQL Query Using Geometry.to_Geometry

```
<view id="LocStream" schema="customerId loc">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.to_Geometry(geom)
    from
        CustomerLocStream
</view>
```

to_JGeometry

Syntax



Purpose

This `com.oracle.cep.cartridge.spatial.Geometry` method converts a `com.oracle.cep.cartridge.spatial.Geometry` object to an `oracle.spatial.geometry.JGeometry` 2D type.

This method takes the following arguments:

- `g`: the `com.oracle.cep.cartridge.spatial.Geometry` object to convert.

Examples

Example 16–29 shows how to use the `to_JGeometry` method.

Example 16–29 Oracle CQL Query Using Geometry.to_JGeometry

```
<view id="LocStream" schema="customerId loc">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.to_JGeometry(geom)
    from
        CustomerLocStream
</view>
```

WITHINDISTANCE

Syntax

Note: This is an Oracle Spatial geometric relation operator and not a method of the `com.oracle.cep.cartridge.spatial.Geometry` class so you invoke this operator as Example 16–13 shows, without a package prefix:

```
WITHINDISTANCE@spatial
```

Purpose

This operator returns `true` if the `GTYPE_POINT` is within the given distance of the geometry, and `false` otherwise.

This operator takes the following arguments:

- `geom`: any supported geometry type.
- `key`: a `GTYPE_POINT` geometry type.

  The geometry type of this geometry must be `GTYPE_POINT` or a `RUNTIME_EXCEPTION` will be thrown.
- `dist`: the distance as a `double` value.

For more information, see "SDO_WITHIN_DISTANCE" in the *Oracle Spatial Developer's Guide*.

Examples

Example 16–27 shows how to use the `WITHINDISTANCE` Oracle Spatial geometric relation operator in an Oracle CQL query.

Example 16–30 Oracle CQL Query Using Geometric Relation Operator WITHINDISTANCE

```
<view id="op_in_where">
    RStream(
        select
            loc.customerId,
            shop.shopId
        from
            LocGeomStream[NOW] as loc,
            ShopGeomRelation as shop
        where
            WITHINDISTANCE@spatial(shop.geom, loc.curLoc, 5.0d) = true
    )
</view>
<view id="op_in_proj">
    RStream(
        select
            loc.customerId,
            shop.shopId,
            WITHINDISTANCE@spatial(shop.geom, loc.curLoc, 5.0d)
        from
```

```
            LocGeomStream[NOW] as loc,
            ShopGeomRelation as shop
    )
</view>
```

17

Oracle Event Processing JDBC Data Cartridge

This chapter describes the Oracle Event Processing JDBC data cartridge, an Oracle Continuous Query Language (Oracle CQL) extension through which you execute a SQL query against a database and use its returned results in a CQL query.

When using functionality provided by the cartridge, you are associating a SQL query with a JDBC cartridge function definition. Then, from a CQL query, you can call the JDBC cartridge function, which executes the associated SQL query against the database. The function call must be enclosed in the TABLE clause, which lets you use the SQL query results as a CQL relation in the CQL query making that function call.

For information the TABLE clause, see Section , "Using the TABLE Clause."

This chapter includes the following sections:

- Understanding the Oracle Event Processing JDBC Data Cartridge
- Using the Oracle Event Processing JDBC Data Cartridge

For more information, see:

- Section , "Understanding Data Cartridges"
- Section , "Oracle CQL Data Cartridge Types"

Understanding the Oracle Event Processing JDBC Data Cartridge

Oracle Event Processing streams contain streaming data, and a database typically stores historical data. Use the Oracle Event Processing JDBC data cartridge to associate historical data (stored in one or more tables) with the streaming data coming from Oracle Event Processing streams. The Oracle Event Processing JDBC data cartridge executes arbitrary SQL query against a database and uses the results in the CQL query. This section describes how to associate streaming and historical data using the Oracle Event Processing JDBC data cartridge.

This section describes:

- Section , "Data Cartridge Name"
- Section , "Scope"
- Section , "Datatype Mapping"
- Section , "Oracle Event Processing JDBC Data Cartridge Application Context"

Data Cartridge Name

The Oracle Event Processing JDBC data cartridge uses the cartridge ID `com.oracle.cep.cartridge.jdbc`. This ID is reserved and cannot be used by any other cartridges.

For more information, see Section , "Oracle Event Processing JDBC Data Cartridge Application Context".

Scope

The Oracle Event Processing JDBC data cartridge supports arbitrarily complex SQL statements with the following restrictions:

- You may use only native SQL types in the `SELECT` list of the SQL query.
- You may not use user-defined types and complex database types in the `SELECT` list.
- You must provide alias names for every `SELECT` list column in the SQL query.

For more information, see Section , "Datatype Mapping".

Note: To use the Oracle Event Processing JDBC data cartridge, your data source must use Oracle JDBC driver version 11.2 or higher.

For more information, see "Configuring Access to a Different Database Driver or Driver Version" in the *Oracle Fusion Middleware Administrator's Guide for Oracle Event Processing*

Datatype Mapping

This section describes Oracle Event Processing JDBC data cartridge datatype mapping.

For reference, consider the Oracle Event Processing JDBC data cartridge context function that Example 17–1 shows.

Example 17–1 Oracle Event Processing JDBC Data Cartridge SQL Statement

```
...
<jc:jdbc-ctx>
    <name>JdbcCartridgeOne</name>
    <data-source>StockDS</data-source>
    <function name="getDetailsByOrderIdName">
        <param name="inpOrderId" type="int" />
        <param name="inpName" type="char" />
        <return-component-type>
            com.oracle.cep.example.jdbc_cartridge.RetEvent
        </return-component-type>
        <sql><![CDATA[
            SELECT
                Employee.empName as employeeName,
                Employee.empEmail as employeeEmail,
                OrderDetails.description as description
            FROM
                 PlacedOrders, OrderDetails , Employee
            WHERE
                PlacedOrders.empId = Employee.empId AND
                PlacedOrders.orderId = OrderDetails.orderId AND
                Employee.empName = :inpName AND
                PlacedOrders.orderId = :inpOrderId
        ]]></sql>
    </function>
</jc:jdbc-ctx>
```

...

For more information, see Section , "Scope".

Oracle Event Processing JDBC Data Cartridge Application Context

To use the Oracle Event Processing JDBC data cartridge, you must declare and configure one or more application-scoped JDBC cartridge context while developing an application, as described in the following steps:

- Section , "Declare a JDBC Cartridge Context in the EPN File"
- Section , "Configure the JDBC Cartridge Context in the Application Configuration File"

Declare a JDBC Cartridge Context in the EPN File

To declare a JDBC cartridge context in the EPN file:

1. Edit your Oracle Event Processing application EPN assembly file to add the required namespace and schema location entries as shown in Example 17–2.
2. Add an entry with the tag `jdbc-context` in the EPN file and specify the `id` attribute, as shown in Example 17–3. The `id` represents the name of this application-scoped context and is used in CQL queries that reference functions defined in this context. The `id` is also used when this context is configured in the application configuration file.

Example 17–2 EPN Assembly File: Oracle Event Processing JDBC Data Cartridge Namespace and Schema Location

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:osgi="http://www.springframework.org/schema/osgi"
       xmlns:wlevs="http://www.bea.com/ns/wlevs/spring"
       xmlns:jdbc="http://www.oracle.com/ns/ocep/jdbc"
       xsi:schemaLocation="
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/osgi
  http://www.springframework.org/schema/osgi/spring-osgi.xsd
  http://www.bea.com/ns/wlevs/spring
  http://www.bea.com/ns/wlevs/spring/spring-wlevs-v11_1_1_6.xsd"
  http://www.oracle.com/ns/ocep/jdbc
  http://www.oracle.com/ns/ocep/jdbc/ocep-jdbc.xsd">
```

Example 17–3 shows how to create an Oracle Event Processing JDBC data cartridge application context named `JdbcCartridgeOne` in an EPN assembly file.

Example 17–3 jdbc:jdbc-context Element in EPN Assembly File

```
<jdbc:jdbc-context id="JdbcCartridgeOne"/>
```

Configure the JDBC Cartridge Context in the Application Configuration File

To configure the JDBC cartridge context, add the configuration details in the component configuration file that is typically placed under the application's /wlevs directory. This configuration is similar to configuring other EPN components such as channel and processor.

To configure the JDBC cartridge context in the application configuration file:

1. Before adding the JDBC context configuration, add the required namespace entry to the configuration XML file, as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<jdbcctxconfig:config
xmlns:jdbcctxconfig="http://www.bea.com/ns/wlevs/config/application"
    xmlns:jc="http://www.oracle.com/ns/ocep/config/jdbc">
```

2. The JDBC cartridge context configuration is done under the parent level tag `jdbc-ctx`. A context defines one or more functions, each of which is associated with a single SQL query. The configuration also specifies the data source representing the database against which the SQL queries are to be executed. Each function can have input parameters that are used to pass arguments to the SQL query defining the function, and each function specifies the return-component-type. Since the call to this function is always enclosed within a TABLE clause, the function always returns a Collection type. The return-component-type property indicates the type of the component of that collection.

   The value of the `name` property must match the value used for the `id` attribute in the EPN file, as shown in Example 17–3.

   Example 17–4 shows how to reference the `jdbc:jbdc-context` in an Oracle CQL query. In this case, the query uses link name `JdbcCartridgeOne` (defined in Example 17–3) to propagate this application context to the Oracle Event Processing JDBC data cartridge. The Oracle CQL query in Example 17–4 invokes the function `getDetailsByOrderIdName` associated with Oracle Event Processing JDBC data cartridge application context `JdbcCartridgeOne`.

Example 17–4 jc:jdbc-ctx Element in Component Configuration File

```
...
<jc:jdbc-ctx>
    <name>JdbcCartridgeOne</name>
    <data-source>StockDS</data-source>
    <function name="getDetailsByOrderIdName">
        <param name="inpOrderId" type="int" />
        <param name="inpName" type="char" />
        <return-component-type>
            com.oracle.cep.example.jdbc_cartridge.RetEvent
        </return-component-type>
        <sql><![CDATA[
            SELECT
                Employee.empName as employeeName,
                Employee.empEmail as employeeEmail,
                OrderDetails.description as description
            FROM
                 PlacedOrders, OrderDetails , Employee
            WHERE
                PlacedOrders.empId = Employee.empId AND
                PlacedOrders.orderId = OrderDetails.orderId AND
                Employee.empName = :inpName AND
                PlacedOrders.orderId = :inpOrderId
        ]]></sql>
    </function>
</jc:jdbc-ctx>
...
<processor>
        <name>Proc</name>
        <rules>
            <query id="q1"><![CDATA[
                RStream(
                   select
```

```
                     currentOrder.orderId,
                     details.orderInfo.employeeName,
                     details.orderInfo.employeeemail,
                     details.orderInfo.description
                   from
                     OrderArrival[now] as currentOrder,
                     TABLE(getDetailsByOrderIdName@JdbcCartridgeOne(
                               currentOrder.orderId, currentOrder.empName
                           ) as orderInfo
                     ) as details
              )
          ]]></query>
      </rules>
</processor>
...
```

For more information, see "How to Configure Oracle Event Processing JDBC Data Cartridge Application Context" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

Using the Oracle Event Processing JDBC Data Cartridge

In general, you use the Oracle Event Processing JDBC data cartridge as follows:

1. Declare and define an Oracle Event Processing JDBC cartridge application-scoped context.

   For more information, see Section , "Oracle Event Processing JDBC Data Cartridge Application Context".

2. Define one or more SQL statements in the `jc:jdbc-ctx` element in the component configuration file.

   For more information, see Section , "Defining SQL Statements: function Element."

3. If you specify the `function` element `return-component-type` child element as a Java bean, implement the bean and ensure that the class is on your Oracle Event Processing application classpath.

   Example 17–5 shows a typical implementation.

Example 17–5 Example return-component-type Class

```
package com.oracle.cep.example.jdbc_cartridge;

public class RetEvent
{
    public String employeeName;
    public String employeeEmail;
    public String description;

    /* Default constructor is mandatory */
    public RetEvent1() {}

    /* May contain getters and setters for the fields */

    public String getEmployeeName() {
        return this.employeeName;
    }

    public void setEmployeeName(String employeeName) {
        this.employeeName = employeeName;
    }

    ...
```

```
    /* May contain other helper methods */

    public int getEmployeeNameLength() {
        return employeeName.length();
    }
}
```

You must declare the fields as public.

The return-component-type class for a JDBC cartridge context function must have a one-to-one mapping for fields in the SELECT list of the SQL query that defines the function. In other words, every field in the SELECT list of the SQL query defining a function must have a corresponding field (matching name) in the Java class that is declared to be the return-component-type for that function; otherwise Oracle Event Processing throws an error. For example, note how the SELECT items in the function in Example 17–4 match the field names in Example 17–5.

For more information, see:

- Section , "return-component-type"
- "Oracle Event Processing IDE for Eclipse Projects" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*

4. Define one or more Oracle CQL queries that call the SQL statements defined in the `jc:jdbc-ctx` element using the Oracle CQL `TABLE` clause and access the returned results by SQL `SELECT` list alias names.

   For more information, see Section , "Defining Oracle CQL Queries With the Oracle Event Processing JDBC Data Cartridge."

Defining SQL Statements: function Element

Within the `jc:jdbc-cxt` element in the component configuration file, you can define a JDBC cartridge context function using the `function` child element as Example 17–6 shows.

Example 17–6 Oracle Event Processing JDBC Data Cartridge SQL Statement

```
...
<jc:jdbc-ctx>
    <name>JdbcCartridgeOne</name>
    <data-source>StockDS</data-source>
    <function name="getDetailsByOrderIdName">
        <param name="inpOrderId" type="int" />
        <param name="inpName" type="char" />
        <return-component-type>
            com.oracle.cep.example.jdbc_cartridge.RetEvent
        </return-component-type>
        <sql><![CDATA[
            SELECT
                Employee.empName as employeeName,
                Employee.empEmail as employeeEmail,
                OrderDetails.description as description
            FROM
                 PlacedOrders, OrderDetails , Employee
            WHERE
                PlacedOrders.empId = Employee.empId AND
                PlacedOrders.orderId = OrderDetails.orderId AND
                Employee.empName = :inpName AND
                PlacedOrders.orderId = :inpOrderId
        ]]></sql>
    </function>
```

```
</jc:jdbc-ctx>
...
```

You may define one or more `function` elements within a given `jc:jdbc-cxt` element.

This section describes:

- Section , "function Element Attributes"
- Section , "function Element Child Elements"
- Section , "function Element Usage"

function Element Attributes

Each `function` element supports the attributes that Table 17–1 lists.

Table 17–1 function Element Attributes

Attribute	Description
`name`	The name of the JDBC cartridge context function. The combination of name and signature must be unique within a given Oracle Event Processing JDBC data cartridge application context. For more information, see Section , "Overloading JDBC Cartridge Context Functions".

function Element Child Elements

Each `function` element supports the following child elements:

- `param`
- `return-component-type`
- `sql`

param The `param` child element specifies an optional input parameter.

The SQL statement may take zero or more parameters. Each parameter is defined in a `param` element.

The `param` child element supports the attributes that Table 17–2 lists.

Table 17–2 param Element Attributes

Attribute	Description
`name`	The name of the input parameter. A valid parameter name is formed by a combination of A-Z,a-z,0-9 and _ (underscore).
`type`	The data type of the parameter.

Datatype Support – You may specify only Oracle CQL native `com.bea.wlevs.ede.api.Type` data types for the input parameter `param` element `type` attribute.

> **Note:** Datatype names are case sensitive. Use the case that the `com.bea.wlevs.ede.api.Type` class specifies.

For more information, see Table 17–3.

return-component-type The `return-component-type` child element specifies the return type of the function. This child element is mandatory.

This represents the component type of the collection type returned by the JDBC data cartridge function. Because the function is always called from within an Oracle CQL `TABLE` clause, it always returns a collection type.

For more information, see Section , "Using the TABLE Clause."

Datatype Support – You may specify any one of the following types as the value of the `return-component-type` element:

- Oracle CQL native `com.bea.wlevs.ede.api.Type` datatype.
- Oracle CQL extensible Java cartridge type, such as a Java bean.

For more information, see:

- Table 17–3
- Chapter 15, "Oracle Java Data Cartridge"

sql The `sql` child element specifies a SQL statement. This child element is mandatory.

Each `function` element may contain one and only one, single-line, SQL statement. You define the SQL statement itself within a `<![CDATA[]]>` block.

Within the SQL statement, you specify input parameters by `param` element `name` attribute using a colon (`:`) prefix as shown in Example 17–6.

Note: You must provide alias names for every `SELECT` list column in the JDBC cartridge context function.

Datatype Support – Table 17–3 lists the SQL types you may use in your Oracle Event Processing JDBC data cartridge context functions and their corresponding Oracle Event Processing Java type and `com.bea.wlevs.ede.api.Type` type.

Table 17–3 SQL Column Types and Oracle Event Processing Type Equivalents

SQL Type	Oracle Event Processing Java Type	com.bea.wlevs.ede.api.Type
NUMBER	java.math.BigDecimal	bigdecimal
NUMBER	long	bigint
RAW	byte[]	byte
CHAR, VARCHAR	java.lang.String	char
NUMBER	double	double
FLOAT, NUMBER	float	float
INTEGER, NUMBER	int	int
TIMESTAMP	java.sql.Timestamp	timestamp

Note: In cases where the size of the Java type exceeds that of the SQL type, your Oracle Event Processing application must restrict values to the maximum size of the SQL type. The choice of type to use on the CQL side should be driven by the range of values in the database column. For example, if the SQL column is a number that contains values in the range of integer, use the "int" type on CQL side. If you choose an incorrect type and encounter out-of-range values, Oracle Event Processing throws a numeric overflow error.

Note: The Oracle Event Processing JDBC data cartridge does not support Oracle Spatial data types.

For more information, see Section , "function Element Usage."

function Element Usage

This section provides examples of different JDBC cartridge context functions you can define using the Oracle Event Processing JDBC data cartridge, including:

- Section , "Multiple Parameter JDBC Cartridge Context Functions"
- Section , "Invoking PL/SQL Functions"
- Section , "Complex JDBC Cartridge Context Functions"
- Section , "Overloading JDBC Cartridge Context Functions"

Multiple Parameter JDBC Cartridge Context Functions Using the Oracle Event Processing JDBC data cartridge, you can define JDBC cartridge context functions that take multiple input parameters.

Example 17–7 shows an Oracle Event Processing JDBC data cartridge application context that defines an JDBC cartridge context function that takes two input parameters.

Example 17–7 Oracle JDBC Data Cartridge Context Functions With Multiple Parameters

```
...
<function name="getDetailsByOrderIdName">
    <param name="inpOrderId" type="int" />
    <param name="inpName" type="char" />
    <return-component-type>
        com.oracle.cep.example.jdbc_cartridge.RetEvent
    </return-component-type>
    <sql><![CDATA[
              SELECT
                    Employee.empName as employeeName,
                    Employee.empEmail as employeeEmail,
                    OrderDetails.description as description
              FROM
                     PlacedOrders, OrderDetails , Employee
              WHERE
                    PlacedOrders.empId = Employee.empId AND
                    PlacedOrders.orderId = OrderDetails.orderId AND
                    Employee.empName = :inpName AND
                    PlacedOrders.orderId = :inpOrderId
    ]]></sql>
</function>
...
```

Invoking PL/SQL Functions Using the Oracle Event Processing JDBC data cartridge, you can define JDBC cartridge context functions that invoke PL/SQL functions that the database defines.

Example 17–8 shows an Oracle Event Processing JDBC data cartridge application context that defines a JDBC cartridge context function that invokes PL/SQL function `getOrderAmt`.

Example 17–8 Oracle JDBC Data Cartridge Context Function Invoking PL/SQL Functions

```
...
<function name="getOrderAmount">
    <param name="inpId" type="int" />
    <return-component-type>
        com.oracle.cep.example.jdbc_cartridge.RetEvent
    </return-component-type>
    <sql><![CDATA[
            SELECT getOrderAmt(:inpId) as orderAmt
            FROM dual
    ]]></sql>
</function>
...
```

Complex JDBC Cartridge Context Functions Using the Oracle Event Processing JDBC data cartridge, you can define arbitrarily complex JDBC cartridge context functions including subqueries, aggregation, `GROUP BY`, `ORDER BY`, and `HAVING`.

Example 17–9 shows an Oracle Event Processing JDBC data cartridge application context that defines a complex JDBC cartridge context function.

Example 17–9 Oracle Event Processing JDBC Data Cartridge Complex JDBC Cartridge Context Function

```
...
<function name="getHighValueOrdersPerEmp">
    <param name="limit" type="int"/>
    <param name="inpName" type="char"/>
    <return-component-type>
        com.oracle.cep.example.jdbc_cartridge.RetEvent
    </return-component-type>
        <sql><![CDATA[
            select description as description,  sum(amt) as totalamt, count(*) as numTimes
            from  OrderDetails
                where orderid in (
                    select orderid from PlacedOrders where empid in (
                        select empid from Employee where  empName = :inpName
                    )
                )
            group by description
            having sum(amt) > :limit
        ]]></sql>
</function>
...
```

Overloading JDBC Cartridge Context Functions Using the Oracle Event Processing JDBC data cartridge, you can define JDBC cartridge context functions with the same name in the same application context provided that each function has a unique signature.

Example 17–10 shows an Oracle Event Processing JDBC data cartridge application context that defines two JDBC cartridge context functions named `getDetails`. Each function is distinguished by a unique signature.

Example 17–10 Oracle JDBC Data Cartridge Context Function Overloading

```
<jc:jdbc-ctx>
    <name>JdbcCartridgeOne</name>
    <data-source>StockDS</data-source>
         <function name="getDetails">
            <param name="inpName" type="char" />
            <return-component-type>
                com.oracle.cep.example.jdbc_cartridge.RetEvent
            </return-component-type>
            <sql><![CDATA[
                      SELECT
                             Employee.empName as employeeName,
                             Employee.empEmail as employeeEmail,
                             OrderDetails.description as description
                      FROM
                             PlacedOrders, OrderDetails , Employee
                      WHERE
                             PlacedOrders.empId = Employee.empId AND
                             PlacedOrders.orderId = OrderDetails.orderId AND
                             Employee.empName=:inpName
                      ORDER BY <!—SQL query using ORDER BY -->
                        description desc
            ]]></sql>
        </function>
        <function name="getDetails">
            <param name="inpOrderId" type="int" />
            <sql><![CDATA[
                      SELECT
                             Employee.empName as employeeName,
                             Employee.empEmail as employeeEmail,
                             OrderDetails.description as description
                      FROM
                             PlacedOrders, OrderDetails , Employee
                      WHERE
                             PlacedOrders.empId= Employee.empId AND
                             PlacedOrders.orderId = OrderDetails.orderId AND
                             PlacedOrders.orderId = :inpOrderId
            ]]></sql>
        </function>
</jc:jdbc-ctx>
```

Defining Oracle CQL Queries With the Oracle Event Processing JDBC Data Cartridge

This section describes how to define Oracle CQL queries that invoke SQL statements using the Oracle Event Processing JDBC data cartridge, including:

- Section , "Using SELECT List Aliases"
- Section , "Using the TABLE Clause"
- Section , "Using a Native CQL Type as a return-component-type"

For more information, see "Querying an Event Stream with Oracle CQL" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

Using SELECT List Aliases

Consider the Oracle Event Processing JDBC data cartridge context function that Example 17–11 shows.

Example 17–11 Oracle Event Processing JDBC Data Cartridge Context Function

```
<jc:jdbc-ctx>
    <name>JdbcCartridgeOne</name>
    <data-source>StockDS</data-source>
```

```
        <function name="getDetailsByOrderIdName">
            <param name="inpOrderId" type="int" />
            <param name="inpName" type="char" />
            <return-component-type>
                com.oracle.cep.example.jdbc_cartridge.RetEvent
            </return-component-type>
            <sql><![CDATA[
                    SELECT
                            Employee.empName as employeeName,
                            Employee.empEmail as employeeEmail,
                            OrderDetails.description as description
                    FROM
                            PlacedOrders, OrderDetails , Employee
                    WHERE
                            PlacedOrders.empId = Employee.empId AND
                            PlacedOrders.orderId = OrderDetails.orderId AND
                            Employee.empName = :inpName AND
                            PlacedOrders.orderId = :inpOrderId
            ]]></sql>
        </function>
</jc:jdbc-ctx>
```

You must assign an alias to each column in the `SELECT` list. When you invoke the JDBC cartridge context function in an Oracle CQL query, you access the columns in the result set by their SQL `SELECT` list aliases.

For more information, see Section , "Using the TABLE Clause".

Using the TABLE Clause

Consider the Oracle Event Processing JDBC data cartridge SQL statement that Example 17–12 shows.

Example 17–12 Oracle Event Processing JDBC Data Cartridge SQL Statement

```
...
<jc:jdbc-ctx>
    <name>JdbcCartridgeOne</name>
    <data-source>StockDS</data-source>
    <function name="getDetailsByOrderIdName">
        <param name="inpOrderId" type="int" />
        <param name="inpName" type="char" />
        <return-component-type>
            com.oracle.cep.example.jdbc_cartridge.RetEvent
        </return-component-type>
        <sql><![CDATA[
            SELECT
                Employee.empName as employeeName,
                Employee.empEmail as employeeEmail,
                OrderDetails.description as description
            FROM
                 PlacedOrders, OrderDetails , Employee
            WHERE
                PlacedOrders.empId = Employee.empId AND
                PlacedOrders.orderId = OrderDetails.orderId AND
                Employee.empName = :inpName AND
                PlacedOrders.orderId = :inpOrderId
        ]]></sql>
    </function>
</jc:jdbc-ctx>
...
```

The Oracle CQL query in Example 17–13 invokes the JDBC cartridge context function that Example 17–12 defines.

Example 17–13 Oracle CQL Query Invoking an Oracle Event Processing JDBC Data Cartridge Context Function

```
<processor>
        <name>Proc</name>
        <rules>
            <query id="q1"><![CDATA[
                RStream(
                   select
                     currentOrder.orderId,
                     details.orderInfo.employeeName,
                     details.orderInfo.employeeemail,
                     details.orderInfo.description
                     details.orderInfo.getEmployeeNameLength()
                   from
                     OrderArrival[now] as currentOrder,
                     TABLE(getDetailsByOrderIdName@JdbcCartridgeOne(
                               currentOrder.orderId, currentOrder.empName
                           ) as orderInfo
                     ) as details
                )
            ]]></query>
        </rules>
</processor>
```

You must wrap the Oracle Event Processing JDBC data cartridge context function invocation in an Oracle CQL query `TABLE` clause.

You access the result set using:

```
TABLE_CLAUSE_ALIAS.JDBC_CARTRIDGE_FUNCTION_ALIAS.SQL_SELECT_LIST_ALIAS
or
TABLE_CLAUSE_ALIAS.JDBC_CARTRIDGE_FUNCTION_ALIAS.METHOD_NAME
```

Where:

- *`TABLE_CLAUSE_ALIAS`*: the outer `AS` alias of the `TABLE` clause.

  In Example 17–13, `details`.

- *`JDBC_CARTRIDGE_FUNCTION_ALIAS`*: the inner `AS` alias of the JDBC cartridge context function.

  In Example 17–13, `orderInfo`.

- *`SQL_SELECT_LIST_ALIAS`*: the JDBC cartridge context function `SELECT` list alias.

  In Example 17–12, `employeeName`, `employeeEmail`, and `description`.

- *`METHOD_NAME`*: the name of the method that the `return-component-type` class provides.

  In Example 17–13, `getEmployeeNameLength()`.

As Example 17–13 shows, you access the JDBC cartridge context function result set in the Oracle CQL query using:

```
details.orderInfo.employeeName
details.orderInfo.employeeemail
details.orderInfo.description
details.orderInfo.getEmployeeNameLength()
```

The component type of the collection type returned by the JDBC data cartridge function is defined by the `function` element `return-component-type` child element. Because the function is always called from within an Oracle CQL `TABLE` clause, it always returns a collection type. If the `getDetailsByORderIdName` JDBC cartridge context function called in Example 17–13 is defined as Example 17–12 shows, then orderInfo is of type com.oracle.cep.example.jdbc_cartridge.RetEvent.

