Oracle® CEP

CQL Language Reference

11*g* Release 1 (11.1.1)

E12048-02

October 2009

ORACLE®

Oracle CEP CQL Language Reference, 11*g* Release 1 (11.1.1)

E12048-02

Primary Author: Peter Purich

Contents

2 Basic Elements of Oracle CQL

3 Pseudocolumns

4 Operators

7 Functions: Colt Single-Row

Index

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 complex event processing (CEP) using Oracle CEP. Oracle CQL is a new technology but it is based on a subset of SQL99.

Oracle CEP (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 CEP 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 CEP supports ultra-high throughput and microsecond latency using JRockit Real Time and provides Oracle CEP Visualizer and Oracle CEP 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

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Deaf/Hard of Hearing Access to Oracle Support Services

To reach Oracle Support Services, use a telecommunications relay service (TRS) to call Oracle Support at 1.800.223.1711. An Oracle Support Services engineer will handle technical issues and provide customer support according to the Oracle service request process. Information about TRS is available at `http://www.fcc.gov/cgb/consumerfacts/trs.html`, and a list of phone numbers is available at `http://www.fcc.gov/cgb/dro/trsphonebk.html`.

Related Documents

For more information, see the following:

- *Oracle CEP Getting Started*
- *Oracle CEP Administrator's Guide*
- *Oracle CEP IDE Developer's Guide for Eclipse*
- *Oracle CEP Visualizer User's Guide*
- *Oracle CEP Java API Reference*
- *Oracle CEP EPL Language Reference*
- *Oracle Database SQL Language Reference*
- SQL99 Specifications (ISO/IEC 9075-1:1999, ISO/IEC 9075-2:1999, ISO/IEC 9075-3:1999, and ISO/IEC 9075-4:1999)
- Oracle CEP Forum: `http://forums.oracle.com/forums/forum.jspa?forumID=820`
- *Oracle CEP Samples*: `http://www.oracle.com/technologies/soa/complex-event-processing.html`
- Oracle Event Driven Architecture Suite sample code: `http://www.oracle.com/technology/sample_code/products/event-driven-architecture`

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.

1 Introduction to Oracle CQL

Oracle Continuous Query Language (Oracle CQL) is 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 complex event processing (CEP) using Oracle CEP.

Oracle CEP (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 CEP Service Engine, which provides a rich, declarative environment based on Oracle CQL to improve the efficiency and effectiveness of managing business operations. Oracle CEP supports ultra-high throughput and microsecond latency using JRockit Real Time and provides Oracle CEP Visualizer and Oracle CEP IDE for Eclipse developer tooling for a complete real time end-to-end Java Event-Driven Architecture (EDA) development platform.

1.1 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 CEP, 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.

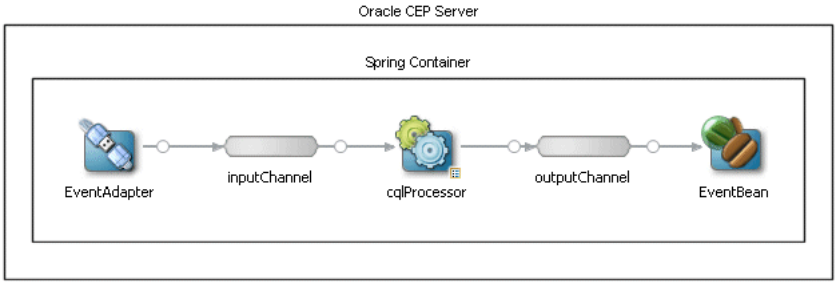
Oracle CQL is designed to be:

- 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 CEP architecture. Oracle CEP server provides the light-weight Spring container for Oracle CEP applications. The Oracle CEP 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 CEP Architecture



Using Oracle CEP, 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 CEP IDE for Eclipse and Oracle CEP 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 CEP application and deploy it to Oracle CEP 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 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>
```

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 1.1.1, "Streams and Relations"
- Section 1.1.2, "Relation-to-Relation Operators"
- Section 1.1.3, "Stream-to-Relation Operators (Windows)"
- Section 1.1.4, "Relation-to-Stream Operators"
- Section 1.1.5, "Stream-to-Stream Operators"
- Section 1.1.6, "Queries, Views, and Joins"
- Section 1.1.7, "Pattern Recognition"
- Section 1.1.8, "Event Sources and Event Sinks"
- Section 1.1.9, "Functions"
- Section 1.1.10, "Time"
- Section 1.2, "Oracle CQL Statements"
- Section 1.2.1, "Oracle CQL Statement Lexical Conventions"
- Section 1.2.2, "Oracle CQL Statement Documentation Conventions"

For more information on Oracle CEP server and tools, see:

- Section 1.4, "Oracle CEP Server and Tools Support."
- *Oracle CEP IDE Developer's Guide for Eclipse*
- *Oracle CEP Visualizer User's Guide*
- *Oracle CEP Administrator's Guide*

1.1.1 Streams and Relations

This section introduces the two fundamental Oracle CEP 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

1.1.1.1 Streams

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

Stream `S` is a bag 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 1.1.10, "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.

Oracle CEP represents a stream as a channel as Figure 1–2 shows. Using Oracle CEP 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 CEP, 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 CEP processor types.

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` assembly source: the `eventTypeName` property specifies event type `PriceEvent`. The `event-type-repository` defines the property names and types for this event.

Example 1–3 Stream Schema Definition

```
<wlevs:event-type-repository>
    <wlevs:event-type type-name="PriceEvent">
        <wlevs:properties>
            <wlevs:property name="cusip" type="java.lang.String" />
            <wlevs:property name="bid" type="java.lang.Double" />
            <wlevs:property name="srcId" type="java.lang.String" />
            <wlevs:property name="bidQty" type="java.lang.Integer" />
            <wlevs:property name="ask" type="java.lang.Double" />
            <wlevs:property name="askQty" type="java.lang.Integer" />
            <wlevs:property name="seq" type="java.lang.Long" />
            <wlevs:property name="sector" type="java.lang.String" />
        </wlevs:properties>
    </wlevs:event-type>
</wlevs:event-type-repository>

<wlevs:adapter advertise="true" id="PriceAdapter"
<wlevs:instance-property name="port" value="9008" />
    <wlevs:instance-property name="eventTypeName"
        value="PriceEvent" />
    <wlevs:instance-property name="eventPropertyNames"
        value="srcId,sector,cusip,bid,ask,bidQty,askQty,seq" />
</wlevs:adapter>
```

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 as Example 1–4 shows.

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>
```

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 source, 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 "Channel Component Configuration" in the *Oracle CEP IDE Developer's Guide for Eclipse*.

Example 1–5 Using selector to Control Which Query Results are Output

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

For more information, see:

- Section 1.1.8, "Event Sources and Event Sinks"
- Section 14.1, "Introduction to Oracle CQL Queries, Views, and Joins"
- *Oracle CEP IDE Developer's Guide for Eclipse*

1.1.1.2 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 CEP 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 CEP 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 CEP, you do not create base relations yourself. The Oracle CEP 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 Section 1.1.10, "Time".

1.1.1.3 Relations and Oracle CEP Tuple Kind Indicator

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

Example 1–6 Oracle CEP 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 CEP 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 CEP Java API Reference*).

To configure silent operation, see attribute `is-silent-relation` in "wlevs:channel" in the *Oracle CEP IDE Developer's Guide for Eclipse*.

In this guide, relations are always shown as not silent: that is, timestamp and Oracle CEP tuple kind indicator are always shown as Example 1–6 shows.

1.1.2 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 CEP 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.

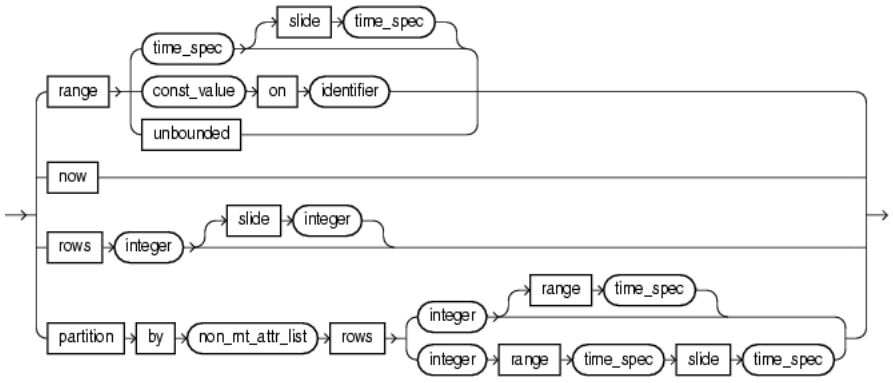
For more information on `distinct`, see:

- Chapter 6, "Functions: Aggregate"
- *select_clause::=* on page 16-3

1.1.3 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]`

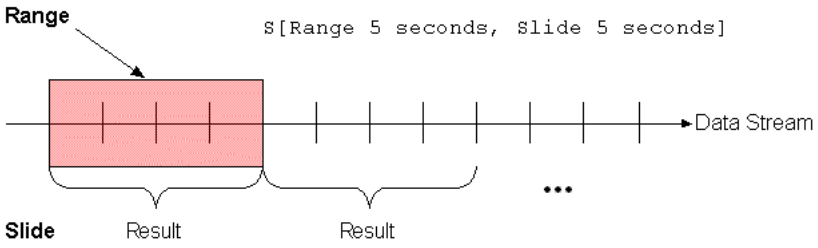
- partitioned:

  `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]
```

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 keyword `Slide` refers to how often you want a result. 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".

Figure 1–3 Range and Slide



The keyword `Partition By` logically partitions a data 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". By default, the range (and slide) is 1 second.

For more information, see:

- "Range-Based Stream-to-Relation Window Operators" on page 4-5
- "Tuple-Based Stream-to-Relation Window Operators" on page 4-13
- "Partitioned Stream-to-Relation Window Operators" on page 4-18

1.1.4 Relation-to-Stream Operators

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

In Example 1–8, 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 1–8 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 4-24.

- `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 4-25.

- `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 4-26.

By default, Oracle CEP 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 1.1.1.3, "Relations and Oracle CEP Tuple Kind Indicator".

1.1.5 Stream-to-Stream Operators

There are no specific operators for stream to stream operations. Instead, 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 1.1.3, "Stream-to-Relation Operators (Windows)".
- A relation-to-relation operator to perform a relational filter. For more information, see Section 1.1.2, "Relation-to-Relation Operators".
- A relation-to-stream operator to turn the relation back into a stream. For more information, see Section 1.1.4, "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 1–9 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 1–9 Stream-to-Stream Operation

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

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 1.1.7, "Pattern Recognition".
- `XMLTABLE`: use this clause to parse data from the `xmltype` stream elements using XPath expressions. For more information, see Section 14.2.5, "XMLTable Query".

1.1.6 Queries, Views, and Joins

An Oracle CQL query is an operation that you express in Oracle CQL syntax and execute on an Oracle CEP 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 14.2, "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 14.3, "Views".

A **join** is a query that combines rows from two or more streams, views, or relations. For more information, see Section 14.4, "Joins".

For more information, see Chapter 14, "Oracle CQL Queries, Views, and Joins".

1.1.7 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 15, "Pattern Recognition With MATCH_RECOGNIZE".

1.1.8 Event Sources and Event Sinks

An Oracle CEP 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.

1.1.8.1 Event Sources

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

In Oracle CEP, the following elements may be event sources:

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

- cache
- relational database table

> **Note:** In Oracle CEP, 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 CEP processor types. For more information, see Section 1.1.1, "Streams and Relations".

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

Oracle CEP `table` and `cache` event sources are pull data sources: that is, Oracle CEP will periodically poll the event source.

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:

- "Configuring Access to a Relational Database" in the *Oracle CEP Administrator's Guide*
- "Configuring an Oracle CQL Processor Table Source" in the *Oracle CEP IDE Developer's Guide for Eclipse*

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. For more information, see:

- "Configuring Caching" in the *Oracle CEP IDE Developer's Guide for Eclipse*
- "Configuring an Oracle CQL Processor Cache Source" in the *Oracle CEP IDE Developer's Guide for Eclipse*
- "Oracle Continuous Query Language (CQL) Example" in the *Oracle CEP Getting Started*

1.1.8.2 Event Sinks

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

In Oracle CEP, the following elements may be event sinks:

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

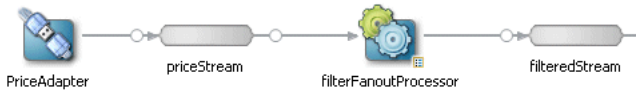
You can associate the same query with more than one event sink and with different types of event sink.

1.1.8.3 Connecting Event Sources and Event Sinks

In Oracle CEP, you define event sources and event sinks using Oracle CEP IDE for Eclipse to create the Event Processing Network (EPN) as Figure 1–4 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 1–4 Event Sources and Event Sinks in the Event Processing Network



For more information, see:

- Section 1.1.1, "Streams and Relations"
- Section 14.1, "Introduction to Oracle CQL Queries, Views, and Joins"
- *Oracle CEP IDE Developer's Guide for Eclipse*

1.1.9 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 CEP attempts to convert the argument to the expected datatype before performing the Oracle CQL function.

For more information, see:

- Chapter 5, "Functions: Single-Row"
- Chapter 6, "Functions: Aggregate"

- Chapter 7, "Functions: Colt Single-Row"
- Chapter 8, "Functions: Colt Aggregate"
- Chapter 9, "Functions: java.lang.Math"
- Chapter 10, "Functions: User-Defined"
- Section 2.3.4, "Data Conversion"

1.1.10 Time

Timestamps are an integral part of an Oracle CEP 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 CEP 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 CEP IDE Developer's Guide for Eclipse*.

For system timestamped relations or streams, time is dependent upon the arrival of data on the relation or stream data source. Oracle CEP 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 CEP 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 CEP IDE Developer's Guide 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 CEP 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 CEP IDE Developer's Guide for Eclipse*.

The Oracle CEP 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 CEP IDE Developer's Guide for Eclipse*.

1.2 Oracle CQL Statements

Oracle CQL provides statements for creating the following:

- queries
- views

- functions
- windows

For more information, see:

- Section 1.2.1, "Oracle CQL Statement Lexical Conventions"
- Section 1.2.2, "Oracle CQL Statement Documentation Conventions"
- Chapter 14, "Oracle CQL Queries, Views, and Joins"
- Chapter 16, "Oracle CQL Statements"

1.2.1 Oracle CQL Statement Lexical Conventions

Using Oracle CEP IDE for Eclipse or Oracle CEP Visualizer, you write Oracle CQL statements in the XML configuration file associated with an Oracle CEP 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 1–10 shows.

Example 1–10 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 `rule`, `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 1–10 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 CEP evaluates the Oracle CQL statement in Example 1–11 and Example 1–12 in the same manner.

Example 1–11 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 1–12 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 text literals and quoted names.

  For more information, see:

  – Section 2.4, "Literals"

  – Section 2.9, "Schema Object Names and Qualifiers"

- Comments are not permitted in Oracle CQL statements. For more information, see Section 2.7, "Comments".

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

1.2.2 Oracle CQL Statement Documentation Conventions

All Oracle CQL statements in this reference (see Chapter 16, "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.

1.3 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.

1.4 Oracle CEP Server and Tools Support

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

1.4.1 Oracle CEP Server

Oracle CEP server provides the light-weight Spring container for Oracle CEP 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 CEP server, see *Oracle CEP Administrator's Guide*.

1.4.2 Oracle CEP Tools

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

- Oracle CEP IDE for Eclipse: an Eclipse-based integrated development environment supporting the complete lifecycle of development for Oracle CEP applications, including Oracle CQL syntax highlighting and validation, graphical Event Processing Network editor, and efficient management of component and application assembly configuration files.

  For more information, see *Oracle CEP IDE Developer's Guide for Eclipse*.

- Oracle CEP Visualizer: a rich graphical user interface for Oracle CEP that provides runtime configuration and administration of Oracle CEP applications, queries, caching, and clustering.

  For more information, see *Oracle CEP Visualizer User's Guide*.

2

Basic Elements of Oracle CQL

This chapter contains reference information on the simplest building blocks of Oracle CQL statements.

2.1 Introduction to Basic Elements of Oracle CQL

The basic elements of Oracle CQL include:

- Section 2.2, "Datatypes"
- Section 2.3, "Datatype Comparison Rules"
- Section 2.4, "Literals"
- Section 2.5, "Format Models"
- Section 2.6, "Nulls"
- Section 2.7, "Comments"
- Section 2.8, "Aliases"
- Section 2.9, "Schema Object Names and Qualifiers"

Before using the statements described in Chapter 14 and Chapter 16, you should familiarize yourself with the concepts covered in this chapter.

2.2 Datatypes

Each value manipulated by Oracle CEP has a datatype. The datatype of a value associates a fixed set of properties with the value. These properties cause Oracle CEP 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.

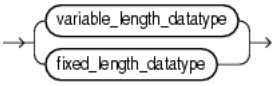
If Oracle CQL does not support a datatype that your events use, you can create a user-defined function to evaluate that datatype in an Oracle CQL query.

For more information, see:

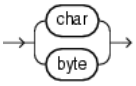
- Section 2.2.1, "Oracle CQL Built-in Datatypes"

- Section 2.2.2, "Handling Other Datatypes Using a User-Defined Function"
- Section 2.3, "Datatype Comparison Rules"
- Section 2.4, "Literals"
- Section 2.5, "Format Models"

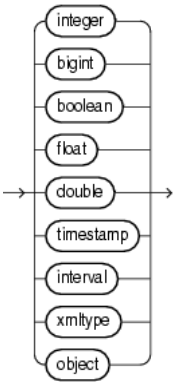
***datatype*::=**



***variable_length_datatype*::=**



***fixed_length_datatype*::=**



2.2.1 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.

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 2.4.2, "Numeric Literals".
`BOOLEAN`	Fixed-length boolean equivalent to a Java `Boolean` type. Valid values are `true` or `false`.
`BYTE[(`*`size`*`)]`	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 2.4.2, "Numeric Literals".
`CHAR [(`*`size`*`)]`	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 2.4.1, "Text Literals".
`DOUBLE`	Fixed-length number equivalent to a Java `double` type. For more information, see Section 2.4.2, "Numeric Literals".
`FLOAT`	Fixed-length number equivalent to a Java `float` type. For more information, see Section 2.4.2, "Numeric Literals".

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

Oracle CQL Datatype	Description
INTEGER	Fixed-length number equivalent to a Java `int` type. For more information, see Section 2.4.2, "Numeric Literals".
INTERVAL	Fixed-length `INTERVAL` datatype specifies a period of time. Oracle CEP supports `DAY TO SECOND`. Maximum length is 64 bytes. For more information, see Section 2.4.4, "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 2.4.3, "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. For more information, see "SQL/XML (SQLX)" on page 11-16.
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 CEP since the Oracle CEP 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 CEP 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.

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

2.2.2 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 10, "Functions: User-Defined".

2.3 Datatype Comparison Rules

This section describes how Oracle CEP compares values of each datatype.

2.3.1 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.

2.3.2 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'.

2.3.3 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'
```

2.3.4 Data 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 CEP 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.

2.3.4.1 Implicit Data Conversion

Oracle CEP 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	--	--	--	--

Table 2–2 (Cont.) Implicit Type Conversion Matrix

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

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

- During `SELECT FROM` operations, Oracle CEP 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 CEP 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 CEP converts the argument to the accepted datatype wherever supported.
- When making assignments, Oracle CEP 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 CEP converts from noncharacter datatypes to `CHAR`.
- During arithmetic operations on and comparisons between character and noncharacter datatypes, Oracle CEP converts from numeric types to `CHAR` as Table 2–2 shows.

2.3.4.2 Explicit Data 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 BIGINT	to_char	--	to_boolean	--	to_double	--	to_float	--	--
from FLOAT	to_char	--	--	--	to_double	--	--	--	--
from TIMESTAMP	to_char	--	--	--	--	--	--	--	--
from INTERVAL	to_char	--	--	--	--	--	--	--	--

2.3.4.3 User-Defined Function Data Conversion

At run time, Oracle CEP 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 10.1.2, "User-Defined Function Datatypes".

2.4 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 CEP supports the following types of literals in Oracle CQL statements:

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

2.4.1 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 CEP 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 CEP supports Java localization. You can specify text literals in the character set specified by your Java locale.

For more information, see:

- Section 1.2.1, "Oracle CQL Statement Lexical Conventions"
- Section 2.9, "Schema Object Names and Qualifiers"
- *const_string::=* on page 13-7

2.4.2 Numeric Literals

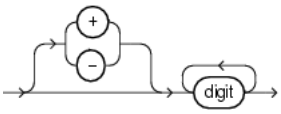
Use numeric literal notation to specify fixed and floating-point numbers.

2.4.2.1 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:

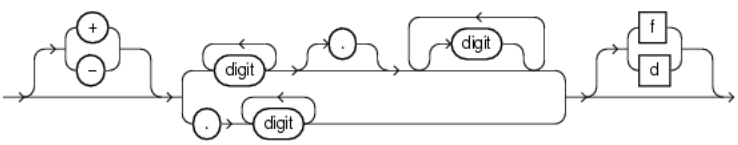
```
7
+255
```

2.4.2.2 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 CEP is expecting an INTEGER and it encounters the string '9', then it converts the string to the Java Integer 9. However, if Oracle CEP 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 CEP truncates the value. If the range of the literal exceeds the range supported by BIGINT or FLOAT, then Oracle CEP 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 CEP 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
```

2.4.3 Datetime Literals

Oracle CEP 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 2.5.2, "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 2.5.2, "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 11-16.

2.4.4 Interval Literals

An interval literal specifies a period of time. Oracle CEP 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.

2.4.4.1 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 2.5.2, "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 Comapring 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>
```

2.5 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 CEP interprets the string. The following format models are relevant to Oracle CQL queries:

- Number Format Models
- Datetime Format Models

2.5.1 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

2.5.2 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 2.4.3, "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.

2.6 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 (see *out_of_line_constraint::=* on page 13-19). Use a null when the actual value is not known or when a value would not be meaningful.

Oracle CEP 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 CEP 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 5-8
- *null_spec::=* on page 16-5

2.6.1 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.

2.6.2 Nulls with Comparison Conditions

To test for nulls, use only the null comparison conditions (see *null_conditions::=* on page 12-8). 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 CEP considers two nulls to be equal when evaluating a decode expression. See *decode::=* on page 11-13 for syntax and additional information.

2.6.3 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 12.6, "Null Conditions".

2.7 Comments

Oracle CQL does not support comments.

2.8 Aliases

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 2.8.1, "Aliases in the relation_variable Clause"
- Section 2.8.2, "Aliases in Window Operators"

For more information, see Chapter 14, "Oracle CQL Queries, Views, and Joins".

2.8.1 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 14.2.6, "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
  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 14.2.1.3, "From Clause".

2.8.2 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 1.1.3, "Stream-to-Relation Operators (Windows)".

2.9 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, windows, streams, views, and user-defined functions. This section provides:

- Section 2.9.1, "Schema Object Naming Rules"
- Section 2.9.2, "Schema Object Naming Guidelines"
- Section 2.9.3, "Schema Object Naming Examples"

For more information, see Section 1.2.1, "Oracle CQL Statement Lexical Conventions".

2.9.1 Schema Object Naming Rules

Every Oracle CEP object has a name. In a Oracle CQL statement, you represent the name of an object with a **quoted identifier** or a **nonquoted identifier**.

- A quoted identifier begins and ends with double quotation marks ("). If you name a schema object using a quoted identifier, then you must use the double quotation marks whenever you refer to that object.
- A nonquoted identifier is not surrounded by any punctuation.

You can use either quoted or nonquoted identifiers to name any Oracle CEP object.

Names are case-sensitive.

The following list of rules applies to both quoted and nonquoted identifiers unless otherwise indicated:

1. Names must be from 1 to 30 characters long.

   If an identifier includes multiple parts separated by periods, then each attribute can be up to 30 bytes long. Each period separator, and any surrounding double quotation marks, counts as one byte. For example, suppose you identify a stream element like this:

   ```
   "stream"."attribute"
   ```

   The stream name can be 30 bytes and the element name can be 30 bytes. Each of the quotation marks and periods is a single-byte character, so the total length of the identifier in this example can be up to 65 bytes.

2. Nonquoted identifiers cannot be Oracle CEP reserved words. Quoted identifiers can be reserved words, although this is not recommended.

   Depending on the Oracle product you plan to use to access an Oracle CEP 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 13-11
   - "unreserved_keyword" on page 13-12
   - "reserved_keyword" on page 13-12

3. Oracle recommends that you use ASCII characters in schema object names because ASCII characters provide optimal compatibility across different platforms and operating systems.
4. Nonquoted identifiers must begin with an alphabetic character from your database character set. Quoted identifiers can begin with any character.
5. Nonquoted identifiers can contain only alphanumeric characters from your Java locale's character set and the underscore (_), dollar sign ($), and pound sign (#). Oracle strongly discourages you from using $ and # in nonquoted identifiers.

   Quoted identifiers can contain any characters and punctuations marks and spaces. However, neither quoted nor nonquoted identifiers can contain double quotation marks or the null character (`\0`).

   For more information, see:

   - *const_string::=* on page 13-7
   - *identifier::=* on page 13-11
6. In general, you should choose names that are unique across an application for the following objects:

   - Streams
   - Queries
   - Views
   - Windows
   - User-defined functions

   Specifically, a query and view cannot have the same name.
7. Nonquoted identifiers are not case sensitive. Oracle CEP interprets them as uppercase. Quoted identifiers are case sensitive.

   By enclosing names in double quotation marks, you can give the following names to different objects in the same namespace:

   ```
   employees
   "employees"
   "Employees"
   "EMPLOYEES"
   ```

   Note that Oracle CEP interprets the following names the same, so they cannot be used for different objects in the same namespace:

   ```
   employees
   EMPLOYEES
   "EMPLOYEES"
   ```
8. 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.
9. 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 10, "Functions: User-Defined"

2.9.2 Schema Object Naming Guidelines

Here are several helpful 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 objective of keeping names short and easy to use with the objective of making names as descriptive as possible. When in doubt, choose the more descriptive name, because the objects in Oracle CEP 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`.

2.9.3 Schema Object Naming Examples

The following examples are valid schema object names:

```
last_name
horse
hr.hire_date
"EVEN THIS & THAT!"
a_very_long_and_valid_name
```

All of these examples adhere to the rules listed in Section 2.9.1, "Schema Object Naming Rules". The following example is not valid, because it exceeds 30 characters:

```
a_very_very_long_and_invalid_name
```

3

Pseudocolumns

A **pseudocolumn** behaves like a stream element, but is not actually part of the tuple.

3.1 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 1.1.9, "Functions").

Oracle CQL supports the following pseudocolumns:

- Section 3.2, "ELEMENT_TIME Pseudocolumn"

3.2 ELEMENT_TIME Pseudocolumn

The `ELEMENT_TIME` pseudocolumn returns the timestamp value associated with a given stream element as a `java.lang.Math.Bigint(Long)`.

For syntax, see *pseudo_column::=* on page 13-2.

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
```

Example 3–4 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 3–4 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

An **operator** manipulates data items and returns a result. Syntactically, an operator appears before or after an operand or between two operands.

4.1 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:

4.1.1 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 (`||`).

4.1.2 What You May Need to Know About Operator Precedence

Precedence is the order in which Oracle CEP evaluates different operators in the same expression. When evaluating an expression containing multiple operators, Oracle CEP evaluates operators with higher precedence before evaluating those with lower precedence. Oracle CEP 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 12, "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 CEP 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 CEP 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 Data 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 CEP 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 2-9
- "Interval Literals" on page 2-10

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 CEP 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 CEP. 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 2.2, "Datatypes"
- "concat" on page 5-3
- "xmlconcat" on page 5-23
- "nvl" on page 5-8

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

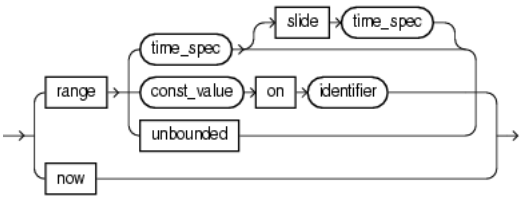
Example 4–1 Concatenation Operator (||)

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

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 13-21)

- 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 16-2
- "Stream-to-Relation Operators (Windows)" on page 1-8
- Section 2.8.2, "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 CEP is nanoseconds and hence all these tuples expire 1 nanosecond later.

For an example, see "S [now] Example" on page 4-6.

Examples

S [now] Example

Consider the query `q1` in Example 4–2 and the data stream `S` in Example 4–3. Timestamps are shown in nanoseconds (`1 sec = 10^9 nanoseconds`). Example 4–4 shows the relation that the query returns at time `5000 ms`. At time `5002 ms`, the query would return an empty relation.

Example 4–2 S [now] Query

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

Example 4–3 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 4–4 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 4-7.

Examples

S [range T] Example

Consider the query `q1` in Example 4–5. Given the data stream `S` in Example 4–6, the query returns the relation in Example 4–7. By default, the range time unit is `second` (see *time_spec::=* on page 13-21) 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 CEP that no further input will have a timestamp lesser than the heartbeat value.

Example 4–5 S [range T] Query

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

Example 4–6 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 4–7 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). So if two tuples arrive between the time period `n*T2` and `(n+1)*T2`, both of them will be visible (enter the window) only at `(n+1)*T2` and will expire from the window at `(n+1)*T2+T1`.

For an example, see "S [range T1 slide T2] Example" on page 4-8.

Examples

S [range T1 slide T2] Example

Consider the query `q1` in Example 4–8. Given the data stream `S` in Example 4–9, the query returns the relation in Example 4–10. By default, the range time unit is `second` (see *time_spec::=* on page 13-21) 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 4–8 S [range T1 slide T2] Query

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

Example 4–9 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 4–10 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 4-10.

Examples

S [range unbounded] Example

Consider the query `q1` in Example 4–11 and the data stream `S` in Example 4–12. 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 4–13 shows the relation that the query returns at time `5000 ms` and Example 4–14 shows the relation that the query returns at time `205000 ms`.

Example 4–11 S [range unbounded] Query

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

Example 4–12 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 4–13 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 4–14 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 4-11
- "S [range C on E] Example: INTERVAL and TIMESTAMP" on page 4-12

Examples

S [range C on E] Example: Constant Value

Consider the query `tkdata56_q0` in Example 4–15 and the data stream `tkdata56_S0` in Example 4–16. Stream `tkdata56_S0` has schema `(c1 integer, c2 float)`. Example 4–17 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 4–15 S [range C on E] Constant Value: Query

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

Example 4–16 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 4–17 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 4–18 and the data stream `tkdata56_S1` in Example 4–19. Stream `tkdata56_S1` has schema `(c1 timestamp, c2 double)`. Example 4–20 shows the relation that the query returns.

Example 4–18 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 4–19 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 4–20 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 4-5
- "Query" on page 16-2
- "Stream-to-Relation Operators (Windows)" on page 1-8
- Section 2.8.2, "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 CEP 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 4-14.

Examples

S [rows N] Example

Consider the query q1 in Example 4–21 and the data stream S in Example 4–22. 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 4-7).

Example 4–23 shows the relation that the query returns at time 1003 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 4–24 shows the relation that the query returns at time 1007 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 1007 ms, namely tuples (15,0.14), (33,4.4), and (23,56.33).

Example 4–25 shows the relation that the query returns at time 2001 ms. At this time, Oracle CEP deletes elements that have been in the window longer than the default range (and slide) of 1000 ms (1 s). 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 tuple (17,1.3).

Example 4–21 S [rows N] Query

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

Example 4–22 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 4–23 S [rows N] Relation Output at Time 1003 ms

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

Example 4–24 S [rows N] Relation Output at Time 1007 ms

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

Example 4–25 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:     -           23,56.33
  1008:     -           34,4.4
  1010:     -           20,0.2
  1012:     -           45,23.44
  1014:     -           30,0.3
  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 CEP 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 CEP delays adding stream elements to the relation until it receives N2 number of elements.

For examples, see "S [rows N] Example" on page 4-14.

Examples

S [rows N1 slide N2] Example

Consider the query tkdata55_q0 in Example 4–26 and the data stream tkdata55_S55 in Example 4–27. Stream tkdata55_S55 has schema (c1 integer, c2 float). The output relation is shown in Example 4–28.

As Example 4–28 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 4–26 S [rows N1 slide N2] Query

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

Example 4–27 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
```

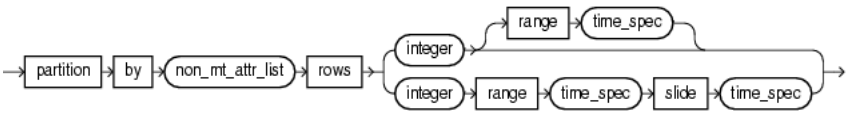
Example 4–28 S [rows N1 slide N2] Relation Output

```
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 13-21, *non_mt_attr_list*::= on page 13-14)

- 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 4-13
- "Query" on page 16-2
- "Stream-to-Relation Operators (Windows)" on page 1-8
- Section 2.8.2, "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 4-19.

Examples

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

Consider the query `qPart_row2` in Example 4–29 and the data stream `SP1` in Example 4–30. Stream `SP1` has schema `(c1 integer, name char(10))`. The query returns the relation in Example 4–31. 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 CEP that no further input will have a timestamp lesser than the heartbeat value.

Example 4–29 S[partition by A1, ..., Ak rows N] Query

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

Example 4–30 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 4–31 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 4-21.

Examples

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

Consider the query `qPart_range2` in Example 4–32 and the data stream `SP5` in Example 4–33. Stream `SP5` has schema `(c1 integer, name char(10))`.The query returns the relation in Example 4–34. By default, the range time unit is `second` (see *time_spec::=* on page 13-21) so `range 2` is equivalent to `range 2 seconds`. Timestamps are shown in milliseconds (`1 s = 1000 ms`).

Example 4–32 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 4–33 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 4–34 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 4-19.

Examples

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

Consider the query `qPart_rangeslide` in Example 4–35 and the data stream `SP1` in Example 4–36. Stream `SP1` has schema `(c1 integer, name char(10))`. The query returns the relation in Example 4–37. By default, the range and slide time unit is `second` (see *time_spec::=* on page 13-21) so `range 1 slide 1` is equivalent to `range 1 second slide 1 second`. Timestamps are shown in milliseconds (`1 s = 1000 ms`).

Example 4–35 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 4–36 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 4–37 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 4–38, 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 4–38 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>
```

For more information, see *xstream_clause::=* on page 16-6.

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 4–39, 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 4–39 DStream

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

Assume that the granularity of time is minutes. Table 4–4 illustrates the contents of the range window operator's relation and the `Dstream` stream for the following input from input stream `TradeInputs`:

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

Table 4–4 Dstream Example Output

Input Stream S	Relation S[Range 10 minutes] Output	Dstream Output
05 1,1	+ 05 1,1	
	- 15 1,1	15 1,1
25 2,2	+ 25 2,2	
	- 35 2,2	35 2,2
50 3,3	+ 50 3,3	
	- 60 3,3	60 3,3

Note that at time 15, the relation is empty: `R(15) = {}` (the empty set).

For more information, see *xstream_clause::=* on page 16-6.

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 4–40, `Rstream` outputs the entire state of the relation at time `Now` and filtered by the `where` clause.

Example 4–40 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 *xstream_clause::=* on page 16-6.

5

Functions: Single-Row

Single-row functions return a single result row for every row of a queried stream or view.

For more information, see Section 1.1.9, "Functions".

5.1 Introduction to Oracle CQL Built-In Single-Row Functions

Table 5–1 lists the built-in single-row functions that Oracle CQL provides.

Table 5–1 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).

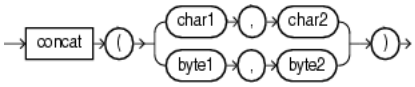
Note: In stream input examples, lines beginning with `h` (such as `h 3800`) are heartbeat input tuples. These inform Oracle CEP that no further input will have a timestamp lesser than the heartbeat value.

For more information, see:

- Section 1.1.9, "Functions"
- Section 2.2, "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 5-23.

Examples

concat Function

Consider the query `chr_concat` in Example 5–1 and data stream `S4` in Example 5–2. Stream `S4` has schema `(c1 char(10))`. The query returns the relation in Example 5–3.

Example 5–1 concat Function Query

```
<query id="chr_concat"><![CDATA[
    select
        concat(c1,c1),
        concat("abc",c1),
        concat(c1,"abc")
    from
        S4[range 5]
]]></query>
```

Example 5–2 concat Function Stream Input

```
Timestamp   Tuple
 1000
 2000       hi
 8000       hi1
 9000
15000       xyz
h 200000000
```

Example 5–3 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 5–4 and the data stream `S10` in Example 5–5. Stream `S10` has schema `(c1 integer, c2 char(10))`. The query returns the relation in Example 5–6.

Example 5–4 Concatenation Operator (||) Query

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

Example 5–5 Concatenation Operator (||) Stream Input

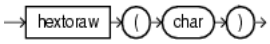
```
Timestamp   Tuple
1           1,abc
2           2,ab
3           3,abc
4           4,a
h 200000000
```

Example 5–6 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 5-13

Examples

Consider the query `q6` in Example 5–7 and the data stream `SinpByte1` in Example 5–8. Stream `SinpByte1` has schema `(c1 byte(10), c2 integer)`.The query returns the relation in Example 5–9.