You can access both fields and methods of the return-component-type in an Oracle CQL query. In Example 17–12, the return-component-type specifies a Java bean implemented as Example 17–14 shows.

Example 17–14 Example return-component-type Class

```
package com.oracle.cep.example.jdbc_cartridge;

public class RetEvent
{
    String employeeName;
    String employeeEmail;
    String description;

    /* Default constructor is mandatory */
    public RetEvent1() {}

    /* May contain getters and setters for the fields */

    public String getEmployeeName() {
        return this.employeeName;
    }

    public void setEmployeeName(String employeeName) {
        this.employeeName = employeeName;
    }

    ...

    /* May contain other helper methods */

    public int getEmployeeNameLength() {
        return employeeName.length();
    }
}
```

This class provides helper methods, like getEmployeeNameLength, that you can invoke within the Oracle CQL query.

For more information, see:

- Section , "return-component-type"
- Section , "Function TABLE Query"
- "table_clause" on page 22-10

Using a Native CQL Type as a return-component-type

Following is a JDBC cartridge context that defines a function that has a native CQL type bigint as return-component-type.

Example 17–15 CQL Type bigint as a return-component-type

```
<jc:jdbc-ctx>
    <name>JdbcCartridgeOne</name>
    <data-source>myJdbcDataSource</data-source>
    <function name="getOrderAmt">
       <param name="inpId" type="int" />
       <return-component-type>bigint</return-component-type> <!-- native CQL as
return component type -->
       <sql><![CDATA[
                SELECT
```

```
                  getOrderAmt(:inpId) as orderAmt
                FROM (select :inpId as iid from
                  dual)]]>
       </sql>
    </function>
</jc:jdbc-ctx>
```

Example 17–16 shows how the `getOrderAmt` function in Example 17–15 can be used in a CQL query.

Example 17–16 getOrderAmt Function in a CQL Query

```
<query id="q1"><![CDATA[
                 RStream(
                   select
                     currentOrder.orderId,
                     details.orderInfo as orderAmt
                   from
                     OrderArrival[now] as currentOrder,
                     TABLE(getOrderAmt@JdbcCartridgeTwo(currentOrder.orderId) as
orderInfo of bigint) as details
                 )
 ]]></query>
```

Note that the alias `orderInfo` itself is of type `bigint` and can be accessed as `details.orderInfo as orderAmt` in the select list of the CQL query.

The "of bigint" clause used inside the TABLE construct is optional. If specified, the type mentioned should match the return-component-type, which is `bigint` in Example 17–15.

18

Oracle Event Processing Hadoop Data Cartridge

This chapter describes how to use the Oracle Hadoop Data Cartridge to integrate a Hadoop system into an Oracle Event Processing event processing network and query data from files on the system.

Hadoop is an open source technology that provides access to large data sets that are distributed across clusters. One strength of the Hadoop software is that it provides access to large quantities of data not stored in a relational database.

The content in this guide assumes that you are already familiar with, and likely running, a Hadoop system. If you need more information about Hadoop, you might start with the Hadoop project web site at `http://hadoop.apache.org/`.

> **Note:** Hadoop functionality in Oracle Event Processing applications is supported only on Unix-based environments.

This chapter includes the following sections:

- Understanding the Oracle Event Processing Hadoop Data Cartridge
- Using Hadoop Data Sources in Oracle CQL

Understanding the Oracle Event Processing Hadoop Data Cartridge

You can use the Hadoop data cartridge to integrate an existing Hadoop data source into an event processing network. With the data source integrated, you can write Oracle CQL query code that incorporates data from files on the Hadoop system.

When integrating a Hadoop system, keep the following guidelines in mind:

- The Hadoop cluster must have been started through its own mechanism and must be accessible. The cluster is not managed directly by Oracle Event Processing.
- A file from a Hadoop system supports only joins using a single key in Oracle CQL. However, any property of the associated event type may be used as key. In other words, with the exception of a key whose type is byte array, you can use keys whose type is other than a String type.
- Joins must use the equals operator. Other operators are not supported in a join condition.
- For the event type you define to represent data from the Hadoop file, only tuple-based event types are supported.

- The order of properties in the event type specification must match the order of fields in the Hadoop file.
- Only the following Oracle CQL to Hadoop types are supported. Any other type will cause a configuration exception to be raised.

Table 18–1 Mapping Between Datatypes for Oracle CQL and Hadoop

Oracle CQL Datatype	Hadoop Datatype
`int`	`int`
`bigint`	`long`
`float`	`float`
`double`	`double`
`char`	`chararray`
`java.lang.String`	`chararray`
`byte`	`bytearray`

Usage Scenario: Using Purchase Data to Develop Buying Incentives

To understand how a Hadoop data source might be used with an Oracle Event Processing application, consider a scenario with an application that requires quick access to a very large amount of customer purchase data in real time.

In this case, the data stored in Hadoop includes all purchases by all customers from all stores. Values in the data include customer identifiers, store identifiers, product identifiers, and so on. The purchase data includes information about which products are selling best in each store. To render the data to a manageable state, a MapReduce function is used to examine the data and produce a list of top buyers (those to whom incentives will be sent).

This data is collected and managed by a mobile application vendor as part of a service designed to send product recommendations and incentives (including coupons) to customers. The data is collected from multiple retailers and maintained separately for each retailer.

The Oracle Event Processing application provides the middle-tier logic for a client-side mobile application that is designed to offer purchase incentives to top buyers. It works in the following way:

1. Retailers arrange with the mobile application vendor to provide purchase data as part of a program to offer incentives to top buyers. The data, regularly refreshed from store sales data, is stored in a Hadoop system and a MapReduce function is used to identify top buyers.
2. The mobile application vendor provides the application for download, noting which retailers support the program.
3. App users each create a user ID that is correlated by the app vendor to data about customers from the retailers.
4. The mobile application is designed to send location data to the Oracle Event Processing application, along with the user ID. This information -- location coordinates and user ID -- forms the event data received by the Oracle Event Processing application.
5. As the Oracle Event Processing application receives event data from the mobile application, it uses Oracle CQL queries to:

- Determine whether the user is near a store from a participating retailer.
- Establish (from Hadoop-based data) whether the user is a top buyer for the retailer.
- Locate purchase information related to that user as a buyer from that retailer.
- If the user is a top buyer, the application correlates products previously purchased with incentives currently being offered to buyers of those products.

6. The Oracle Event Processing application pushes an incentive announcement to the user.

Data Cartridge Name

The Oracle Event Processing Hadoop cartridge uses the cartridge ID `com.oracle.cep.cartridge.hadoop`.

Using Hadoop Data Sources in Oracle CQL

You use the Hadoop support included with Oracle Event Processing by integrating a file in an existing Hadoop system into an event processing network. With the file integrated, you have access to data in the file from Oracle CQL code.

This section describes the following:

- Section , "Configuring Integration of Oracle Event Processing and Hadoop"
- Section , "Integrating a File from a Hadoop System Into an EPN"
- Section , "Using Hadoop Data in Oracle CQL"

Configuring Integration of Oracle Event Processing and Hadoop

In order to use Hadoop from Oracle Event Processing, you must first make configuration changes on both the Oracle Event Processing and Hadoop servers:

- On the Oracle Event Processing server, add the following Hadoop configuration files at the server's bootclasspath: core-site.xml, hdfs.xml, and mapred.xml.
- To the Hadoop server, copy the Pig JAR file to the lib directory and include it as part of the HADOOP_CLASSPATH defined in the hadoop-env.sh file.

Note: A connection with a Hadoop data source through the cartridge might require many input/output operations, such that undeploying the application can time out or generate errors that prevent the application from being deployed again. Before undeploying an application that uses a Hadoop cartridge, be sure to discontinue event flow into the application.

Integrating a File from a Hadoop System Into an EPN

Integrating a file from an existing Hadoop system is similar to the way you might integrate a table from an existing relational database. For a Hadoop file, you use the `file` XML element from the Oracle Event Processing schema specifically added for Hadoop support.

The file element is from the http://www.oracle.com/ns/ocep/hadoop namespace. So your EPN assembly file needs to reference that namespace. The `file` element includes the following attributes:

- `id` -- Uniquely identifies the file in the EPN. You will use this attribute's value to reference the data source in a processor.
- `event-type` -- A reference to the event-type to which data from the file should be bound. The event-type must be defined in the EPN.
- `path` -- The path to the file in the Hadoop file system.
- `separator` -- Optional. The character delimiter to use when parsing the lines in the Hadoop file into separate fields. The default delimiter is the comma (',') character.
- `operation-timeout` -- Optional. The maximum amount of time, in milliseconds, to wait for the operation to complete.

With the Hadoop file to integrate specified with the file element, you use the table-source element to add the file as a data source for the Oracle CQL processor in which you will be using the file's data.

In Example 18–1, "EPN Integrating a File from Hadoop", note that the `http://www.oracle.com/ns/ocep/hadoop` namespace (and `hadoop` prefix) is referenced in the beans element. The `file` element references a CustomerDescription.txt file for data, along with a `CustomerDescription` event type defined in the event type repository.

Example 18–1 EPN Integrating a File from Hadoop

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:wlevs="http://www.bea.com/ns/wlevs/spring"
    xmlns:hadoop="http://www.oracle.com/ns/ocep/hadoop"
    xsi:schemaLocation="
        http://www.bea.com/ns/wlevs/spring
        http://www.bea.com/ns/wlevs/spring/spring-wlevs-v11_1_1_6.xsd
        http://www.oracle.com/ns/ocep/hadoop
        http://www.oracle.com/ns/ocep/hadoop/ocep-hadoop.xsd">
<!-- Some schema references omitted for brevity. -->

    <!-- Event types that will be used in the query. -->
    <wlevs:event-type-repository>
        <wlevs:event-type type-name="SalesEvent">
            <wlevs:class>com.bea.wlevs.example.SalesEvent</wlevs:class>
        </wlevs:event-type>
        <wlevs:event-type type-name="CustomerDescription">
            <wlevs:properties>
                <wlevs:property name="userId" type="char"/>
                <wlevs:property name="creditScore" type="int"/>
                <wlevs:property name="address" type="char"/>
                <wlevs:property name="customerName" type="char"/>
            </wlevs:properties>
        </wlevs:event-type>
    </wlevs:event-type-repository>

    <!-- Input adapter omitted for brevity. -->

    <!-- Channel sending SalesEvent instances to the processor. -->
    <wlevs:channel id="S1" event-type="SalesEvent" >
        <wlevs:listener ref="P1"/>
    </wlevs:channel>

    <!-- The file element to integrate CustomerDescription.text file from
        the Hadoop system into the EPN. -->
    <hadoop:file id="CustomerDescription" event-type="CustomerDescription"
```

```
        path="CustomerDescription.txt" />

    <!-- The file from the Hadoop system tied into the query processor
        with the table-source element. -->
    <wlevs:processor id="P1">
        <wlevs:table-source ref="CustomerDescription" />
    </wlevs:processor>

    <!-- Other stages omitted for brevity. -->

</beans>
```

Using Hadoop Data in Oracle CQL

After you have integrated a Hadoop file into an event processing network, you can query the file from Oracle CQL code.

Example 18–1, "EPN Integrating a File from Hadoop" illustrates how you can add a file from a Hadoop system into an EPN. With the file added to the EPN, you can query it from Oracle CQL code, as shown in Example 18–2, "Oracle CQL Query Using Data from a Hadoop File".

In the following example, the processor receives SalesEvent instances from a channel, but also has access to a file in the Hadoop system as CustomerDescription instances. The Hadoop file is essentially a CSV file that lists customers. Both event types have a `userId` property.

Example 18–2 Oracle CQL Query Using Data from a Hadoop File

```
<n1:config
    xsi:schemaLocation="http://www.bea.com/ns/wlevs/config/application wlevs_
application_config.xsd"
    xmlns:n1="http://www.bea.com/ns/wlevs/config/application"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <processor>
        <name>P1</name>
        <rules>
            <query id="q1"><![CDATA[
                SELECT customerName, creditScore, price, item
                FROM S1 [Now], CustomerDescription as cust
                WHERE S1.userId = cust.userId
                AND S1.price > 1000
            ]]></query>
        </rules>
    </processor>
</n1:config>
```

19

Oracle Event Processing NoSQL Database Data Cartridge

This chapter describes how to use the Oracle NoSQL Data Cartridge to integrate an Oracle NoSQL database into an Oracle Event Processing event processing network and query data from the database.

The Oracle NoSQL Database is a distributed key-value database. In it, data is stored as key-value pairs, which are written to particular storage node(s). Storage nodes are replicated to ensure high availability, rapid failover in the event of a node failure and optimal load balancing of queries.

The content in this guide assumes that you are already familiar with, and likely running, an Oracle NoSQL database. If you need more information about Oracle NoSQL, be sure to see its Oracle Technology Network page at `http://www.oracle.com/technetwork/products/nosqldb/`.

> **Note:** To use the NoSQL Data Cartridge, you must have a license for NoSQL Enterprise Edition.

This chapter includes the following sections:

Understanding the Oracle Event Processing NoSQL Database Data Cartridge

You can use the Oracle Event Processing NoSQL Database data cartridge to refer to data stored in Oracle NoSQL Database as part of an Oracle CQL query. The cartridge makes it possible for queries to retrieve values from an Oracle NoSQL Database store by specifying a key in the query and then referring to fields of the value associated with the key.

When integrating an Oracle NoSQL database, keep the following guidelines in mind:

- The NoSQL database must have been started through its own mechanisms and must be accessible. It is not managed directly by Oracle Event Processing.
- This release of the cartridge provides access to the database using release 1.2.123 of the Oracle NoSQL Database API.
- The property used as a key in queries must be of type `String`. Joins can use a single key only.

- Joins must use the equals operator. Other operators are not supported in a join condition.
- Run-away queries involving the NoSQL database are not supported.

Data Cartridge Name

The Oracle Event Processing NoSQL cartridge uses the cartridge ID `com.oracle.cep.cartridge.nosqldb`.

Using a NoSQL Database in Oracle CQL

To use the Oracle Event Processing NoSQL Database data cartridge in a CQL application, you must declare and configure it in one or more application-scoped cartridge contexts for the application.

Integrating a NoSQL Database Into an EPN

Integrating an existing NoSQL database is similar to the way you might integrate a table from a relational database. For a NoSQL database, you update the EPN assembly file in the following ways (see Example 19–1, "EPN integrating a NoSQL Database"):

1. Add namespace declarations to support for the `store` element for referencing the NoSQL data source.

   Your changes should add a namespace schema location to the schemaLocation attribute, along with a namespace and prefix declaration:

   - http://www.oracle.com/ns/oep/nosqldb http://www.oracle.com/ns/oep/nosqldb/oep-nosqldb.xsd
   - xmlns:nosqldb="http://www.oracle.com/ns/oep/nosqldb"

2. Add the `store` element to integrate the NoSQL database into the event processing network as a relation source.

   The store element supports the following attributes, all of which are required:

   - `id` -- The name that will be used to refer to the key-value store in CQL queries.
   - `store-name` -- The name of the key-value store, which should match the name specified in the KVStoreConfig class when creating the store.
   - `store-locations` -- One or more host names and ports of active nodes in the store. The attribute value is a space-separated list in which each entry is formatted as "hostname:port". Nodes with the specified host name and port values will be contacted in order when connecting to the store initially.
   - `event-type` -- The object type for all objects retrieved for this relation from values in the store. The attribute value should correspond to the name of a wlevs:event-type entry specified in a wlevs:event-type-repository entry.

3. Add a `table-source` element to connect the NoSQL database to the processor in which queries will be executed.

Example 19–1, "EPN integrating a NoSQL Database" illustrates how you can connect an event processing network to a NoSQL database. The `store` element provides access to a store named "kvstore-customers", using port 5000 on host kvhost-alpha or port 5010 on host kvhost-beta to make the initial connection. It defines Oracle CQL processor P1 and makes the data in the key-value store available to it as a relation named "CustomerDescription".

The store can be referred to within Oracle CQL queries using the name "CustomerDescription". All values retrieved from the store should be serialized instances of the CustomerDescription class.

Example 19–1 EPN integrating a NoSQL Database

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:osgi="http://www.springframework.org/schema/osgi"
       xmlns:wlevs="http://www.bea.com/ns/wlevs/spring"
       xmlns:nosqldb="http://www.oracle.com/ns/oep/nosqldb"
       xsi:schemaLocation="
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/osgi
  http://www.springframework.org/schema/osgi/spring-osgi.xsd
  http://www.bea.com/ns/wlevs/spring
  http://www.bea.com/ns/wlevs/spring/ocep-epn.xsd
  http://www.oracle.com/ns/oep/nosqldb
  http://www.oracle.com/ns/oep/nosqldb/oep-nosqldb.xsd">

 <!-- Provide access to the CustomerDescription class, which represents
     the type of values in the store. -->
 <wlevs:event-type-repository>
     <wlevs:event-type type-name="CustomerDescription">
         <wlevs:class>com.bea.wlevs.example.CustomerDescription</wlevs:class>
     </wlevs:event-type>
     <wlevs:event-type type-name="SalesEvent">
         <wlevs:class>com.bea.wlevs.example.SalesEvent</wlevs:class>
     </wlevs:event-type>
 </wlevs:event-type-repository>

 <!-- The store element declares the key-value store, along with the
     event type to which incoming NoSQL data will be bound. -->
 <nosqldb:store store-name="kvstore-customers"
     store-locations="kvhost-alpha:5000, kvhost-beta:5010"
     id="CustomerDescription"
     event-type="CustomerDescription"/>


 <wlevs:channel id="S1" event-type="SalesEvent">
     <wlevs:listener ref="P1"/>
 </wlevs:channel>

 <!- The table-source element links the store to the CQL processor. -->
 <wlevs:processor id="P1">
     <wlevs:table-source ref="CustomerDescription" />
 </wlevs:processor>

</beans>
```

The event types defined in Example 19–1, "EPN integrating a NoSQL Database" are implemented as JavaBeans classes as described in "Defining and Using Event Types" in Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse.

If Oracle CQL queries refer to entries in a store specified by a `store` element, then the values of those entries must be serialized instances of the type specified by the event-type attribute. The event type class must implement `java.io.Serializable`.

If a query retrieves a value from the store that is not a valid serialized form, or if the value is not the serialized form for the specified class, then Oracle Event Processing throws an exception and event processing is halted. You can declare multiple `store` elements to return values of different types from the same or different stores.

Using NoSQL Data in Oracle CQL

After you have integrated a NoSQL database into an event processing network, you can access data from Oracle CQL code. The query can look up an entry from the store by specifying an equality relation in the query's `WHERE` clause.

In the Oracle CQL code shown in Example 19–2, "Oracle CQL Query Using Data from a NoSQL Database", the `CustomerDescription` in the `FROM` clause corresponds to the `id` attribute value in the store element shown in Example 19–1, "EPN integrating a NoSQL Database".

Example 19–2 Oracle CQL Query Using Data from a NoSQL Database

```
<n1:config
    xsi:schemaLocation="http://www.bea.com/ns/wlevs/config/application wlevs_
application_config.xsd"
    xmlns:n1="http://www.bea.com/ns/wlevs/config/application"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <processor>
        <name>P1</name>
        <rules>
            <query id="q1"><![CDATA[
                SELECT customerName, creditScore, price, item
                FROM S1 [Now], CustomerDescription as cust
                WHERE S1.userId = cust.userId
                AND creditScore 5
            ]]></query>
        </rules>
    </processor>
</n1:config>
```

In this example, the event type instances representing data from the S1 channel and CustomerDescription NoSQL data source are both implemented as JavaBeans classes as described in "Defining and Using Event Types" in Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse. Because both event types are JavaBeans classes, the Oracle CQL query can access the customer description associated with a particular event by equating the event's user ID with that of the customer description in the `WHERE` clause, treating both as JavaBeans properties:

```
WHERE S1.userId = CustomerDescription.userId
```

This clause requests that an entry be retrieved from the store that has the key specified by the value of the event's `userId` field. Only equality relations are supported for obtaining entries from the store.

Once an entry from the store has been selected, fields from the value retrieved from the store can be referred to in the `SELECT` portion of the query or in additional clauses in the `WHERE` clause.

The `creditScore` value specified in the `SELECT` clause will include the value of the `creditScore` field of the `CustomerDescription` object retrieved from the store in the query output. The reference to `creditScore` in the `WHERE` clause will also further restrict the query to events where the value of the `CustomerDescription` `creditScore` field is greater than 5.

Formatting the Key Used to Obtain Entries from the NoSQL Store

The key used to obtain entries from the store can be formatted in one of two ways: by beginning the value with a forward slash ('/') or by omitting a slash.

If the value specified on the left hand side of the equality relation starts with a forward slash, then the key is treated as a full key path that specifies one or more major components, as well as minor components if desired. For more details on the syntax of key paths, see the information about the `oracle.kv.Key` class in the Oracle NoSQL Database API documentation at
http://docs.oracle.com/cd/NOSQL/html/javadoc/index.html.

For example, if the `userId` field of a `SalesEvent` object has the value "/users/user42/-/custDesc", then that value will be treated as a full key path that specifies "users" as the first major component, the user ID "user42" as the second major component, and a minor component named "custDesc".

As a convenience, if the value specified on the left hand side of the equality relation does not start with a forward slash, then it is treated as a single major component that comprises the entire key.

Note that keys used to retrieve entries from the store must be specified in full by a single field accessed by the Oracle CQL query. In particular, if a key path with multiple components is required to access entries in the key-value store, then the full key path expression must be stored in a single field that is accessed by the query.

Part IV

Using Oracle CQL

This part contains the following chapters:

20

Oracle CQL Queries, Views, and Joins

This chapter provides reference and usage guidelines for queries, views, and joins in Oracle Continuous Query Language (Oracle CQL). You select, process, and filter element data from streams and relations using Oracle CQL queries and views.

A top-level `SELECT` statement that you create using the `QUERY` statement is called a **query**.

A top-level `VIEW` statement that you create using the `VIEW` statement is called a **view** (the Oracle CQL equivalent of a subquery).

A **join** is a query that combines rows from two or more streams, views, or relations.

This chapter includes the following sections:

- Introduction to Oracle CQL Queries, Views, and Joins
- Queries
- Views
- Joins
- Oracle CQL Queries and the Oracle Event Processing Server Cache
- Oracle CQL Queries and Relational Database Tables
- Oracle CQL Queries and Oracle Data Cartridges

For more information, see:

- Section , "Lexical Conventions"
- Section , "Documentation Conventions"
- Chapter 2, "Basic Elements of Oracle CQL"
- Chapter 7, "Common Oracle CQL DDL Clauses"
- Chapter 22, "Oracle CQL Statements"

Introduction to Oracle CQL Queries, Views, and Joins

An Oracle CQL query is an operation that you express in Oracle CQL syntax and execute on an Oracle Event Processing CQL Processor to process data from one or more streams or views. For more information, see Section , "Queries".

An Oracle CQL view represents an alternative selection on a stream or relation. In Oracle CQL, you use a view instead of a subquery. For more information, see Section , "Views".

Oracle Event Processing performs a join whenever multiple streams appear in the FROM clause of the query. For more information, see Section , "Joins".

Example 20–1 shows typical Oracle CQL queries defined in an Oracle CQL processor component configuration file for the processor named proc.

Example 20–1 Typical Oracle CQL Query

```
<?xml version="1.0" encoding="UTF-8"?>
<n1:config
    xsi:schemaLocation="http://www.bea.com/ns/wlevs/config/application wlevs_application_
config.xsd"
    xmlns:n1="http://www.bea.com/ns/wlevs/config/application"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <processor>
        <name>proc</name>
        <rules>
            <view id="lastEvents" schema="cusip mbid srcId bidQty ask askQty seq"><![CDATA[
                select cusip, mod(bid) as mbid, srcId, bidQty, ask, askQty, seq
                from filteredStream[partition by srcId, cusip rows 1]
            ]]></view>
            <query id="q1"><![CDATA[
                SELECT *
                FROM   lastEvents [Range Unbounded]
                WHERE  price > 10000
            ]]></query>
        </rules>
    </processor>
</n1:config>
```

As Example 20–1 shows, the rules element contains each Oracle CQL statement in a view or query child element:

- view: contains Oracle CQL view statements (the Oracle CQL equivalent of subqueries). The view element id attribute defines the name of the view.

  In Example 20–1, the view element specifies an Oracle CQL view statement (the Oracle CQL equivalent of a subquery).

- query: contains Oracle CQL select statements. The query element id attribute defines the name of the query.

  In Example 20–1, the query element specifies an Oracle CQL query statement. The query statement selects from the view. By default, the results of a query are output to a down-stream channel. You can control this behavior in the channel configuration using a selector element.

  For more information, see "Configuring a Channel" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

Each Oracle CQL statement is contained in a <![CDATA[...]]> tag and does *not* end in a semicolon (;).

For more information, see:

- Section , "How to Create an Oracle CQL Query"
- Section , "Lexical Conventions"
- Chapter 22, "Oracle CQL Statements"

How to Create an Oracle CQL Query

Typically, you create an Oracle CQL query or view using the Oracle Event Processing IDE for Eclipse. After deployment, you can add, change, and delete Oracle CQL queries using the Oracle Event Processing Visualizer.

To create an Oracle CQL query:

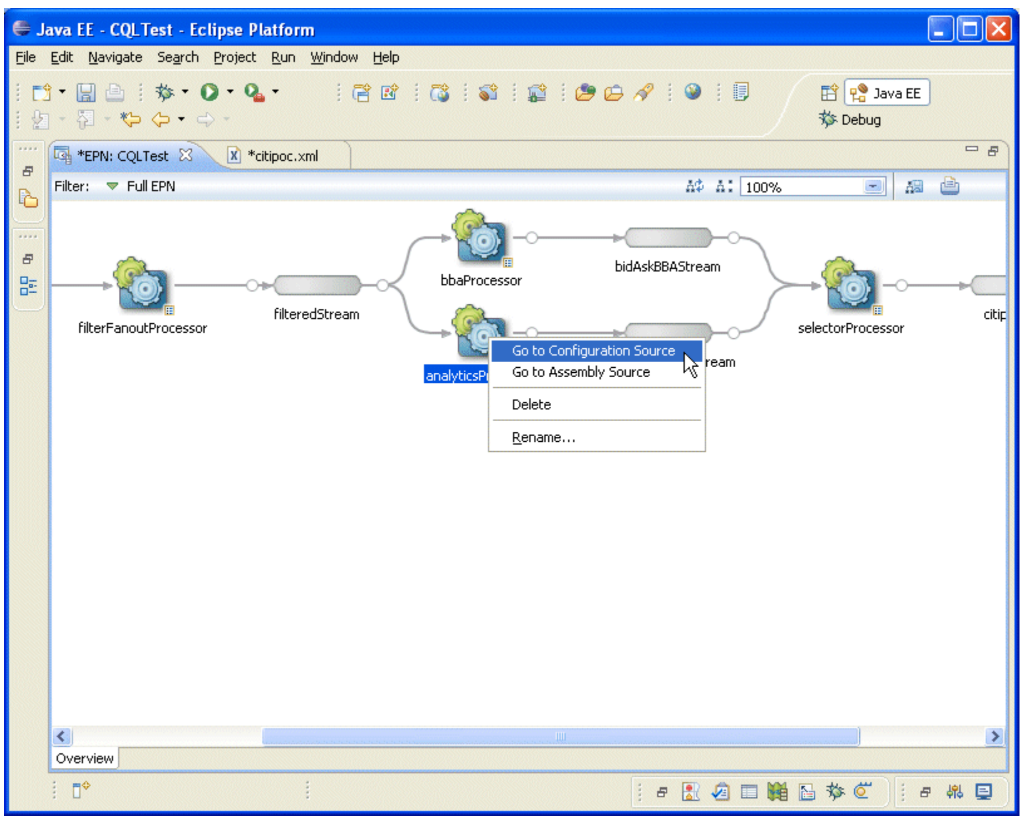
1. Using Oracle Event Processing IDE for Eclipse, create an Oracle Event Processing application and Event Processing Network (EPN).