Example 5–7 hextoraw Function Query

```
<query id="q6"><![CDATA[
    select * from SByt[range 2]
    where
        bytTest(c2) between hextoraw("5200") and hextoraw("5600")
]]></query>
```

Example 5–8 hextoraw Function Stream Input

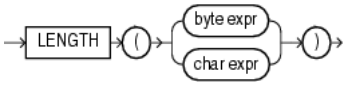
```
Timestamp   Tuple
1000        1,"51c1"
2000        2,"52"
3000        3,"53aa"
4000        4,"5"
5000         ,"55ef"
6000        6,
h 8000
h 200000000
```

Example 5–9 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 5–10 and the data stream `S2` in Example 5–11. Stream `S2` has schema `(c1 integer, c2 integer)`. The query returns the relation that Example 5–12.

Example 5–10 length Function Query

```
<query id="chr_len"><![CDATA[
    select length(c1) from S4[range 5]
]]></query>
```

Example 5–11 length Function Stream Input

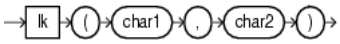
```
Timestamp   Tuple
 1000
 2000       hi
 8000       hi1
 9000
15000       xyz
h 200000000
```

Example 5–12 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 12.4, "LIKE Condition".

Examples

Consider the query `q291` in Example 5–13 and the data stream `SLk1` in Example 5–14. Stream `SLk1` has schema `(first1 char(20), last1 char(20))`. The query returns the relation in Example 5–15.

Example 5–13 lk Function Query

```
<query id="q291"><![CDATA[
    select * from SLk1
    where
        lk(first1,"^Ste(v|ph)en$")
]]></query>
```

Example 5–14 lk Function Stream Input

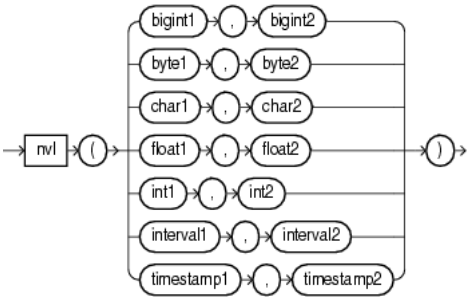
```
Timestamp   Tuple
1           Steven,King
2           Sten,Harley
3           Stephen,Stiles
4           Steven,Markles
h 200000000
```

Example 5–15 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 CEP implicitly converts one to the other. If they cannot be converted implicitly, Oracle CEP returns an error. The implicit conversion is implemented as follows:

- If *`expr1`* is character data, then Oracle CEP converts *`expr2`* to character data before comparing them and returns `VARCHAR2` in the character set of *`expr1`*.
- If *`expr1`* is numeric, then Oracle CEP 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 5–16 and the data stream `SNVL` in Example 5–17. Stream `SNVL` has schema `(c1 char(20), c2 integer)`. The query returns the relation in Example 5–18.

Example 5–16 nvl Function Query

```
<query id="q281"><![CDATA[
    select nvl(c1,"abcd") from SNVL
]]></query>
```

Example 5–17 nvl Function Stream Input

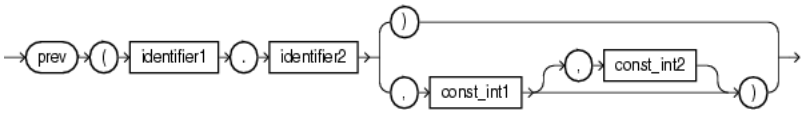
```
Timestamp   Tuple
1              ,1
2             ab,2
3            abc,3
4              ,4
h 200000000
```

Example 5–18 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 specified stream element before the 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 the following arguments:

- prev(identifier1.identifier2)
- prev(identifier1.identifier2, const_int1)
- prev(identifier1.identifier2, const_int1, const_int2)

Where:

- `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.
- `const_int1`: the index of the stream element before the current stream element to compare against. Default: 1.
- `const_int2`: 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 3.2, "ELEMENT_TIME Pseudocolumn").

See Also:

- "first" on page 6-6
- "last" on page 6-8
- "func_expr" on page 11-15
- *pattern_recognition_clause*::= on page 15-1

Examples

prev(identifier1.identifier2)

Consider query `q2` in Example 5–19 and the data stream `S1` in Example 5–20. 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 relation in Example 5–21.

Example 5–19 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 5–20 prev(identifier1.identifier2) Function Stream Input

```
Timestamp   Tuple
1000        35
3000        35
4000        10
5000         7
```

Example 5–21 prev(identifier1.identifier2) Function Relation Output

```
Timestamp   Tuple Kind  Tuple
5000:       +           35,7
```

prev(identifier1.identifier2, const_int1)

Consider query `q35` in Example 5–22 and the data stream `S15` in Example 5–23. 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 relation in Example 5–24.

Example 5–22 prev(identifier1.identifier2, const_int1) 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 5–23 prev(identifier1.identifier2, const_int1) 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 5–24 prev(identifier1.identifier2, const_int1) Function Relation Output

```
Timestamp   Tuple Kind  Tuple
 3000:       +          45
 4000:       +          45
 5000:       +          45
 6000:       +          45
 7000:       +          45
 8000:       +          45
13000:       +          52
15000:       +          43
```

prev(identifier1.identifier2, const_int1, const_int2)

Consider query `q36` in Example 5–25 and the data stream `S15` in Example 5–23. Stream `S15` has schema `(c1 integer, c2 integer)`. This example defines pattern `A` as `A.c1 = prev(A.c1,3,5000)`. 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 if the timestamp of this prior stream element is `5000`. The query returns the relation in Example 5–24.

Example 5–25 shows how to use the `prev` function to compare against a particular prior stream element with a given timestamp.

Example 5–25 prev(identifier1.identifier2, const_int1, const_int2) 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.c1,3,5000) )
    ) as T
]]></query>
```

Example 5–26 prev(identifier1.identifier2, const_int1, const_int2) Function Stream Input

```
Timestamp   Tuple
 1000       45,20
 2000       45,30
 3000       45,30
 4000       45,30
 5000       45,30
```

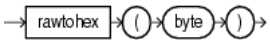
```
 6000       45,20
 7000       45,20
 8000       45,20
 9000       43,40
10000       52,10
11000       52,30
12000       43,40
13000       52,50
14000       43,40
15000       43,40
```

Example 5–27 prev(identifier1.identifier2, const_int1, const_int2) Function Relation 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 5-5

Examples

Consider the query `byte_to_hex` in Example 5–28 and the data stream `S5` in Example 5–29. 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 5–30.

Example 5–28 rawtohex Function Query

```
<query id="byte_to_hex"><![CDATA[
    select rawtohex(c2) from S5[range 4]
]]></query>
```

Example 5–29 rawtohex Function Stream Input

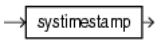
```
Timestamp   Tuple
1000        1,"51c1"
2000        2,"52"
2500        7,"axc"
3000        3,"53aa"
4000        4,"5"
5000         ,"55ef"
6000        6,
h 8000
h 200000000
```

Example 5–30 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 CEP server resides. The return type is `TIMESTAMP WITH TIME ZONE`.

Examples

Consider the query `q106` in Example 5–31 and the data stream `S0` in Example 5–32. Stream `S0` has schema `(c1 float, c2 integer)`. The query returns the relation in Example 5–33. For more information about `case`, see "case_expr" on page 11-9.

Example 5–31 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 5–32 systimestamp Function Stream Input

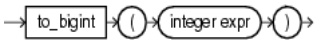
```
Timestamp   Tuple
     1000   0.1 ,10
     1002   0.14,15
   200000   0.2 ,20
   400000   0.3 ,30
   500000   0.3 ,35
   600000       ,35
h 800000
100000000   4.04,40
h 200000000
```

Example 5–33 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 11-6
- Section 2.3.4.2, "Explicit Data Conversion"

Examples

Consider the query `q282` in Example 5–34 and the data stream `S11` in Example 5–35. Stream `S11` has schema `(c1 integer, name char(10))`. The query returns the relation in Example 5–36.

Example 5–34 to_bigint Function Query

```
<query id="q282"><![CDATA[
    select nvl(to_bigint(c1), 5.2) from S11
]]></query>
```

Example 5–35 to_bigint Function Stream Input

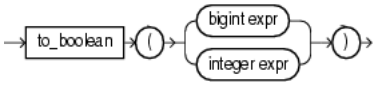
```
Timestamp   Tuple
  10        1,abc
2000         ,ab
3400        3,abc
4700         ,a
h 8000
h 200000000
```

Example 5–36 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 11-6
- Section 2.3.4.2, "Explicit Data Conversion"

Examples

Consider the query `q282` in Example 5–34 and the data stream `S11` in Example 5–35. Stream `S11` has schema `(c1 integer, name char(10))`. The query returns the relation in Example 5–36.

Example 5–37 to_boolean Function Query

```
<view id="v2" schema="c1 boolean, c2 bigint" ><![CDATA[
    select to_boolean(c1), c1 from tkboolean_S3 [now] where c2 = 0.1
]]></view>
<query id="q1"><![CDATA[
    select * from v2
]]></query>
```

Example 5–38 to_boolean Function Stream Input

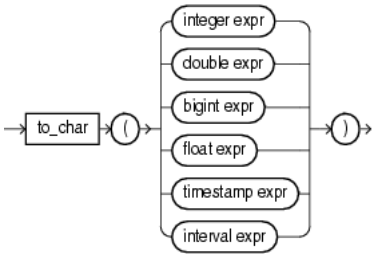
```
Timestamp   Tuple
1000        -2147483648, 0.1
2000        2147483647, 0.2
3000        12345678901, 0.3
4000        -12345678901, 0.1
5000        9223372036854775799, 0.2
6000        -9223372036854775799, 0.3
7000        , 0.1
8000        10000000000, 0.2
9000        60000000000, 0.3
h 200000000
```

Example 5–39 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 CEP returns an error.

For more information, see:

- *arith_expr::=* on page 11-6
- Section 2.3.4.2, "Explicit Data Conversion"

Examples

Consider the query `q282` in Example 5–34 and the data stream `S11` in Example 5–35. Stream `S11` has schema `(c1 integer, name char(10))`. The query returns the relation in Example 5–36.

Example 5–40 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 5–41 to_char Function Stream Input

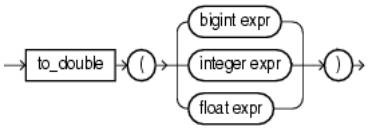
```
Timestamp   Tuple
1000        99,99999, 99.9, 99.9999, "4 1:13:48.10", "08/07/2004 11:13:48", cep
```

Example 5–42 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 CEP returns an error.

For more information, see:

- *arith_expr::=* on page 11-6
- Section 2.3.4.2, "Explicit Data Conversion"

Examples

Consider the query `q282` in Example 5–34 and the data stream `S11` in Example 5–35. Stream `S11` has schema `(c1 integer, name char(10))`. The query returns the relation in Example 5–36.

Example 5–43 to_double Function Query

```
<query id="q282"><![CDATA[
    select nvl(to_double(c1), 5.2) from S11
]]></query>
```

Example 5–44 to_double Function Stream Input

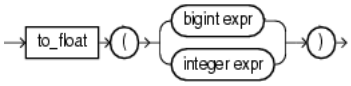
```
Timestamp   Tuple
  10        1,abc
2000         ,ab
3400        3,abc
4700         ,a
h 8000
h 200000000
```

Example 5–45 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 CEP returns an error.

For more information, see:

- *arith_expr::=* on page 11-6
- Section 2.3.4.2, "Explicit Data Conversion"

Examples

Consider the query `q1` in Example 5–46 and the data stream `S11` in Example 5–47. Stream `S1` has schema `(c1 integer, name char(10))`. The query returns the relation in Example 5–48.

Example 5–46 to_float Function Query

```
<query id="q1"><![CDATA[
    select nvl(to_float(c1), 5.2) from S11
]]></query>
```

Example 5–47 to_float Function Stream Input

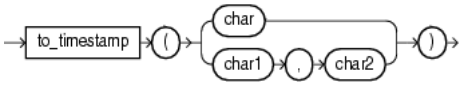
```
Timestamp   Tuple
  10        1, abc
2000         , ab
3400        3, abc
4700         , a
h 8000
h 200000000
```

Example 5–48 to_float Function Relation Output

```
Timestamp   Tuple Kind  Tuple
10:+ 1.0
2000:+ 5.2
3400:+ 3.0
4700:+ 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.

For more information, see:

- Section 2.2.1, "Oracle CQL Built-in Datatypes"
- Section 2.4.3, "Datetime Literals"
- Section 2.5.2, "Datetime Format Models"

Examples

Consider the query `q277` in Example 5–49 and the data stream `STs2` in Example 5–50. Stream `STs2` has schema `(c1 integer, c2 char(20))`. The query returns the relation that Example 5–51.

Example 5–49 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 5–50 to_timestamp Function Stream Input

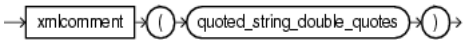
```
Timestamp   Tuple
1           1,"040807111348"
2           2,"050907101348"
3           3,"041007111348"
4           4,"060806111248"
h 200000000
```

Example 5–51 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 13-7
- "SQL/XML (SQLX)" on page 11-16

Examples

Consider the query `tkdata64_q1` in Example 5–52 and data stream `tkdata64_S` in Example 5–53. Stream `tkdata64_S` has schema `(c1 char(30))`. The query returns the relation in Example 5–54.

Example 5–52 xmlcomment Function Query

```
<query id="tkdata64_q1"><![CDATA[
    xmlconcat(xmlelement("parent", c1), xmlcomment("this is a comment"))
from tkdata64_S
]]></query>
```

Example 5–53 xmlcomment Function Stream Input

```
Timestamp   Tuple
c 30
1000        "san jose"
1000        "mountain view"
1000
1000        "sunnyvale"
1003
1004        "belmont"
```

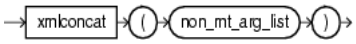
Example 5–54 xmlcomment Function Relation Output

```
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 13-13.

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 11-16.

To concatenate datatypes other than `xmltype`, use `CONCAT`. For more information, see "concat" on page 5-3.

See Also: "SQL/XML (SQLX)" on page 11-16

Examples

Consider the query `tkdata64_q1` in Example 5–52 and the data stream `tkdata64_S` in Example 5–53. Stream `tkdata64_S` has schema `(c1 char(30))`. The query returns the relation in Example 5–54.

Example 5–55 xmlconcat Function Query

```
<query id="tkdata64_q1"><![CDATA[
    select
        xmlconcat(xmlelement("parent", c1), xmlcomment("this is a comment"))
    from tkdata64_S
]]></query>
```

Example 5–56 xmlconcat Function Stream Input

```
Timestamp   Tuple
c 30
1000        "san jose"
1000        "mountain view"
1000
1000        "sunnyvale"
1003
1004        "belmont"
```

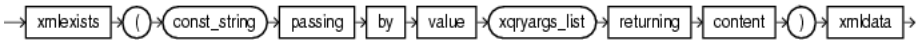
Example 5–57 xmlconcat Function Relation Output

```
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 CEP applies to the XML stream element data that you bind in `xqryargs_list`. For more information, see *const_string::=* on page 13-7.
- `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 13-25.

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 11-16.

See Also:

- "xmlquery" on page 5-27
- *func_expr::=* on page 11-15
- *xmltable_clause::=* on page 16-6
- "SQL/XML (SQLX)" on page 11-16

Examples

Consider the query `q1` in Example 5–58 and the XML data stream `S` in Example 5–59. 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 5–60.

Example 5–58 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 5–59 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 5–60 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 CEP applies to the XML stream element data that you bind in `xqryargs_list`. For more information, see *const_string::=* on page 13-7.
- `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 13-25.

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 11-16.

See Also:

- "xmlexists" on page 5-25
- *func_expr::=* on page 11-15
- *xmltable_clause::=* on page 16-6
- "SQL/XML (SQLX)" on page 11-16

Examples

Consider the query `q1` in Example 5–61 and the XML data stream `S` in Example 5–62. 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 5–63.

Example 5–61 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 5–62 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 5–63 xmlquery Function Relation Output

```
Timestamp   Tuple Kind  Tuple
3:          +
4:          +
5:          +           "<PDName>hello1</PDName>"
6:          +           "<PDName>hello2</PDName>"
```

6

Functions: Aggregate

Aggregate functions perform a summary operation on all the values that a query returns.

For more information, see Section 1.1.9, "Functions".

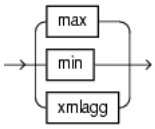
6.1 Introduction to Oracle CQL Built-In Aggregate Functions

Table 6–1 lists the built-in aggregate functions that Oracle CQL provides:

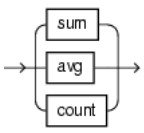
Table 6–1 Oracle CQL Built-in Aggregate Functions

Type	Function
Aggregate	▪ `max` ▪ `min` ▪ `xmlagg`
Aggregate (incremental computation)	▪ `avg` ▪ `count` ▪ `sum`
Extended aggregate	▪ `first` ▪ `last`

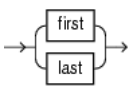
***builtin_aggr*::=**



***builtin_aggr_incr*::=**



***extended_builtin_aggr*::=**



Specify `distinct` if you want Oracle CEP 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 11-3.

Oracle CEP 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).

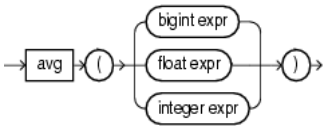
Note: In stream input examples, lines beginning with `h` (such as `h 3800`) are heartbeat input tuples. These inform Oracle CEP that no further input will have a timestamp lesser than the heartbeat value.

For more information, see:

- Section 1.1.2, "Relation-to-Relation Operators"
- Section 1.1.9, "Functions"
- Section 2.2, "Datatypes"
- *select_clause::=* on page 16-3

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 6–1 and the data stream `S3` in Example 6–2. Stream `S3` has schema `(c1 float)`. The query returns the relation in Example 6–3. Note that the `avg` function returns a result of `NaN` if the average value is not a number. For more information, see Section 2.4.2, "Numeric Literals".

Example 6–1 avg Function Query

```
<query id="float_avg"><![CDATA[
    select avg(c1) from S3[range 5]
]]></query>
```

Example 6–2 avg Function Stream Input

```
Timestamp   Tuple
 1000
 2000        5.5
 8000        4.4
 9000
15000       44.2
h 200000000
```

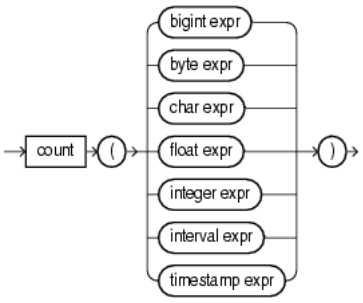
Example 6–3 avg Function Relation Output

```
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



Purpose

`count` returns the number of tuples returned by the query as an `int` value.

If you specify *`expr`*, then `count` returns the number of tuples where *`expr`* is not null.

If you specify the asterisk (`*`), then this function returns a count of all tuples, including duplicates and nulls. `count` never returns null.

Example

Consider the query `q2` in Example 6–4 and the data stream `S2` in Example 6–5. Stream `S2` has schema `(c1 integer, c2 integer)`. The query returns the relation in Example 6–6. For more information on `range` windows, see "Range-Based Stream-to-Relation Window Operators" on page 4-5.

Example 6–4 count Function Query

```
<query id="q2"><![CDATA[
    SELECT COUNT(c2), COUNT(*) FROM  S [RANGE 10]
]]></query>
```

Example 6–5 count Function Stream Input

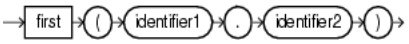
```
Timestamp   Tuple
1000        1,2
2000        1,
3000        1,4
6000        1,6
```

Example 6–6 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
```

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 6-8
- "prev" on page 5-9
- *pattern_recognition_clause::=* on page 15-1

Examples

Consider the query `q9` in Example 6–7 and the data stream `S0` in Example 6–8. 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 6–9.