   For more information, see:

   - "Oracle Event Processing IDE for Eclipse Projects" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.
   - "Oracle Event Processing IDE for Eclipse and the Event Processing Network" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

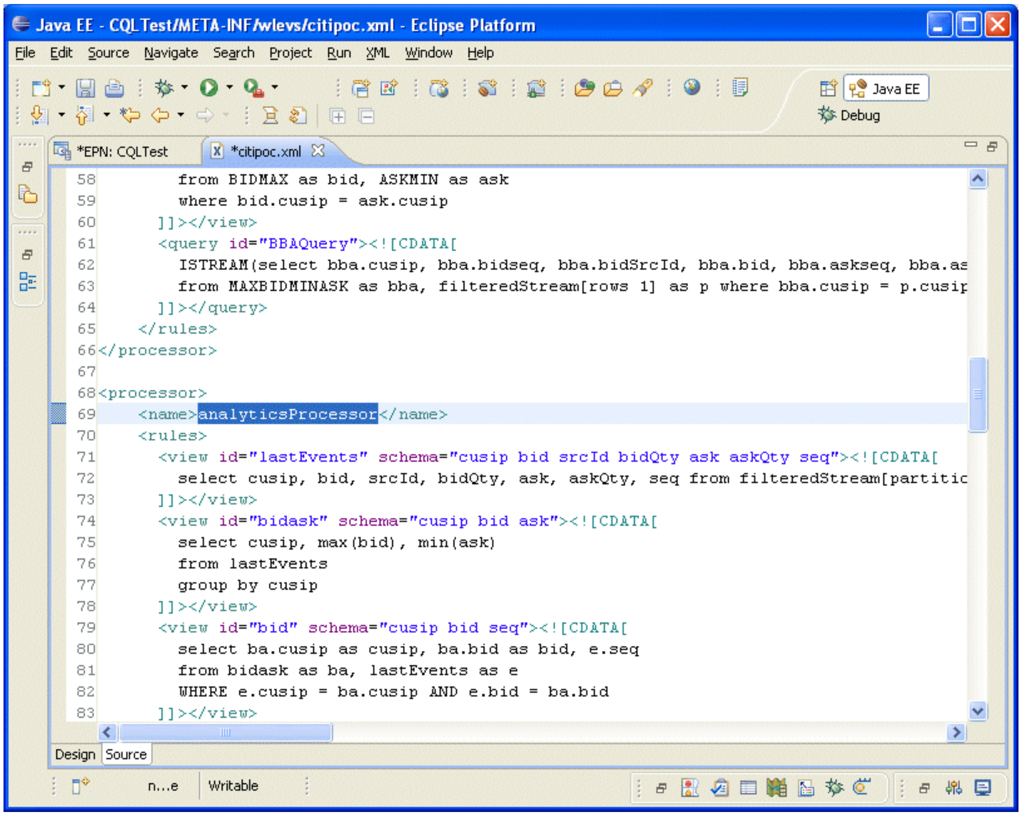
2. In the EPN Editor, right-click an Oracle CQL processor and select **Go to Configuration Source** as Figure 20–1 shows.

Figure 20–1 Navigating to the Configuration Source of a Processor from the EPN Editor



The EPN Editor opens the corresponding component configuration file for this processor and positions the cursor in the appropriate `processor` element as Figure 20–2 shows.

Figure 20–2 Editing the Configuration Source for a Processor



3. Create queries and views and register user-defined functions and windows.

   For examples, see

   - Section , "Queries"
   - Section , "Views"
   - Section , "Joins"
   - Section , "Oracle CQL Queries and Oracle Data Cartridges"
   - Section , "Oracle CQL Queries and the Oracle Event Processing Server Cache"
   - Chapter 4, "Operators"
   - Chapter 5, "Expressions"
   - Chapter 6, "Conditions"
   - Chapter 8, "Built-In Single-Row Functions"
   - Chapter 9, "Built-In Aggregate Functions"
   - Chapter 10, "Colt Single-Row Functions"
   - Chapter 11, "Colt Aggregate Functions"
   - Chapter 12, "java.lang.Math Functions"
   - Chapter 13, "User-Defined Functions"
   - Chapter 21, "Pattern Recognition With MATCH_RECOGNIZE"
   - Chapter 22, "Oracle CQL Statements"

- *Oracle Fusion Middleware Getting Started Guide for Oracle Event Processing*

4. Using Oracle Event Processing IDE for Eclipse, package your Oracle Event Processing application and deploy to the Oracle Event Processing server.

   For more information, see "Assembling and Deploying Oracle Event Processing Applications" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

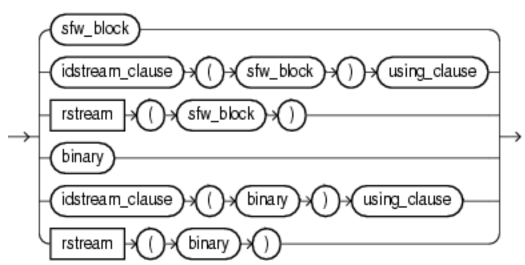
5. After deployment, use the Oracle Event Processing Visualizer to change, add, and delete queries in the Oracle Event Processing application.

   For more information, see "Managing Oracle CQL Rules" in the *Oracle Fusion Middleware Visualizer User's Guide for Oracle Event Processing*.

Queries

Queries are the principle means of extracting information from data streams and views.

***query*::=**



The `query` clause itself is made up of one of the following parts:

- `sfw_block`: use this select-from-where clause to express a CQL query.

  For more information, see Section , "Select, From, Where Block".

- `idstream_clause`: use this clause to specify an input `IStream` or delete `DStream` relation-to-stream operator that applies to the query.

  For more information, see Section , "IDStream Clause"

- `rstream`: use this clause to specify an `RStream` relation-to-stream operator that applies to the query.

  For more information, see "RStream Relation-to-Stream Operator" on page 4-27

- `binary`: use this clause to perform set operations on the tuples that two queries or views return.

  For more information, see Section , "Binary Clause"

The following sections discuss the basic query types that you can create:

- Section , "Simple Query"
- Section , "Built-In Window Query"
- Section , "MATCH_RECOGNIZE Query"
- Section , "Relational Database Table Query"
- Section , "XMLTABLE Query"
- Section , "Function TABLE Query"

- Section , "Cache Query"

For more information, see:

- Section , "Sorting Query Results"
- Section , "Detecting Differences in Query Results"
- Section , "Parameterized Queries"

Query Building Blocks

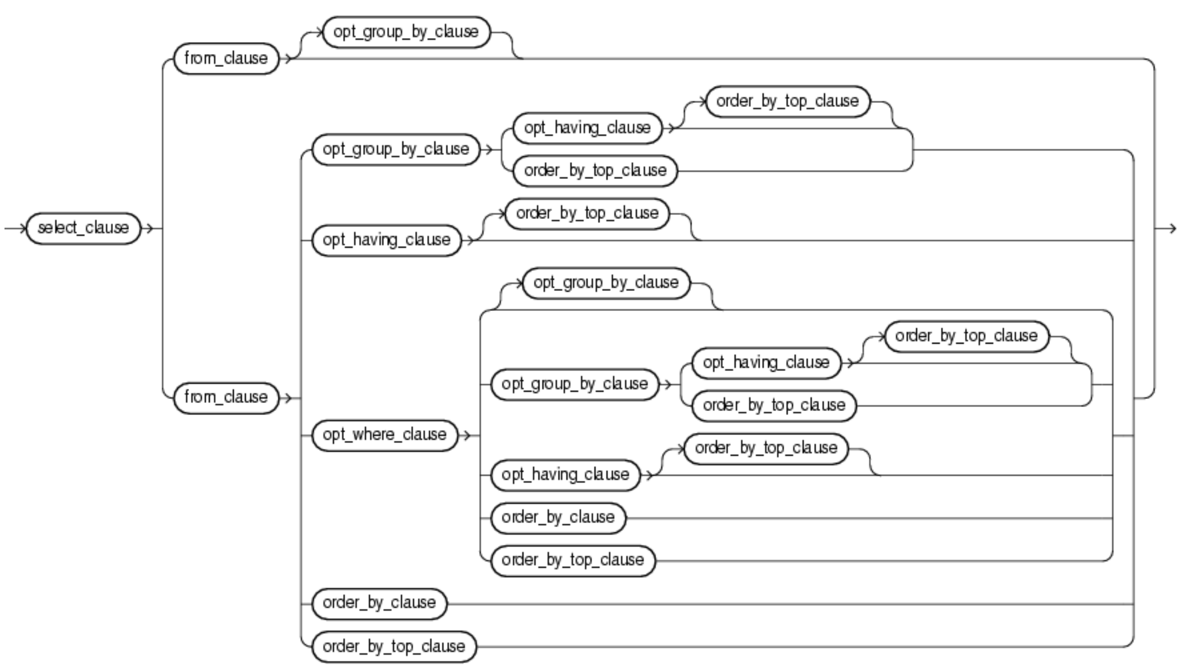
This section summarizes the basic building blocks that you use to construct an Oracle CQL query, including:

- Section , "Select, From, Where Block"
- Section , "Select Clause"
- Section , "From Clause"
- Section , "Where Clause"
- Section , "Group By Clause"
- Section , "Order By Clause"
- Section , "Having Clause"
- Section , "Binary Clause"
- Section , "IDStream Clause"

Select, From, Where Block

Use the `sfw_block` to specify the select, from, and optional where clauses of your Oracle CQL query.

***sfw_block*::=**



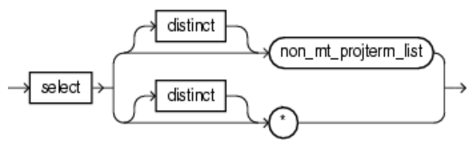
The `sfw_block` is made up of the following parts:

- Section , "Select Clause"
- Section , "From Clause"
- Section , "Where Clause"
- Section , "Group By Clause"
- Section , "Order By Clause"
- Section , "Having Clause"

Select Clause

Use this clause to specify the stream elements you want in the query's result set. The `select_clause` may specify all stream elements using the `*` operator or a list of one or more stream elements.

***select_clause*::=**



The list of expressions that appears after the `SELECT` keyword and before the *`from_clause`* is called the **select list**. Within the select list, you specify one or more stream elements in the set of elements you want Oracle Event Processing to return from one or more streams or views. The number of stream elements, and their datatype and length, are determined by the elements of the select list.

Optionally, specify `distinct` if you want Oracle Event Processing to return only one copy of each set of duplicate tuples selected. Duplicate tuples are those with matching values for each expression in the select list.

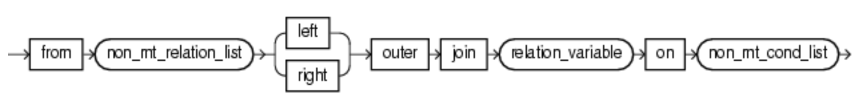
For more information, see *select_clause::=* on page 22-3

From Clause

Use this clause to specify the streams and views that provide the stream elements you specify in the `select_clause` (see Section , "Select Clause").

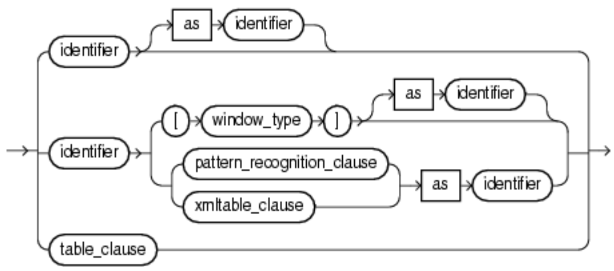
The `from_clause` may specify one or more comma-delimited `relation_variable` clauses.

***from_clause*::=**



For more information, see *from_clause::=* on page 22-3

***relation_variable*::=**



You can select from any of the data sources that your `relation_variable` clause specifies.

You can use the `relation_variable` clause `AS` operator to define an alias to label the immediately preceding expression in the select list so that you can reference the result by that (see Section , "Aliases in the relation_variable Clause").

If you create a join (see Section , "Joins") between two or more streams, view, or relations that have some stream element names in common, then you must qualify stream element names with the name of their stream, view, or relation. Example 20–2 shows how to use stream names to distinguish between the `customerID` stream element in the `OrderStream` and the `customerID` stream element in the `CustomerStream`.

Example 20–2 Fully Qualified Stream Element Names

```
<query id="q0"><![CDATA[
    select * from OrderStream, CustomerStream
    where
        OrderStream.customerID = CustomerStream.customerID
]]></query>
```

Otherwise, fully qualified stream element names are optional. However, Oracle recommends that you always qualify stream element references explicitly. Oracle Event Processing often does less work with fully qualified stream element names.

For more information, see:

- Section , "MATCH_RECOGNIZE Query"
- Section , "XMLTABLE Query"
- Section , "Function TABLE Query"
- *relation_variable::=* on page 22-4

Where Clause

Use this optional clause to specify conditions that determine when the `select_clause` returns results (see Section , "Select Clause").

Because Oracle CQL applies the `WHERE` clause before `GROUP BY` or `HAVING`, if you specify an aggregate function in the `SELECT` clause, you must test the aggregate function result in a `HAVING` clause, not the `WHERE` clause.

For more information, see:

- *opt_where_clause::=* on page 22-4
- Section , "Built-In Aggregate Functions and the Where, Group By, and Having Clauses"

- Section , "Colt Aggregate Functions and the Where, Group By, and Having Clauses"

Group By Clause

Use this optional clause to group (partition) results. This clause does not guarantee the order of the result set. To order the groupings, use the order by clause.

Because Oracle CQL applies the `WHERE` clause before `GROUP BY` or `HAVING`, if you specify an aggregate function in the `SELECT` clause, you must test the aggregate function result in a `HAVING` clause, not the `WHERE` clause.

For more information, see:

- *opt_group_by_clause::=* on page 22-5
- Section , "Group By Clause"
- Section , "Built-In Aggregate Functions and the Where, Group By, and Having Clauses"
- Section , "Colt Aggregate Functions and the Where, Group By, and Having Clauses"

Order By Clause

Use this optional clause to order all results or the top-`n` results.

For more information, see:

- *order_by_clause::=* on page 22-5
- *order_by_top_clause::=* on page 22-5
- Section , "Sorting Query Results"

Having Clause

Use this optional clause to restrict the groups of returned stream elements to those groups for which the specified *`condition`* is `TRUE`. If you omit this clause, then Oracle Event Processing returns summary results for all groups.

Because Oracle CQL applies the `WHERE` clause before `GROUP BY` or `HAVING`, if you specify an aggregate function in the `SELECT` clause, you must test the aggregate function result in a `HAVING` clause, not the `WHERE` clause.

For more information, see:

- *opt_having_clause::=* on page 22-5
- Section , "Built-In Aggregate Functions and the Where, Group By, and Having Clauses"
- Section , "Colt Aggregate Functions and the Where, Group By, and Having Clauses"

Binary Clause

Use the `binary` clause to perform set operations on the tuples that two queries or views return, including:

- `EXCEPT`
- `MINUS`
- `INTERSECT`

- UNION and UNION ALL
- IN and NOT IN

For more information, see *binary::=* on page 22-6.

IDStream Clause

Use this clause to take either a select-from-where clause or binary clause and return its results as one of IStream or DStream relation-to-stream operators.

You can succinctly detect differences in query output by combining an IStream or Dstream operator with the using_clause.

For more information, see:

- *idstream_clause::=* on page 22-6
- "IStream Relation-to-Stream Operator" on page 4-25
- "DStream Relation-to-Stream Operator" on page 4-26
- Section , "Detecting Differences in Query Results"

Simple Query

Example 20–3 shows a simple query that selects all stream elements from a single stream.

Example 20–3 Simple Query

```
<query id="q0"><![CDATA[
    select * from OrderStream where orderAmount > 10000.0
]]></query>
```

For more information, see "Query" on page 22-2.

Built-In Window Query

Example 20–4 shows a query that selects all stream elements from stream S2, with schema (c1 integer, c2 float), using a built-in tuple-based stream-to-relation window operator.

Example 20–4 Built-In Window Query

```
<query id="BBAQuery"><![CDATA[
    select * from S2 [range 5 minutes] where S2.c1 > 10
]]></query>
```

For more information, see:

- Section , "Stream-to-Relation Operators (Windows)"
- *window_type::=* on page 22-4

MATCH_RECOGNIZE Query

Example 20–5 shows a query that uses the MATCH_RECOGNIZE clause to express complex relationships among the stream elements of ItemTempStream.

Example 20–5 MATCH_RECOGNIZE Query

```
<query id="detectPerish"><![CDATA[
  select its.itemId
```

```
  from tkrfid_ItemTempStream MATCH_RECOGNIZE (
      PARTITION BY itemId
      MEASURES A.itemId as itemId
      PATTERN (A B* C)
      DEFINE
          A  AS  (A.temp >= 25),
          B  AS  ((B.temp >= 25) and (to_timestamp(B.element_time) - to_timestamp(A.element_time) < INTERVAL "0
00:00:05.00" DAY TO SECOND)),
          C  AS  (to_timestamp(C.element_time) - to_timestamp(A.element_time) >= INTERVAL "0 00:00:05.00" DAY TO
SECOND)
  ) as its
]]></query>
```

For more information, see:

- Chapter 21, "Pattern Recognition With MATCH_RECOGNIZE"
- *pattern_recognition_clause::=* on page 21-2

Relational Database Table Query

Using an Oracle CQL processor, you can specify a relational database table as an event source. You can query this event source, join it with other event sources, and so on.

For more information, see, Section , "Oracle CQL Queries and Relational Database Tables"

XMLTABLE Query

Use this query to map the results of an XPath or XQuery expression into tuples.

XMLTABLE has the following sub-clauses:

- XMLNAMESPACES -- Optional. A string with a set of XML namespace declarations that can be used in the query expression.
- XQuery_string -- A string with the XQuery or XPath string to use to query the XML.
- PASSING BY VALUE -- Points to the XML that is being used for input.
- COLUMNS -- Optional. Defines the output properties of the result.
  - PATH -- Optional. A subclause of COLUMNS that specifies an XPath expression that points to where values for that property should be drawn from the XML.

Example 20–6 shows a view `v1` and a query `q1` on that view. The view selects from a stream `S1` of `xmltype` stream elements. The view `v1` uses the `XMLTABLE` clause to parse data from the `xmltype` stream elements using XPath expressions. Note that the data types in the view's schema match the datatypes of the parsed data in the `COLUMNS` clause. The query `q1` selects from this view as it would from any other data source. The `XMLTABLE` clause also supports XML namespaces.

For other examples, see "XMLTABLE Query Example" and "XMLTABLE With XML Namespaces Query Example".

Example 20–6 XMLTABLE Query

```
<view id="v1" schema="orderId LastShares LastPrice"><![CDATA[
    SELECT
        X.OrderId,
        X.LastShares,
        X.LastPrice
```

```
    FROM S1,
    XMLTABLE (
        "//FILL"
        PASSING BY VALUE
        S1.c1 as "."
        COLUMNS
            OrderId char(16) PATH "fn:data(../@ID)",
            LastShares integer PATH "fn:data(@LastShares)",
            LastPrice float PATH "fn:data(@LastPx)"
    ) as X
]]></view>

<query id="q1"><![CDATA[
    IStream(
        select
            orderId,
            sum(LastShares * LastPrice),
            sum(LastShares * LastPrice) / sum(LastShares)
        from
            v1[now]
        group by orderId
    )
]]></query>
```

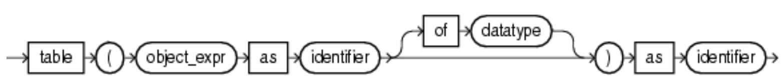
For more information, see:

- Section , "Stream-to-Stream Operators"
- *xmltable_clause::=* on page 22-6
- "SQL/XML (SQLX)" on page 5-16

Function TABLE Query

Use the `TABLE` clause to access the multiple rows returned by a built-in or user-defined function in the `FROM` clause of an Oracle CQL query. The `TABLE` clause converts the set of returned rows into an Oracle CQL relation. Because this is an external relation, you must join the `TABLE` function clause with a stream.

table_clause::=



(*object_expr::=* on page 5-19, *identifier::=* on page 7-17, *datatype::=* on page 2-2)

Note the following:

- The function must return an array type or `Collection` type.
- You must join the `TABLE` function clause with a stream.

Example 20–7 shows a data cartridge `TABLE` clause that invokes the Oracle Spatial method `getContainingGeometries`, passing in one parameter (`InputPoints.point`). The return value of this method, a `Collection`, is aliased as `validGeometries`. The relation that the `TABLE` clause returns is aliased as `R2`.

Example 20–7 Data Cartridge TABLE Query

```
<query id="q1"><![CDATA[
RSTREAM (
    SELECT
        R2.validGeometries.shape as containingGeometry,
        R1.point as inputPoint
```

```
    FROM
        InputPoints[now] as R1,
        TABLE (getContainingGeometries@spatial (InputPoints.point) as validGeometries) AS R2
)
]]></query>
```

Example 20–8 shows an invalid data cartridge TABLE query that fails to join the data cartridge TABLE clause with another stream because the function getAllGeometries@spatial was called without any parameters. Oracle Event Processing invokes the data cartridge method only on the arrival of elements on the joined stream.

Example 20–8 Invalid Data Cartridge TABLE Query

```
<query id="q2"><![CDATA[
RSTREAM (
    SELECT
        R2.validGeometries.shape as containingGeometry
    FROM
        TABLE (getAllGeometries@spatial () as validGeometries) AS R2
)
]]></query>
```

For more examples, see:

- "Data Cartridge TABLE Query Example: Iterator" on page 22-22
- "Data Cartridge TABLE Query Example: Array" on page 22-23
- "Data Cartridge TABLE Query Example: Collection" on page 22-24

For more information, see:

- "table_clause" on page 22-10
- Section , "Oracle CQL Queries and Oracle Data Cartridges"
- Section , "Functions"
- Section , "Arrays"
- Section , "Collections"

Cache Query

Using an Oracle CQL processor, you can specify a cache as an event source. You can query this event source and join it with other event sources using a Now window only.

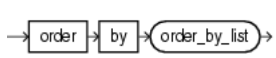
Oracle Event Processing cache event sources are pull data sources: that is, Oracle Event Processing polls the event source on arrival of an event on the data stream.

For more information, see Section , "Oracle CQL Queries and the Oracle Event Processing Server Cache".

Sorting Query Results

Use the ORDER BY clause to order the rows selected by a query.

***order_by_clause*::=**



(*order_by_list::=* on page 22-5)

Sorting by position is useful in the following cases:

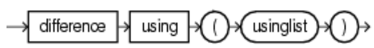
- To order by a lengthy select list expression, you can specify its position in the ORDER BY clause rather than duplicate the entire expression.
- For compound queries containing set operators UNION, INTERSECT, MINUS, or UNION ALL, the ORDER BY clause must specify positions or aliases rather than explicit expressions. Also, the ORDER BY clause can appear only in the last component query. The ORDER BY clause orders all rows returned by the entire compound query.

The mechanism by which Oracle Event Processing sorts values for the ORDER BY clause is specified by your Java locale.

Detecting Differences in Query Results

Use the DIFFERENCE USING clause to succinctly detect differences in the IStream or DStream of a query.

using_clause::=



(*usinglist::=* on page 22-6)

Consider the query that Example 20–9 shows.

Example 20–9 DIFFERENCE USING Clause

```
<query id="q0">
    ISTREAM (
        SELECT c1 FROM S [RANGE 1 NANOSECONDS]
    )    DIFFERENCE USING (c1)
</query>
```

Table 20–1 shows sample input for this query. The Relation column shows the contents of the relation S [RANGE 1 NANOSECONDS] and the Output column shows the query results after the DIFFERENCE USING clause is applied. This clause allows you to succinctly detect only differences in the IStream output.

Table 20–1 DIFFERENCE USING Clause Affect on IStream

Input	Relation	Output
1000: +5	{5}	+5
1000: +6	{5, 6}	+6
1000: +7	{5, 6, 7}	+7
1001: +5	{5, 6, 7, 5}	
1001: +6	{5, 6, 7, 5, 6}	
1001: +7	{5, 6, 7, 5, 6, 7}	
1001: +8	{5, 6, 7, 5, 6, 7, 8}	+8
1002: +5	{5, 6, 7, 5, 6, 7, 8, 5}	
1003: -5	{5, 6, 7, 5, 6, 7, 8}	
1003: -5	{5, 6, 7, 6, 7, 8}	
1003: -5	{6, 7, 6, 7, 8}	
1003: -6	{6, 7, 7, 8}	
1003: -6	{7, 7, 8}	

Table 20–1 (Cont.) DIFFERENCE USING Clause Affect on IStream

Input	Relation	Output
`1003: -7`	`{7, 8}`	
`1003: -7`	`{8}`	
`1003: -8`	`{}`	
`1004: +5`	`{5}`	`+5`

When you specify the `usinglist` in the `DIFFERENCE USING` clause, you may specify columns by:

- attribute name: use this option when you are selecting by attribute name.

  Example 20–10 shows attribute name `c1` in the `DIFFERENCE USING` clause `usinglist`.

- alias: use this option when you want to include the results of an expression where an alias is specified.

  Example 20–10 shows alias `logval` in the `DIFFERENCE USING` clause `usinglist`.

- position: use this option when you want to include the results of an expression where no alias is specified.

  Specify position as a constant, positive integer starting at 1, reading from left to right.

  Example 20–10 specifies the result of expression `funct(c2, c3)` by its position (3) in the `DIFFERENCE USING` clause `usinglist`.

Example 20–10 Specifying the usinglist in a DIFFERENCE USING Clause

```
<query id="q1">
    ISTREAM (
        SELECT c1, log(c4) as logval, funct(c2, c3) FROM S [RANGE 1 NANOSECONDS]
    )    DIFFERENCE USING (c1, logval, 3)
</query>
```

You can use the `DIFFERENCE USING` clause with both `IStream` and `DStream` operators.

For more information, see:

- *using_clause::=* on page 22-6
- "IStream Relation-to-Stream Operator" on page 4-25
- "DStream Relation-to-Stream Operator" on page 4-26

Parameterized Queries

You can parameterize an Oracle CQL query and bind parameter values at run time using the `:n` character string as a placeholder, where *n* is a positive integer that corresponds to the position of the replacement value in a `params` element.

> **Note:** You cannot parameterize a view.

Example 20–11 shows a parameterized Oracle CQL query.

Example 20–11 Parameterized Oracle CQL Query

```
<n1:config xmlns:n1="http://www.bea.com/ns/wlevs/config/application"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    ...
    <processor>
        <name>myProcessor</name>
        <rules>
            <query id="MarketRule"><![CDATA[
                 SELECT symbol, AVG(price) AS average, :1 AS market
                 FROM StockTick [RANGE 5 SECONDS]
                 WHERE symbol = :2
            ]]></query>
        </rules>
        <bindings>
            <binding id="MarketRule">
                <params id="nasORCL">NASDAQ, ORCL</params>
                <params id="nyJPM">NYSE, JPM</params>
                <params id="nyWFC">NYSE, WFC</params>
            </binding>
        </bindings>
    </processor>
    <processor>
        <name>summarizeResults</name>
        <rules>
            <query id="SummarizeResultsRule"><![CDATA[
                select
                    crossRate1 || crossRate2 as crossRatePair,
                    count(*) as totalCount,
                    :1 as averageInternalPrice
                from CrossRateStream
                group by crossRate1,crossRate2
                having :2
            ]]></query>
        </rules>
        <bindings>
            <binding id="SummarizeResultsRule">
                <params id="avgcount">avg(internalPrice), count(*) > 0</params>
            </binding>
        </bindings>
    </processor>
</n1:config>
```

In this example, the:

- `MarketRule` query specifies two parameters: the third term in the `SELECT` and the value of `symbol` in the `WHERE` clause
- `SummarizeResultsRule` query specifies two parameters: the third term in the `SELECT` and the value of the `HAVING` clause.

This section describes:

- Section , "Parameterized Queries in Oracle CQL Statements"
- Section , "The bindings Element"
- Section , "Run-Time Query Naming"
- Section , "Lexical Conventions for Parameter Values"
- Section , "Parameterized Queries at Runtime"
- Section , "Replacing Parameters Programmatically"

Parameterized Queries in Oracle CQL Statements

You may specify a placeholder anywhere an arithmetic expression or a `String` literal is legal in an Oracle CQL statement. For example:

- `SELECT` list items
- `WHERE` clause predicates
- `WINDOW` constructs (such as `RANGE`, `SLIDE`, `ROWS`, and `PARTITION BY`)
- `PATTERN` duration clause

For more information, see:

- "arith_expr" on page 5-6
- "Literals" on page 2-7

The bindings Element

Parameter values are contained by a `bindings` element. There may be one `bindings` element per `processor` element.

For each parameterized query, the `bindings` element must contain a `binding` element with the same `id` as the query.

The `binding` element must contain one or more `params` elements. Each `params` element must have a unique `id` and must contain a comma separated list of parameter values equal in number to the number of placeholder characters (`:`*n*) in the corresponding query.

The order of the parameter values corresponds to placeholder characters (`:`*n*) in the parameterized query, such that `:1` corresponds to the first parameter value, `:2` corresponds to the second parameter value, and so on. You may use placeholder characters (`:`*n*) in any order. That is, `:1` corresponds to the first parameter value whether it precedes or follows `:2` in a query. A placeholder number can be used only once in a query.

For more information, see:

- Section , "Lexical Conventions for Parameter Values"
- Section , "Parameterized Queries at Runtime"

Run-Time Query Naming

When a binding instantiates a parameterized query, Oracle Event Processing creates a new query at run time with the name queryId_paramId. For example, in Example 20–11, the run-time name of the first query instantiated by the MarketRule binding is `MarketRule_nasORCL`.

To avoid run-time naming conflicts, be sure query ID and parameter ID combinations are unique.

Lexical Conventions for Parameter Values

Each `params` element must have a unique `id` and must contain a comma separated list of parameter values equal in number to the number of placeholder characters (`:`*n*) in the corresponding query.

Table 20–2 Parameterized Query Parameter Value Lexical Conventions

<table>
<tr><th>Convention</th><th>Example</th><th>Replacement Value</th></tr>
<tr><td>Primitive type literals</td><td><params id="p1">NASDAQ, 200.0</params></td><td>:1 = NASDAQ
:2 = 200.0</td></tr>
<tr><td>Oracle CQL fragments</td><td><params id="p1">count(*), avg(val)</params></td><td>:1 = count(*)
:2 = avg(val)</td></tr>
<tr><td>Quotes</td><td><params id="p1">'alert', "Seattle, WA", 'fun' || "house", one "two" 3</params></td><td>:1 = 'alert'
:2 = "Seattle, WA"
:3 = 'fun' || "house"
:4 = one "two" 3</td></tr>
</table>

In an Oracle CQL query, a placeholder within single or double quotes is a `String` literal. The following query is not a parameterized query:

```
SELECT ":1" as symbol, price FROM StockTick [RANGE 5 SECONDS]
```

Oracle Event Processing parses this query as assigning the `String` literal `":1"` to alias `symbol`. To make this query into a parameterized query, use:

```
SELECT :1 as symbol, price FROM StockTick [RANGE :2 SECONDS]
```

And define a `params` element like this:

```
<params id="p1">"ORCL", 5</params>
```

Because the parameter value (`ORCL`) does not contain a comma, the quotes are not required. You could specify a `params` element like this:

```
<params id="p1">ORCL, 5</params>
```

However, if the parameter value does contain a comma, then you must use quotes around the parameter value. Consider this parameterized query:

```
SELECT :1 = cityAndState AS cityOfInterest FROM channel1 [RANGE :2 SECONDS]
```

Where `cityAndState` has values like `"Seattle, WA"` or `"Ottawa, ON"`. In this case, you must specify a `params` element like this:

```
<params id="p1">"Seattle, WA", 5</params>
<params id="p1">"Ottawa, ON", 5</params>
```

Commas are allowed only in quoted parameter values that signify string values. Commas are not allowed as a separator character in unquoted parameter values. For example:

`"Seattle, WA"` is valid, because the comma is part of the string.

`PARTITION BY fromRate,toRate ROWS 10` is invalid. Create the following two parameters instead:

```
PARTITION BY fromRate ROWS 10
PARTITION BY toRate ROWS 10
```

Parameterized Queries at Runtime

Each `params` element effectively causes a new Oracle CQL query to execute with the new parameters. At rule execution time, Oracle CQL substitutes parameter values for placeholder characters, from left to right. Example 20–11 is effectively equivalent to the queries that Example 20–12 shows.

Example 20–12 Equivalent Queries at Runtime

```
SELECT symbol, AVG(price) AS average, NASDAQ AS market
FROM StockTick [RANGE 5 SECONDS]
WHERE symbol = ORCL

SELECT symbol, AVG(price) AS average, NYSE AS market
FROM StockTick [RANGE 5 SECONDS]
WHERE symbol = JPM

SELECT symbol, AVG(price) AS average, NYSE AS market
FROM StockTick [RANGE 5 SECONDS]
WHERE symbol = WFC
```

Replacing Parameters Programmatically

If you use the CQLProcessorMBean.replaceAllBoundParameters() method to programmatically replace parameters in a parameterized query, any existing parameters not replaced by the method are automatically removed from the query.

Views

Queries are the principle means of extracting information from data streams and relations. A view represents an alternative selection on a stream or relation that you can use to create subqueries.

A view is only accessible by the queries that reside in the same processor and cannot be exposed beyond that boundary.

You can specify any query type in the definition of your view. For more information, see Section , "Queries".

For complete details on the view statement, see "View" on page 22-29.

In Example 20–13, query `BBAQuery` selects from view `MAXBIDMINASK` which in turn selects from other views such as `BIDMAX` which in turn selects from other views. Finally, views such as `lastEvents` select from an actual event source: `filteredStream`. Each such view represents a separate derived stream drawn from one or more base streams.

Example 20–13 Using Views Instead of Subqueries

```
<view id="lastEvents" schema="cusip bid srcId bidQty ask askQty seq"><![CDATA[
    select cusip, bid, srcId, bidQty, ask, askQty, seq
    from filteredStream[partition by srcId, cusip rows 1]
]]></view>
<view id="bidask" schema="cusip bid ask"><![CDATA[
    select cusip, max(bid), min(ask)
    from lastEvents
    group by cusip
]]></view>
<view id="bid" schema="cusip bid seq"><![CDATA[
    select ba.cusip as cusip, ba.bid as bid, e.seq
    from bidask as ba, lastEvents as e
    WHERE e.cusip = ba.cusip AND e.bid = ba.bid
]]></view>
<view id="bid1" schema="cusip maxseq"><![CDATA[
    select b.cusip, max(seq) as maxseq
    from bid as b
    group by b.cusip
]]></view>
```

```
<view id="BIDMAX" schema="cusip seq srcId bid bidQty"><![CDATA[
    select e.cusip, e.seq, e.srcId, e.bid, e.bidQty
    from bid1 as b, lastEvents as e
    where (e.seq = b.maxseq)
]]></view>
<view id="ask" schema="cusip ask seq"><![CDATA[
    select ba.cusip as cusip, ba.ask as ask, e.seq
    from bidask as ba, lastEvents as e
    WHERE e.cusip = ba.cusip AND e.ask = ba.ask
]]></view>
<view id="ask1" schema="cusip maxseq"><![CDATA[
    select a.cusip, max(seq) as maxseq
    from ask as a
    group by a.cusip
]]></view>
<view id="ASKMIN" schema="cusip seq srcId ask askQty"><![CDATA[
    select e.cusip, e.seq, e.srcId, e.ask, e.askQty
    from ask1 as a, lastEvents as e
    where (e.seq = a.maxseq)
]]></view>
<view id="MAXBIDMINASK" schema="cusip bidseq bidSrcId bid askseq askSrcId ask bidQty
askQty"><![CDATA[
    select bid.cusip, bid.seq, bid.srcId as bidSrcId, bid.bid, ask.seq, ask.srcId as
askSrcId, ask.ask, bid.bidQty, ask.askQty
    from BIDMAX as bid, ASKMIN as ask
    where bid.cusip = ask.cusip
]]></view>
<query id="BBAQuery"><![CDATA[
    ISTREAM(select bba.cusip, bba.bidseq, bba.bidSrcId, bba.bid, bba.askseq, bba.askSrcId,
bba.ask, bba.bidQty, bba.askQty, "BBAStrategy" as intermediateStrategy, p.seq as
correlationId, 1 as priority
    from MAXBIDMINASK as bba, filteredStream[rows 1] as p where bba.cusip = p.cusip)
]]></query>
```

Using this technique, you can achieve the same results as in the subquery case. However, using views you can better control the complexity of queries and reuse views by name in other queries.

Views and Joins

If you create a join between two or more views that have some stream element names in common, then you must qualify stream element names with names of streams. Example 20–14 shows how to use view names to distinguish between the `seq` stream element in the `BIDMAX` view and the `seq` stream element in the `ASKMIN` view.

Example 20–14 Using View Names to Distinguish Between Stream Elements of the Same Name

```
<view id="MAXBIDMINASK" schema="cusip bidseq bidSrcId bid askseq askSrcId ask bidQty
askQty"><![CDATA[
    select bid.cusip, bid.seq, bid.srcId as bidSrcId, bid.bid, ask.seq, ask.srcId as
askSrcId, ask.ask, bid.bidQty, ask.askQty
    from BIDMAX as bid, ASKMIN as ask
    where bid.cusip = ask.cusip
]]></view>
```

Otherwise, fully qualified stream element names are optional. However, it is a best practice to always qualify stream element references explicitly. Oracle Event Processing often does less work with fully qualified stream element names.

For more information, see Section , "Joins".

Views and Schemas

You may define the optional schema of the view using a space delimited list of event attribute names as Example 20–15 shows.

Example 20–15 Schema With Event Attribute Names Only

```
<view id="MAXBIDMINASK" schema="cusip bidseq"><![CDATA[
    select ...
]]></view>
```

Joins

A **join** is a query that combines rows from two or more streams, views, or relations. Oracle Event Processing performs a join whenever multiple streams appear in the `FROM` clause of the query. The select list of the query can select any stream elements from any of these streams. If any two of these streams have a stream element name in common, then you must qualify all references to these stream elements throughout the query with stream names to avoid ambiguity.

If you create a join between two or more streams, view, or relations that have some stream element names in common, then you must qualify stream element names with the name of their stream, view, or relation. Example 20–16 shows how to use stream names to distinguish between the `customerID` stream element in the `OrderStream` stream and the `customerID` stream element in the `CustomerStream` stream.

Example 20–16 Fully Qualified Stream Element Names

```
<query id="q0"><![CDATA[
    select * from OrderStream[range 5] as orders, CustomerStream[range 3] as customers where
        orders.customerID = customers.customerID
]]></query>
```

Otherwise, fully qualified stream element names are optional. However, Oracle recommends that you always qualify stream element references explicitly. Oracle Event Processing often does less work with fully qualified stream element names.

Oracle Event Processing supports the following types of joins:

- Inner Joins
- Outer Joins

> **Note:** When joining against a cache, you must observe additional query restrictions as Section , "Creating Joins Against the Cache" describes.

Inner Joins

By default, Oracle Event Processing performs an inner join (sometimes called a simple join): a join of two or more streams that returns only those stream elements that satisfy the join condition.

Example 20–17 shows how to create a query `q4` that uses an inner join between streams `S0`, with schema `(c1 integer, c2 float)`, and `S1`, with schema `(c1 integer, c2 float)`.

Example 20–17 Inner Joins

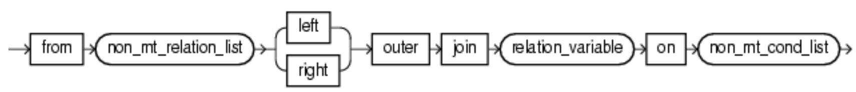
```
<query id="q4"><![CDATA[
    select *
        from
            S0[range 5] as a,
            S1[range 3] as b
        where
            a.c1+a.c2+4.9 = b.c1 + 10
]]></query>
```

Outer Joins

An outer join extends the result of a simple join. An outer join returns all rows that satisfy the join condition and also returns some or all of those rows from one table for which no rows from the other satisfy the join condition.

You specify an outer join in the `FROM` clause of a query using `LEFT` or `RIGHT OUTER JOIN ... ON` syntax.

***from_clause*::=**



(*non_mt_relation_list::=* on page 22-4, *relation_variable::=* on page 22-4, *non_mt_cond_list::=* on page 7-25)

Example 20–18 shows how to create a query `q5` that uses a left outer join between streams `S0`, with schema `(c1 integer, c2 float)`, and `S1`, with schema `(c1 integer, c2 float)`.

Example 20–18 Outer Joins

```
<query id="q5"><![CDATA[
    SELECT a.c1+b.c1
    FROM S0[range 5] AS a LEFT OUTER JOIN S1[range 3] AS b ON b.c2 = a.c2
    WHERE b.c2 > 3
]]></query>
```

Use the `ON` clause to specify a join condition. Doing so lets you specify join conditions separate from any search or filter conditions in the `WHERE` clause.

You can perform the following types of outer join:

- Section , "Left Outer Join"
- Section , "Right Outer Join"
- Section , "Outer Join Look-Back"

Left Outer Join

To write a query that performs an outer join of streams A and B and returns all stream elements from A (a left outer join), use the `LEFT OUTER JOIN` syntax in the `FROM` clause as Example 20–19 shows. For all stream elements in A that have no matching stream elements in B, Oracle Event Processing returns null for any select list expressions containing stream elements of B.

Example 20–19 Left Outer Joins

```
<query id="q5"><![CDATA[
    SELECT a.c1+b.c1
```

```
    FROM S0[range 5] AS a LEFT OUTER JOIN S1[range 3] AS b  ON b.c2 = a.c2
    WHERE b.c2 > 3
]]></query>
```

Right Outer Join

To write a query that performs an outer join of streams A and B and returns all stream elements from B (a right outer join), use the `RIGHT OUTER JOIN` syntax in the `FROM` clause as Example 20–20 shows. For all stream elements in B that have no matching stream elements in A, Oracle Event Processing returns null for any select list expressions containing stream elements of A.

Example 20–20 Right Outer Joins

```
<query id="q5"><![CDATA[
    SELECT a.c1+b.c1
    FROM S0[range 5] AS a RIGHT OUTER JOIN S1[range 3] AS b ON b.c2 = a.c2
    WHERE b.c2 > 3
]]></query>
```

Outer Join Look-Back

You can create an outer join that refers or looks-back to a previous outer join as Example 20–21 shows.

Example 20–21 Outer Join Look-Back

```
<query id="q5"><![CDATA[
    SELECT R1.c1+R2.c1
    FROM S0[rows 2] as R1 LEFT OUTER JOIN S1[rows 2] as R2 on R1.c2 = R2.c2 RIGHT OUTER JOIN
S2[rows 2] as R3 on R2.c2 = R3.c22
    WHERE R2.c2 > 3
]]></query>
```

Oracle CQL Queries and the Oracle Event Processing Server Cache

You can access an Oracle Event Processing cache from an Oracle CQL statement or user-defined function.

This section describes:

- Section , "Creating Joins Against the Cache"

For more information, see:

- "Configuring Oracle Event Processing Caching" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*
- "Accessing a Cache From an Oracle CQL Statement" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*
- "Accessing a Cache From an Oracle CQL User-Defined Function" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*
- "Oracle Continuous Query Language (CQL) Example" in the *Oracle Fusion Middleware Getting Started Guide for Oracle Event Processing*

Creating Joins Against the Cache

When writing Oracle CQL queries that join against a cache, you must observe the following restrictions:

- Section , "Cache Key First and Simple Equality"
- Section , "No Arithmetic Operations on Cache Keys"
- Section , "No Full Scans"
- Section , "Multiple Conditions and Inequality"

For more information, see Section , "Joins".

Cache Key First and Simple Equality

The complex predicate's first subclause (from the left) with a comparison operation over a cache key attribute may only be a simple equality predicate.

The following predicate is invalid because the predicate is not the first sub-clause (from the left) which refers to cache attributes:

```
... S.c1 = 5 AND CACHE.C2 = S.C2 AND CACHE.C1 = S.C1 ...
```

However, the following predicate is valid:

```
... S.c1 = 5 AND CACHE.C1 = S.C1 AND CACHE.C2 = S.C2 ...
```

No Arithmetic Operations on Cache Keys

The subclause may not have any arithmetic operations on a cache key attribute.

The following predicate is invalid because arithmetic operations are not allowed on cache key attributes:

```
... CACHE.C1 + 5 = S.C1 AND CACHE.C2 = S.C2 ...
```

No Full Scans

The complex predicate must not require a full scan of the cache.

Assume that your cache has cache key `C1`.

The following predicates are invalid. Because they do not use the cache key attribute in comparisons, they must scan through the whole cache which is not allowed.

```
... CACHE.C2 = S.C1 ...

... CACHE.C2 > S.C1 ...

... S.C1 = S.C2 ...

... S.C1 = CACHE.C2 AND S.C2 = CACHE.C2 ...
```

The following predicates are also invalid. Although they do use the cache key attribute in comparisons, they use inequality operations that must scan through the whole cache which is not allowed.

```
... CACHE.C1 != S.C1 ...

... CACHE.C1 > 5 ...

... CACHE.C1 + 5 = S.C1 ...
```

The following predicate is also invalid. Although they do use the cache key attribute in comparisons, the first subclause referring to the cache attributes does not refer to the cache key attribute (in this example, `C1`). That is, the first subclause refers to `C2` which is not a cache key and the cache key comparison subclause (`CACHE.C1 = S.C1`) appears after the non-key comparison subclause.

```
... CACHE.C2 = S.C2 AND CACHE.C1 = S.C1 ...
```

Multiple Conditions and Inequality

To support multiple conditions, inequality, or both, you must make the first sub-clause an equality predicate comparing a cache key value and specify the rest of the predicate subclauses separated by one AND operator.

The following are valid predicates:

```
... S.c1 = 5 AND CACHE.C1 = S.C1 AND CACHE.C2 > S.C2 ...

... CACHE.C1 = S.C1 AND CACHE.C2 = S.C2 ...

... S.c1 = 5 AND CACHE.C1 = S.C1 AND CACHE.C2 != S.C2 ...
```

Oracle CQL Queries and Relational Database Tables

You can access a relational database table from an Oracle CQL query using:

- table source: using a table source, you may join a stream only with a NOW window and only to a single database table.

  > **Note:** Because changes in the table source are not coordinated in time with stream data, you may only join the table source to an event stream using a Now window and you may only join to a single database table. For more information, see "S[now]" on page 4-7.
  >
  > To integrate arbitrarily complex SQL queries and multiple tables with your Oracle CQL queries, consider using the Oracle JDBC data cartridge instead.

  For more information, see "Configuring an Oracle CQL Processor Table Source" in the *Oracle Fusion Middleware Developer's Guide for Oracle Event Processing for Eclipse*.

- Oracle JDBC data cartridge: using the Oracle JDBC data cartridge, you may integrate arbitrarily complex SQL queries and multiple tables and datasources with your Oracle CQL queries.

  For more information, see Section , "Understanding the Oracle Event Processing JDBC Data Cartridge".

  > **Note:** Oracle recommends that you use the Oracle JDBC data cartridge to access relational database tables from an Oracle CQL statement.

In all cases, you must define datasources in the Oracle Event Processing server config.xml file. For more information, see "Configuring Access to a Relational Database" in the *Oracle Fusion Middleware Administrator's Guide for Oracle Event Processing*.

Oracle Event Processing relational database table event sources are pull data sources: that is, Oracle Event Processing polls the event source on arrival of an event on the data stream.

Oracle CQL Queries and Oracle Data Cartridges

You can access Oracle CQL data cartridge types in Oracle CQL queries just as you would Oracle CQL native types.

For more information, see:

- Section , "Oracle CQL Data Cartridge Types"
- Chapter 15, "Oracle Java Data Cartridge"
- Chapter 16, "Oracle Spatial"
- Chapter 17, "Oracle Event Processing JDBC Data Cartridge"

21

Pattern Recognition With MATCH_RECOGNIZE

This chapter provides reference and usage information about the MATCH_RECOGNIZE clause in Oracle Continuous Query Language (Oracle CQL). This clause and its sub-clauses perform pattern recognition in Oracle CQL queries.

This chapter includes the following sections:

Understanding Pattern Recognition With MATCH_RECOGNIZE

The MATCH_RECOGNIZE clause performs pattern recognition in an Oracle CQL query as Example 21–1 shows. This query will export (make available for inclusion in the SELECT) the MEASURES clause values for events (tuples) that satisfy the PATTERN clause regular expression over the DEFINE clause conditions.

Example 21–1 Pattern Matching With MATCH_RECOGNIZE

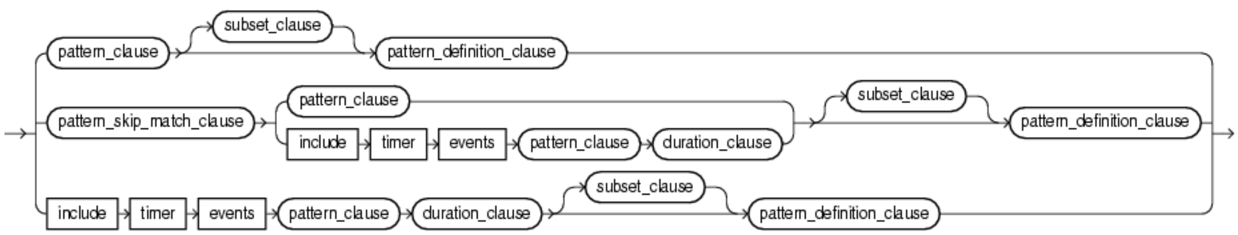
```
<query id="detectPerish"><![CDATA[
  select its.badItemId
  from tkrfid_ItemTempStream
  MATCH_RECOGNIZE (
      PARTITION BY itemId
      MEASURES A.itemId as badItemId
      PATTERN (A B* C)
      DEFINE
          A  AS  (A.temp >= 25),
          B  AS  ((B.temp >= 25) and (to_timestamp(B.element_time) - to_timestamp(A.element_time) < INTERVAL "0
00:00:05.00" DAY TO SECOND)),
```

```
         C  AS  (to_timestamp(C.element_time) - to_timestamp(A.element_time) >= INTERVAL "0 00:00:05.00" DAY TO
SECOND)
  ) as its
]]></query>
```

pattern_recognition_clause::=

(*pattern_partition_clause::=* on page 21-17, *pattern_measures_clause::=* on page 21-9, *pattern_def_dur_clause::=* on page 21-2)

pattern_def_dur_clause::=



(*pattern_clause::=* on page 21-11, *pattern_skip_match_clause::=* on page 21-19, *pattern_definition_clause::=* on page 21-14, *duration_clause::=* on page 21-22, *subset_clause::=* on page 21-25)

Using `MATCH_RECOGNIZE`, you define conditions on the attributes of incoming events and identify these conditions by using *`identifiers`* called correlation variables. Example 21–1 defines correlation variables `A`, `B`, and `C`. A sequence of consecutive events in the input stream satisfying these conditions constitutes a pattern.

The output of a `MATCH_RECOGNIZE` query is always a stream.

The principle `MATCH_RECOGNIZE` sub-clauses are:

- `MEASURES`: exports (makes available for inclusion in the `SELECT`) attribute values of events that successfully match the pattern you specify.

  See Section , "MEASURES Clause".

- `PATTERN`: specifies the pattern to be matched as a regular expression over one ore more correlation variables.

  See Section , "PATTERN Clause".

- `DEFINE`: specifies the condition for one or more correlation variables.

  See Section , "DEFINE Clause".

To refine pattern recognition, you may use the optional `MATCH_RECOGNIZE` sub-clauses, including:

- Section , "PARTITION BY Clause"
- Section , "ORDER BY Clause"
- Section , "ALL MATCHES Clause"
- Section , "WITHIN Clause"
- Section , "DURATION Clause"

- Section , "INCLUDE TIMER EVENTS Clause"
- Section , "SUBSET Clause"

For more information, see:

- Section , "MATCH_RECOGNIZE and the WHERE Clause"
- Section , "Referencing Singleton and Group Matches"
- Section , "Referencing Aggregates"
  - Section , "Using count With *, identifier.*, and identifier.attr"
  - Section , "Using first and last"
- Section , "Using prev"
- Section , "MATCH_RECOGNIZE Examples"

MATCH_RECOGNIZE and the WHERE Clause

In Oracle CQL (as in SQL), the `FROM` clause is evaluated before the `WHERE` clause.

Consider the following Oracle CQL query:

```
SELECT ... FROM S MATCH_RECOGNIZE ( .... ) as T WHERE ...
```

In this query, the `S MATCH_RECOGNIZE ( .... ) as T` is like a subquery in the `FROM` clause and is evaluated first, before the `WHERE` clause.

Consequently, you rarely use both a `MATCH_RECOGNIZE` clause and a `WHERE` clause in the same Oracle CQL query. Instead, you typically use a view to apply the required `WHERE` clause to a stream and then select from the view in a query that applies the `MATCH_RECOGNIZE` clause.

Example 21–2 shows two views, `e1p1` and `e2p2`, each applying a `WHERE` clause to stream `S` to pre-filter the stream for the required events. The query `q` then selects from these two views and applies the `MATCH_RECOGNIZE` on this filtered stream of events.

Example 21–2 MATCH_RECOGNIZE and the WHERE Clause

```
<view id="e1p1">
    SELECT * FROM S WHERE eventName = 'E1' and path = 'P1' and statName = 'countValue'
</view>
<view id="e2p2">
    SELECT * FROM S WHERE eventName = 'E2' and path = 'P2' and statName = 'countValue'
</view>

<query id="q">
    SELECT
        T.e1p1Stat as e1p1Stat, T.e2p2Stat as e2p2Stat
    FROM
        e1p1, e2p2
    MATCH_RECOGNIZE(
        ALL MATCHES
        PATTERN(A+)
        DURATION 60 MINUTES
        DEFINE
            A as (A.e1p1Stat < 1000 and A.e2p2Stat > 2000 and count(A) > 3)
        ) as T
</query>
```

For more information, see *opt_where_clause::=* on page 22-4

Referencing Singleton and Group Matches

The MATCH_RECOGNIZE clause identifies the following types of matches:

- singleton: a correlation variable is a singleton if it occurs exactly once in a pattern, is not defined by a SUBSET, is not in the scope of an alternation, and is not quantified by a pattern quantifier.

  References to such a correlation variable refer to this single event.

- group: a correlation variable is a group if it occurs in more than one pattern, is defined by a SUBSET, is in the scope of an alternation, or is quantified by a pattern quantifier.

  References to such a correlation variable refer to this group of events.

When you reference singleton and group correlation variables in the MEASURES and DEFINE clauses, observe the following rules:

- For singleton correlation variables, you may reference individual event attributes only, not aggregates.
- For group correlation variables:
  - If you reference an individual event attribute, then the value of the last event to match the correlation variable is returned.

    If the correlation variable is not yet matched, NULL is returned. In the case of count(A.*), if the correlation variable A is not yet matched, 0 is returned.

    If the correlation variable is being referenced in a definition of the same variable (such as DEFINE A as A.balance > 1000), then the value of the current event is returned.

  - If you reference an aggregate, then the aggregate is performed over all events that have matched the correlation variable so far.

For more information, see:

- Section , "Using count With *, identifier.*, and identifier.attr"
- Section , "Pattern Quantifiers and Regular Expressions"
- Section , "Referencing Attributes in the DEFINE Clause"

Referencing Aggregates

You can use any built-in, Colt, or user-defined aggregate function in the MEASURES and DEFINE clause of a MATCH_RECOGNIZE query.

When using aggregate functions, consider the following:

- Section , "Running Aggregates and Final Aggregates"
- Section , "Operating on the Same Correlation Variable"
- Section , "Referencing Variables That Have not Been Matched Yet"
- Section , "Referencing Attributes not Qualified by Correlation Variable"

For more information, see:

- Section , "Using count With *, identifier.*, and identifier.attr"
- Section , "Using first and last"
- Section , "Introduction to Oracle CQL Built-In Aggregate Functions"

- Section , "Introduction to Oracle CQL Built-In Aggregate Colt Functions"
- Section , "User-Defined Aggregate Functions"
- Section , "MEASURES Clause"
- Section , "DEFINE Clause"

Running Aggregates and Final Aggregates

In the DEFINE clause, any aggregate function on a correlation variable X is a running aggregate: that is, the aggregate includes all preceding matches of X up to and including the current match. If the correlation variable X has been completely matched so far, then the aggregate is final, otherwise it is running.