Example 6–7 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 6–8 first Function Stream Input

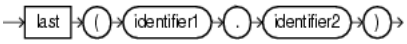
```
Timestamp   Tuple
 1000       33,0.9
 3000       44,0.4
 4000       30,0.3
 5000       10,10.0
 6000        7,0.9
 7000        7,2.3
 9000        7,8.7
11000       40,6.6
15000       19,8.8
17000       30,5.5
20000        5,10.0
23000       40,6.6
25000        3,5.5
30000       30,2.2
35000        2,10.0
40000        7,5.5
44000       40,8.9
```

Example 6–9 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 6-6
- "prev" on page 5-9
- *pattern_recognition_clause::=* on page 15-1

Examples

Consider the query `q9` in Example 6–10 and the data stream `S0` in Example 6–11. 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 6–12.

Example 6–10 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 6–11 last Function Stream Input

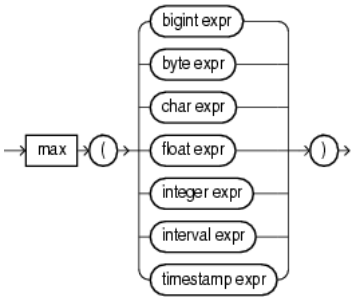
```
Timestamp   Tuple
 1000       33,0.9
 3000       44,0.4
 4000       30,0.3
 5000       10,10.0
 6000        7,0.9
 7000        7,2.3
 9000        7,8.7
11000       40,6.6
15000       19,8.8
17000       30,5.5
20000        5,10.0
23000       40,6.6
25000        3,5.5
30000       30,2.2
35000        2,10.0
40000        7,5.5
44000       40,8.9
```

Example 6–12 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
```

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 6–13 and the data stream `S15` in Example 6–14. Stream `S15` has schema `(c1 int, c2 timestamp)`. The query returns the relation in Example 6–15.

Example 6–13 max Function Query

```
<query id="test_max_timestamp"><![CDATA[
    select max(c2) from S15[range 2]
]]></query>
```

Example 6–14 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
```

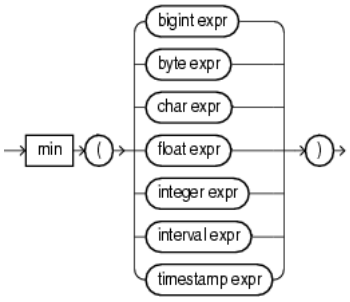
Example 6–15 max Function Relation Output

```
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 6–16 and the data stream `S15` in Example 6–17. Stream `S15` has schema `(c1 int, c2 timestamp)`. The query returns the relation in Example 6–18.

Example 6–16 min Function Query

```
<query id="test_min_timestamp"><![CDATA[
    select min(c2) from S15[range 2]
]]></query>
```

Example 6–17 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
```

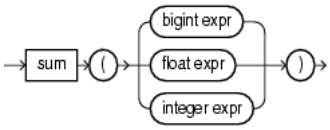
Example 6–18 min Function Relation Output

```
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 6–19 and the data stream `S1` in Example 6–20. Stream `S1` has schema `(c1 integer, c2 bigint)`. The query returns the relation in Example 6–21. For more information on `range`, see "Range-Based Stream-to-Relation Window Operators" on page 4-5.

Example 6–19 sum Query

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

Example 6–20 sum Stream Input

```
Timestamp   Tuple
1000         5,
1000        10,5
2000          ,4
3000        30,6
5000        45,44
7000        55,3
h 200000000
```

Example 6–21 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 14.2.6, "Sorting Query Results".

Examples

This section describes the following `xmlagg` examples:

- "xmlagg Function and the xmlelement Function" on page 6-15
- "xmlagg Function and the ORDER BY Clause" on page 6-16

xmlagg Function and the xmlelement Function

Consider the query `tkdata67_q1` in Example 6–22 and the input relation in Example 6–23. 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 6–24.

For more information about `xmlelement`, see "xmlelement_expr" on page 11-24.

Example 6–22 xmlagg Query

```
<query id="tkdata67_q1"><![CDATA[
    select
        c1,
        xmlagg(xmlelement("c2",c2))
    from
        tkdata67_S0[rows 10]
    group by c1
]]></query>
```

Example 6–23 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 6–24 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 6–25 and the input relation in Example 6–26. 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 6–27.

Example 6–25 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 6–26 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 6–27 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>
```

7

Functions: Colt Single-Row

Oracle CQL provides a variety of built-in single-row functions based on the Colt open source libraries for high performance scientific and technical computing.

For more information, see Section 1.1.9, "Functions".

7.1 Introduction to Oracle CQLBuilt-In Single-Row Colt Functions

Table 7–1 lists the built-in single-row Colt functions that Oracle CQL provides.

Table 7–1 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`
`cern.jet.math.Bessel` A set of Bessel functions.	▪ `i0` ▪ `i0e` ▪ `i1` ▪ `i1e` ▪ `j0` ▪ `j1` ▪ `jn` ▪ `k0` ▪ `k0e` ▪ `k1` ▪ `k1e` ▪ `kn` ▪ `y0` ▪ `y1` ▪ `yn`

Table 7–1 (Cont.) Oracle CQL Built-in Single-Row Colt-Based Functions

Colt Package	Function
`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`
`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 CEP that no further input will have a timestamp lesser than the heartbeat value.

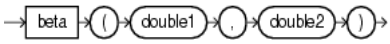
For more information, see:

- Section 1.1.9, "Functions"

- Section 2.2, "Datatypes"
- http://dsd.lbl.gov/~hoschek/colt/

beta

Syntax



Purpose

`beta` is based on `cern.jet.stat.Gamma`. It returns the beta function (see Figure 7–1) of the input arguments as a `double`.

Figure 7–1 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Gamma.html#beta(double,%20double)`.

Examples

Consider the query `qColt28` in Example 7–1. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–2, the query returns the relation in Example 7–3.

Example 7–1 beta Function Query

```
<query id="qColt28"><![CDATA[
    select beta(c2,c2) from SColtFunc
]]></query>
```

Example 7–2 beta Function Stream Input

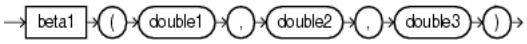
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–3 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 7–2) as a `double`.

Figure 7–2 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Probability.html#beta(double,%20double,%20double)`.

Examples

Consider the query `qColt35` in Example 7–4. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–5, the query returns the relation in Example 7–6.

Example 7–4 beta1 Function Query

```
<query id="qColt35"><![CDATA[
    select beta1(c2,c2,c2) from SColtFunc
]]></query>
```

Example 7–5 beta1 Function Stream Input

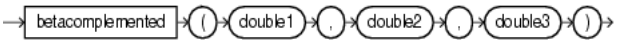
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–6 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 7–2) 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 7-32
- `http://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Probability.html#betaComplemented(double,%20double,%20double)`

Examples

Consider the query `qColt37` in Example 7–7. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–8, the query returns the relation in Example 7–9.

Example 7–7 betacomplemented Function Query

```
<query id="qColt37"><![CDATA[
    select betacomplemented(c2,c2,c2) from SColtFunc
]]></query>
```

Example 7–8 betacomplemented Function Stream Input

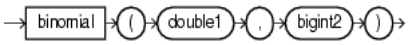
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–9 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 7–3) as a `double`.

Figure 7–3 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 7–2 lists the `binomial` function return values for various values of `k`.

Table 7–2 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 7–3.

For more information, see `http://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Arithmetic.html#binomial(double,%20long)`.

Examples

Consider the query `qColt6` in Example 7–10. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 long)` in Example 7–11, the query returns the relation in Example 7–12.

Example 7–10 binomial Function Query

```
<query id="qColt6"><![CDATA[
    select binomial(c2,c3) from SColtFunc
]]></query>
```

Example 7–11 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 7–12 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 7–3) as a `double`.

This function takes the following arguments:

- `long1`: the `n` value.
- `long2`: the `k` value.

Table 7–3 lists the `BINOMIAL` function return values for various values of `k`.

Table 7–3 cern.jet.math.Arithmetic Binomial1 Return Values

Arguments	Return Value
`k < 0`	0
`k = 0 | | k = n`	1
`k = 1 | | k = n-1`	`n`
Any other value of `k`	Computed binomial coefficient as given in Figure 7–3.

For more information, see
`http://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Arithmetic.html#binomial(long,%20long)`.

Examples

Consider the query `qColt7` in Example 7–13. Given the data stream `SColtFunc` with schema `(c1 integer, c2 float, c3 long)` in Example 7–14, the query returns the relation in Example 7–15.

Example 7–13 binomial1 Function Query

```
<query id="qColt7"><![CDATA[
    select binomial1(c3,c3) from SColtFunc
]]></query>
```

Example 7–14 binomial1 Function Stream Input

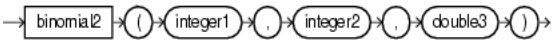
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–15 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 7–4) as a `double`.

Figure 7–4 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Probability.html#binomial(int,%20int,%20double)`.

Examples

Consider the query `qColt34` in Example 7–16. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–17, the query returns the relation in Example 7–18.

Example 7–16 binomial2 Function Query

```
<query id="qColt34"><![CDATA[
    select binomial2(c1,c1,c2) from SColtFunc
]]></query>
```

Example 7–17 binomial2 Function Stream Input

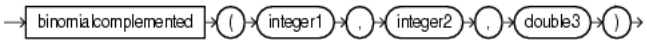
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–18 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 7–5) as a `double`.

Figure 7–5 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Probability.html#binomialComplemented(int,%20int,%20double)`.

Examples

Consider the query `qColt38` in Example 7–19. Given the data stream `SColtFunc` with schema `(integer, c2 double, c3 bigint)` in Example 7–20, the query returns the relation in Example 7–21.

Example 7–19 binomialcomplemented Function Query

```
<query id="qColt38"><![CDATA[
    select binomialcomplemented(c1,c1,c2) from SColtFunc
]]></query>
```

Example 7–20 binomialcomplemented Function Stream Input

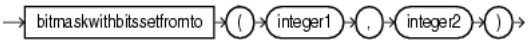
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–21 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://dsd.lbl.gov/~hoschek/colt/api/cern/colt/bitvector/QuickBitVector.html#bitMaskWithBitsSetFromTo(int,%20int)`
- "leastsignificantbit" on page 7-43
- "mostsignificantbit" on page 7-50

Examples

Consider the query `qColt53` in Example 7–22. Given the data stream `SColtFunc` with schema `(c1 integer, c2 float, c3 bigint)` in Example 7–23, the query returns the relation in Example 7–24.

Example 7–22 bitmaskwithbitssetfromto Function Query

```
<query id="qColt53"><![CDATA[
    select bitmaskwithbitssetfromto(c1,c1) from SColtFunc
]]></query>
```

Example 7–23 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 7–24 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Arithmetic.html#ceil(double)`
- "ceil1" on page 9-12

Examples

Consider the query `qColt1` in Example 7–25. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–26, the query returns the relation in Example 7–27.

Example 7–25 ceil Function Query

```
<query id="qColt1"><![CDATA[
    select ceil(c2) from SColtFunc
]]></query>
```

Example 7–26 ceil Function Stream Input

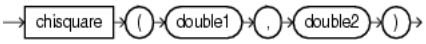
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–27 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 7–6) as a `double`.

Figure 7–6 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Probability.html#chiSquare(double,%20double).

Examples

Consider the query `qColt39` in Example 7–28. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–29, the query returns the relation in Example 7–30.

Example 7–28 chisquare Function Query

```
<query id="qColt39"><![CDATA[
    select chisquare(c2,c2) from SColtFunc
]]></query>
```

Example 7–29 chisquare Function Stream Input

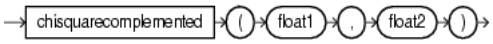
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–30 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 7–6) 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Probability.html#chiSquareComplemented(double,%20double)`.

Examples

Consider the query `qColt40` in Example 7–31. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–32, the query returns the relation in Example 7–33.

Example 7–31 chisquarecomplemented Function Query

```
<query id="qColt40"><![CDATA[
    select chisquarecomplemented(c2,c2) from SColtFunc
]]></query>
```

Example 7–32 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 7–33 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 7–7 shows.

Figure 7–7 cern.jet.stat.Probability errorfunction

$$f(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2)\,dt$$

For more information, see `http://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Probability.html#errorFunction(double)`.

Examples

Consider the query `qColt41` in Example 7–34. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–35, the query returns the relation in Example 7–36.

Example 7–34 errorfunction Function Query

```
<query id="qColt41"><![CDATA[
    select errorfunction(c2) from SColtFunc
]]></query>
```

Example 7–35 errorfunction Function Stream Input

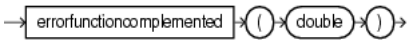
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–36 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 7–8 shows.

Figure 7–8 cern.jet.stat.Probability errorfunctioncompelemented

$$f(x) = \frac{2}{\sqrt{\pi}} \int_{x}^{\infty} \exp(-t^2)\,dt$$

For more information, see http://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Probability.html#errorFunctionComplemented(double).

Examples

Consider the query `qColt42` in Example 7–37. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–38, the query returns the relation in Example 7–39.

Example 7–37 errorfunctioncomplemented Function Query

```
<query id="qColt42"><![CDATA[
    select errorfunctioncomplemented(c2) from SColtFunc
]]></query>
```

Example 7–38 errorfunctioncomplemented Function Stream Input

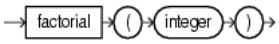
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–39 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Arithmetic.html#factorial(int)`.

Examples

Consider the query `qColt8` in Example 7–40. Given the data stream `SColtFunc` with schema `(c1 integer, c2 float, c3 bigint)` in Example 7–41, the query returns the relation in Example 7–42.

Example 7–40 factorial Function Query

```
<query id="qColt8"><![CDATA[
    select factorial(c1) from SColtFunc
]]></query>
```

Example 7–41 factorial Function Stream Input

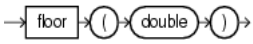
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–42 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Arithmetic.html#floor(double)`
- "floor1" on page 9-17

Examples

Consider the query `qColt2` in Example 7–43. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–44, the query returns the relation in Example 7–45.

Example 7–43 floor Function Query

```
<query id="qColt2"><![CDATA[
    select floor(c2) from SColtFunc
]]></query>
```

Example 7–44 floor Function Stream Input

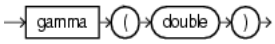
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–45 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Gamma.html#gamma(double)`.

Examples

Consider the query `qColt29` in Example 7–46. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–47, the query returns the relation in Example 7–48.

Example 7–46 gamma Function Query

```
<query id="qColt29"><![CDATA[
    select gamma(c2) from SColtFunc
]]></query>
```

Example 7–47 gamma Function Stream Input

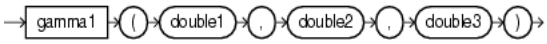
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–48 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 7–9) as a `double`.

Figure 7–9 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Probability.html#gamma(double,%20double,%20double)`.

Examples

Consider the query `qColt36` in Example 7–49. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–50, the query returns the relation in Example 7–51.

Example 7–49 gamma1 Function Query

```
<query id="qColt36"><![CDATA[
    select gamma1(c2,c2,c2) from SColtFunc
]]></query>
```

Example 7–50 gamma1 Function Stream Input

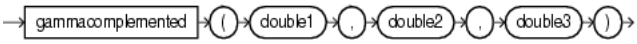
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–51 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 7–10) as a `double`.

Figure 7–10 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Probability.html#gammaComplemented(double,%20double,%20double)`.

Examples

Consider the query `qColt43` in Example 7–52. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–53, the query returns the relation in Example 7–54.

Example 7–52 gammacomplemented Function Query

```
<query id="qColt43"><![CDATA[
    select gammacomplemented(c2,c2,c2) from SColtFunc
]]></query>
```

Example 7–53 gammacomplemented Function Stream Input

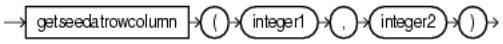
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–54 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/random/engine/RandomSeedTable.html#getSeedAtRowColumn(int,%20int)`.

Examples

Consider the query `qColt27` in Example 7–55. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–56, the query returns the relation in Example 7–57.

Example 7–55 getseedatrowcolumn Function Query

```
<query id="qColt27"><![CDATA[
    select getseedatrowcolumn(c1,c1) from SColtFunc
]]></query>
```

Example 7–56 getseedatrowcolumn Function Stream Input

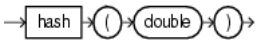
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–57 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://dsd.lbl.gov/~hoschek/colt/api/cern/colt/map/HashFunctions.html#hash(double)`.

Examples

Consider the query `qColt56` in Example 7–58. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–59, the query returns the relation in Example 7–60.

Example 7–58 hash Function Query

```
<query id="qColt56"><![CDATA[
    select hash(c2) from SColtFunc
]]></query>
```

Example 7–59 hash Function Stream Input

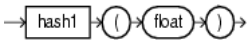
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–60 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://dsd.lbl.gov/~hoschek/colt/api/cern/colt/map/HashFunctions.html#hash(float)`.

Examples

Consider the query `qColt57` in Example 7–61. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–62, the query returns the relation in Example 7–63.

Example 7–61 hash1 Function Query

```
<query id="qColt57"><![CDATA[
    select hash1(c2) from SColtFunc
]]></query>
```

Example 7–62 hash1 Function Stream Input

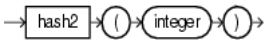
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–63 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://dsd.lbl.gov/~hoschek/colt/api/cern/colt/map/HashFunctions.html#hash(int)`.

Examples

Consider the query `qColt58` in Example 7–64. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–65, the query returns the relation in Example 7–66.

Example 7–64 hash2 Function Query

```
<query id="qColt58"><![CDATA[
    select hash2(c1) from SColtFunc
]]></query>
```

Example 7–65 hash2 Function Stream Input

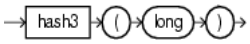
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–66 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://dsd.lbl.gov/~hoschek/colt/api/cern/colt/map/HashFunctions.html#hash(long)`.

Examples

Consider the query `qColt59` in Example 7–67. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–68, the query returns the relation in Example 7–69.

Example 7–67 hash3 Function Query

```
<query id="qColt59"><![CDATA[
    select hash3(c3) from SColtFunc
]]></query>
```

Example 7–68 hash3 Function Stream Input

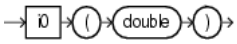
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–69 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Bessel.html#i0(double)`
- "j0" on page 7-35

Examples

Consider the query `qColt12` in Example 7–70. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–71, the query returns the relation in Example 7–72.