In the MEASURES clause, because it is evaluated after the match has been found, all aggregates are final because they are computed over the final match.

When using a SUBSET clause, be aware of the fact that you may inadvertently imply a running aggregate as Example 21–3 shows.

Example 21–3 Implied Running Aggregate

```
...
PATTERN (X+ Y+)
SUBSET Z = (X, Y)
DEFINE
    X AS X.price > 100,
    Y AS sum(Z.price) < 1000
...
```

Because correlation variable Z involves Y, the definition of Y involves a running aggregate on Y.

For more information, see:

- Section , "MEASURES Clause"
- Section , "DEFINE Clause"
- Section , "SUBSET Clause"

Operating on the Same Correlation Variable

In both the MEASURES and DEFINE clause, you may only apply an aggregate function to attributes of the same correlation variable.

For example: the use of aggregate function correlation in Example 21–4 is invalid.

Example 21–4 Invalid Use of Aggregate Function

```
...
MEASURES xycorr AS correlation(X.price, Y.price)
PATTERN (X+ Y+)
DEFINE
    X AS X.price <= 10,
    Y AS Y.price > 10
...
```

The correlation aggregate function may not operate on more than one correlation variable.

Referencing Variables That Have not Been Matched Yet

In the DEFINE clause, you may reference a correlation variable that has not been matched yet. However, you should use caution when doing so. Consider Example 21–5.

Example 21–5 Referencing a Variable That has not Been Matched Yet: Invalid

```
PATTERN (X+ Y+)
DEFINE
    X AS count(Y.*) >= 3
    Y AS Y.price > 10,
```

Although this syntax is legal, note that in this particular example, the pattern will never match because at the time X is matched, Y has not yet been matched, and count(Y.*) is 0.

To implement the desired behavior ("Match when the price of Y has been greater than 10, 3 or more times in a row"), implement this pattern as Example 21–6 shows.

Example 21–6 Referencing a Variable That has not Been Matched Yet: Valid

```
PATTERN (Y+ X+)
DEFINE
    Y AS Y.price > 10,
    X AS count(Y.*) >= 3
```

For more information, see Section , "Using count With *, identifier.*, and identifier.attr".

Referencing Attributes not Qualified by Correlation Variable

In the DEFINE clause, if you apply an aggregate function to an event attribute not qualified by correlation variable, the aggregate is a running aggregate as Example 21–7 shows.

Example 21–7 Referencing Attributes not Qualified by Correlation Variable

```
PATTERN ((RISE FALL)+)
DEFINE
    RISE AS count(RISE.*) = 1 or RISE.price > FALL.price,
    FALL AS FALL.price < RISE.price and count(*) > 1000
```

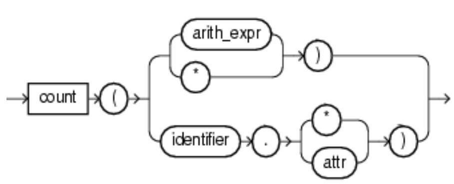
This query detects a pattern in which a price alternately goes up and down, for as long as possible, but for at least more than 1000 matches.

For more information, see:

- Section , "Running Aggregates and Final Aggregates"
- Section , "Using count With *, identifier.*, and identifier.attr"

Using count With *, *identifier*.*, and *identifier.attr*

The built-in aggregate function count has syntax:



(*arith_expr::=* on page 5-6, *attr::=* on page 7-5, *identifier::=* on page 7-17)

The return value of `count` depends on the argument as Table 21–1 shows.

Table 21–1 Return Values for count Aggregate Function

Input Argument	Return Value
arith_expr	The number of tuples where *arith_expr* is not `NULL`.
`*`	The number of all tuples, including duplicates and nulls.
identifier.`*`	The number of all tuples that match the correlation variable *identifier*, including duplicates and nulls. Note the following: ■ `count(A.*) = 1` is true for the first event that matches `A`. ■ `count(A.*) = 0` is true while `A` has not been matched yet.
identifier.*attr*	The number of tuples that match correlation variable *identifier*, where *attr* is not `NULL`.

Consider Example 21–8. Assume that the schema of `S` includes attributes `account` and `balance`. This query returns an event for each `account` that has not received 3 or more events in 60 minutes.

Example 21–8 MATCH_RECOGNIZE Query Using count(A.*)

```
select
    T.account,
    T.Atime
FROM S
    MATCH_RECOGNIZE(
        PARTITION BY account
        MEASURES
            A.account has account
            A.ELEMENT_TIME as Atime
        ALL MATCHES
        INCLUDE TIMER EVENTS
        PATTERN (A+)
        DURATION 60 MINUTES
        DEFINE
            A AS count(A.*) < 3
    ) as T
```

The `PATTERN (A+)` specifies the pattern "Match `A` one or more times".

The `DEFINE` clause specifies the condition:

```
A AS count(A.*) < 3
```

This condition for `A` places no restrictions on input tuples (such as `A.balance > 1000`). The only restrictions are imposed by the `PARTITION BY account` and `DURATION 60 MINUTES` clauses. In the `DEFINE` clause, the `A.*` means, "Match all input tuples for the group `A+`". This group includes the one or more input tuples with a particular `account` received in the 60 minutes starting with the first input tuple. The `count(A.*)` is a running aggregate that returns the total number of events in this group.

If the `DEFINE` clause specifies the condition:

```
A AS A.balance > 1000 and count(A.*) < 3
```

Then `A.*` still means "Match all input tuples for the group `A+`". In this case, this group includes the one or more input tuples with a particular `account` received in the 60 minutes starting with the first input tuple and with `balance > 1000`.

In contrast:

- `count(*)` means "The number of all tuples, including duplicates and nulls". That is, the number of all tuples received on `S`, whether they satisfy the `MATCH_RECOGNIZE` clause or not.
- `count(A.balance)` means "The number of all tuples that match the correlation variable `A` where the `balance` is not `NULL`".

For more information, see:

- "count" on page 9-5
- Section , "Range, Rows, and Slide at Query Start-Up and for Empty Relations"
- Section , "Referencing Singleton and Group Matches"
- Section , "Referencing Aggregates"
- Section , "Referencing Variables That Have not Been Matched Yet"
- Section , "Referencing Attributes not Qualified by Correlation Variable"

Using first and last

Use the `first` and `last` built-in aggregate functions to access event attributes of the first or last event match, respectively:

`first` returns the value of the first match of a group in the order defined by the `ORDER BY` clause or the default order.

`last` returns the value of the last match of a group in the order defined by the `ORDER BY` clause or the default order.

The `first` and `last` functions accept an optional non-negative, constant integer argument (`N`) that indicates the offset following the first and the offset preceding the last match of the variable, respectively. If you specify this offset, the `first` function returns the `N`-th matching event following the first match and the `last` function returns the `N`-th matching event preceding the last match. If the offset does not fall within the match of the variable, the `first` and `last` functions return `NULL`.

For more information, see:

- "first" on page 9-7
- "last" on page 9-9
- Section , "Referencing Aggregates"
- Section , "ORDER BY Clause"

Using prev

Use the `prev` built-in single-row function to access event attributes of a previous event match. If there is no previous event match, the `prev` function returns `NULL`.

The `prev` function accepts an optional non-negative, constant integer argument (`N`) that indicates the offset to a previous match. If you specify this offset, the `prev` function returns the `N`-th matching event preceding the current match. If there is no such previous match, the `prev` functions returns `NULL`.

When you use the prev function in the DEFINE clause, this function may only access the currently defined correlation variable.

For example: the correlation variable definition in Example 21–9 is valid:

Example 21–9 Use of the prev Function: Valid

```
Y AS Y.price < prev(Y.price, 2)
```

However, the correlation variable definition in Example 21–10 is invalid because while defining correlation variable Y, it references correlation variable X inside the prev function.

Example 21–10 Use of the prev Function: Invalid

```
Y AS Y.price < prev(X.price, 2)
```

For more information, see:

- "prev" on page 8-10
- Section , "DEFINE Clause"

MEASURES Clause

The MEASURES clause exports (makes available for inclusion in the SELECT) attribute values of events that successfully match the pattern you specify.

You may specify expressions over correlation variables that reference partition attributes, order by attributes, singleton variables and aggregates on group variables, and aggregates on the attributes of the stream that is the source of the MATCH_RECOGNIZE clause.

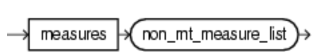
You qualify attribute values by correlation variable to export the value of the attribute for the event that matches the correlation variable's condition. For example, within the MEASURES clause, A.c1 refers to the value of event attribute c1:

- In the tuple that last matched the condition corresponding to correlation variable A, if A is specified in the DEFINE clause.
- In the last processed tuple, if A is not specified in the DEFINE clause.

  This is because if A is not specified in the DEFINE clause, then A is considered as TRUE always. So effectively all the tuples in the input match to A.

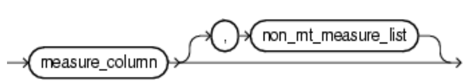
You may include in the SELECT statement only attributes you specify in the MEASURES clause.

***pattern_measures_clause*::=**



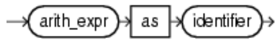
(*non_mt_measure_list::=* on page 21-9)

***non_mt_measure_list*::=**



(*measure_column::=* on page 21-10)

***measure_column*::=**



(*arith_expr::=* on page 5-6, *identifier::=* on page 7-17)

In Example 21–1, the *`pattern_measures_clause`* is:

```
MEASURES
    A.itemId as itemId
```

This section describes:

- Section , "Functions Over Correlation Variables in the MEASURES Clause"

For more information, see:

- Section , "Referencing Singleton and Group Matches"
- Section , "Referencing Aggregates"
- Section , "DEFINE Clause"
- Section , "Functions"

Functions Over Correlation Variables in the MEASURES Clause

In the `MEASURES` clause, you may apply any single-row or aggregate function to the attributes of events that match a condition.

Example 21–11 applies the `last` function over correlation variable `Z.c1` in the `MEASURES` clause.

Example 21–11 Using Functions Over Correlation Variables

```
<query id="tkpattern_q41"><![CDATA[
    select
        T.firstW, T.lastZ
    from
        tkpattern_S11
    MATCH_RECOGNIZE (
        MEASURES A.c1 as firstW, last(Z.c1) as lastZ
        ALL MATCHES
        PATTERN(A W+ X+ Y+ Z+)
        DEFINE
            W as W.c2 < prev(W.c2),
            X as X.c2 > prev(X.c2),
            Y as Y.c2 < prev(Y.c2),
            Z as Z.c2 > prev(Z.c2)
    ) as T
]]></query>
```

Note the following in the `MEASURES` clause in Example 21–11:

- `A.c1` will export the value of `c1` in the first and only the first event that the query processes because:
  - `A` is not specified in the `DEFINE` clause, therefor it is always true.
  - `A` has no pattern quantifiers, therefor it will match exactly once.
- The built-in aggregate function `last` will export the value of `c1` in the last event that matched `Z` at the time the `PATTERN` clause was satisfied.

For more information, see:

- Section , "Referencing Aggregates"

- Section , "Using count With *, identifier.*, and identifier.attr"
- Section , "Using first and last"
- Section , "Using prev"

PATTERN Clause

The `PATTERN` clause specifies the pattern to be matched as a regular expression over one ore more correlation variables.

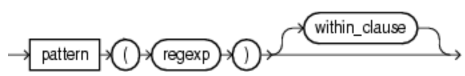
Incoming events must match these conditions in the order given (from left to right).

The regular expression may contain correlation variables that are:

- Defined in the `DEFINE` clause: such correlation variables are considered true only if their condition definition evaluates to `TRUE`.

  See Section , "DEFINE Clause".
- Not defined in the `DEFINE` clause: such correlation variables are considered as always `TRUE`; that is, they match every input.

***pattern_clause*::=**



(*regexp::=* on page 21-11, *within_clause::=* on page 21-21)

This section describes:

- Section , "Pattern Quantifiers and Regular Expressions"
- Section , "Grouping and Alternation in the PATTERN Clause"

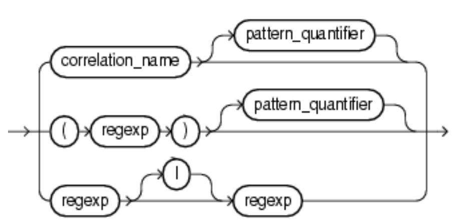
For more information, see:

- Section , "Pattern Detection"
- Section , "Pattern Detection With PARTITION BY"
- Section , "Pattern Detection With Aggregates"

Pattern Quantifiers and Regular Expressions

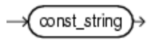
You express the pattern as a regular expression composed of correlation variables and pattern quantifiers.

***regexp*::=**



(*correlation_name::=* on page 21-12, *pattern_quantifier::=* on page 21-12)

correlation_name::=



(*const_string::=* on page 7-13)

pattern_quantifier::=

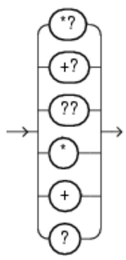


Table 21–2 lists the pattern quantifiers (*pattern_quantifier::=* on page 21-12) Oracle CQL supports.

Table 21–2 MATCH_RECOGNIZE Pattern Quantifiers

Maximal	Minimal	Description
`*`	`*?`	0 or more times
`+`	`+?`	1 or more times.
`?`	`??`	0 or 1 time.
None	None	An unquantified pattern, such as `A`, is assumed to have a quantifier that requires exactly 1 match.

Use the pattern quantifiers to specify the pattern as a regular expression, such as `A*` or `A+?`.

The one-character pattern quantifiers are maximal or "greedy"; they will attempt to match as many instances of the regular expression on which they are applied as possible.

The two-character pattern quantifiers are minimal or "reluctant"; they will attempt to match as few instances of the regular expression on which they are applied as possible.

Consider the following *`pattern_clause`*:

```
PATTERN (A B* C)
```

This pattern clause means a pattern match will be recognized when the following conditions are met by consecutive incoming input tuples:

1. Exactly one tuple matches the condition that defines correlation variable `A`, followed by
2. Zero or more tuples that match the correlation variable `B`, followed by
3. Exactly one tuple that matches correlation variable `C`.

While in state 2, if a tuple arrives that matches both the correlation variables `B` and `C` (since it satisfies the defining conditions of both of them) then as the quantifier `*` for `B` is greedy that tuple will be considered to have matched `B` instead of `C`. Thus due to the greedy property `B` gets preference over `C` and we match a greater number of `B`. Had the pattern expression be `A B*? C`, one that uses a lazy or reluctant quantifier over `B`, then a tuple matching both B and C will be treated as matching C only. Thus C would get preference over B and we will match fewer B.

For more information, see:

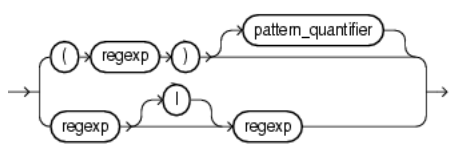
- Section , "Referencing Singleton and Group Matches"
- Section , "Grouping and Alternation in the PATTERN Clause"

Grouping and Alternation in the PATTERN Clause

As shown in the *regexp_grp_alt* syntax, you can use:

- open and close round brackets ((and)) to group correlation variables
- alternation operators (|) to match either one correlation variable (or group of correlation variables) or another

***regexp_grp_alt*::=**



(*correlation_name::=* on page 21-12, *pattern_quantifier::=* on page 21-12, *regexp::=* on page 21-11)

Consider the following *pattern_clause*:

```
PATTERN (A+ B+)
```

This means "A one or more times followed by B one or more times".

You can group correlation variables. For example:

```
PATTERN (A+ (C+ B+)*)
```

This means "A one or more times followed by zero or more occurrences of C one or more times and B one or more times".

Using the PATTERN clause alternation operator (|), you can refine the sense of the *pattern_clause*. For example:

```
PATTERN (A+ | B+)
```

This means "A one or more times or B one or more times, whichever comes first".

Similarly, you can both group correlation variables and use the alternation operator. For example:

```
PATTERN (A+ (C+ | B+))
```

This means "A one or more times followed by either C one or more times or B one or more times, whichever comes first".

To match every permutation you can use:

```
PATTERN ((A B) | (B A))
```

This means "A followed by B or B followed by A, which ever comes first".

For more information, see:

- Section , "Pattern Quantifiers and Regular Expressions"

- "Alternation Operator" on page 4-5

DEFINE Clause

The DEFINE clause specifies the boolean condition for each correlation variable.

You may specify any logical or arithmetic expression and apply any single-row or aggregate function to the attributes of events that match a condition.

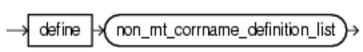
On receiving a new tuple from the base stream, the conditions of the correlation variables that are relevant at that point in time are evaluated. A tuple is said to have matched a correlation variable if it satisfies its defining condition. A particular input can match zero, one, or more correlation variables. The relevant conditions to be evaluated on receiving an input are determined by logic governed by the PATTERN clause regular expression and the state in pattern recognition process that we have reached after processing the earlier inputs.

The condition can refer to any of the attributes of the schema of the stream or view that evaluates to a stream on which the MATCH_RECOGNIZE clause is being applied.

A correlation variable in the PATTERN clause need not be specified in the DEFINE clause: the default for such a correlation variable is a predicate that is always true. Such a correlation variable matches every event. It is an error to specify a correlation variable in the DEFINE clause which is not used in a PATTERN clause

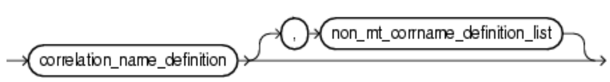
No correlation variable defined by a SUBSET clause may be defined in the DEFINE clause.

pattern_definition_clause::=



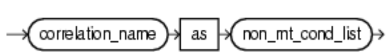
(*non_mt_corrname_definition_list::=* on page 21-14)

non_mt_corrname_definition_list::=



(*correlation_name_definition::=* on page 21-14)

correlation_name_definition::=



(*correlation_name::=* on page 21-12, *non_mt_cond_list::=* on page 7-25)

This section describes:

- Section , "Functions Over Correlation Variables in the DEFINE Clause"
- Section , "Referencing Attributes in the DEFINE Clause"
- Section , "Referencing One Correlation Variable From Another in the DEFINE Clause"

For more information, see:

- Section , "Referencing Singleton and Group Matches"
- Section , "Referencing Aggregates"

- Section , "Using first and last"
- Section , "Using prev"
- Section , "PATTERN Clause"
- Section , "SUBSET Clause"
- Section , "Functions"

Functions Over Correlation Variables in the DEFINE Clause

You can use functions over the correlation variables while defining them.

Example 21–12 applies the `to_timestamp` function to correlation variables.

Example 21–12 Using Functions Over Correlation Variables: to_timestamp

```
...
      PATTERN (A B* C)
      DEFINE
          A  AS  (A.temp >= 25),
          B  AS  ((B.temp >= 25) and (to_timestamp(B.element_time) - to_timestamp(A.element_time) < INTERVAL "0
00:00:05.00" DAY TO SECOND)),
          C  AS  (to_timestamp(C.element_time) - to_timestamp(A.element_time) >= INTERVAL "0 00:00:05.00" DAY TO
SECOND)
...
```

Example 21–13 applies the `count` function to correlation variable `B` to count the number of times its definition was satisfied. A match is recognized when `totalCountValue` is less than 1000 two or more times in 30 minutes.

Example 21–13 Using Functions Over Correlation Variables: count

```
...
    MATCH_RECOGNIZE(
        ...
        PATTERN(B*)
        DURATION 30 MINUTES
        DEFINE
            B as (B.totalCountValue < 1000 and count(B.*) >= 2)
...
```

For more information, see:

- Section , "Referencing Aggregates"
- Section , "Using count With *, identifier.*, and identifier.attr"
- Section , "Using first and last"
- Section , "Using prev"

Referencing Attributes in the DEFINE Clause

You can refer to the attributes of a base stream:

- Without a correlation variable: `c1 < 20`.
- With a correlation variable: `A.c1 < 20`.

When you refer to the attributes without a correlation variable, a tuple that last matched any of the correlation variables is consulted for evaluation.

Consider the following definitions:

- `DEFINE A as c1 < 20`
- `DEFINE A as A.c1 < 20`

Both refer to `c1` in the same tuple which is the latest input tuple. This is because on receiving an input we evaluate the condition of a correlation variable assuming that the latest input matches that correlation variable.

If you specify a correlation name that is not defined in the `DEFINE` clause, it is considered to be true for every input.

In Example 21–14, correlation variable `A` appears in the `PATTERN` clause but is not specified in the `DEFINE` clause. This means the correlation name `A` is true for every input. It is an error to define a correlation name which is not used in a `PATTERN` clause.

Example 21–14 Undefined Correlation Name

```
<query id="q"><![CDATA[
    SELECT
        T.firstW,
        T.lastZ
    FROM
        S2
    MATCH_RECOGNIZE (
        MEASURES
            A.c1 as firstW,
            last(Z) as lastZ
        PATTERN(A W+ X+ Y+ Z+)
        DEFINE
            W as W.c2 < prev(W.c2),
            X as X.c2 > prev(X.c2),
            Y as Y.c2 < prev(Y.c2),
            Z as Z.c2 > prev(Z.c2)
    ) as T
]]></query>
```

For more information, see:

- Section , "Referencing One Correlation Variable From Another in the DEFINE Clause"
- Section , "Referencing Singleton and Group Matches"
- Section , "Referencing Variables That Have not Been Matched Yet"
- Section , "Referencing Attributes not Qualified by Correlation Variable"
- Section , "PATTERN Clause"

Referencing One Correlation Variable From Another in the DEFINE Clause

A definition of one correlation variable can refer to another correlation variable. Consider the query that Example 21–15 shows:

Example 21–15 Referencing One Correlation Variable From Another

```
...
Select
    a_firsttime, d_lasttime, b_avgprice, d_avgprice
FROM
    S
MATCH_RECOGNIZE (
    PARTITION BY symbol
    MEASURES
```

```
        first(a.time) as a_firsttime,
        last(d.time) as d_lasttime,
        avg(b.price) as b_avgprice,
        avg(d.price) as d_avgprice
    PATTERN (A B+ C+ D)
    DEFINE
        A as A.price > 100,
        B as B.price > A.price,
        C as C.price < avg(B.price),
        D as D.price > prev(D.price)
)
...
```

Note the following:

- Because correlation variable `A` defines a single attribute, `B` can refer to this single attribute.
- Because `B` defines more than one attribute, `C` cannot reference a single attribute of `B`. In this case, `C` may only reference an aggregate of `B`.
- `D` is defined in terms of itself: in this case, you may refer to a single attribute or an aggregate. In this example, the `prev` function is used to access the match of `D` prior to the current match.

For more information, see:

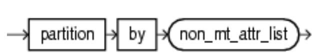
- Section , "Referencing Attributes in the DEFINE Clause"
- Section , "Referencing Singleton and Group Matches"
- Section , "Referencing Variables That Have not Been Matched Yet"
- Section , "Referencing Attributes not Qualified by Correlation Variable"
- Section , "Referencing Attributes in the DEFINE Clause"

PARTITION BY Clause

Use this optional clause to specify the stream attributes by which a `MATCH_RECOGNIZE` clause should partition its results.

Without a `PARTITION BY` clause, all stream attributes belong to the same partition.

pattern_partition_clause::=



(*non_mt_attr_list::=* on page 7-22)

In Example 21–1, the *`pattern_partition_clause`* is:

```
PARTITION BY
    itemId
```

The partition by clause in pattern means the input stream is logically divided based on the attributes mentioned in the partition list and pattern matching is done within a partition.

Consider a stream `S` with schema `(c1 integer, c2 integer)` with the input data that Example 21–16 shows.

Example 21–16 Input Stream S1

```
     c1  c2
1000 10, 1
2000 10, 2
3000 20, 2
4000 20, 1
```

Consider the `MATCH_RECOGNIZE` query that Example 21–17 shows.

Example 21–17 MATCH_RECOGNIZE Query Using Input Stream S1

```
select T.p1, T.p2, T.p3 from S MATCH_RECOGNIZE(
    MEASURES
        A.ELEMENT_TIME as p1,
        B.ELEMENT_TIME as p2
        B.c2 as p3
    PATTERN (A B)
    DEFINE
        A as A.c1 = 10,
        B as B.c1 = 20
) as T
```

This query would output the following:

```
3000:+ 2000, 3000, 2
```

If we add `PARTITION BY c2` to the query that Example 21–17 shows, then the output would change to:

```
3000:+ 2000, 3000, 2
4000:+ 1000, 4000, 1
```

This is because by adding the `PARTITION BY` clause, matches are done within partition only. Tuples at 1000 and 4000 belong to one partition and tuples at 2000 and 3000 belong to another partition owing to the value of `c2` attribute in them. In the first partition `A` matches tuple at 1000 and `B` matches tuple at 4000. Even though a tuple at 3000 matches the `B` definition, it is not presented as a match for the first partition since that tuple belongs to different partition.

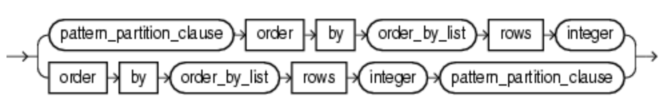
When you partition by more than one attribute, you can control the order of partitions using the `ORDER BY` clause. For more information, see Section , "ORDER BY Clause".

ORDER BY Clause

Use this optional clause to specify the stream attributes by which a `MATCH_RECOGNIZE` clause should order partitions when using a `PARTITION BY` clause.

Without an `ORDER BY` clause, the results of `MATCH_RECOGNIZE` are nondeterministic.

pattern_order_by_top_clause::=



You may only use the `ORDER BY` clause with a `PARTITION BY` clause.

For more information, see Section , "PARTITION BY Clause,"*pattern_partition_clause::=* on page 21-17, and *order_by_list::=* on page 22-5.

ALL MATCHES Clause

Use this optional clause to configure Oracle Event Processing to match overlapping patterns.

With the `ALL MATCHES` clause, Oracle Event Processing finds all possible matches. Matches may overlap and may start at the same event. In this case, there is no distinction between greedy and reluctant pattern quantifiers. For example, the following pattern:

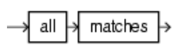
```
ALL MATCHES
PATTERN (A* B)
```

produces the same result as:

```
ALL MATCHES
PATTERN (A*? B)
```

Without the `ALL MATCHES` clause, overlapping matches are not returned, and quantifiers such as the asterisk determine which among a set of candidate (and overlapping) matches is the preferred one for output. The rest of the overlapping matches are discarded.

pattern_skip_match_clause::=



Consider the query `tkpattern_q41` in Example 21–18 that uses `ALL MATCHES` and the data stream `tkpattern_S11` in Example 21–19. Stream `tkpattern_S11` has schema `(c1 integer, c2 integer)`. The query returns the stream in Example 21–20.

The query `tkpattern_q41` in Example 21–18 will report a match when the input stream values, when plotted, form the shape of the English letter **W**. The relation in Example 21–20 shows an example of overlapping instances of this W-pattern match.

There are two types of overlapping pattern instances:

- Total: Example of total overlapping: Rows from time 3000-9000 and 4000-9000 in the input, both match the given pattern expression. Here the longest one (3000-9000) will be preferred if `ALL MATCHES` clause is not present.
- Partial: Example of Partial overlapping: Rows from time 12000-21000 and 16000-23000 in the input, both match the given pattern expression. Here the one which appears earlier is preferred when `ALL MATCHES` clause is not present. This is because when `ALL MATCHES` clause is omitted, we start looking for the next instance of pattern match at a tuple which is next to the last tuple in the previous matched instance of the pattern.