Example 7–70 i0 Function Query

```
<query id="qColt12"><![CDATA[
    select i0(c2) from SColtFunc
]]></query>
```

Example 7–71 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 7–72 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Bessel.html#i0e(double)`
- "j0" on page 7-35

Examples

Consider the query `qColt13` in Example 7–73. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–74, the query returns the relation in Example 7–75.

Example 7–73 i0e Function Query

```
<query id="qColt13"><![CDATA[
    select i0e(c2) from SColtFunc
]]></query>
```

Example 7–74 i0e Function Stream Input

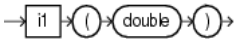
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–75 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Bessel.html#i1(double)`
- "j1" on page 7-36

Examples

Consider the query `qColt14` in Example 7–76. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–77, the query returns the relation in Example 7–78.

Example 7–76 i1 Function Query

```
<query id="qColt14"><![CDATA[
    select i1(c2) from SColtFunc
]]></query>
```

Example 7–77 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 7–78 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Bessel.html#i1e(double)`
- "j1" on page 7-36

Examples

Consider the query `qColt15` in Example 7–79. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–80, the query returns the relation in Example 7–81.

Example 7–79 i1e Function Query

```
<query id="qColt15"><![CDATA[
    select i1e(c2) from SColtFunc
]]></query>
```

Example 7–80 i1e Function Stream Input

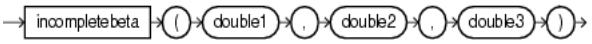
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–81 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Gamma.html#incompleteBeta(double,%20double,%20double).

Examples

Consider the query `qColt30` in Example 7–82. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–83, the query returns the relation in Example 7–84.

Example 7–82 incompletebeta Function Query

```
<query id="qColt30"><![CDATA[
    select incompletebeta(c2,c2,c2) from SColtFunc
]]></query>
```

Example 7–83 incompletebeta Function Stream Input

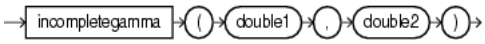
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–84 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Gamma.html#incompleteGamma(double,%20double).

Examples

Consider the query `qColt31` in Example 7–85. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–86, the query returns the relation in Example 7–87.

Example 7–85 incompletegamma Function Query

```
<query id="qColt31"><![CDATA[
    select incompletegamma(c2,c2) from SColtFunc
]]></query>
```

Example 7–86 incompletegamma Function Stream Input

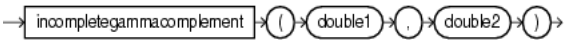
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–87 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Gamma.html#incompleteGammaComplement(double,%20double).

Examples

Consider the query `qColt32` in Example 7–88. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–89, the query returns the relation in Example 7–90.

Example 7–88 incompletegammacomplement Function Query

```
<query id="qColt32"><![CDATA[
    select incompletegammacomplement(c2,c2) from SColtFunc
]]></query>
```

Example 7–89 incompletegammacomplement Function Stream Input

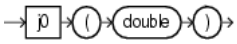
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–90 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Bessel.html#j0(double).

Examples

Consider the query `qColt16` in Example 7–91. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–92, the query returns the relation in Example 7–93.

Example 7–91 j0 Function Query

```
<query id="qColt16"><![CDATA[
    select j0(c2) from SColtFunc
]]></query>
```

Example 7–92 j0 Function Stream Input

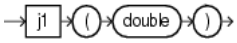
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–93 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Bessel.html#j1(double)`.

Examples

Consider the query `qColt17` in Example 7–94. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–95, the query returns the relation in Example 7–96.

Example 7–94 j1 Function Query

```
<query id="qColt17"><![CDATA[
    select j1(c2) from SColtFunc
]]></query>
```

Example 7–95 j1 Function Stream Input

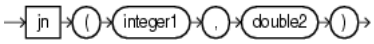
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–96 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Bessel.html#jn(int,%20double).

Examples

Consider the query `qColt18` in Example 7–97. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–98, the query returns the relation in Example 7–99.

Example 7–97 jn Function Query

```
<query id="qColt18"><![CDATA[
    select jn(c1,c2) from SColtFunc
]]></query>
```

Example 7–98 jn Function Stream Input

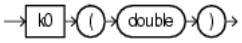
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–99 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Bessel.html#k0(double).

Examples

Consider the query `qColt19` in Example 7–100. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–101, the query returns the relation in Example 7–102.

Example 7–100 k0 Function Query

```
<query id="qColt19"><![CDATA[
    select k0(c2) from SColtFunc
]]></query>
```

Example 7–101 k0 Function Stream Input

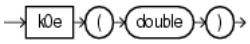
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–102 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Bessel.html#k0e(double).

Examples

Consider the query `qColt20` in Example 7–103. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–104, the query returns the relation in Example 7–105.

Example 7–103 k0e Function Query

```
<query id="qColt20"><![CDATA[
    select k0e(c2) from SColtFunc
]]></query>
```

Example 7–104 k0e Function Stream Input

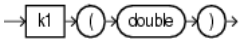
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–105 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Bessel.html#k1(double).

Examples

Consider the query `qColt21` in Example 7–106. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–107, the query returns the relation in Example 7–108.

Example 7–106 k1 Function Query

```
<query id="qColt21"><![CDATA[
    select k1(c2) from SColtFunc
]]></query>
```

Example 7–107 k1 Function Stream Input

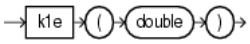
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–108 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Bessel.html#k1e(double)`
- "k1" on page 7-40

Examples

Consider the query `qColt22` in Example 7–109. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–110, the query returns the relation in Example 7–111.

Example 7–109 k1e Function Query

```
<query id="qColt22"><![CDATA[
    select k1e(c2) from SColtFunc
]]></query>
```

Example 7–110 k1e Function Stream Input

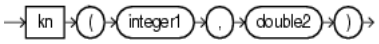
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–111 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Bessel.html#kn(int,%20double).

Examples

Consider the query `qColt23` in Example 7–112. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–113, the query returns the relation in Example 7–114.

Example 7–112 kn Function Query

```
<query id="qColt23"><![CDATA[
    select kn(c1,c2) from SColtFunc
]]></query>
```

Example 7–113 kn Function Stream Input

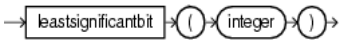
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–114 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://dsd.lbl.gov/~hoschek/colt/api/cern/colt/bitvector/QuickBitVector.html#leastSignificantBit(int)`
- "bitmaskwithbitssetfromto" on page 7-12
- "mostsignificantbit" on page 7-50

Examples

Consider the query `qColt54` in Example 7–115. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–116, the query returns the relation in Example 7–117.

Example 7–115 leastsignificantbit Function Query

```
<query id="qColt54"><![CDATA[
    select leastsignificantbit(c1) from SColtFunc
]]></query>
```

Example 7–116 leastsignificantbit Function Stream Input

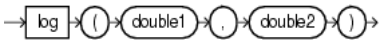
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–117 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 7–11 shows as a `double`.

Figure 7–11 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Arithmetic.html#log(double,%20double)`.

Examples

Consider the query `qColt3` in Example 7–118. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–119, the query returns the relation in Example 7–120.

Example 7–118 log Function Query

```
<query id="qColt3"><![CDATA[
    select log(c2,c2) from SColtFunc
]]></query>
```

Example 7–119 log Function Stream Input

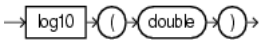
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–120 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Arithmetic.html#log10(double).

Examples

Consider the query `qColt4` in Example 7–121. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–122, the query returns the relation in Example 7–123.

Example 7–121 log10 Function Query

```
<query id="qColt4"><![CDATA[
    select log10(c2) from SColtFunc
]]></query>
```

Example 7–122 log10 Function Stream Input

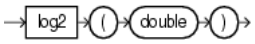
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–123 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Arithmetic.html#log2(double).

Examples

Consider the query `qColt9` in Example 7–124. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–125, the query returns the relation in Example 7–126.

Example 7–124 log2 Function Query

```
<query id="qColt9"><![CDATA[
    select log2(c2) from SColtFunc
]]></query>
```

Example 7–125 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 7–126 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Arithmetic.html#logFactorial(int).

Examples

Consider the query `qColt10` in Example 7–127. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–128, the query returns the relation in Example 7–129.

Example 7–127 logfactorial Function Query

```
<query id="qColt10"><![CDATA[
    select logfactorial(c1) from SColtFunc
]]></query>
```

Example 7–128 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 7–129 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Gamma.html#logGamma(double).

Examples

Consider the query `qColt33` in Example 7–130. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–131, the query returns the relation in Example 7–132.

Example 7–130 loggamma Function Query

```
<query id="qColt33"><![CDATA[
    select loggamma(c2) from SColtFunc
]]></query>
```

Example 7–131 loggamma Function Stream Input

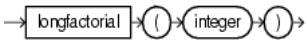
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–132 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Arithmetic.html#longFactorial(int).

Examples

Consider the query `qColt11` in Example 7–133. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–134, the query returns the relation in Example 7–135.

Example 7–133 longfactorial Function Query

```
<query id="qColt11"><![CDATA[
    select longfactorial(c1) from SColtFunc
]]></query>
```

Example 7–134 longfactorial Function Stream Input

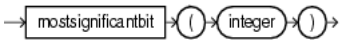
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–135 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://dsd.lbl.gov/~hoschek/colt/api/cern/colt/bitvector/QuickBitVector.html#mostSignificantBit(int)`
- "bitmaskwithbitssetfromto" on page 7-12
- "leastsignificantbit" on page 7-43

Examples

Consider the query `qColt55` in Example 7–136. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–137, the query returns the relation in Example 7–138.

Example 7–136 mostsignificantbit Function Query

```
<query id="qColt55"><![CDATA[
    select mostsignificantbit(c1) from SColtFunc
]]></view>
```

Example 7–137 mostsignificantbit Function Stream Input

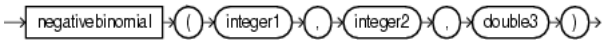
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–138 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 7–12) as a `double`.

Figure 7–12 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Probability.html#negativeBinomial(int,%20int,%20double)`.

Examples

Consider the query `qColt44` in Example 7–139. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–140, the query returns the relation in Example 7–141.

Example 7–139 negativebinomial Function Query

```
<query id="qColt44"><![CDATA[
    select negativebinomial(c1,c1,c2) from SColtFunc
]]></query>
```

Example 7–140 negativebinomial Function Stream Input

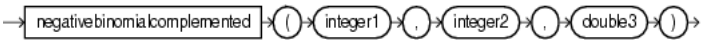
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–141 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 7–13) as a `double`.

Figure 7–13 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Probability.html#negativeBinomialComplemented(int,%20int,%20double).

Examples

Consider the query `qColt45` in Example 7–142. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–143, the query returns the relation in Example 7–144.

Example 7–142 negativebinomialcomplemented Function Query

```
<query id="qColt45"><![CDATA[
    select negativebinomialcomplemented(c1,c1,c2) from SColtFunc
]]></query>
```

Example 7–143 negativebinomialcomplemented Function Stream Input

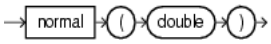
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–144 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 7–14) as a `double`.

Figure 7–14 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Probability.html#normal(double)`.

Examples

Consider the query `qColt46` in Example 7–145. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–146, the query returns the relation in Example 7–147.

Example 7–145 normal Function Query

```
<query id="qColt46"><![CDATA[
    select normal(c2) from SColtFunc
]]></query>
```

Example 7–146 normal Function Stream Input

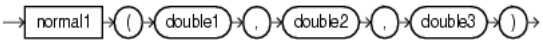
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–147 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 7–15) as a `double`.

Figure 7–15 cern.jet.stat.Probability normal1

$$f(x) = \frac{1}{\sqrt{(2\pi * v)}} \int_{-\infty}^{x} \exp(-\frac{(t-mean)^2}{2v})dt$$

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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Probability.html#normal(double,%20double,%20double)`.

Examples

Consider the query `qColt47` in Example 7–148. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–149, the query returns the relation in Example 7–150.

Example 7–148 normal1 Function Query

```
<query id="qColt47"><![CDATA[
    select normal1(c2,c2,c2) from SColtFunc
]]></query>
```

Example 7–149 normal1 Function Stream Input

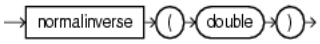
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–150 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Probability.html#normalInverse(double)`.

Examples

Consider the query `qColt48` in Example 7–151. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–152, the query returns the relation in Example 7–153.

Example 7–151 normalinverse Function Query

```
<query id="qColt48"><![CDATA[
    select normalinverse(c2) from SColtFunc
]]></view>
```

Example 7–152 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 7–153 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 7–16) as a `double`.

Figure 7–16 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Probability.html#poisson(int,%20double)`.

Examples

Consider the query `qColt49` in Example 7–154. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–155, the query returns the relation in Example 7–156.

Example 7–154 poisson Function Query

```
<query id="qColt49"><![CDATA[
    select poisson(c1,c2) from SColtFunc
]]></query>
```

Example 7–155 poisson Function Stream Input

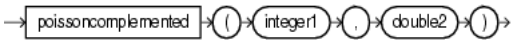
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–156 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 7–17) as a `double`.

Figure 7–17 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Probability.html#poissonComplemented(int,%20double)`.

Examples

Consider the query `qColt50` in Example 7–157. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–158, the query returns the relation in Example 7–159.

Example 7–157 poissoncomplemented Function Query

```
<query id="qColt50"><![CDATA[
    select poissoncomplemented(c1,c2) from SColtFunc
]]></query>
```

Example 7–158 poissoncomplemented Function Stream Input

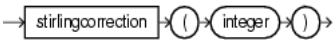
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–159 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 7–18) as a `double`.

Figure 7–18 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Arithmetic.html#stirlingCorrection(int).

Examples

Consider the query `qColt5` in Example 7–160. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–161, the query returns the relation in Example 7–162.

Example 7–160 stirlingcorrection Function Query

```
<query id="qColt5"><![CDATA[
    select stirlingcorrection(c1) from SColtFunc
]]></query>
```

Example 7–161 stirlingcorrection Function Stream Input

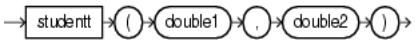
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–162 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 7–19) as a `double`.

Figure 7–19 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Probability.html#studentT(double,%20double)`.

Examples

Consider the query `qColt51` in Example 7–163. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–164, the query returns the relation in Example 7–165.

Example 7–163 studentt Function Query

```
<query id="qColt51"><![CDATA[
    select studentt(c2,c2) from SColtFunc
]]></query>
```

Example 7–164 studentt Function Stream Input

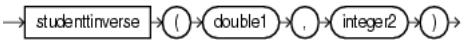
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–165 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/stat/Probability.html#studentTInverse(double,%20int)`
- "studentt" on page 7-59

Examples

Consider the query `qColt52` in Example 7–166. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–167, the query returns the relation in Example 7–168.

Example 7–166 studenttinverse Function Query

```
<query id="qColt52"><![CDATA[
    select studenttinverse(c2,c1) from SColtFunc
]]></query>
```

Example 7–167 studenttinverse Function Stream Input

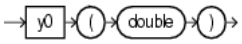
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–168 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Bessel.html#y0(double).

Examples

Consider the query `qColt24` in Example 7–169. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–170, the query returns the relation in Example 7–171.

Example 7–169 y0 Function Query

```
<query id="qColt24"><![CDATA[
    select y0(c2) from SColtFunc
]]></query>
```

Example 7–170 y0 Function Stream Input

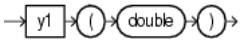
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–171 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Bessel.html#y1(double)`.

Examples

Consider the query `qColt25` in Example 7–172. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–173, the query returns the relation in Example 7–174.

Example 7–172 y1 Function Query

```
<query id="qColt25"><![CDATA[
    select y1(c2) from SColtFunc
]]></query>
```

Example 7–173 y1 Function Stream Input

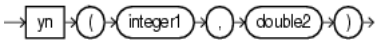
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 7–174 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://dsd.lbl.gov/~hoschek/colt/api/cern/jet/math/Bessel.html#yn(int,%20double).

Examples

Consider the query `qColt26` in Example 7–175. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 7–176, the query returns the relation in Example 7–177.

Example 7–175 yn Function Query

```
<query id="qColt26"><![CDATA[
    select yn(c1,c2) from SColtFunc
]]></query>
```

Example 7–176 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 7–177 yn Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +             -1.4714724
1000:       +           -132.63406
1200:       +             -8.020442
2000:       +             -6.3026547E8
```

8

Functions: Colt Aggregate

Oracle CQL provides a set of built-in aggregate functions based on the Colt open source libraries for high performance scientific and technical computing.

For more information, see Section 1.1.9, "Functions".

8.1 Introduction to Oracle CQL Built-In Aggregate Colt Functions

Table 8–1 lists the built-in aggregate Colt functions that Oracle CQL provides.

Table 8–1 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 CEP 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"
- Section 1.1.9, "Functions"
- Section 2.2, "Datatypes"
- http://dsd.lbl.gov/~hoschek/colt/

8.1.1 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 CEP, `data` will never come in the form of a `Collection`. The Oracle CQL function receives input data in a stream of tuples.

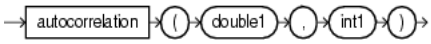
So suppose our stream is defined as `S:(double val, integer lag)`. On each input tuple, the Oracle CQL `autocorrelation` function will compute two intermediate aggregates, `mean` and `variance`, and one final aggregate, `autocorrelation`.

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)
```

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 8-15

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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr1` in Example 8–1. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–2, the query returns the relation in Example 8–3.

Example 8–1 autocorrelation Function Query

```
<query id="qColtAggr1"><![CDATA[
     select autocorrelation(c3, c1) from SColtAggrFunc
]]></query>
```

Example 8–2 autocorrelation Function Stream Input

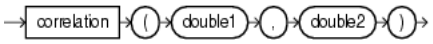
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 8–3 autocorrelation Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
```

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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr2` in Example 8–4. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–5, the query returns the relation in Example 8–6.

Example 8–4 correlation Function Query

```
<query id="qColtAggr2"><![CDATA[
     select correlation(c3, c3) from SColtAggrFunc
]]></query>
```

Example 8–5 correlation Function Stream Input

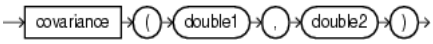
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 8–6 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:       +           1.333333333333333
```

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 8–1) of the input arguments as a `double`.

Figure 8–1 cern.jet.stat.Descriptive.covariance

$$\mathrm{cov}(x, y) = \left(\frac{1}{size() - 1}\right) * Sum(x[i] - mean(x)) * (y[i] - mean(y))$$

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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr3` in Example 8–7. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–8, the query returns the relation in Example 8–9.

Example 8–7 covariance Function Query

```
<query id="qColtAggr3"><![CDATA[
     select covariance(c3, c3) from SColtAggrFunc
]]></query>
```

Example 8–8 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
```

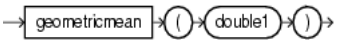
Example 8–9 covariance Function Relation Output

```
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 8–2) of the input argument as a `double`.