Example 21–18 ALL MATCHES Clause Query

```
<query id="tkpattern_q41"><![CDATA[
    select
        T.firstW, T.lastZ
    from
        tkpattern_S11
    MATCH_RECOGNIZE (
        MEASURES A.c1 as firstW, last(Z.c1) as lastZ
        ALL MATCHES
        PATTERN(A W+ X+ Y+ Z+)
        DEFINE
            W as W.c2 < prev(W.c2),
            X as X.c2 > prev(X.c2),
```

```
            Y as Y.c2 < prev(Y.c2),
            Z as Z.c2 > prev(Z.c2)
    ) as T
]]></query>
```

Example 21–19 ALL MATCHES Clause Stream Input

```
Timestamp   Tuple
 1000         1,8
 2000         2,8
 3000         3,8
 4000         4,6
 5000         5,3
 6000         6,7
 7000         7,6
 8000         8,2
 9000         9,6
10000        10,2
11000        11,9
12000        12,9
13000        13,8
14000        14,5
15000        15,0
16000        16,9
17000        17,2
18000        18,0
19000        19,2
20000        20,3
21000        21,8
22000        22,5
23000        23,9
24000        24,9
25000        25,4
26000        26,7
27000        27,2
28000        28,8
29000        29,0
30000        30,4
31000        31,4
32000        32,7
33000        33,8
34000        34,6
35000        35,4
36000        36,5
37000        37,1
38000        38,7
39000        39,5
40000        40,8
41000        41,6
42000        42,6
43000        43,0
44000        44,6
45000        45,8
46000        46,4
47000        47,3
48000        48,8
49000        49,2
50000        50,5
51000        51,3
52000        52,3
53000        53,9
54000        54,8
55000        55,5
56000        56,5
57000        57,9
58000        58,7
```

```
59000        59,3
60000        60,3
```

Example 21–20   ALL MATCHES Clause Stream Output

```
Timestamp   Tuple Kind  Tuple
 9000:      +      3,9
 9000:      +      4,9
11000:      +      6,11
11000:      +      7,11
19000:      +     12,19
19000:      +     13,19
19000:      +     14,19
20000:      +     12,20
20000:      +     13,20
20000:      +     14,20
21000:      +     12,21
21000:      +     13,21
21000:      +     14,21
23000:      +     16,23
23000:      +     17,23
28000:      +     24,28
30000:      +     26,30
38000:      +     33,38
38000:      +     34,38
40000:      +     36,40
48000:      +     42,48
50000:      +     45,50
50000:      +     46,50
```

As Example 21–20 shows, the `ALL MATCHES` clause reports all the matched pattern instances on receiving a particular input. For example, at time 20000, all of the tuples `{12,20}`, `{13,20}`, and `{14,20}` are output.

For more information, see Section , "Pattern Quantifiers and Regular Expressions".

WITHIN Clause

The `WITHIN` clause is an optional clause that outputs a *`pattern_clause`* match if and only if the match occurs within the specified time duration.

***within_clause*::=**

(*time_spec::=* on page 7-30)

That is, if and only if:

```
TL - TF < WD
```

Where:

- `TL` - Timestamp of last event matching the pattern.
- `TF` - Timestamp of first event matching the pattern.
- `WD` - Duration specified in the `WITHIN` clause.

The `WITHIN INCLUSIVE` clause tries to match events at the boundary case as well. That is, it outputs a match if and only if:

```
TL - TF <= WD
```

If the match completes within the specified time duration, then the event is output as soon as it happens. That is, if the match can be output, it is output with the timestamp at which it completes. The WITHIN clause does not wait for the time duration to expire as the DURATION clause does.

When the WITHIN clause duration expires, it discards any potential candidate matches which are incomplete.

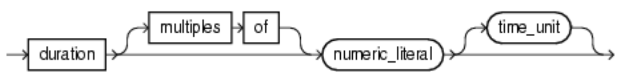
For more information, see Section , "Pattern Detection With the WITHIN Clause".

> **Note:** You cannot use a WITHIN clause with a DURATION clause. For more information, see Section , "DURATION Clause".

DURATION Clause

The DURATION clause is an optional clause that you should use only when writing a query involving non-event detection. Non-event detection is the detection of a situation when a certain event which should have occurred in a particular time limit does not occur in that time frame.

duration_clause::=



(*time_unit::=* on page 7-30)

Using this clause, a match is reported only when the regular expression in the PATTERN clause is matched completely and no other event or input arrives until the duration specified in the DURATION clause expires. The duration is measured from the time of arrival of the first event in the pattern match.

You must use the INCLUDE TIMER EVENTS clause when using the DURATION clause. For more information, see Section , "INCLUDE TIMER EVENTS Clause".

This section describes:

- Section , "Fixed Duration Non-Event Detection"
- Section , "Recurring Non-Event Detection"

> **Note:** You cannot use a DURATION clause with a WITHIN clause. For more information, see Section , "WITHIN Clause".

Fixed Duration Non-Event Detection

The duration can be specified as a constant value, such as 10. Optionally, you may specify a time unit such as seconds or minutes (see *time_unit::=* on page 7-30); the default time unit is seconds.

Consider the query tkpattern_q59 in Example 21–21 that uses DURATION 10 to specify a delay of 10 s (10000 ms) and the data stream tkpattern_S19 in Example 21–22. Stream tkpattern_S19 has schema (c1 integer). The query returns the stream in Example 21–23.

Example 21–21 MATCH_RECOGNIZE with Fixed Duration DURATION Clause Query

```
<query id="BBAQuery"><![CDATA[
    select
        T.p1, T.p2
    from
        tkpattern_S19
    MATCH_RECOGNIZE (
        MEASURES A.c1 as p1, B.c1 as p2
        include timer events
        PATTERN(A B*)
        duration 10
        DEFINE A as A.c1 = 10, B as B.c1 != A.c1
    ) as T
]]></query>
```

Example 21–22 MATCH_RECOGNIZE with Fixed Duration DURATION Clause Stream Input

```
Timestamp   Tuple
 1000        10
 4000        22
 6000       444
 7000        83
 9000        88
11000        12
11000        22
11000        15
12000        13
15000        10
27000        11
28000        10
30000        18
40000        10
44000        19
52000        10
h 100000
```

Example 21–23 MATCH_RECOGNIZE with Fixed DURATION Clause Stream Output

```
Timestamp   Tuple Kind  Tuple
11000:      +           10,88
25000:      +           10,
38000:      +           10,18
50000:      +           10,19
62000:      +           10,
```

The tuple at time 1000 matches `A`.

Since the duration is 10 we output a match as soon as input at time `1000+10000=11000` is received (the one with the value 12). Since the sequence of tuples from 1000 through 9000 match the pattern `AB*` and nothing else a match is reported as soon as input at time 11000 is received.

The next match starts at 15000 with the tuple at that time matching `A`. The next tuple that arrives is at 27000. So here also we have tuples satisfying pattern `AB*` and nothing else and hence a match is reported at time `15000+10000=25000`. Further output is generated by following similar logic.

For more information, see "Fixed Duration Non-Event Detection" on page 21-33.

Recurring Non-Event Detection

When you specify a `MULTIPLES OF` clause, it indicates recurring non-event detection. In this case an output is sent at the multiples of duration value as long as there is no event after the pattern matches completely.

Consider the query `tkpattern_q75` in Example 21–24 that uses `DURATION MULTIPLES OF 10` to specify a delay of 10 s (10000 ms) and the data stream `tkpattern_S23` in Example 21–25. Stream `tkpattern_S23` has schema `(c1 integer)`. The query returns the stream in Example 21–26.

Example 21–24 MATCH_RECOGNIZE with Variable Duration DURATION MULTIPLES OF Clause Query

```
<query id="tkpattern_q75"><![CDATA[
    select
        T.p1, T.p2, T.p3
    from
        tkpattern_S23
    MATCH_RECOGNIZE (
        MEASURES A.c1 as p1, B.c1 as p2, sum(B.c1) as p3
        ALL MATCHES
        include timer events
        PATTERN(A B*)
        duration multiples of 10
        DEFINE A as A.c1 = 10, B as B.c1 != A.c1
    ) as T
]]></query>
```

Example 21–25 MATCH_RECOGNIZE with Variable Duration DURATION MULTIPLES OF Clause Stream Input

```
Timestamp   Tuple
 1000        10
 4000        22
 6000       444
 7000        83
 9000        88
11000        12
11000        22
11000        15
12000        13
15000        10
27000        11
28000        10
30000        18
44000        19
62000        20
72000        10
h 120000
```

Example 21–26 MATCH_RECOGNIZE with Variable Duration DURATION MULTIPLES OF Clause Stream Output

```
Timestamp   Tuple Kind  Tuple
 11000:     +           10,88,637
 25000:     +           10,,
 38000:     +           10,18,18
 48000:     +           10,19,37
 58000:     +           10,19,37
 68000:     +           10,20,57
 82000:     +           10,,
 92000:     +           10,,
102000:     +           10,,
112000:     +           10,,
```

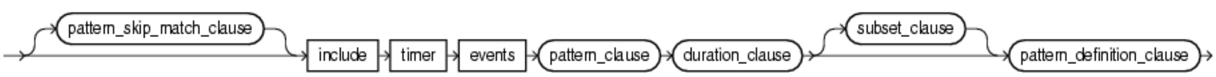
The execution here follows similar logic to that of the example above for just the `DURATION` clause (see "Fixed Duration Non-Event Detection" on page 21-22). The difference comes for the later outputs. The tuple at 72000 matches `A` and then there is nothing else after that. So the pattern `AB*` is matched and we get output at 82000. Since we have the `MULTIPLES OF` clause and duration 10 we see outputs at time 92000, 102000, and so on.

INCLUDE TIMER EVENTS Clause

Use this clause in conjunction with the `DURATION` clause for non-event detection queries.

Typically, in most pattern match queries, a pattern match output is always triggered by an input event on the input stream over which pattern is being matched. The only exception is non-event detection queries where there could be an output triggered by a timer expiry event (as opposed to an explicit input event on the input stream).

pattern_inc_timer_evs_clause::=



(*pattern_clause::=* on page 21-11, *pattern_skip_match_clause::=* on page 21-19, *pattern_definition_clause::=* on page 21-14, *duration_clause::=* on page 21-22, *subset_clause::=* on page 21-25)

For more information, see Section , "DURATION Clause".

SUBSET Clause

Using this clause, you can group together one or more correlation variables that are defined in the `DEFINE` clause. You can use this named subset in the `MEASURES` and `DEFINE` clauses just like any other correlation variable.

For example:

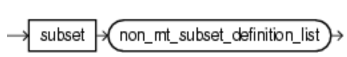
```
SUBSET S1 = (Z,X)
```

The right-hand side of the subset (`(Z,X)`) is a comma-separated list of one or more correlation variables as defined in the `PATTERN` clause.

The left-hand side of the subset (`S1`) is the union of the correlation variables on the right-hand side.

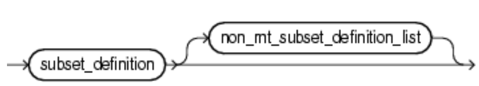
You cannot include a subset variable in the right-hand side of a subset.

subset_clause::=



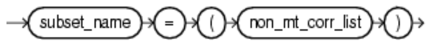
(*non_mt_subset_definition_list::=* on page 21-25)

non_mt_subset_definition_list::=



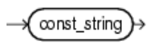
(*subset_definition::=* on page 21-26)

subset_definition::=



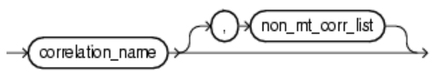
(*subset_name::=* on page 21-26, *non_mt_corr_list::=* on page 21-26)

subset_name::=



(*const_string::=* on page 7-13)

non_mt_corr_list::=



(*correlation_name::=* on page 21-12)

Consider the query `q55` in Example 21–27 and the data stream `S11` in Example 21–28. Stream `S11` has schema `(c1 integer, c2 integer)`. This example defines subsets `S1` through `S6`. This query outputs a match if the `c2` attribute values in the input stream form the shape of the English letter **W**. Now suppose we want to know the sum of the values of `c2` for those tuples which form the incrementing arms of this **W** shape. The correlation variable `X` represents tuples that are part of the first incrementing arm and `Z` represent the tuples that are part of the second incrementing arm. So we need some way to group the tuples that match both. Such a requirement can be captured by defining a `SUBSET` clause as the example shows.

Subset `S4` is defined as `(X,Z)`. It refers to the tuples in the input stream that match either `X` or `Z`. This subset is used in the `MEASURES` clause statement `sum(S4.c2) as sumIncrArm`. This computes the sum of the value of `c2` attribute in the tuples that match either `X` or `Z`. A reference to `S4.c2` in a `DEFINE` clause like `S4.c2 = 10` will refer to the value of `c2` in the latest among the last tuple that matched `X` and the last tuple that matched `Z`.

Subset `S6` is defined as `(Y)`. It refers to all the tuples that match correlation variable `Y`.

The query returns the stream in Example 21–29.

Example 21–27 MATCH_RECOGNIZE with SUBSET Clause Query

```
<query id="q55"><![CDATA[
    select
        T.firstW,
        T.lastZ,
        T.sumDecrArm,
        T.sumIncrArm,
        T.overallAvg
    from
        S11
    MATCH_RECOGNIZE (
        MEASURES
            S2.c1 as firstW,
            last(S1.c1) as lastZ,
            sum(S3.c2) as sumDecrArm,
            sum(S4.c2) as sumIncrArm,
            avg(S5.c2) as overallAvg
```

```
        PATTERN(A W+ X+ Y+ Z+)
        SUBSET S1 = (Z) S2 = (A) S3 = (A,W,Y) S4 = (X,Z) S5 = (A,W,X,Y,Z) S6 = (Y)
        DEFINE
            W as W.c2 < prev(W.c2),
            X as X.c2 > prev(X.c2),
            Y as S6.c2 < prev(Y.c2),
            Z as Z.c2 > prev(Z.c2)
    ) as T
]]></query>
```

Example 21–28 MATCH_RECOGNIZE with SUBSET Clause Stream Input

```
Timestamp   Tuple
 1000        1,8
 2000        2,8
 3000        3,8
 4000        4,6
 5000        5,3
 6000        6,7
 7000        7,6
 8000        8,2
 9000        9,6
10000       10,2
11000       11,9
12000       12,9
13000       13,8
14000       14,5
15000       15,0
16000       16,9
17000       17,2
18000       18,0
19000       19,2
20000       20,3
21000       21,8
22000       22,5
23000       23,9
24000       24,9
25000       25,4
26000       26,7
27000       27,2
28000       28,8
29000       29,0
30000       30,4
31000       31,4
32000       32,7
33000       33,8
34000       34,6
35000       35,4
36000       36,5
37000       37,1
38000       38,7
39000       39,5
40000       40,8
41000       41,6
42000       42,6
43000       43,0
44000       44,6
45000       45,8
46000       46,4
47000       47,3
48000       48,8
49000       49,2
50000       50,5
51000       51,3
52000       52,3
53000       53,9
```

```
54000        54,8
55000        55,5
56000        56,5
57000        57,9
58000        58,7
59000        59,3
60000        60,3
```

Example 21–29 MATCH_RECOGNIZE with SUBSET Clause Stream Output

```
Timestamp   Tuple Kind  Tuple
 9000:      +           3,9,25,13,5.428571
21000:      +           12,21,24,22,4.6
28000:      +           24,28,15,15,6.0
38000:      +           33,38,19,12,5.1666665
48000:      +           42,48,13,22,5.0
```

For more information, see:

- Section , "Running Aggregates and Final Aggregates"
- Section , "MEASURES Clause"
- Section , "PATTERN Clause"
- Section , "DEFINE Clause"

MATCH_RECOGNIZE Examples

The following examples illustrate basic MATCH_RECOGNIZE practices:

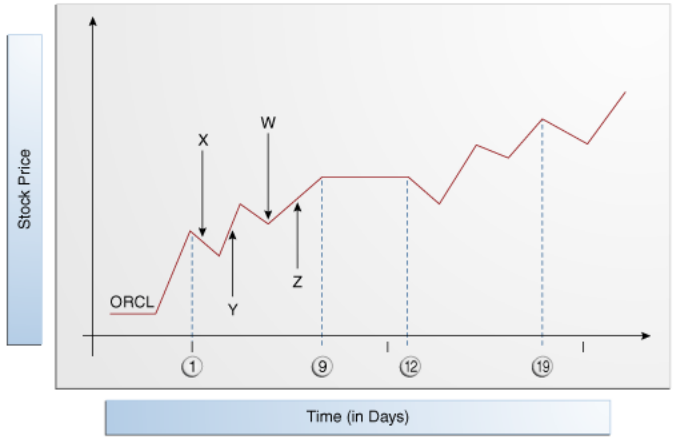
- "Pattern Detection" on page 21-28
- "Pattern Detection With PARTITION BY" on page 21-30
- "Pattern Detection With Aggregates" on page 21-31
- "Fixed Duration Non-Event Detection" on page 21-33

For more examples, see *Oracle Fusion Middleware Getting Started Guide for Oracle Event Processing*.

Pattern Detection

Consider the stock fluctuations that Figure 21–1 shows. This data can be represented as a stream of stock ticks (index number or time) and stock price. Figure 21–1 shows a common trading behavior known as a double bottom pattern between days 1 and 9 and between days 12 and 19. This pattern can be visualized as a W-shaped change in stock price: a fall (X), a rise (Y), a fall (W), and another rise (Z).

Figure 21–1 Pattern Detection: Double Bottom Stock Fluctuations



Example 21–30 shows a query `q` on stream `S2` of stock price events with schema `symbol`, `stockTick`, and `price`. This query detects double bottom patterns on the incoming stock trades using the `PATTERN` clause (`A W+ X+ Y+ Z+`). The correlation names in this clause are:

- `A`: corresponds to the start point of the double bottom pattern.

  Because correlation name `A` is true for every input, it is not defined in the `DEFINE` clause. If you specify a correlation name that is not defined in the `DEFINE` clause, it is considered to be true for every input.

- `W+`: corresponds to the first decreasing arm of the double bottom pattern.

  It is defined by `W.price < prev(W.price)`. This definition implies that the current price is less than the previous one.

- `X+`: corresponds to the first increasing arm of the double bottom pattern.
- `Y+`: corresponds to the second decreasing arm of the double bottom pattern.
- `Z+`: corresponds to the second increasing arm of the double bottom pattern.

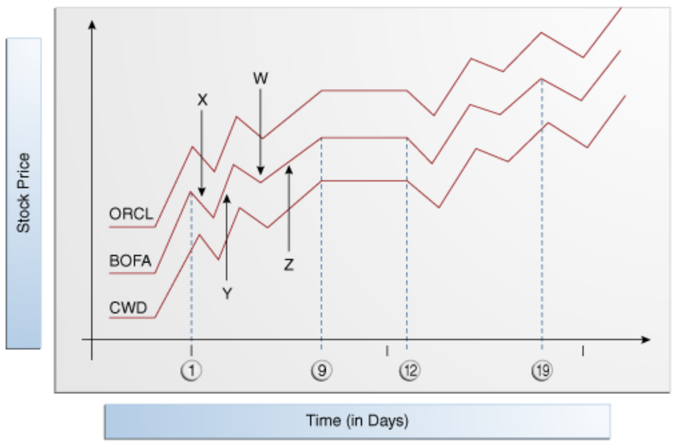
Example 21–30 Simple Pattern Detection: Query

```
<query id="q"><![CDATA[
    SELECT
        T.firstW,
        T.lastZ
    FROM
        S2
    MATCH_RECOGNIZE (
        MEASURES
            A.stockTick as firstW,
            last(Z) as lastZ
        PATTERN(A W+ X+ Y+ Z+)
        DEFINE
            W as W.price < prev(W.price),
            X as X.price > prev(X.price),
            Y as Y.price < prev(Y.price),
            Z as Z.price > prev(Z.price)
    ) as T
    WHERE
        S2.symbol = "oracle"
]]></query>
```

Pattern Detection With PARTITION BY

Consider the stock fluctuations that Figure 21–2 shows. This data can be represented as a stream of stock ticks (index number or time) and stock price. In this case, the stream contains data for more than one stock ticker symbol. Figure 21–2 shows a common trading behavior known as a double bottom pattern between days 1 and 9 and between days 12 and 19 for stock BOFA. This pattern can be visualized as a W-shaped change in stock price: a fall (X), a rise (Y), a fall (W), and another rise (Z).

Figure 21–2 Pattern Detection With Partition By: Stock Fluctuations



Example 21–31 shows a query q on stream S2 of stock price events with schema symbol, stockTick, and price. This query detects double bottom patterns on the incoming stock trades using the PATTERN clause (A W+ X+ Y+ Z+). The correlation names in this clause are:

- A: corresponds to the start point of the double bottom pattern.
- W+: corresponds to the first decreasing arm of the double bottom pattern as defined by W.price < prev(W.price), which implies that the current price is less than the previous one.
- X+: corresponds to the first increasing arm of the double bottom pattern.
- Y+: corresponds to the second decreasing arm of the double bottom pattern.
- Z+: corresponds to the second increasing arm of the double bottom pattern.

The query partitions the input stream by stock ticker symbol using the PARTITION BY clause and applies this PATTERN clause to each logical stream.

Example 21–31 Pattern Detection With PARTITION BY: Query

```
<query id="q"><![CDATA[
    SELECT
        T.firstW,
        T.lastZ
    FROM
        S2
    MATCH_RECOGNIZE (
        PARTITION BY
            A.symbol
        MEASURES
            A.stockTick as firstW,
            last(Z) as lastZ
        PATTERN(A W+ X+ Y+ Z+)
```

```
        DEFINE
            W as W.price < prev(W.price),
            X as X.price > prev(X.price),
            Y as Y.price < prev(Y.price),
            Z as Z.price > prev(Z.price)
    ) as T
]]></query>
```

Pattern Detection With Aggregates

Consider the query `q1` in Example 21–32 and the data stream `S` in Example 21–33. Stream `S` has schema `(c1 integer)`. The query returns the stream in Example 21–34.

Example 21–32 Pattern Detection With Aggregates: Query

```
<query id="q1"><![CDATA[
    SELECT
        T.sumB
    FROM
        S
    MATCH_RECOGNIZE (
        MEASURES
            sum(B.c1) as sumB
        PATTERN(A B* C)
        DEFINE
            A as ((A.c1 < 50) AND (A.c1 > 35)),
            B as B.c1 > avg(A.c1),
            C as C.c1 > prev(C.c1)
    ) as T
]]></query>
```

Example 21–33 Pattern Detection With Aggregates: Stream Input

```
Timestamp   Tuple
 1000       40
 2000       52
 3000       60
 4000       58
 5000       57
 6000       56
 7000       55
 8000       59
 9000       30
10000       40
11000       52
12000       60
13000       58
14000       57
15000       56
16000       55
17000       30
18000       10
19000       20
20000       30
21000       10
22000       25
23000       25
24000       25
25000       25
```

Example 21–34 Pattern Detection With Aggregates: Stream Output

```
Timestamp   Tuple
8000        338
12000       52
```

Pattern Detection With the WITHIN Clause

Consider the queries in Example 21–35 and Example 21–36 and the data stream S in Example 21–37. Stream S has schema (c1 integer, c2 integer). Table 21–3 compares the output of these queries.

Example 21–35 PATTERN Clause and WITHIN Clause

```
<query id="queryWithin"><![CDATA[
    SELECT T.Ac2, T.Bc2, T.Cc2
    FROM S
    MATCH_RECOGNIZE(
        MEASURES A.c2 as Ac2, B.c2 as Bc2, C.c2 as Cc2
        PATTERN (A (B+ | C)) within 3000 milliseconds
        DEFINE
            A as A.c1=10 or A.c1=25,
            B as B.c1=20 or B.c1=15 or B.c1=25,
            C as C.c1=15
    ) as T
]]></query>
```

Example 21–36 PATTERN Clause and WITHIN INCLUSIVE Clause

```
<query id="queryWithinInclusive"><![CDATA[
    SELECT T.Ac2, T.Bc2, T.Cc2
    FROM S
    MATCH_RECOGNIZE(
        MEASURES A.c2 as Ac2, B.c2 as Bc2, C.c2 as Cc2
        PATTERN (A (B+ | C)) within inclusive 3000 milliseconds
        DEFINE
            A as A.c1=10 or A.c1=25,
            B as B.c1=20 or B.c1=15 or B.c1=25,
            C as C.c1=15
    ) as T
]]></query>
```

Example 21–37 Pattern Detection With the WITHIN Clause: Stream Input

```
Timestamp   Tuple
  1000       10,100
h 2000
  3000       15,200
  3000       20,300
  4000       25,400
  5000       20,500
  6000       20,600
  7000       35,700
  8000       10,800
  9000       15,900
h 11000
 11000       20,1000
 11000       50,1100
```

Table 21–3 WITHIN and WITHIN INCLUSIVE Query Output

Query queryWithin	Query queryWithinInclusive
Timestamp Tuple Kind Tuple 3000: + 100,300, 6000: + 400,600, 9000: + 800,900,	Timestamp Tuple Kind Tuple 4000: + 100,400, 11000: + 800,1000,

As Table 21–3 shows for the queryWithin query, the candidate match starts with the event at TimeStamp=1000 and since the WITHIN clause duration is 3 seconds, the query will output the match only if it completes before the event at TimeStamp=4000. When the query receives the event at TimeStampe=4000, the longest match up to that point (since we are not using ALL MATCHES) is output. Note that although the event at TimeStamp=4000 matches B, it is not included in the match. The next match starts with the event at TimeStamp=4000 since that event also matches A and the previous match ends at TimeStamp=3000.

As Table 21–3 shows for the queryWithinInclusive query, the candidate match starts with the event at TimeStamp=1000. When the query receives the event at TimeStamp=4000, that event is included in the match because the query uses WITHIN INCLUSIVE and the event matches B. Note that although the event at TimeStamp=5000 matches B, the pattern is not grown further since it exceeds the duration (3 seconds) measured from the start of the match (TimeStamp=1000). Since this match ends at TimeStamp=4000 and we are not using ALL MATCHES, the next match does not start at TimeStamp=4000, even though it matches A.

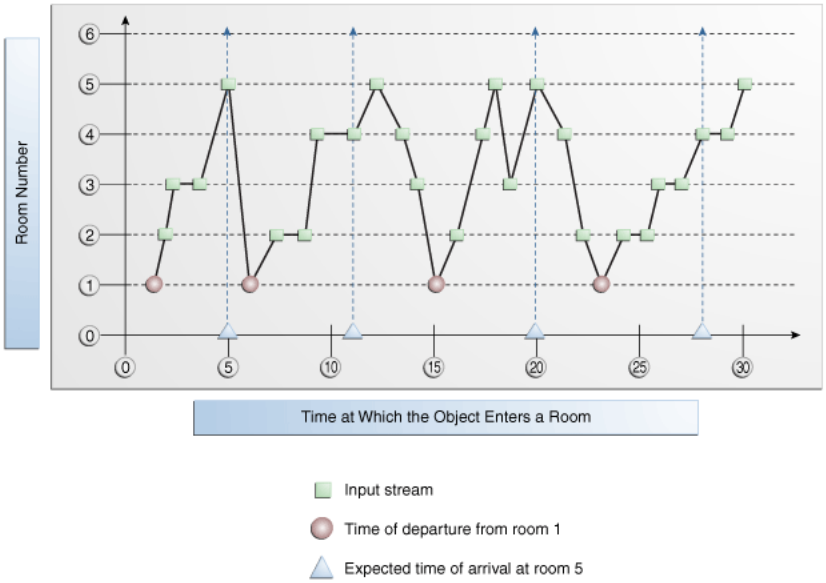
For more information, see:

- Section , "WITHIN Clause"
- Section , "ALL MATCHES Clause"

Fixed Duration Non-Event Detection

Consider an object that moves among five different rooms. Each time it starts from room 1, it must reach room 5 within 5 minutes. Figure 21–3 shows the object's performance. This data can be represented as a stream of time and room number. Note that when the object started from room 1 at time 1, it reached room 5 at time 5, as expected. However, when the object started from room 1 at time 6, it failed to reach room 5 at time 11; it reached room 5 at time 12. When the object started from room 1 at time 15, it was in room 5 at time 20, as expected. However, when the object started from room 1 at time 23, it failed to reach room 5 at time 28; it reached room 5 at time 30. The successes at times 5 and 20 are considered events: the arrival of the object in room 5 at the appropriate time. The failures at time 11 and 28 are considered non-events: the expected arrival event did not occur. Using Oracle CQL, you can query for such non-events.

Figure 21–3 Fixed Duration Non-Event Detection



Example 21–38 shows query `q` on stream `S` (with schema `c1` integer representing room number) that detects these non-events. Each time the object fails to reach room 5 within 5 minutes of leaving room 1, the query returns the time of departure from room 1.