Figure 8–2 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr6` in Example 8–10. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–11, the query returns the relation in Example 8–12.

Example 8–10 geometricmean Function Query

```
<query id="qColtAggr6"><![CDATA[
    select geometricmean(c3) from SColtAggrFunc
]]></query>
```

Example 8–11 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
```

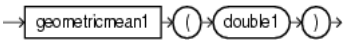
Example 8–12 geometricmean Function Relation Output

```
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 8–3) of the input arguments as a `double`.

Figure 8–3 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr7` in Example 8–13. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–14, the query returns the relation in Example 8–15.

Example 8–13 geometricmean1 Function Query

```
<query id="qColtAggr7"><![CDATA[
    select geometricmean1(c3) from SColtAggrFunc
]]></query>
```

Example 8–14 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
```

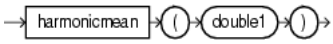
Example 8–15 geometricmean1 Function Relation Output

```
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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr8` in Example 8–16. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–17, the query returns the relation in Example 8–18.

Example 8–16 harmonicmean Function Query

```
<query id="qColtAggr8"><![CDATA[
    select harmonicmean(c3) from SColtAggrFunc
]]></query>
```

Example 8–17 harmonicmean Function Stream Input

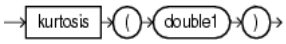
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 8–18 harmonicmean Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           0.0
1000:       -           0.0
1000:       +           0.0
1200:       -           0.0
1200:       +           0.0
2000:       -           0.0
2000:       +           0.0
```

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 8–4) of a data sequence as a `double`.

Figure 8–4 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr12` in Example 8–19. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–20, the query returns the relation in Example 8–21.

Example 8–19 kurtosis Function Query

```
<query id="qColtAggr12"><![CDATA[
    select kurtosis(c3) from SColtAggrFunc
]]></query>
```

Example 8–20 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
```

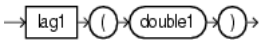
Example 8–21 kurtosis Function Relation Output

```
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 8-4.

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#lag1(cern.colt.list.DoubleArrayList,%20double)`
- Section 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr14` in Example 8–22. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–23, the query returns the relation in Example 8–24.

Example 8–22 lag1 Function Query

```
<query id="qColtAggr14"><![CDATA[
    select lag1(c3) from SColtAggrFunc
]]></query>
```

Example 8–23 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
```

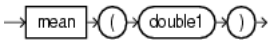
Example 8–24 lag1 Function Relation Output

```
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 8–5) as a `double`.

Figure 8–5 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr16` in Example 8–25. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–26, the query returns the relation in Example 8–27.

Example 8–25 mean Function Query

```
<query id="qColtAggr16"><![CDATA[
    select mean(c3) from SColtAggrFunc
]]></query>
```

Example 8–26 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
```

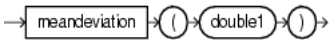
Example 8–27 mean Function Relation Output

```
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 8–6) as a `double`.

Figure 8–6 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr17` in Example 8–28. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–29, the query returns the relation in Example 8–30.

Example 8–28 meandeviation Function Query

```
<query id="qColtAggr17"><![CDATA[
    select meandeviation(c3) from SColtAggrFunc
]]></query>
```

Example 8–29 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
```

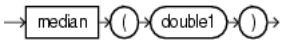
Example 8–30 meandeviation Function Relation Output

```
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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr18` in Example 8–31. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–32, the query returns the relation in Example 8–33.

Example 8–31 median Function Query

```
<query id="qColtAggr18"><![CDATA[
    select median(c3) from SColtAggrFunc
]]></query>
```

Example 8–32 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 8–33 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 8–7) as a `double`.

Figure 8–7 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr21` in Example 8–34. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–35, the query returns the relation in Example 8–36.

Example 8–34 moment Function Query

```
<query id="qColtAggr21"><![CDATA[
    select moment(c3, c1, c3) from SColtAggrFunc
]]></query>
```

Example 8–35 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
```

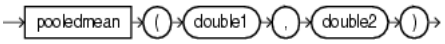
Example 8–36 moment Function Relation Output

```
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 8–8) as a `double`.

Figure 8–8 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr22` in Example 8–37. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–38, the query returns the relation in Example 8–39.

Example 8–37 pooledmean Function Query

```
<query id="qColtAggr22"><![CDATA[
    select pooledmean(c3, c3) from SColtAggrFunc
]]></query>
```

Example 8–38 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
```

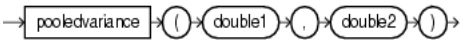
Example 8–39 pooledmean Function Relation Output

```
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 8–9) as a `double`.

Figure 8–9 cern.jet.stat.Descriptive.pooledVariance(int size1, double variance1, int size2, double variance2)

$$\frac{(size1 * variance1 + size2 * variance2)}{(size1 + size2)}$$

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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr23` in Example 8–40. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–41, the query returns the relation in Example 8–42.

Example 8–40 pooledvariance Function Query

```
<query id="qColtAggr23"><![CDATA[
    select pooledvariance(c3, c3) from SColtAggrFunc
]]></query>
```

Example 8–41 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
```

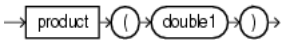
Example 8–42 pooledvariance Function Relation Output

```
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 8–10) as a `double`.

Figure 8–10 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr24` in Example 8–43. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–44, the query returns the relation in Example 8–45.

Example 8–43 product Function Query

```
<query id="qColtAggr24"><![CDATA[
    select product(c3) from SColtAggrFunc
]]></query>
```

Example 8–44 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
```

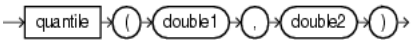
Example 8–45 product Function Relation Output

```
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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr26` in Example 8–46. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–47, the query returns the relation in Example 8–48.

Example 8–46 quantile Function Query

```
<query id="qColtAggr26"><![CDATA[
    select quantile(c3, c2) from SColtAggrFunc
]]></query>
```

Example 8–47 quantile Function Stream Input

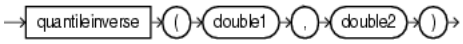
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 8–48 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr27` in Example 8–49. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–50, the query returns the relation in Example 8–51.

Example 8–49 quantileinverse Function Query

```
<query id="qColtAggr27"><![CDATA[
    select quantileinverse(c3, c3) from SColtAggrFunc
]]></query>
```

Example 8–50 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
```

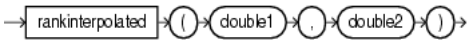
Example 8–51 quantileinverse Function Relation Output

```
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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr29` in Example 8–52. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–53, the query returns the relation in Example 8–54.

Example 8–52 rankinterpolated Function Query

```
<query id="qColtAggr29"><![CDATA[
    select rankinterpolated(c3, c3) from SColtAggrFunc
]]></query>
```

Example 8–53 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
```

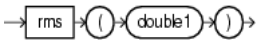
Example 8–54 rankinterpolated Function Relation Output

```
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 8–11) as a `double`.

Figure 8–11 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr30` in Example 8–55. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–56, the query returns the relation in Example 8–57.

Example 8–55 rms Function Query

```
<query id="qColtAggr30"><![CDATA[
    select rms(c3) from SColtAggrFunc
]]></query>
```

Example 8–56 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
```

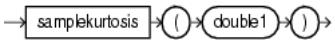
Example 8–57 rms Function Relation Output

```
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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr31` in Example 8–58. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–59, the query returns the relation in Example 8–60.

Example 8–58 samplekurtosis Function Query

```
<query id="qColtAggr31"><![CDATA[
     select samplekurtosis(c3) from SColtAggrFunc
]]></query>
```

Example 8–59 samplekurtosis Function Stream Input

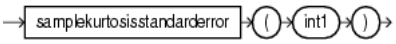
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 8–60 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr33` in Example 8–61. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–62, the query returns the relation in Example 8–63.

Example 8–61 samplekurtosisstandarderror Function Query

```
<query id="qColtAggr33"><![CDATA[
     select samplekurtosisstandarderror(c1) from SColtAggrFunc
]]></query>
```

Example 8–62 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 8–63 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr34` in Example 8–64. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–65, the query returns the relation in Example 8–66.

Example 8–64 sampleskew Function Query

```
<query id="qColtAggr34"><![CDATA[
    select sampleskew(c3) from SColtAggrFunc
]]></query>
```

Example 8–65 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 8–66 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr36` in Example 8–67. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–68, the query returns the relation in Example 8–69.

Example 8–67 sampleskewstandarderror Function Query

```
<query id="qColtAggr36"><![CDATA[
    select sampleskewstandarderror(c1) from SColtAggrFunc
]]></query>
```

Example 8–68 sampleskewstandarderror Function Stream Input

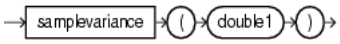
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 8–69 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 8–12) as a `double`.

Figure 8–12 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr38` in Example 8–70. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–71, the query returns the relation in Example 8–72.

Example 8–70 samplevariance Function Query

```
<query id="qColtAggr38"><![CDATA[
    select samplevariance(c3) from SColtAggrFunc
]]></query>
```

Example 8–71 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
```

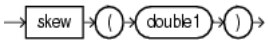
Example 8–72 samplevariance Function Relation Output

```
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 8–13) as a `double`.

Figure 8–13 cern.jet.stat.Descriptive.skew(DoubleArrayList data, double mean, double standardDeviation)

$$\frac{moment(data,3,mean)}{standardDeviation^3}$$

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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr41` in Example 8–73. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–74, the query returns the relation in Example 8–75.

Example 8–73 skew Function Query

```
<query id="qColtAggr41"><![CDATA[
    select skew(c3) from SColtAggrFunc
]]></query>
```

Example 8–74 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
```

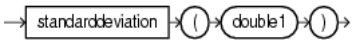
Example 8–75 skew Function Relation Output

```
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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr44` in Example 8–76. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–77, the query returns the relation in Example 8–78.

Example 8–76 standarddeviation Function Query

```
<query id="qColtAggr44"><![CDATA[
    select standarddeviation(c3) from SColtAggrFunc
]]></query>
```

Example 8–77 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 8–78 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 8–14) as a `double`.

Figure 8–14 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr45` in Example 8–79. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–80, the query returns the relation in Example 8–81.

Example 8–79 standarderror Function Query

```
<query id="qColtAggr45"><![CDATA[
     select standarderror(c3) from SColtAggrFunc
]]></query>
```

Example 8–80 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
```

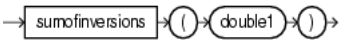
Example 8–81 standarderror Function Relation Output

```
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 8–15) as a `double`.

Figure 8–15 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr48` in Example 8–82. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–83, the query returns the relation in Example 8–84.

Example 8–82 sumofinversions Function Query

```
<query id="qColtAggr48"><![CDATA[
     select sumofinversions(c3) from SColtAggrFunc
]]></query>
```

Example 8–83 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 8–84 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 8–16) as a `double`.

Figure 8–16 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr49` in Example 8–85. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–86, the query returns the relation in Example 8–87.

Example 8–85 sumoflogarithms Function Query

```
<query id="qColtAggr49"><![CDATA[
    select sumoflogarithms(c3) from SColtAggrFunc
]]></query>
```

Example 8–86 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
```

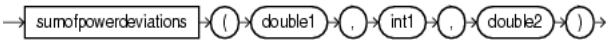
Example 8–87 sumoflogarithms Function Relation Output

```
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 8–17) as a `double`.

Figure 8–17 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr50` in Example 8–88. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–89, the query returns the relation in Example 8–90.

Example 8–88 sumofpowerdeviations Function Query

```
<query id="qColtAggr50"><![CDATA[
    select sumofpowerdeviations(c3, c1, c3) from SColtAggrFunc
]]></query>
```

Example 8–89 sumofpowerdeviations Function Stream Input

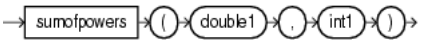
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 8–90 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 8–18) as a `double`.

Figure 8–18 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr52` in Example 8–91. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–92, the query returns the relation in Example 8–93.

Example 8–91 sumofpowers Function Query

```
<query id="qColtAggr52"><![CDATA[
    select sumofpowers(c3, c1) from SColtAggrFunc
]]></query>
```

Example 8–92 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
```

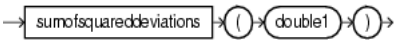
Example 8–93 sumofpowers Function Relation Output

```
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 8–19) as a `double`.

Figure 8–19 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr53` in Example 8–94. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–95, the query returns the relation in Example 8–96.

Example 8–94 sumofsquareddeviations Function Query

```
<query id="qColtAggr53"><![CDATA[
    select sumofsquareddeviations(c3) from SColtAggrFunc
]]></query>
```

Example 8–95 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 8–96 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 8–20) as a `double`.

Figure 8–20 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr54` in Example 8–97. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–98, the query returns the relation in Example 8–99.

Example 8–97 sumofsquares Function Query

```
<query id="qColtAggr54"><![CDATA[
    select sumofsquares(c3) from SColtAggrFunc
]]></query>
```

Example 8–98 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
```

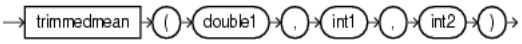
Example 8–99 sumofsquares Function Relation Output

```
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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr55` in Example 8–100. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–101, the query returns the relation in Example 8–102.

Example 8–100 trimmedmean Function Query

```
<query id="qColtAggr55"><![CDATA[
    select trimmedmean(c3, c1, c1) from SColtAggrFunc
]]></query>
```

Example 8–101 trimmedmean Function Stream Input

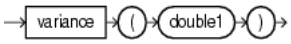
```
Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        4, 0.7, 30.0, 6
1200        3, 0.89, 20.0, 12
2000        8, 0.4, 10.0, 4
h 8000
h 200000000
```

Example 8–102 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 8–21) as a `double`.

Figure 8–21 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr57` in Example 8–103. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–104, the query returns the relation in Example 8–105.

Example 8–103 variance Function Query

```
<query id="qColtAggr57"><![CDATA[
    select variance(c3) from SColtAggrFunc
]]></query>
```

Example 8–104 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
```

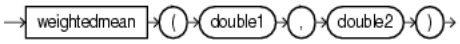
Example 8–105 variance Function Relation Output

```
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 8–22) as a `double`.

Figure 8–22 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr58` in Example 8–106. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–107, the query returns the relation in Example 8–108.

Example 8–106 weightedmean Function Query

```
<query id="qColtAggr58"><![CDATA[
    select weightedmean(c3, c3) from SColtAggrFunc
]]></query>
```

Example 8–107 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 8–108 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 8.1.1, "Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments"

Examples

Consider the query `qColtAggr60` in Example 8–109. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)` in Example 8–110, the query returns the relation in Example 8–111.

Example 8–109 winsorizedmean Function Query

```
<query id="qColtAggr60"><![CDATA[
    select winsorizedmean(c3, c1, c1) from SColtAggrFunc
]]></query>
```

Example 8–110 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 8–111 winsorizedmean Function Relation Output

```
Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
```

9

Functions: java.lang.Math

Oracle CQL provides a variety of built-in functions based on the `java.lang.Math` class.

For more information, see Section 1.1.9, "Functions".

9.1 Introduction to Oracle CQL Built-In java.lang.Math Functions

Table 9–1 lists the built-in `java.lang.Math` functions that Oracle CQL provides.

Table 9–1 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 9–1 (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).

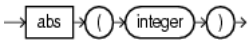
Note: In stream input examples, lines beginning with `h` (such as `h 3800`) are heartbeat input tuples. These inform Oracle CEP that no further input will have a timestamp lesser than the heartbeat value.

For more information, see:

- Section 1.1.9, "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 9–1. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–2, the query returns the stream in Example 9–3.

Example 9–1 abs Function Query

```
<query id="q66"><![CDATA[
    select abs(c1) from SFunc
]]></query>
```

Example 9–2 abs Function Stream Input

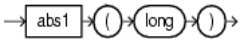
```
Timestamp   Tuple
  10         1,0.5,8
1000        -4,0.7,6
1200        -3,0.89,12
2000         8,0.4,4
```

Example 9–3 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 9–4. Given the data stream `SFunc` with schema `(c1 integer, c2 float, c3 long)` in Example 9–5, the query returns the stream in Example 9–6.

Example 9–4 abs1 Function Query

```
<query id="q67"><![CDATA[
    select abs1(c3) from SFunc
]]></query>
```

Example 9–5 abs1 Function Stream Input

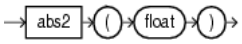
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,-6
1200        3,0.89,-12
2000        8,0.4,4
```

Example 9–6 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 9–7. Given the data stream `SFunc` with schema `(c1 integer, c2 float, c3 bigint)` in Example 9–8, the query returns the stream in Example 9–9.