Example 21–38 Fixed Duration Non-Event Detection: Query

```
<query id="q"><![CDATA[
select T.Atime FROM S
    MATCH_RECOGNIZE(
        MEASURES
            A.ELEMENT_TIME as Atime
       INCLUDE TIMER EVENTS
       PATTERN (A B*)
       DURATION 5 MINUTES
       DEFINE
           A as A.c1 = 1,
           B as B.c1 != 5
    ) as T
]]></query>
```

For more information, see Section , "DURATION Clause".

22

Oracle CQL Statements

This chapter describes data definition language (DDL) statements in Oracle Continuous Query Language (Oracle CQL).

This chapter includes the following section:

- Introduction to Oracle CQL Statements

Introduction to Oracle CQL Statements

Oracle CQL supports the following DDL statements:

- Query
- View

> **Note:** In stream input examples, lines beginning with `h` (such as `h 3800`) are heartbeat input tuples. These inform Oracle Event Processing that no further input will have a timestamp lesser than the heartbeat value.

For more information, see:

- Section , "Lexical Conventions"
- Section , "Syntactic Shortcuts and Defaults"
- Section , "Documentation Conventions"
- Chapter 2, "Basic Elements of Oracle CQL"
- Chapter 7, "Common Oracle CQL DDL Clauses"
- Chapter 20, "Oracle CQL Queries, Views, and Joins"

Query

Purpose

Use the query statement to define a Oracle CQL query that you reference by `identifier` in subsequent Oracle CQL statements.

Prerequisites

If your query references a stream or view, then the stream or view must already exist.

If the query already exists, Oracle Event Processing server throws an exception.

For more information, see:

- "View" on page 22-29
- Chapter 20, "Oracle CQL Queries, Views, and Joins"

Syntax

You express a query in a `<query></query>` element as Example 22–1 shows.

The `query` element has one attribute:

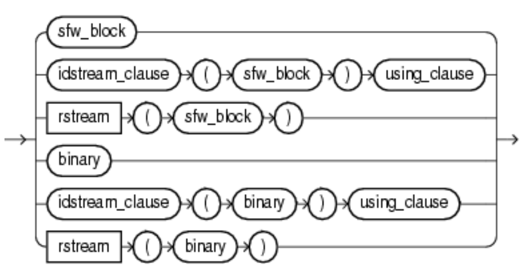
- `id`: Specify the `identifier` as the `query` element `id` attribute.

  The `id` value must conform with the specification given by *identifier::=* on page 7-17.

Example 22–1 Query in a <query></query> Element

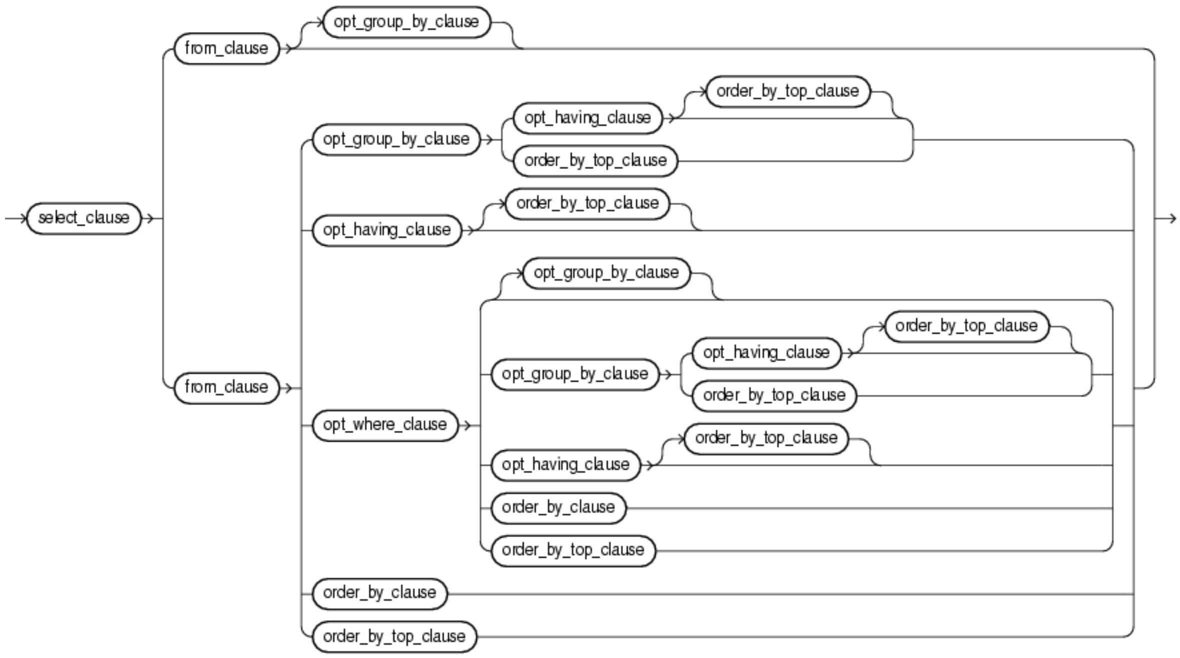
```
<query id="q0"><![CDATA[
    select * from OrderStream where orderAmount > 10000.0
]]></query>
```

***query*::=**



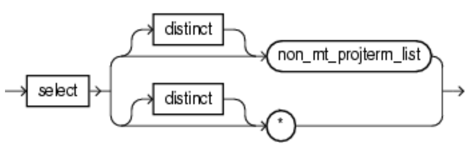
(*sfw_block::=* on page 22-3, *idstream_clause::=* on page 22-6, *binary::=* on page 22-6, *using_clause::=* on page 22-6)

sfw_block::=



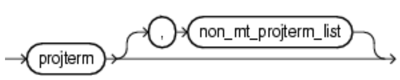
(*select_clause::=* on page 22-3, *from_clause::=* on page 22-3, *opt_where_clause::=* on page 22-4, *opt_group_by_clause::=* on page 22-5, *order_by_clause::=* on page 22-5, *order_by_top_clause::=* on page 22-5, *opt_having_clause::=* on page 22-5)

select_clause::=



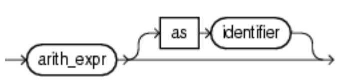
(*non_mt_projterm_list::=* on page 22-3)

non_mt_projterm_list::=



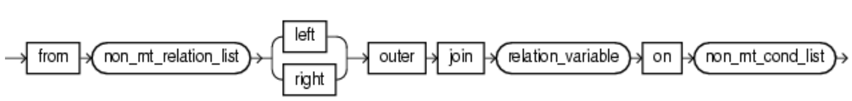
(*projterm::=* on page 22-3)

projterm::=



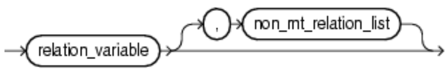
(*arith_expr* on page 5-6, *identifier::=* on page 7-17)

from_clause::=



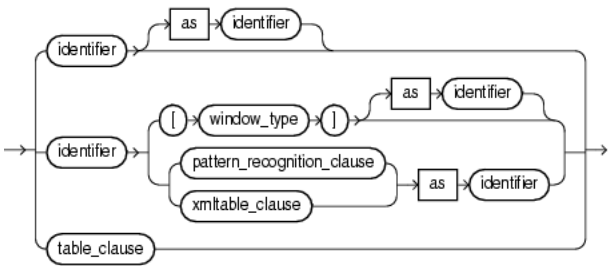
(*non_mt_relation_list::=* on page 22-4, *relation_variable::=* on page 22-4, *non_mt_cond_list::=* on page 7-25)

non_mt_relation_list::=



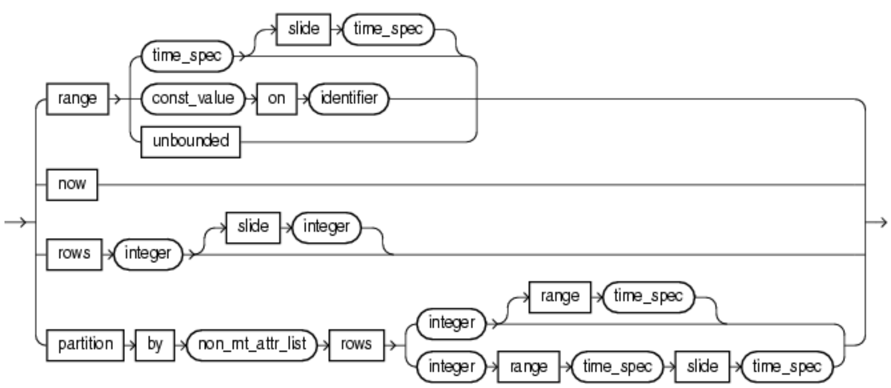
(*relation_variable::=* on page 22-4)

relation_variable::=



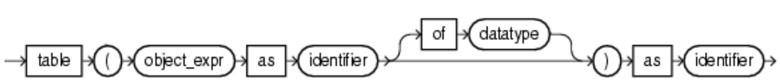
(*identifier::=* on page 7-17, *window_type::=* on page 22-4, *pattern_recognition_clause::=* on page 21-2, *xmltable_clause::=* on page 22-6, *object_expr::=* on page 5-19, *datatype::=* on page 2-2, *table_clause::=* on page 22-4)

window_type::=



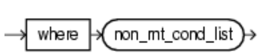
(*identifier::=* on page 7-17, *non_mt_attr_list::=* on page 7-22, *time_spec::=* on page 7-30)

table_clause::=



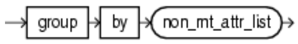
(*object_expr::=* on page 5-19, *identifier::=* on page 7-17, *datatype::=* on page 2-2)

opt_where_clause::=



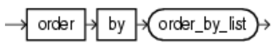
(*non_mt_cond_list::=* on page 7-25)

opt_group_by_clause::=



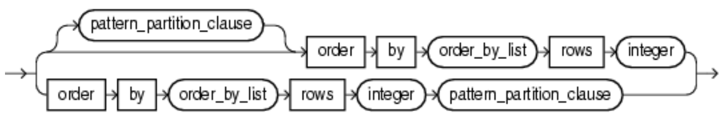
(*non_mt_attr_list::=* on page 7-22)

order_by_clause::=



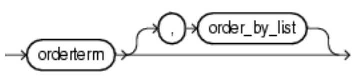
(*order_by_list::=* on page 22-5)

order_by_top_clause::=



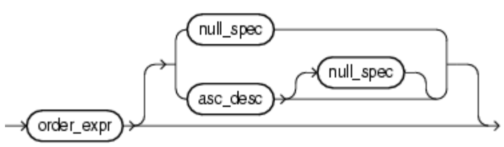
(*pattern_partition_clause::=* on page 21-17, *order_by_list::=* on page 22-5)

order_by_list::=



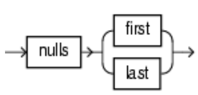
(*orderterm::=* on page 22-5)

orderterm::=

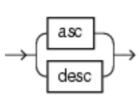


(*order_expr::=* on page 5-23, *null_spec::=* on page 22-5, *asc_desc::=* on page 22-5)

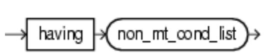
null_spec::=



asc_desc::=

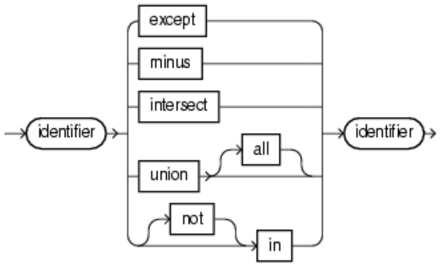


opt_having_clause::=



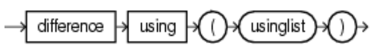
(*non_mt_cond_list::=* on page 7-25)

binary::=



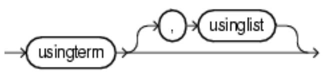
idstream_clause::=

using_clause::=



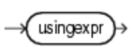
(*usinglist::=* on page 22-6)

usinglist::=



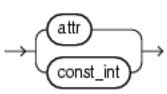
(*usingterm::=* on page 22-6)

usingterm::=



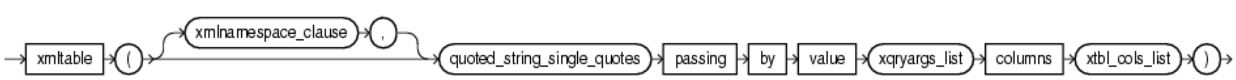
(*usingexpr::=* on page 22-6)

usingexpr::=



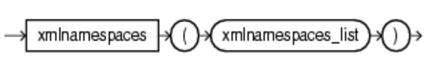
(*attr::=* on page 7-5, *const_int::=* on page 7-12)

xmltable_clause::=



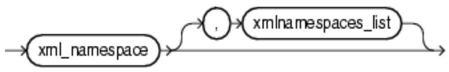
(*xmlnamespace_clause::=* on page 22-6, *const_string::=* on page 7-13, *xqryargs_list::=* on page 7-34, *xtbl_cols_list::=* on page 22-7)

xmlnamespace_clause::=



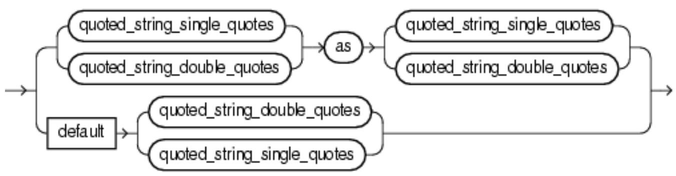
(*xmlnamespaces_list::=* on page 22-7)

xmlnamespaces_list::=



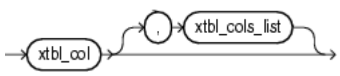
(*xml_namespace::=* on page 22-7)

xml_namespace::=



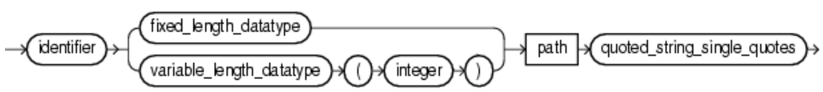
(*const_string::=* on page 7-13)

xtbl_cols_list::=



(*xtbl_col::=* on page 22-7)

xtbl_col::=



Semantics

named_query

Specify the Oracle CQL query statement itself (see "query" on page 22-7).

For syntax, see "Query" on page 22-2.

query

You can create an Oracle CQL query from any of the following clauses:

- `sfw_block`: a select, from, and other optional clauses (see "sfw_block" on page 22-7).
- `binary`: an optional clause, often a set operation (see "binary" on page 22-13).
- `xstream_clause`: apply an optional relation-to-stream operator to your `sfw_block` or `binary` clause to control how the query returns its results (see "idstream_clause" on page 22-13).

For syntax, see *query::=* on page 22-2.

sfw_block

Specify the select, from, and other optional clauses of the Oracle CQL query. You can specify any of the following clauses:

- `select_clause`: the stream elements to select from the stream or view you specify (see "select_clause" on page 22-8).
- `from_clause`: the stream or view from which your query selects (see "from_clause" on page 22-9).
- `opt_where_clause`: optional conditions your query applies to its selection (see "opt_where_clause" on page 22-10)
- `opt_group_by_clause`: optional grouping conditions your query applies to its results (see "opt_group_by_clause" on page 22-11)
- `order_by_clause`: optional ordering conditions your query applies to its results (see "order_by_clause" on page 22-11)
- `order_by_top_clause`: optional ordering conditions your query applies to the top-`n` elements in its results (see "order_by_top_clause" on page 22-12)
- `opt_having_clause`: optional clause your query uses to restrict the groups of returned stream elements to those groups for which the specified `condition` is `TRUE` (see "opt_having_clause" on page 22-13)

For syntax, see *sfw_block::=* on page 22-3 (parent: *query::=* on page 22-2).

select_clause

Specify the select clause of the Oracle CQL query statement.

If you specify the asterisk (`*`), then this clause returns all tuples, including duplicates and nulls.

Otherwise, specify the individual stream elements you want (see "non_mt_projterm_list" on page 22-8).

Optionally, specify `distinct` if you want Oracle Event Processing to return only one copy of each set of duplicate tuples selected. Duplicate tuples are those with matching values for each expression in the select list. For an example, see "Select and Distinct Examples" on page 22-21.

For syntax, see *select_clause::=* on page 22-3 (parent: *sfw_block::=* on page 22-3).

non_mt_projterm_list

Specify the projection term ("projterm" on page 22-8) or comma separated list of projection terms in the select clause of the Oracle CQL query statement.

For syntax, see *non_mt_projterm_list::=* on page 22-3 (parent: *select_clause::=* on page 22-3).

projterm

Specify a projection term in the select clause of the Oracle CQL query statement. You can select any element from any of stream or view in the `from_clause` (see "from_clause" on page 22-9) using the `identifier` of the element.

Optionally, you can specify an arithmetic expression on the projection term.

Optionally, use the `AS` keyword to specify an alias for the projection term instead of using the stream element name as is.

For syntax, see *projterm::=* on page 22-3 (parent: *non_mt_projterm_list::=* on page 22-3).

from_clause

Specify the from clause of the Oracle CQL query statement by specifying the individual streams or views from which your query selects (see "non_mt_relation_list" on page 22-9).

To perform an outer join, use the `LEFT` or `RIGHT OUTER JOIN ... ON` syntax. To perform an inner join, use the `WHERE` clause.

For more information, see:

- "opt_where_clause" on page 22-10
- "Joins" on page 20-21

For syntax, see *from_clause::=* on page 22-3 (parent: *sfw_block::=* on page 22-3).

non_mt_relation_list

Specify the stream or view ("relation_variable" on page 22-9) or comma separated list of streams or views in the from clause of the Oracle CQL query statement.

For syntax, see *non_mt_relation_list::=* on page 22-4 (parent: *from_clause::=* on page 22-3).

relation_variable

Use the *`relation_variable`* statement to specify a stream or view from which the Oracle CQL query statement selects.

You can specify a previously registered or created stream or view directly by its *`identifier`* you used when you registered or created the stream or view. Optionally, use the `AS` keyword to specify an alias for the stream or view instead of using its name as is.

To specify a built-in stream-to-relation operator, use a *`window_type`* clause (see "window_type" on page 22-9). Optionally, use the `AS` keyword to specify an alias for the stream or view instead of using its name as is.

To apply advanced comparisons optimized for data streams to the stream or view, use a *`pattern_recognition_clause`* (see "pattern_recognition_clause" on page 22-14). Optionally, use the `AS` keyword to specify an alias for the stream or view instead of using its name as is.

To process `xmltype` stream elements using XPath and XQuery, use an *`xmltable_clause`* (see "xmltable_clause" on page 22-14). Optionally, use the `AS` keyword to specify an alias for the stream or view instead of using its name as is.

To access, as a relation, the multiple rows returned by a data cartridge function in the `FROM` clause of an Oracle CQL query, use a *`table_clause`* (see "table_clause" on page 22-10).

For more information, see:

- "View" on page 22-29

For syntax, see *relation_variable::=* on page 22-4 (parent: *non_mt_relation_list::=* on page 22-4).

window_type

Specify a built-in stream-to-relation operator.

For more information, see Section , "Stream-to-Relation Operators (Windows)".

For syntax, see *window_type::=* on page 22-4 (parent: *relation_variable::=* on page 22-4).

table_clause

Use the data cartridge TABLE clause to access the multiple rows returned by a data cartridge function in the FROM clause of an Oracle CQL query.

The TABLE clause converts the set of returned rows into an Oracle CQL relation. Because this is an external relation, you must join the TABLE function clause with a stream. Oracle Event Processing invokes the data cartridge method only on the arrival of elements on the joined stream.

Use the optional OF keyword to specify the type contained by the returned array type or Collection type.

Use the AS keyword to specify an alias for the *object_expr* and for the returned relation.

Note the following:

- The data cartridge method must return an array type or Collection type.
- You must join the TABLE function clause with a stream.

For examples, see:

- "Data Cartridge TABLE Query Example: Iterator" on page 22-22
- "Data Cartridge TABLE Query Example: Array" on page 22-23
- "Data Cartridge TABLE Query Example: Collection" on page 22-24

For more information, see:

- Section , "Arrays"
- Section , "Collections"
- *object_expr::=* on page 5-19

For syntax, see *table_clause::=* on page 22-4 (parent: *relation_variable::=* on page 22-4).

time_spec

Specify the time over which a range or partitioned range sliding window should slide.

Default: if units are not specified, Oracle Event Processing assumes [second|seconds].

For more information, see "Range-Based Stream-to-Relation Window Operators" on page 4-6 and "Partitioned Stream-to-Relation Window Operators" on page 4-19.

For syntax, see *time_spec::=* on page 7-30 (parent: *window_type::=* on page 22-4).

opt_where_clause

Specify the (optional) where clause of the Oracle CQL query statement.

Because Oracle CQL applies the WHERE clause before GROUP BY or HAVING, if you specify an aggregate function in the SELECT clause, you must test the aggregate function result in a HAVING clause, not the WHERE clause.

In Oracle CQL (as in SQL), the FROM clause is evaluated before the WHERE clause. Consider the following Oracle CQL query:

```
SELECT ... FROM S MATCH_RECOGNIZE ( .... ) as T WHERE ...
```

In this query, the S MATCH_RECOGNIZE (....) as T is like a subquery in the FROM clause and is evaluated first, before the WHERE clause. Consequently, you rarely use both a MATCH_RECOGNIZE clause and a WHERE clause in the same Oracle CQL query.

Instead, you typically use views to apply the required `WHERE` clause to a stream and then select from the views in a query that applies the `MATCH_RECOGNIZE` clause.

For more information, see:

- Section , "Built-In Aggregate Functions and the Where, Group By, and Having Clauses"
- Section , "Colt Aggregate Functions and the Where, Group By, and Having Clauses"
- Section , "MATCH_RECOGNIZE and the WHERE Clause"

For syntax, see *opt_where_clause::=* on page 22-4 (parent: *sfw_block::=* on page 22-3).

opt_group_by_clause

Specify the (optional) `GROUP BY` clause of the Oracle CQL query statement. Use the `GROUP BY` clause if you want Oracle Event Processing to group the selected stream elements based on the value of *expr*(s) and return a single (aggregate) summary result for each group.

Expressions in the `GROUP BY` clause can contain any stream elements or views in the `FROM` clause, regardless of whether the stream elements appear in the select list.

The `GROUP BY` clause groups stream elements but does not guarantee the order of the result set. To order the groupings, use the `ORDER BY` clause.

Because Oracle CQL applies the `WHERE` clause before `GROUP BY` or `HAVING`, if you specify an aggregate function in the `SELECT` clause, you must test the aggregate function result in a `HAVING` clause, not the `WHERE` clause.

For more information, see:

- Section , "Built-In Aggregate Functions and the Where, Group By, and Having Clauses"
- Section , "Colt Aggregate Functions and the Where, Group By, and Having Clauses"

For syntax, see *opt_group_by_clause::=* on page 22-5 (parent: *sfw_block::=* on page 22-3).

order_by_clause

Specify the `ORDER BY` clause of the Oracle CQL query statement as a comma-delimited list ("order_by_list" on page 22-12) of one or more order terms (see "orderterm" on page 22-12). Use the `ORDER BY` clause to specify the order in which stream elements on the left-hand side of the rule are to be evaluated. The *expr* must resolve to a dimension or measure column. This clause returns a stream.

Both ORDER BY and ORDER BY ROWS support specifying the direction of sort as ascending or descending by using the ASC or DESC keywords. They also support specifying whether null items should be listed first or last when sorting by using NULLS FIRST or NULLS LAST.

For more information, see:

- "ORDER BY Query Example"
- Section , "Sorting Query Results"

For syntax, see *order_by_clause::=* on page 22-5 (parent: *sfw_block::=* on page 22-3).

order_by_top_clause

Specify the `ORDER BY` clause of the Oracle CQL query statement as a comma-delimited list ("order_by_list" on page 22-12) of one or more order terms (see "orderterm" on page 22-12) followed by a `ROWS` keyword and integer number (`n`) of elements. Use this form of the `ORDER BY` clause to select the top-`n` elements over a stream or relation. This clause returns a relation.

Consider the following example queries:

- At any point of time, the output of the following example query will be a relation having top 10 stock symbols throughout the stream.

  ```
  select stock_symbols from StockQuotes order by stock_price rows 10
  ```

- At any point of time, the output of the following example query will be a relation having top 10 stock symbols from last 1 hour of data.

  ```
  select stock_symbols from StockQuotes[range 1 hour] order by stock_price rows
  10
  ```

See "order_by_clause::=" for more about what is supported by ORDER BY.

For more information, see:

- "ORDER BY ROWS Query Example" on page 22-26
- Section , "Sorting Query Results"

For syntax, see *order_by_top_clause::=* on page 22-5 (parent: *sfw_block::=* on page 22-3).

order_by_list

Specify a comma-delimited list of one ore more order terms (see "orderterm" on page 22-12) in an (optional) `ORDER BY` clause.

For syntax, see *order_by_list::=* on page 22-5 (parent: *order_by_clause::=* on page 22-5).

orderterm

A stream element (*attr::=* on page 7-5) or positional index (constant int) to a stream element. Optionally, you can configure whether or not nulls are ordered first or last using the `NULLS` keyword (see "null_spec" on page 22-12).

order_expr (*order_expr::=* on page 5-23) can be an `attr` or `constant_int`. The `attr` (*attr::=* on page 7-5) can be any stream element or pseudo column.

For syntax, see *orderterm::=* on page 22-5 (parent: *order_by_list::=* on page 22-5).

null_spec

Specify whether or not nulls are ordered first (`NULLS FIRST`) or last (`NULLS LAST`) for a given order term (see "orderterm" on page 22-12).

For syntax, see *null_spec::=* on page 22-5 (parent: *orderterm::=* on page 22-5).

asc_desc

Specify whether an order term is ordered in ascending (`ASC`) or descending (`DESC`) order.

For syntax, see *asc_desc::=* on page 22-5 (parent: *orderterm::=* on page 22-5).

opt_having_clause

Use the HAVING clause to restrict the groups of returned stream elements to those groups for which the specified *condition* is TRUE. If you omit this clause, then Oracle Event Processing returns summary results for all groups.

Specify GROUP BY and HAVING after the *opt_where_clause*. If you specify both GROUP BY and HAVING, then they can appear in either order.

Because Oracle CQL applies the WHERE clause before GROUP BY or HAVING, if you specify an aggregate function in the SELECT clause, you must test the aggregate function result in a HAVING clause, not the WHERE clause.

For more information, see:

- Section , "Built-In Aggregate Functions and the Where, Group By, and Having Clauses"
- Section , "Colt Aggregate Functions and the Where, Group By, and Having Clauses"

For an example, see "HAVING Example" on page 22-15.

For syntax, see *opt_having_clause::=* on page 22-5 (parent: *sfw_block::=* on page 22-3).

binary

Use the *binary* clause to perform operations on the tuples that two streams or views return. Most of these perform set operations, receiving two relations as operands. However, the UNION ALL operator can instead receive two streams, which are by nature unbounded.

For examples, see:

- "BINARY Example: UNION and UNION ALL" on page 22-16
- "BINARY Example: INTERSECT" on page 22-17
- "BINARY Example: EXCEPT and MINUS" on page 22-18
- "BINARY Example: IN and NOT IN" on page 22-19

For syntax, see *binary::=* on page 22-6 (parent: *query::=* on page 22-2).

idstream_clause

Use an *idstream_clause* to specify an IStream or DStream relation-to-stream operator that applies to the query.

For more information, see Section , "Relation-to-Stream Operators".

For syntax, see *idstream_clause::=* on page 22-6 (parent: *query::=* on page 22-2).

using_clause

Use a DIFFERENCE USING clause to succinctly detect differences in the IStream or DStream of a query.

For more information, see Section , "Detecting Differences in Query Results".

For syntax, see *using_clause::=* on page 22-6 (parent: *query::=* on page 22-2).

usinglist

Use a usinglist clause to specify the columns to use to detect differences in the IStream or DStream of a query. You may specify columns by:

- attribute name: use this option when you are selecting by attribute name.

  Example 22–2 shows attribute name `c1` in the `DIFFERENCE USING` clause `usinglist`.

- alias: use this option when you want to include the results of an expression where an alias is specified.

  Example 22–2 shows alias `logval` in the `DIFFERENCE USING` clause `usinglist`.