Example 9–7 abs2 Function Query

```
<query id="q68"><![CDATA[
    select abs2(c2) from SFunc
]]></query>
```

Example 9–8 abs2 Function Stream Input

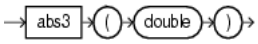
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,-0.7,6
1200        3,-0.89,12
2000        8,0.4,4
```

Example 9–9 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 9–10. Given the data stream `SFunc` with schema `(c1 integer, c2 float, c3 bigint, c4 double)` in Example 9–11, the query returns the stream in Example 9–12.

Example 9–10 abs3 Function Query

```
<query id="q69"><![CDATA[
    select abs3(c4) from SFunc
]]></query>
```

Example 9–11 abs3 Function Stream Input

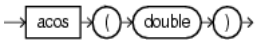
```
Timestamp   Tuple
  10        1,0.5,8,0.25334
1000        4,0.7,6,-4.64322
1200        3,0.89,12,-1.4672272
2000        8,0.4,4,2.66777
```

Example 9–12 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 9–13. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–14, the query returns the stream in Example 9–15.

Example 9–13 acos Function Query

```
<query id="q73"><![CDATA[
    select acos(c2) from SFunc
]]></query>
```

Example 9–14 acos Function Stream Input

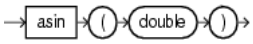
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–15 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 9–16. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–17, the query returns the stream in Example 9–18.

Example 9–16 asin Function Query

```
<query id="q74"><![CDATA[
    select asin(c2) from SFunc
]]></query>
```

Example 9–17 asin Function Stream Input

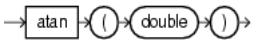
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–18 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 9–19. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–20, the query returns the stream in Example 9–21.

Example 9–19 atan Function Query

```
<query id="q75"><![CDATA[
    select atan(c2) from SFunc
]]></query>
```

Example 9–20 atan Function Stream Input

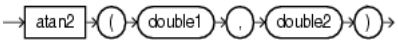
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–21 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,%20double).

Examples

Consider the query `q63` in Example 9–22. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–23, the query returns the stream in Example 9–24.

Example 9–22 atan2 Function Query

```
<query id="q63"><![CDATA[
    select atan2(c2,c2) from SFunc
]]></query>
```

Example 9–23 atan2 Function Stream Input

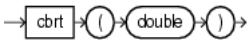
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–24 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 9–25. Given the data stream `SFunc` with schema `(c1 integer, c2 float, c3 bigint)` in Example 9–26, the query returns the stream in Example 9–27.

Example 9–25 cbrt Function Query

```
<query id="q76"><![CDATA[
    select cbrt(c2) from SFunc
]]></query>
```

Example 9–26 cbrt Function Stream Input

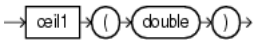
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–27 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 7-13

Examples

Consider the query `q77` in Example 9–28. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–29, the query returns the stream in Example 9–30.

Example 9–28 ceil1 Function Query

```
<query id="q77"><![CDATA[
    select ceil1(c2) from SFunc
]]></query>
```

Example 9–29 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 9–30 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 9–31. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–32, the query returns the stream in Example 9–33.

Example 9–31 cos Function Query

```
<query id="q61"><![CDATA[
    select cos(c2) from SFunc
]]></query>
```

Example 9–32 cos Function Stream Input

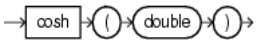
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–33 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 9–34. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–35, the query returns the stream in Example 9–36.

Example 9–34 cosh Function Query

```
<query id="q78"><![CDATA[
    select cosh(c2) from SFunc
]]></query>
```

Example 9–35 cosh Function Stream Input

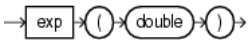
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–36 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 9-16

Examples

Consider the query `q79` in Example 9–37. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–38, the query returns the stream in Example 9–39.

Example 9–37 exp Function Query

```
<query id="q79"><![CDATA[
    select exp(c2) from SFunc
]]></query>
```

Example 9–38 exp Function Stream Input

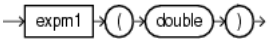
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–39 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 9–1 shows as a `double`, where `x` is the `double` argument and *e* is Euler's number.

Figure 9–1 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 9-15

Examples

Consider the query `q80` in Example 9–40. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–41, the query returns the stream in Example 9–42.

Example 9–40 expm1 Function Query

```
<query id="q80"><![CDATA[
    select expm1(c2) from SFunc
]]></query>
```

Example 9–41 expm1 Function Stream Input

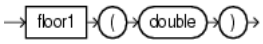
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–42 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 7-19

Examples

Consider the query `q81` in Example 9–43. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–44, the query returns the stream in Example 9–45.

Example 9–43 floor1 Function Query

```
<query id="q81"><![CDATA[
    select floor1(c2) from SFunc
]]></query>
```

Example 9–44 floor1 Function Stream Input

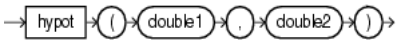
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–45 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 9–2) of the `double` arguments as a `double`.

Figure 9–2 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 9–46. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–47, the query returns the stream in Example 9–48.

Example 9–46 hypot Function Query

```
<query id="q82"><![CDATA[
    select hypot(c2,c2) from SFunc
]]></query>
```

Example 9–47 hypot Function Stream Input

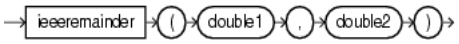
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–48 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 9–49. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–50, the query returns the stream in Example 9–51.

Example 9–49 ieeeremainder Function Query

```
<query id="q72"><![CDATA[
    select ieeeremainder(c2,c2) from SFunc
]]></query>
```

Example 9–50 ieeeremainder Function Stream Input

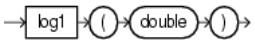
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–51 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 9-22

Examples

Consider the query `q83` in Example 9–52. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–53, the query returns the stream in Example 9–54.

Example 9–52 log1 Function Query

```
<query id="q83"><![CDATA[
    select log1(c2) from SFunc
]]></query>
```

Example 9–53 log1 Function Stream Input

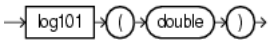
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–54 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 9–55. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–56, the query returns the stream in Example 9–57.

Example 9–55 log101 Function Query

```
<query id="q84"><![CDATA[
    select log101(c2) from SFunc
]]></query>
```

Example 9–56 log101 Function Stream Input

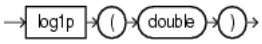
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–57 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 9-20

Examples

Consider the query `q85` in Example 9–58. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–59, the query returns the stream in Example 9–60.

Example 9–58 log1p Function Query

```
<query id="q85"><![CDATA[
    select log1p(c2) from SFunc
]]></query>
```

Example 9–59 log1p Function Stream Input

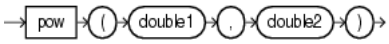
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–60 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 9–61. Given the data stream SFunc with schema (c1 integer, c2 double, c3 bigint) in Example 9–62, the query returns the stream in Example 9–63.

Example 9–61 pow Function Query

```
<query id="q65"><![CDATA[
    select pow(c2,c2) from SFunc
]]></query>
```

Example 9–62 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 9–63 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 9–64. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–65, the query returns the stream in Example 9–66.

Example 9–64 rint Function Query

```
<query id="q86"><![CDATA[
    select rint(c2) from SFunc
]]></query>
```

Example 9–65 rint Function Stream Input

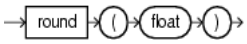
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–66 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 9–67. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–68, the query returns the stream in Example 9–69.

Example 9–67 round Function Query

```
<query id="q87"><![CDATA[
    select round(c2) from SFunc
]]></query>
```

Example 9–68 round Function Stream Input

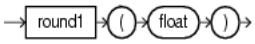
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–69 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 9–70. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–71, the query returns the stream in Example 9–72.

Example 9–70 round1 Function Query

```
<query id="q88"><![CDATA[
    select round1(c2) from SFunc
]]></query>
```

Example 9–71 round1 Function Stream Input

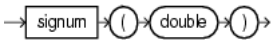
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–72 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 9–73. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–74, the query returns the stream in Example 9–75.

Example 9–73 signum Function Query

```
<query id="q70"><![CDATA[
    select signum(c2) from SFunc
]]></query>
```

Example 9–74 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 9–75 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 9–76. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–77, the query returns the relation in Example 9–78.

Example 9–76 signum1 Function Query

```
<query id="q71"><![CDATA[
    select signum1(c2) from SFunc
]]></query>
```

Example 9–77 signum1 Function Stream Input

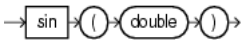
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,-0.7,6
1200        3,-0.89,12
2000        8,0.4,4
```

Example 9–78 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 9–79. Given the data stream `SFunc` with schema (`c1 integer`, `c2 float`, `c3 bigint`) in Example 9–80, the query returns the stream in Example 9–81.

Example 9–79 sin Function Query

```
<query id="q60"><![CDATA[
    select sin(c2) from SFunc
]]></query>
```

Example 9–80 sin Function Stream Input

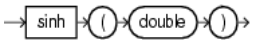
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–81 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 9–82. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–83, the query returns the stream in Example 9–84.

Example 9–82 sinh Function Query

```
<query id="q89"><![CDATA[
    select sinh(c2) from SFunc
]]></query>
```

Example 9–83 sinh Function Stream Input

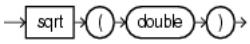
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–84 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 9–85. Given the data stream `SFunc` with schema (`c1 integer`, `c2 float`, `c3 bigint`) in Example 9–86, the query returns the stream in Example 9–87.

Example 9–85 sqrt Function Query

```
<query id="q64"><![CDATA[
    select sqrt(c2) from SFunc
]]></query>
```

Example 9–86 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 9–87 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 9–88. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–89, the query returns the stream in Example 9–90.

Example 9–88 tan Function Query

```
<query id="q62"><![CDATA[
    select tan(c2) from SFunc
]]></query>
```

Example 9–89 tan Function Stream Input

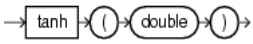
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–90 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 9–91. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–92, the query returns the stream in Example 9–93.

Example 9–91 tanh Function Query

```
<query id="q90"><![CDATA[
    select tanh(c2) from SFunc
]]></query>
```

Example 9–92 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 9–93 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 9-35
- "cos" on page 9-13

Examples

Consider the query `q91` in Example 9–94. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–95, the query returns the stream in Example 9–96.

Example 9–94 todegrees Function Query

```
<query id="q91"><![CDATA[
    select todegrees(c2) from SFunc
]]></query>
```

Example 9–95 todegrees Function Stream Input

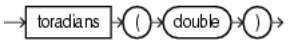
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–96 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 9-34
- "cos" on page 9-13

Examples

Consider the query `q92` in Example 9–97. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–98, the query returns the stream in Example 9–99.

Example 9–97 toradians Function Query

```
<query id="q92"><![CDATA[
    select toradians(c2) from SFunc
]]></query>
```

Example 9–98 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 9–99 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 9–100. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–101, the query returns the stream in Example 9–102.

Example 9–100 ulp Function Query

```
<query id="q93"><![CDATA[
    select ulp(c2) from SFunc
]]></query>
```

Example 9–101 ulp Function Stream Input

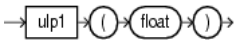
```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4
```

Example 9–102 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 9–103. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)` in Example 9–104, the query returns the relation in Example 9–105.

Example 9–103 ulp1 Function Query

```
<query id="q94"><![CDATA[
    select ulp1(c2) from SFunc
]]></query>
```

Example 9–104 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 9–105 ulp1 Function Relation Output

```
Timestamp   Tuple Kind  Tuple
  10:       +           5.9604645E-8
1000:       +           5.9604645E-8
1200:       +           5.9604645E-8
2000:       +           2.9802322E-8
```

10

Functions: User-Defined

Use user-defined functions to perform more advanced or application-specific operations on stream data than is possible using built-in functions.

For more information, see Section 1.1.9, "Functions".

10.1 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 CEP server classpath or the Oracle CEP server classpath must be modified to include the JAR file.

For more information, see:

- Section 10.1.1, "Types of User-Defined Functions"
- Section 10.1.2, "User-Defined Function Datatypes"
- Section 10.1.3, "User-Defined Functions and the Oracle CEP Server Cache"
- Section 10.2, "Implementing a User-Defined Function"
- Section 1.1.9, "Functions"

10.1.1 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:

- Single-row: 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 10-3.

- Aggregate: 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 10-4.

You can create overloaded functions and you can override built-in functions.

10.1.2 User-Defined Function Datatypes

Table 10–1 lists the datatypes you can specify when you implement and register a user-defined function.

Table 10–1 User-Defined Function Datatypes

Oracle CQL Datatype	Equivalent Java Datatype
`int`	`java.lang.Integer`
`bigint`	`java.lang.Long`
`float`	`java.lang.Float`
`double`	`java.lang.Double`
`char`	`java.lang.String`

The **Oracle CQL Datatype** column lists the datatypes you can specify in the Oracle CQL statement you use to register your user-defined function and the **Equivalent Java Datatype** column lists the Java datatype equivalents you can use in your user-defined function implementation.

At run time, Oracle CEP 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 CEP will throw a `ClassCastException`.

For more information about data conversion, see Section 2.3.4, "Data Conversion".

10.1.3 User-Defined Functions and the Oracle CEP Server Cache

You can access an Oracle CEP cache from an Oracle CQL statement or user-defined function.

For more information, see:

- "Configuring Oracle CEP Caching" in the *Oracle CEP IDE Developer's Guide for Eclipse*
- "Accessing a Cache From an Oracle CQL Statement" in the *Oracle CEP IDE Developer's Guide for Eclipse*
- "Accessing a Cache From an Oracle CQL User-Defined Function" in the *Oracle CEP IDE Developer's Guide for Eclipse*

10.2 Implementing a User-Defined Function

This section describes:

- Section 10.2.1, "How to Implement a User-Defined Single-Row Function"
- Section 10.2.2, "How to Implement a User-Defined Aggregate Function"

For more information, see Section 10.1, "Introduction to Oracle CQL User-Defined Functions".

10.2.1 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 10–1 shows.

   Ensure that the data type of the return value corresponds to a supported data type as Section 10.1.2, "User-Defined Function Datatypes" decribes.

   For more information on accessing the Oracle CEP cache from a user-defined function, see Section 10.1.3, "User-Defined Functions and the Oracle CEP Server Cache".

Example 10–1 MyMod.java User-Defined Single-Row Function

```
package com.bea.wlevs.example.function;

public class MyMod {
    public Object execute(Object[] args) {
        int arg0 = ((Integer)args[0]).intValue();
        int arg1 = ((Integer)args[1]).intValue();
        return new Integer(arg0 % arg1);
    }
}
```

2. Compile the user-defined function Java implementation class and register the class in your Oracle CEP application assembly file as Example 10–2 shows.

Example 10–2 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 CEP IDE Developer's Guide 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 10–3 shows.

Example 10–3 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>
...
```

10.2.2 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 10–4 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 10–4 supports incremental processing.

   Ensure that the data type of the return value corresponds to a supported data type as Section 10.1.2, "User-Defined Function Datatypes" decribes.

   For more information on accessing the Oracle CEP cache from a user-defined function, see Section 10.1.3, "User-Defined Functions and the Oracle CEP Server Cache".

Example 10–4 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 CEP application assembly file as Example 10–5 shows.

Example 10–5 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 CEP IDE Developer's Guide 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 10–6 shows.

Example 10–6 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 CEP passes only the new event (rather than the entire active set) to the appropriate handler context by invoking the appropriate `handlePlus*` method in Example 10–4. 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 CEP passes only the expired event (rather than the entire active set) to the appropriate handler context by invoking the appropriate `handleMinus` method in Example 10–4. As before, the state in the handler context can be incrementally updated to accommodate expiry of the event in an incremental fashion.

11 Expressions

An **expression** is a combination of one or more values and one or more operations. A value can be 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))
```

11.1 Introduction to Expressions

Oracle CEP provides the following expressions:

- Aggregate distinct expressions: "aggr_distinct_expr" on page 11-3.
- Aggregate expressions: "aggr_expr" on page 11-4.
- Arithmetic expressions: "arith_expr" on page 11-6.
- Arithmetic expression list: "arith_expr_list" on page 11-8.
- Case expressions: "case_expr" on page 11-9.
- Decode expressions: "decode" on page 11-13.
- Function expressions: "func_expr" on page 11-15.
- Order expressions: "order_expr" on page 11-19.
- XML aggregate expressions: "xml_agg_expr" on page 11-20
- XML column attribute value expressions: "xmlcolattval_expr" on page 11-22.
- XML element expressions: "xmlelement_expr" on page 11-24.
- XML forest expressions: "xmlforest_expr" on page 11-26.
- XML parse expressions: "xml_parse_expr" on page 11-28

You can use expressions in:

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

- The `VALUES` clause of the `INSERT` statement

Oracle CEP does not accept all forms of expressions in all parts of all Oracle CQL statements. Refer to the individual Oracle CQL statements in Chapter 16, "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.

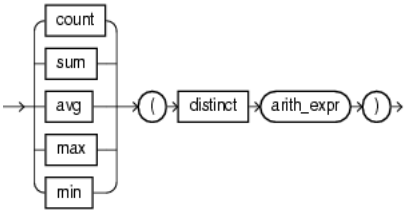
Note: In stream input examples, lines beginning with `h` (such as `h 3800`) are heartbeat input tuples. These inform Oracle CEP that no further input will have a timestamp lesser than the heartbeat value.

---

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 11-4.

aggr_distinct_expr::=



(*arith_expr*::= on page 11-6)

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 11-6

For more information, see Chapter 6, "Functions: Aggregate".

Examples

Example 11–2 shows how to use a `COUNT` aggregate distinct expression.

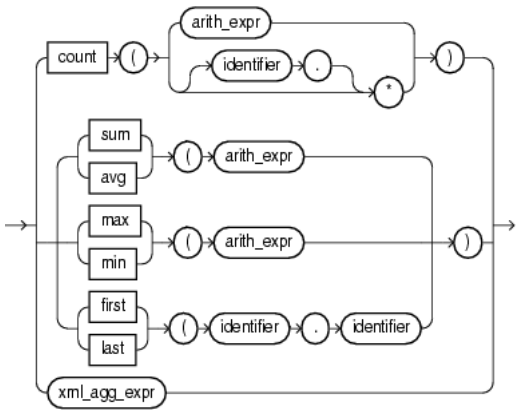
Example 11–1 aggr_distinct_expr for COUNT

```
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 11-3

aggr_expr::=



(*arith_expr::=* on page 11-6, *identifier::=* on page 13-11, *xml_agg_expr::=* on page 11-20)

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

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 11-6

For more information, see:

- Chapter 6, "Functions: Aggregate"
- "first" on page 6-6
- "last" on page 6-8

Examples

Example 11–2 shows how to use a `COUNT` aggregate expression.

Example 11–2 aggr_expr for COUNT

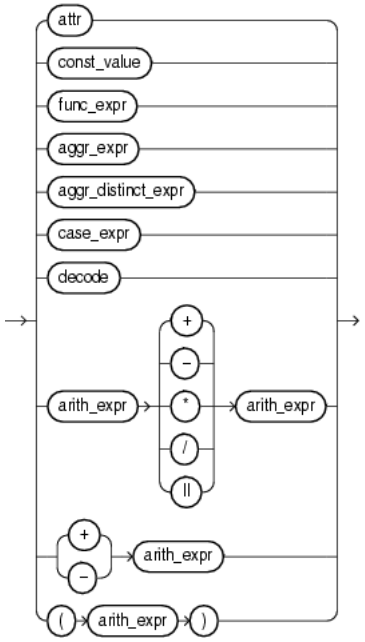
```
<view id="SegVol" schema="exp_way lane dir seg volume"><![CDATA[
    select
        exp_way,
        lane,
        dir,
        seg,
        count(*) as volume
    from
        CurCarSeg
```

```
    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 13-2, *const_value*::= on page 13-8, *func_expr*::= on page 11-15, *aggr_expr*::= on page 11-4, *aggr_distinct_expr*::= on page 11-3, *case_expr*::= on page 11-9, *decode*::= on page 11-13, *arith_expr*::= on page 11-6)

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

- *aggr_distinct_expr*::= on page 11-3
- *aggr_expr*::= on page 11-4
- *arith_expr*::= on page 11-6
- *case_expr*::= on page 11-9
- *searched_case*::= on page 11-9
- *simple_case*::= on page 11-9
- *between_condition*::= on page 12-7
- *non_mt_arg_list*::= on page 13-13
- *condition*::= on page 12-4
- *measure_column*::= on page 15-9
- *projterm*::= on page 16-3

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

Examples

Example 11–2 shows how to use a *arith_expr* expression.