- position: use this option when you want to include the results of an expression where no alias is specified.

  Specify position as a constant, positive integer starting at 1, reading from left to right.

  Example 22–2 specifies the result of expression `funct(c2, c3)` by its position (3) in the `DIFFERENCE USING` clause `usinglist`.

Example 22–2 Specifying the usinglist in a DIFFERENCE USING Clause

```
<query id="q1">
    ISTREAM (
        SELECT c1, log(c4) as logval, funct(c2, c3) FROM S [RANGE 1 NANOSECONDS]
    )    DIFFERENCE USING (c1, logval, 3)
</query>
```

For more information, see Section , "Detecting Differences in Query Results".

For syntax, see *usinglist::=* on page 22-6 (parent: *using_clause::=* on page 22-6).

xmltable_clause

Use an *`xmltable_clause`* to process `xmltype` stream elements using XPath and XQuery. You can specify a comma separated list (see *xtbl_cols_list::=* on page 22-7) of one or more XML table columns (see *xtbl_col::=* on page 22-7), with or without an XML namespace.

For examples, see:

- "XMLTABLE Query Example" on page 22-21
- "XMLTABLE With XML Namespaces Query Example" on page 22-22

For syntax, see *xmltable_clause::=* on page 22-6 (parent: *relation_variable::=* on page 22-4).

pattern_recognition_clause

Use a *`pattern_recognition_clause`* to perform advanced comparisons optimized for data streams.

For more information and examples, see Chapter 21, "Pattern Recognition With MATCH_RECOGNIZE".

For syntax, see *pattern_recognition_clause::=* on page 21-2 (parent: *relation_variable::=* on page 22-4).

Examples

The following examples illustrate the various semantics that this statement supports:

- "Simple Query Example" on page 22-15
- "HAVING Example" on page 22-15
- "BINARY Example: UNION and UNION ALL" on page 22-16

- "BINARY Example: INTERSECT" on page 22-17
- "BINARY Example: EXCEPT and MINUS" on page 22-18
- "Select and Distinct Examples" on page 22-21
- "XMLTABLE Query Example" on page 22-21
- "XMLTABLE With XML Namespaces Query Example" on page 22-22
- "Data Cartridge TABLE Query Example: Iterator" on page 22-22
- "Data Cartridge TABLE Query Example: Array" on page 22-23
- "Data Cartridge TABLE Query Example: Collection" on page 22-24
- "ORDER BY ROWS Query Example" on page 22-26

For more examples, see Chapter 20, "Oracle CQL Queries, Views, and Joins".

Simple Query Example

Example 22–3 shows how to register a simple query `q0` that selects all (`*`) tuples from stream `OrderStream` where stream element `orderAmount` is greater than 10000.

Example 22–3 REGISTER QUERY

```
<query id="q0"><![CDATA[
    select * from OrderStream where orderAmount > 10000.0
]]></query>
```

HAVING Example

Consider the query `q4` in Example 22–4 and the data stream `S2` in Example 22–5. Stream `S2` has schema `(c1 integer, c2 integer)`. The query returns the relation in Example 22–6.

Example 22–4 HAVING Query

```
<query id="q4"><![CDATA[
    select
        c1,
        sum(c1)
    from
        S2[range 10]
    group by
        c1
    having
        c1 > 0 and sum(c1) > 1
]]></query>
```

Example 22–5 HAVING Stream Input

```
Timestamp   Tuple
1000        ,2
2000        ,4
3000        1,4
5000        1,
6000        1,6
7000        ,9
8000        ,
```

Example 22–6 HAVING Relation Output

```
Timestamp   Tuple Kind  Tuple
5000:       +           1,2
6000:       -           1,2
```

```
6000:       +           1,3
```

BINARY Example: UNION and UNION ALL

The UNION and UNION ALL operators both take two operands and combine their elements. The result of the UNION ALL operator includes all of the elements from the two operands, including duplicates. The result of the UNION operator omits duplicates.

The UNION operator accepts only two relations and produces a relation as its output. This operator cannot accept streams because in order to remove duplicates, the Oracle CQL engine must keep track of all of the elements contained in both operands. This is not possible with streams, which are by nature unbounded.

The UNION ALL operator accepts either two streams (and producing a stream) or two relations (producing a relation) as its operands. Using one stream and one relation as operands is invalid for both operators.

Given the relations `R1` and `R2` in Example 22–8 and Example 22–9, respectively, the `UNION` query `q1` in Example 22–7 returns the relation in Example 22–10 and the `UNION ALL` query `q2` in Example 22–7 returns the relation in Example 22–11.

See Example 22–12, Example 22–13, and Example 22–14 for an example of UNION ALL with streams instead of relations.

Example 22–7 UNION and UNION ALL Queries

```
<query id="q1"><![CDATA[
    R1 UNION R2
]]></query>
<query id="q2"><![CDATA[
    R1 UNION ALL R2
]]></query>
```

Example 22–8 UNION Relation Input R1

```
Timestamp   Tuple Kind  Tuple
   200000:  +           20,0.2
   201000:  -           20,0.2
   400000:  +           30,0.3
   401000:  -           30,0.3
100000000:  +           40,4.04
100001000:  -           40,4.04
```

Example 22–9 UNION Relation Input R2

```
Timestamp   Tuple Kind  Tuple
     1002:  +           15,0.14
     2002:  -           15,0.14
   200000:  +           20,0.2
   201000:  -           20,0.2
   400000:  +           30,0.3
   401000:  -           30,0.3
100000000:  +           40,4.04
100001000:  -           40,4.04
```

Example 22–10 UNION Relation Output

```
Timestamp   Tuple Kind  Tuple
     1002:  +           15,0.14
     2002:  -           15,0.14
   200000:  +           20,0.2
   201000:  -           20,0.2
   400000:  +           30,0.3
   401000:  -           30,0.3
```

```
100000000:  +           40,4.04
100001000:  -           40,4.04
```

Example 22–11 UNION ALL Output from Relation Input

The following output is from using UNION ALL with two relations as operands.

```
Timestamp   Tuple Kind  Tuple
     1002:  +           15,0.14
     2002:  -           15,0.14
   200000:  +           20,0.2
   200000:  +           20,0.2
    20100:  -           20,0.2
   201000:  -           20,0.2
   400000:  +           30,0.3
   400000:  +           30,0.3
   401000:  -           30,0.3
   401000:  -           30,0.3
100000000:  +           40,4.04
100000000:  +           40,4.04
 10001000:  -           40,4.04
100001000:  -           40,4.04
```

Example 22–12 UNION ALL Stream Input S1

```
Timestamp     Tuple
   100000:    20,0.1
   150000:    15,0.1
   200000:    5,0.2
   400000:    30,0.3
100000000:    8,4.04
```

Example 22–13 UNION ALL Stream Input S2

```
Timestamp     Tuple
     1001:    10,0.1
     1002:    15,0.14
   200000:    20,0.2
   400000:    30,0.3
100000000:    40,4.04
```

Example 22–14 UNION ALL Output from Stream Input

The following output is from using UNION ALL with the two preceding streams as operands. Note that all of the elements are inserted events.

```
Timestamp   Tuple Kind  Tuple
     1001:  +           10,0.1
     1002:  +           15,0.14
   100000:  +           20,0.1
   150000:  +           15,0.14
   200000:  +           20,0.2
   200000:  +           5,0.2
   400000:  +           30,0.3
   400000:  +           30,0.3
100000000:  +           40,4.04
100000000:  +           8,4.04
```

BINARY Example: INTERSECT

The INTERSECT operator returns a relation (with duplicates removed) with only those elements that appear in both of its operand relations.

Given the relations `R1` and `R2` in Example 22–16 and Example 22–17, respectively, the `INTERSECT` query `q1` in Example 22–15 returns the relation in Example 22–18.

Example 22–15 INTERSECT Query

```
<query id="q1"><![CDATA[
    R1 INTERSECT R2
]]></query>
```

Example 22–16 INTERSECT Relation Input R1

```
Timestamp   Tuple Kind  Tuple
1000:        +          10,30
1000:        +          10,40
2000:        +          11,20
3000:        -          10,30
3000:        -          10,40
```

Example 22–17 INTERSECT Relation Input R2

```
Timestamp   Tuple Kind  Tuple
1000:       +           10,40
2000:       +           10,30
2000:       -           10,40
3000:       -           10,30
```

Example 22–18 INTERSECT Relation Output

```
Timestamp   Tuple Kind  Tuple
1000:       +           10,40
2000:       +           10,30
2000:       -           10,40
3000:       -           10,30
```

BINARY Example: EXCEPT and MINUS

As in database programming, the `EXCEPT` and `MINUS` operators are very similar. They both take relations as their two operands. Both result in a relation that is essentially elements of the first operand relation that are not also in the second operand relation.

An important difference between `EXCEPT` and `MINUS` is in how they handle duplicate occurrences between the first and second relation operands, as follows:

- The EXCEPT operator results in a relation made up of elements from the first relation, removing elements that were also found in the second relation up to the number of duplicate elements found in the second relation. In other words, if an element occurs m times in the first relation and n times in the second, the number of that element in the result will be n subtracted from m, or 0 if there were fewer in m than n.
- The MINUS operator results in a relation made up of elements in the first relation minus those elements that were also found in the second relation, regardless of how many of those elements were found in each. The MINUS operator also removes duplicate elements found in the first relation, so that each duplicate item is unique in the result.

The following examples illustrate the MINUS operator. Given the relations `R1` and `R2` in Example 22–20 and Example 22–21, respectively, the `MINUS` query `q1` in Example 22–19 returns the relation in Example 22–22.

Example 22–19 MINUS Query

```
<query id="q1BBAQuery"><![CDATA[
    R1 MINUS R2
]]></query>
```

Example 22–20 MINUS Relation Input R1

```
Timestamp   Tuple Kind  Tuple
1500:       +           10,40
1800:       +           10,30
2000:       +           10,40
2000:       +           10,40
2100:       -           10,40
3000:       -           10,30
```

Example 22–21 MINUS Relation Input R2

```
Timestamp   Tuple Kind  Tuple
1000:       +           11,20
2000:       +           10,40
3000:       -           10,30
```

Example 22–22 MINUS Relation Output

```
Timestamp   Tuple Kind  Tuple
1500:       +           10,40
1800:       +           10,30
2100:       -           10,40
```

The following examples illustrate the EXCEPT operator. Given the relations `R1` and `R2` in Example 22–24 and Example 22–25, respectively, the `EXCEPT` query `q1` in Example 22–23 returns the relation in Example 22–26.

Example 22–23 EXCEPT Query

```
<query id="exceptQuery"><![CDATA[
    R1 EXCEPT R2
]]></query>
```

Example 22–24 EXCEPT Relation Input R1

```
Timestamp   Tuple Kind  Tuple
1500:       +           10,40
1800:       +           10,30
2000:       +           10,40
2000:       +           10,40
2100:       -           10,40
3000:       -           10,30
```

Example 22–25 EXCEPT Relation Input R2

```
Timestamp   Tuple Kind  Tuple
1000:       +           11,20
2000:       +           10,40
3000:       -           10,30
```

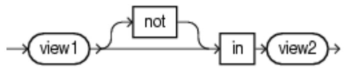
Example 22–26 EXCEPT Relation Output

```
Timestamp   Tuple Kind  Tuple
1500:       +           10,40
1800:       +           10,30
2000:       +           10,40
2100:       -           10,40
```

BINARY Example: IN and NOT IN

In this usage, the query will be a binary query.

in_condition_set::=



Note: You cannot combine this usage with *in_condition_membership* as Section , "Using IN and NOT IN as a Membership Condition" describes.

Consider the views `V3` and `V4` and the query `Q1` in Example 22–27 and the data streams `S3` in Example 22–28 (with schema `(c1 integer, c2 integer)`) and `S4` in Example 22–29 (with schema `(c1 integer, c2 integer)`). In this condition test, the numbers and data types of attributes in left relation should be same as number and types of attributes of the right relation. Example 22–30 shows the relation that the query returns.

Example 22–27 IN and NOT IN as a Set Operation: Query

```
<view id="V3" schema="c1 c2"><![CDATA[
    select * from S3[range 2]
]]></query>
<view id="V4" schema="c1 d1"><![CDATA[
    select * from S4[range 1]
]]></query>
<query id="Q1"><![CDATA[
     v3 not in v4
]]></query>
```

Example 22–28 IN and NOT IN as a Set Operation: Stream S3 Input

```
Timestamp   Tuple
1000        10, 30
1000        10, 40
2000        11, 20
3000        12, 40
3000        12, 30
3000        15, 50
h 2000000
```

Example 22–29 IN and NOT IN as a Set Operation: Stream S4 Input

```
Timestamp   Tuple
1000        10, 40
2000        10, 30
2000        12, 40
h 2000000
```

Example 22–30 IN and NOT IN as a Set Operation: Relation Output

```
Timestamp   Tuple Kind  Tuple
1000:       +           10,30
1000:       +           10,40
1000:       -           10,30
1000:       -           10,40
2000:       +           11,20
2000:       +           10,30
2000:       +           10,40
2000:       -           10,30
2000:       -           10,40
3000:       +           15,50
3000:       +           12,40
```

```
3000:       +          12,30
4000:       -          11,20
5000:       -          12,40
5000:       -          12,30
5000:       -          15,50
```

Select and Distinct Examples

Consider the query `q1` in Example 22–31. Given the data stream `S` in Example 22–32, the query returns the relation in Example 22–33.

Example 22–31 Select DISTINCT Query

```
<query id="q1"><![CDATA[
    SELECT DISTINCT FROM S WHERE c1 > 10
]]></query>
```

Example 22–32 Select DISTINCT Stream Input

```
Timestamp   Tuple
 1000        23
 2000        14
 3000        13
 5000        22
 6000        11
 7000        10
 8000         9
10000         8
11000         7
12000        13
13000        14
```

Example 22–33 Select DISTINCT Stream Output

```
Timestamp   Tuple
1000         23
2000         14
3000         13
5000         22
6000         11
```

XMLTABLE Query Example

Consider the query `q1` in Example 22–34 and the data stream `S` in Example 22–35. Stream `S` has schema `(c1 xmltype)`. The query returns the relation in Example 22–36.

For a more complete description of XMLTABLE, see Section , "XMLTABLE Query".

Example 22–34 XMLTABLE Query

```
<query id="q1"><![CDATA[
    SELECT
        X.Name,
        X.Quantity
    FROM S1,
        XMLTABLE (
            "//item" PASSING BY VALUE S1.c2 as "."
            COLUMNS
                Name CHAR(16) PATH "/item/productName",
                Quantity INTEGER PATH "/item/quantity"
        ) AS X
]]></query>
```

Example 22–35 XMLTABLE Stream Input

```
Timestamp   Tuple
```

```
3000        "<purchaseOrder><shipTo><name>Alice Smith</name><street>123 Maple
Street</street><city>Mill Valley</city><state>CA</state><zip>90952</zip>
</shipTo><billTo><name>Robert Smith</name><street>8 Oak Avenue</street><city>Old
Town</city><state>PA</state> <zip>95819</zip> </billTo><comment>Hurry, my lawn is going
wild!</comment><items> <item><productName>Lawnmower
</productName><quantity>1</quantity><USPrice>148.95</USPrice><comment>Confirm this is
electric</comment></item><item> <productName>Baby Monitor</productName><quantity>1</quantity>
<USPrice>39.98</USPrice> <shipDate>1999-05-21</shipDate></item></items> </purchaseOrder>"
4000        "<a>hello</a>"
```

Example 22–36 XMLTABLE Relation Output

```
Timestamp   Tuple Kind  Tuple
3000:       +           <productName>Lawnmower</productName>,<quantity>1</quantity>
3000:       +           <productName>Baby Monitor</productName>,<quantity>1</quantity>
```

XMLTABLE With XML Namespaces Query Example

Consider the query q1 in Example 22–37 and the data stream S1 in Example 22–38. Stream S1 has schema (c1 xmltype). The query returns the relation in Example 22–39.

For a more complete description of XMLTABLE, see Section , "XMLTABLE Query".

Example 22–37 XMLTABLE With XML Namespaces Query

```
<query id="q1"><![CDATA[
    SELECT *
    FROM S1,
        XMLTABLE (
            XMLNAMESPACES('http://example.com' as 'e'),
            'for $i in //e:emps return $i/e:emp' PASSING BY VALUE S1.c1 as "."
            COLUMNS
                empName char(16) PATH 'fn:data(@ename)',
                empId integer PATH 'fn:data(@empno)'
        ) AS X
]]></query>
```

Example 22–38 XMLTABLE With XML Namespaces Stream Input

```
Timestamp   Tuple
3000        "<emps xmlns=\"http://example.com\"><emp empno=\"1\" deptno=\"10\" ename=\"John\"
salary=\"21000\"/><emp empno=\"2\" deptno=\"10\" ename=\"Jack\" salary=\"310000\"/><emp
empno=\"3\" deptno=\"20\" ename=\"Jill\" salary=\"100001\"/></emps>"
h 4000
```

Example 22–39 XMLTABLE With XML Namespaces Relation Output

```
Timestamp   Tuple Kind  Tuple
3000:       +     John,1
3000:       +     Jack,2
3000:       +     Jill,3
```

Data Cartridge TABLE Query Example: Iterator

Consider a data cartridge (MyCartridge) with method getIterator as Example 22–40 shows.

Example 22–40 MyCartridge Method getIterator

```
...
    public static Iterator<Integer> getIterator() {
        ArrayList<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        return list.iterator();
```

```
    }
...
```

Consider the query `q1` in Example 22–41. Given the data stream `S0` in Example 22–42, the query returns the relation in Example 22–43.

Example 22–41 TABLE Query: Iterator

```
<query id="q1"><![CDATA[
    select S1.c1, S1.c2, S2.c1
    from
        S0[now] as S1,
        table (com.acme.MyCartridge.getIterator() as c1) of integer as S2
]]></query>
```

Example 22–42 TABLE Query Stream Input: Iterator

```
Timestamp   Tuple
1           1, abc
2           2, ab
3           3, abc
4           4, a
h 200000000
```

Example 22–43 TABLE Query Output: Iterator

```
Timestamp   Tuple Kind  Tuple
1:          +           1,abc,1
1:          +           1,abc,2
1:          -           1,abc,1
1:          -           1,abc,2
2:          +           2,ab,1
2:          +           2,ab,2
2:          -           2,ab,1
2:          -           2,ab,2
3:          +           3,abc,1
3:          +           3,abc,2
3:          -           3,abc,1
3:          -           3,abc,2
4:          +           4,a,1
4:          +           4,a,2
4:          -           4,a,1
4:          -           4,a,2
```

Data Cartridge TABLE Query Example: Array

Consider a data cartridge (`MyCartridge`) with method `getArray` as Example 22–44 shows.

Example 22–44 MyCartridge Method getArray

```
...
    public static Integer[] getArray(int c1) {
        ArrayList<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        return list.toArray(new Integer[2]);;
    }
...
```

Consider the query q1 in Example 22–45. Given the data stream S0 in Example 22–46, the query returns the relation in Example 22–47.

Example 22–45 TABLE Query: Array

```
<query id="q1"><![CDATA[
    select S1.c1, S1.c2, S2.c1
    from
        S0[now] as S1,
        table (com.acme.MyCartridge.getArrayS1.c1) as c1) of integer as S2
]]></query>
```

Example 22–46 TABLE Query Stream Input: Array

```
Timestamp   Tuple
1           1, abc
2           2, ab
3           3, abc
4           4, a
h 200000000
```

Example 22–47 TABLE Query Output: Array

```
Timestamp   Tuple Kind  Tuple
1:          +           1,abc,1
1:          +           1,abc,2
1:          -           1,abc,1
1:          -           1,abc,2
2:          +           2,ab,2
2:          +           2,ab,4
2:          -           2,ab,2
2:          -           2,ab,4
3:          +           3,abc,3
3:          +           3,abc,6
3:          -           3,abc,3
3:          -           3,abc,6
4:          +           4,a,4
4:          +           4,a,8
4:          -           4,a,4
4:          -           4,a,8
```

Data Cartridge TABLE Query Example: Collection

Consider a data cartridge (MyCartridge) with method getCollection as Example 22–48 shows.

Example 22–48 MyCartridge Method getCollection

```
...
    public HashMap<Integer,String> developers;
    developers = new HashMap<Integer,String>();
    developers.put(2, "Mohit");
    developers.put(4, "Unmesh");
    developers.put(3, "Sandeep");
    developers.put(1, "Swagat");

    public HashMap<Integer,String> qaengineers;
    qaengineers = new HashMap<Integer,String>();
    qaengineers.put(4, "Terry");
    qaengineers.put(5, "Tony");
    qaengineers.put(3, "Junger");
```

```
    qaengineers.put(1, "Arthur");
...
    public Collection<String> getEmployees(int exp_yrs) {
        LinkedList<String> employees = new LinkedList<String>();
        employees.add(developers.get(exp_yrs));
        employees.add(qaengineers.get(exp_yrs));
        return employees;
  }
...
```

Consider the query q1 in Example 22–49. Given the data stream S0 in Example 22–50, the query returns the relation in Example 22–51.

Example 22–49 TABLE Query: Collection

```
<query id="q1"><![CDATA[
    RStream(
        select S1.c1, S2.c1
        from
            S0[now] as S1,
            table(S1.c2.getEmployees(S1.c1) as c1) of char as S2
    )
]]></query>
```

Example 22–50 TABLE Query Stream Input: Collection

```
Timestamp   Tuple
1           1, abc
2           2, ab
3           3, abc
4           4, a
h 200000000
```

Example 22–51 TABLE Query Output: Collection

```
Timestamp   Tuple Kind  Tuple
1:          +           1,Swagat
1:          +           1,Arthur
2:          +           2,Mohit
3:          +           3,Sandeep
3:          +           3,Junger
4:          +           4,Unmesh
4:          +           4,Terry
```

ORDER BY Query Example

Use the ORDER BY clause with stream input to sort events that have duplicate timestamps. ORDER BY is only valid when the input is a stream and only sorts among events of the same timestamp. Its output is a stream with the sorted events.

For more information, see "order_by_clause::=".

Consider the query q1 in Example 22–55. Given the data stream S0 in Example 22–56, the query returns the relation in Example 22–57. The query sorts events of duplicate timestamps in ascending order by tuple values.

Example 22–52 ORDER BY Query

```
<query id="q1"><![CDATA[
    SELECT *
    FROM S0
```

```
    ORDER BY c1,c2 ASC
]]></query>
```

Example 22–53 ORDER BY Stream Input

```
Timestamp   Tuple
1000        7, 15
2000        7, 14
2000        5, 23
2000        5, 15
2000        5, 15
2000        5, 25
3000        3, 12
3000        2, 13
4000        4, 17
5000        1, 9
h 1000000000
```

Example 22–54 ORDER BY Stream Output

```
Timestamp   Tuple Kind  Tuple
1000:       +           7,15
2000:       +           5,15
2000:       +           5,15
2000:       +           5,23
2000:       +           5,25
3000:       +           2,13
3000:       +           3,19
4000:       +           4,17
5000:       +           1,9
```

ORDER BY ROWS Query Example

Use the ORDER BY clause with the ROWS keyword to use ordering criteria to determine whether an event received by the query should be included in output. ORDER BY ROWS accepts either stream or relation input and outputs a relation.

The ORDER BY ROWS clause maintains a set of events whose maximum size is the number specified by the ROWS keyword. As new events are received, they are evaluated, based on thr order criteria and the ROWS limit, to determine whether they will be added to the output.

Note that the output of ORDER BY ROWS is not arranged based on the ordering criteria, as is the output of the ORDER BY clause. Instead, ORDER BY ROWS uses the ordering criteria and specified number of rows to determine whether to admit events into the output as they are received.

For more information, see "order_by_top_clause::=".

Consider the query `q1` in Example 22–55. Given the data stream `S0` in Example 22–56, the query returns the relation in Example 22–57.

Example 22–55 ORDER BY ROWS Query

```
<query id="q1"><![CDATA[
    SELECT c1 ,c2
    FROM S0
    ORDER BY c1,c2 ROWS 5
]]></query>
```

Example 22–56 ORDER BY ROWS Stream Input

```
Timestamp   Tuple
1000        7, 15
2000        7, 14
```

```
2000       5, 23
2000       5, 15
2000       5, 15
2000       5, 25
3000       2, 13
3000       3, 19
4000       4, 17
5000       1, 9
h 1000000000
```

Example 22–57 ORDER BY ROWS Output

```
Timestamp   Tuple Kind  Tuple
1000:       +           7,15
2000:       +           7,14
2000:       +           5,23
2000:       +           5,15
2000:       +           5,15
2000:       -           7,15
2000:       +           5,25
3000:       -           7,14
3000:       +           2,13
3000:       -           5,25
3000:       +           3,19
4000:       -           5,23
4000:       +           4,17
5000:       -           5,15
5000:       +           1,9
```

In the following example, the query uses the PARTITION keyword to specify the tuple property within which to sort events and constrain output size. Here, the PARTITION keyword specifies that events in the input should be evaluated based on their symbol value.

In other words, when determining whether to include an event in the output, the query looks at the existing set of events in output that have the same symbol. The ROWS limit is two, meaning that the query will maintain a set of sorted events that has no more than two events in it. For example, if there are already two events with the ORCL symbol, adding another ORCL event to the output will require deleting the oldest element in output having the ORCL symbol.

Also, the query is ordering events by the value property, so that is also considered when a new event is being considered for output. Here, the DESC keyword specifies that event be ordered in descending order. A new event that does not come after events already in the output set will not be included in output.

Example 22–58 ORDER BY ROWS with PARTITION Query

```
<query id="q1"><![CDATA[
    SELECT symbol, value
    FROM S0
    ORDER BY value DESC ROWS 2
    PARTITION BY symbol
]]></query>
```

Example 22–59 ORDER BY ROWS with PARTITION Input

```
Timestamp   Tuple
1000        ORCL, 500
1100        MSFT, 400
1200        INFY, 200
1300        ORCL, 503
1400        ORCL, 509
```

```
1500        ORCL, 502
1600        MSFT, 405
1700        INFY, 212
1800        INFY, 209
1900        ORCL, 512
2000        ORCL, 499
2100        MSFT, 404
2200        MSFT, 403
2300        INFY, 215
2400        MSFT, 415
2500        ORCL, 499
2600        INFY, 211
```

Example 22–60   ORDER BY ROWS with PARTITION Output

```
Timestamp    Tuple Kind    Tuple
1000         +             ORCL,500
1100         +             MSFT,400
1200         +             INFY,200
1300         +             ORCL,503
1400         -             ORCL,500
1400         +             ORCL,509
1600         +             MSFT,405
1700         +             INFY,212
1800         -             INFY,200
1800         +             INFY,209
1900         -             ORCL,503
1900         +             ORCL,512
2100         -             MSFT,400
2100         +             MSFT,404
2300         -             INFY,209
2300         +             INFY,215
2400         -             MSFT,404
2400         +             MSFT,415
```

View

Purpose

Use view statement to create a view over a base stream or relation that you reference by *identifier* in subsequent Oracle CQL statements.

Prerequisites

For more information, see:

- "Query" on page 22-2
- Chapter 20, "Oracle CQL Queries, Views, and Joins".

Syntax

You express the a view in a `<view></view>` element as Example 22–61 shows.

The `view` element has two attributes:

- `id`: Specify the `identifier` as the `view` element `id` attribute.

  The `id` value must conform with the specification given by *identifier::=* on page 7-17.

- `schema`: Optionally, specify the schema of the view as a space delimited list of attribute names.

  Oracle Event Processing server infers the types.

Example 22–61 View in a <view></view> Element

```
<view id="v2" schema="cusip bid ask"><![CDATA[
    IStream(select * from S1[range 10 slide 10])
]]></view>
```

The body of the view has the same syntax as a query. For more information, see "Query" on page 22-2.

Examples

The following examples illustrate the various semantics that this statement supports. For more examples, see Chapter 20, "Oracle CQL Queries, Views, and Joins".

Registering a View Example

Example 22–62 shows how to register view `v2`.

Example 22–62 REGISTER VIEW

```
<view id="v2" schema="cusip bid ask"><![CDATA[
    IStream(select * from S1[range 10 slide 10])
]]></view>
```