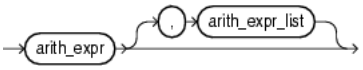
Example 11–3 arith_expr

```
<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_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 11-6)

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

- *xmlelement_expr::=* on page 11-24

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

Examples

Example 11–4 shows how to use a *arith_expr_list* expression.

Example 11–4 arith_expr_list

```
<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>
```

case_expr

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

case_expr::=

(*searched_case_list::=* on page 11-9, *arith_expr::=* on page 11-6, *simple_case_list::=* on page 11-9)

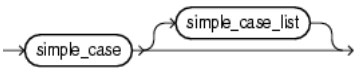
searched_case_list::=

(*searched_case::=* on page 11-9)

searched_case::=

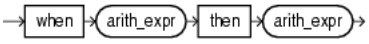
(*non_mt_cond_list::=* on page 13-17, *arith_expr::=* on page 11-6)

simple_case_list::=



(*simple_case::=* on page 11-9)

simple_case::=



(*arith_expr::=* on page 11-6)

The *case_expr* is similar to the DECODE clause of an arithmetic expression (see "decode" on page 11-13).

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 11-6
- *opt_where_clause::=* on page 16-4
- *select_clause::=* on page 16-3

Examples

This section describes the following `case_expr` examples:

case_expr with SELECT *

Consider the query `q97` in Example 11–5 and the data stream `S0` in Example 11–6. Stream `S1` has schema `(c1 float, c2 integer)`. The query returns the relation in Example 11–7.

Example 11–5 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 11–6 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 11–7 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 11–8 and the data streams `S0` in Example 11–9 and `S1` in Example 11–10. Stream `S0` has schema `(c1 float, c2 integer)` and stream `S1` has schema `(c1 float, c2 integer)`. The query returns the relation in Example 11–11.

Example 11–8 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 11–9 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 11–10 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 11–11 CASE Expression: SELECT Relation Output

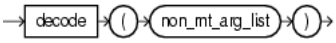
```
Timestamp   Tuple Kind  Tuple
     1000:  +           0.6
     1002:  +           0.44
     1002:  +           0.4
     1002:  +           0.14
   200000:  +           0.5
   200000:  +           0.5
   200000:  +           0.4
   200000:  +           0.44
   200000:  +           0.7
   300000:  +           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 13-13)

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 2.2, "Datatypes").

The *search*, *result*, and *default* values can be derived from expressions. Oracle CEP 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 CEP never evaluates a search i, if a previous search `j (0 < j < i)` equals *expr*.

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

In a DECODE expression, Oracle CEP considers two nulls to be equivalent. If *expr* is null, then Oracle CEP 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 11-9).

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

- *arith_expr*::= on page 11-6
- *opt_where_clause*::= on page 16-4
- *select_clause*::= on page 16-3

Examples

Consider the query q in Example 11–12 and the input relation R in Example 11–13. Relation R has schema `(c1 float, c2 integer)`. The query returns the relation in Example 11–14.

Example 11–12 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 11–13 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
```

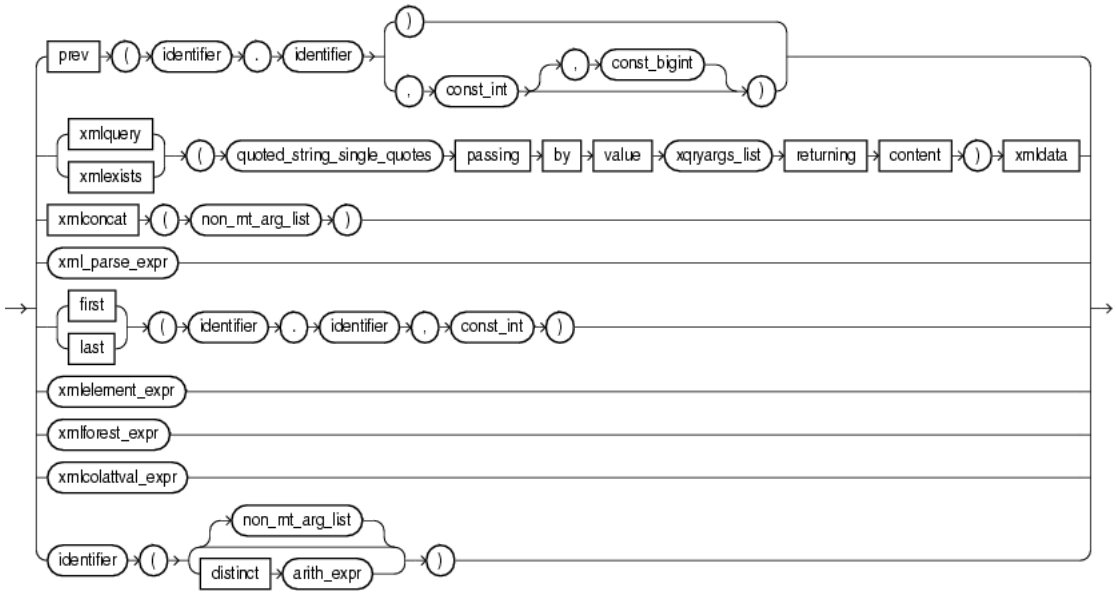
Example 11–14 Arithmetic Expression and DECODE Relation Output

```
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 or user-defined function.

func_expr::=



(*identifier*::= on page 13-11, *const_int*::= on page 13-6, *const_bigint*::= on page 13-5, *const_string*::= on page 13-7, *xqryargs_list*::= on page 13-25, *non_mt_arg_list*::= on page 13-13, *xml_parse_expr*::= on page 11-28, *xmlelement_expr*::= on page 11-24, *xmlforest_expr*::= on page 11-26, *xmlcolattval_expr*::= on page 11-22, *arith_expr*::= on page 11-6)

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 3.2, "ELEMENT_TIME Pseudocolumn").

For more information, see "prev" on page 5-9. For an example, see "func_expr PREV Function Example" on page 11-17.

XQuery: XMLEXISTS and XMLQUERY

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

- "xmlexists" on page 5-25
- "xmlquery" on page 5-27

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 13-25.

For an example, see "func_expr XMLQUERY Function Example" on page 11-17.

XMLCONCAT

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

For more information, see "xmlconcat" on page 5-23.

SQL/XML (SQLX)

The SQLX specification extends SQL to support XML data. 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.

For more information on Oracle CQL support for SQLX, see:

- "xml_agg_expr" on page 11-20
- "xmlcolattval_expr" on page 11-22
- "xmlelement_expr" on page 11-24
- "xmlforest_expr" on page 11-26
- "xml_parse_expr" on page 11-28
- "XQuery: XMLEXISTS and XMLQUERY" on page 11-16
- "xmlcomment" on page 5-21
- "xmlconcat" on page 5-23
- "xmlagg" on page 6-15

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

- Section 2.4.3, "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 6-6
- "last" on page 6-8

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.

IDENTIFIER

You can specify the identifier of a function explicitly with a distinct arithmetic expression. For more information, see *aggr_distinct_expr* on page 11-3.

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

- *arith_expr*::= on page 11-6

For more information, see Section 1.1.9, "Functions".

Examples

This section describes the following func_expr examples:

- "func_expr PREV Function Example" on page 11-17
- "func_expr XMLQUERY Function Example" on page 11-17

func_expr PREV Function Example

Example 11–15 shows how to compose a *func_expr* to invoke the PREV function.

Example 11–15 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 11–16 shows how to compose a *func_expr* to invoke the XMLQUERY function.

Example 11–16 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 11–17 shows how to compose a *func_expr* to invoke the `SUM` function.

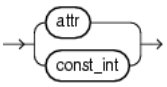
Example 11–17 func_expr for SUM

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

order_expr

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

order_expr::=



(*attr::=* on page 13-2, *const_int::=* on page 13-6)

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 16-5

Examples

Consider Example 11–18. 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 11–18 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 11-6, *order_by_clause::=* on page 16-4)

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 11-4

For more information, see:

- Chapter 6, "Functions: Aggregate"
- "xmlagg" on page 6-15
- "XMLAGG" in the *Oracle Database SQL Language Reference*

Examples

Consider the query `tkdata67_q1` in Example 11–19 and the input relation `tkdata67_S0` in Example 11–20. Relation `tkdata67_S0` has schema `(c1 integer, c2 float)`. The query returns the relation in Example 11–21.

Example 11–19 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 11–20 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 11–21 xml_agg_expr Relation Output

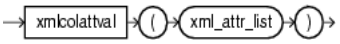
```
Timestamp   Tuple Kind  Tuple
1000:       +           15,<c2>0.1</c2>
                           <c2>0.2</c2>
1000:       +           20,<c2>0.14</c2>
4000:       -           20,<c2>0.14</c2>
4000:       +           20,<c2>0.14</c2>
                           <c2>0.3</c2>
```

```
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 13-24)

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 11-16.

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

- *func_expr*::= on page 11-15

For more information, see "XMLCOLATTVAL" in the *Oracle Database SQL Language Reference*.

Examples

Consider the query `tkdata53_q1` in Example 11–19 and the input relation `tkdata53_S0` in Example 11–20. Relation `tkdata53_S0` has schema `(c1 integer, c2 float)`. The query returns the relation in Example 11–21.

Example 11–22 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 11–23 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 11–24 xmlcolattval_expr Relation Output

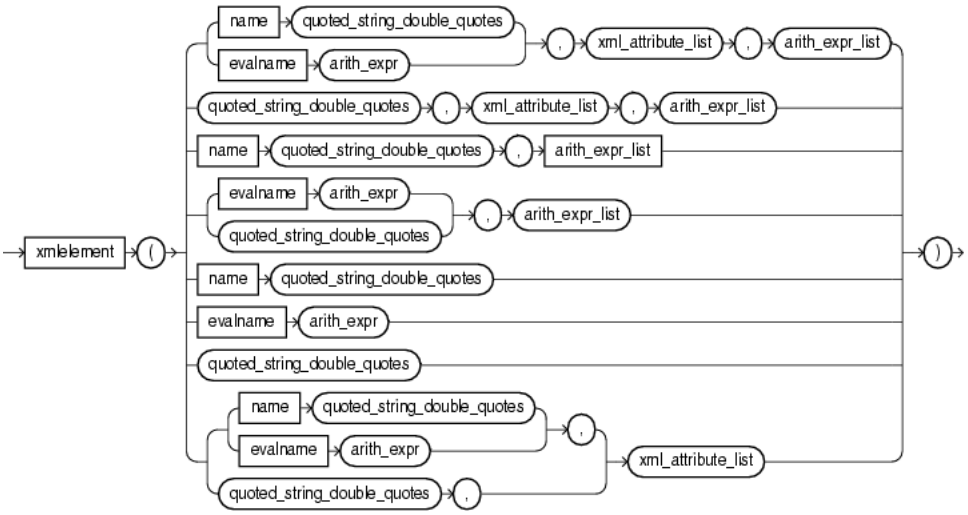
```
Timestamp   Tuple Kind  Tuple
     1000:       +      <tkdata53_S0>
                            <column name="c1">10</column>
                            <column name="c2">0.1</column>
                        </tkdata53_S0>
     1002:       +      <tkdata53_S0>
                            <column 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 13-7, *arith_expr*::= on page 11-6, *xml_attribute_list*::= on page 13-23, *arith_expr_list*::= on page 11-8)

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 11-16.

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

- *func_expr*::= on page 11-15

For more information, see "XMLELEMENT" in the *Oracle Database SQL Language Reference*.

Examples

Consider the query `tkdata51_q0` in Example 11–25 and the input relation `tkdata51_S0` in Example 11–26. Relation `tkdata51_S0` has schema `(c1 integer, c2 float)`. The query returns the relation in Example 11–27.

Example 11–25 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 11–26 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 11–27 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 13-24)

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 11-16.

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

- *func_expr::=* on page 11-15

For more information, see "XMLFOREST" in the *Oracle Database SQL Language Reference*.

Examples

Consider the query `tkdata52_q0` in Example 11–28 and the input relation `tkdata52_S0` in Example 11–29. Relation `tkdata52_S0` has schema `(c1 integer, c2 float)`. The query returns the relation in Example 11–30.

Example 11–28 xmlforest_expr Query

```
<query id="tkdata52_q0"><![CDATA[
    select
        XMLFOREST( tkdata52_S0.c1, tkdata52_S0.c2)
    from
        tkdata52_S0 [range 1]
]]></query>
```

Example 11–29 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 11–30 xmlforest_expr Relation Output

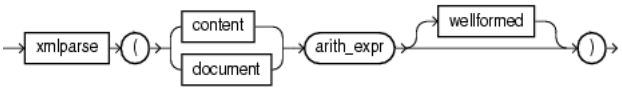
```
Timestamp   Tuple Kind  Tuple
     1000:  +           <c1>10</c1>
                        <c2>0.1</c2>
     1002:  +           <c1>15</c1>
                        <c2>0.14</c2>
     2000:  -           <c1>10</c1>
                        <c2>0.1</c2>
     2002:  -           <c1>15</c1>
                        <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 11-6)

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 11-29
- 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 11-28.
- 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 11-29.

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 11-16.

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

- *func_expr::=* on page 11-15

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 11-28
- "xml_parse_expr Document Example" on page 11-29
- "xml_parse_expr Wellformed Example" on page 11-29

xml_parse_expr Content Example

Consider the query `tkdata62_q3` in Example 11–31 and the input relation `tkdata62_S1` in Example 11–32. Relation `tkdata62_S1` has schema `(c1 char(30))`.The query returns the relation in Example 11–33.

Example 11–31 xml_parse_expr Content: Query

```
<query id="tkdata62_q3"><![CDATA[
    select XMLPARSE(CONTENT c1) from tkdata62_S1
]]></query>
```

Example 11–32 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 11–33 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 11–34 and the input relation `tkdata62_S1` in Example 11–35. Relation `tkdata62_S1` has schema `(c1 char(30))`. The query returns the relation in Example 11–36.

Example 11–34 xml_parse_expr Document: Query

```
<query id="tkdata62_q4"><![CDATA[
    select XMLPARSE(DOCUMENT c1) from tkdata62_S1
]]></query>
```

Example 11–35 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 11–36 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 11–37 and the input relation `tkdata62_S` in Example 11–38. Relation `tkdata62_S` has schema `(c char(30))`. The query returns the relation in Example 11–39.

Example 11–37 xml_parse_expr Wellformed: Query

```
<query id="tkdata62_q2"><![CDATA[
    select XMLPARSE(DOCUMENT c WELLFORMED) from tkdata62_S
]]></query>
```

Example 11–38 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 11–39 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>
```

12 Conditions

A **condition** specifies a combination of one or more expressions and logical operators and returns a value of TRUE, FALSE, or UNKNOWN.

12.1 Introduction to Conditions

Oracle CQL provides the following conditions:

- Section 12.2, "Comparison Conditions"
- Section 12.3, "Logical Conditions"
- Section 12.4, "LIKE Condition"
- Section 12.5, "Range Conditions"
- Section 12.6, "Null Conditions"
- Section 12.7, "Compound Conditions"
- Section 12.8, "IN Condition"

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 16-2

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
```

12.1.1 Condition Precedence

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

Table 12–1 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 12–1 Oracle CQL Condition Precedence

Type of Condition	Purpose
Oracle CQL operators are evaluated before Oracle CQL conditions	See Section 4.1.2, "What You May Need to Know About Operator Precedence".
`=, <>, <, >, <=, >=`	comparison
`IS NULL, IS NOT NULL, LIKE, BETWEEN`	comparison
`NOT`	exponentiation, logical negation
`AND`	conjunction
`OR`	disjunction
`XOR`	disjunction

12.2 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 CEP 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.

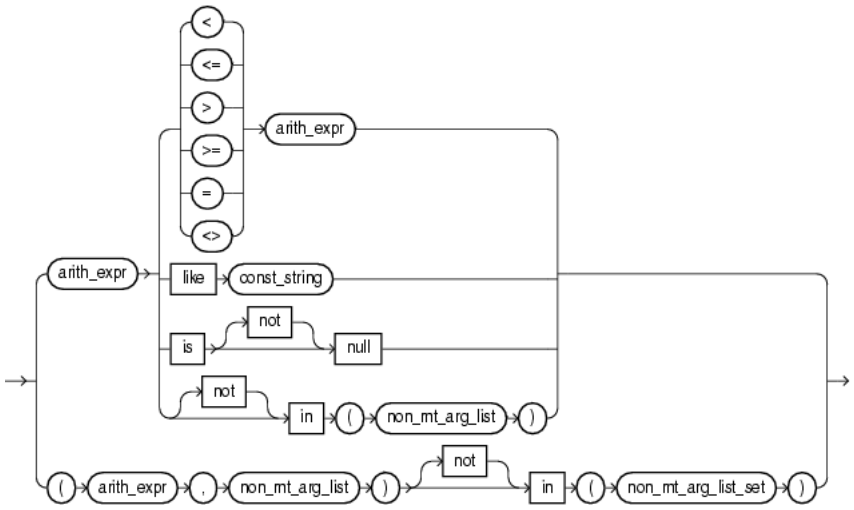
A comparison condition specifies a comparison with expressions or view results.

Table 12–2 lists comparison conditions.

Table 12–2 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 12.4, "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 12.6, "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>
[not] in	Membership tests. For more information, see Section 12.8, "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> <query id="Q2"><![CDATA[SELECT * FROM S0 WHERE salary NOT IN (V1)]]></query>

***condition*::=**



(*arith_expr::=* on page 11-6, *const_string::=* on page 13-7, *non_mt_arg_list::=* on page 13-13, *non_mt_arg_list_set::=* on page 13-17)

12.3 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 12–3 lists logical conditions.

Table 12–3 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 12–4 shows the result of applying the NOT condition to an expression.

Table 12–4 NOT Truth Table

--	TRUE	FALSE	UNKNOWN
NOT	FALSE	TRUE	UNKNOWN

Table 12–5 shows the results of combining the AND condition to two expressions.

Table 12–5 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 12–6 shows the results of applying OR to two expressions.

Table 12–6 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 12–7 shows the results of applying XOR to two expressions.

Table 12–7 XOR Truth Table

XOR	TRUE	FALSE	UNKNOWN
TRUE	FALSE	TRUE	
FALSE	TRUE	FALSE	

Table 12–7 (Cont.) XOR Truth Table

XOR	TRUE	FALSE	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>
```

12.4 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 11-6, *const_string::=* on page 13-7)

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 12–8 describes the `LIKE` conditions.

Table 12–8 LIKE Conditions

Type of Condition	Operation	Example
`x LIKE y`	`TRUE` if *x* does match the pattern *y*, `FALSE` otherwise.	`create query q291 as select * from SLk1 where first1 like "^Ste(v|ph)en$"`

See Also: "lk" on page 5-7

12.4.1 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.

12.5 Range Conditions

A range condition tests for inclusion in a range.

between_condition::=

(*arith_expr*::= on page 11-6)

Table 12–9 describes the range conditions.

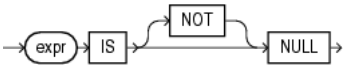
Table 12–9 Range Conditions

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>`

12.6 Null Conditions

A `NULL` condition tests for nulls. This is the only condition that you should use to test for nulls.

***null_conditions*::=**



(Chapter 11, "Expressions")

Table 12–10 lists the null conditions.

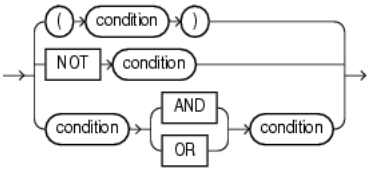
Table 12–10 Null Conditions

Type of Condition	Operation	Example
`IS [NOT] NULL`	Tests for nulls. **See Also:** Section 2.6, "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>`

12.7 Compound Conditions

A compound condition specifies a combination of other conditions.

***compound_conditions*::=**

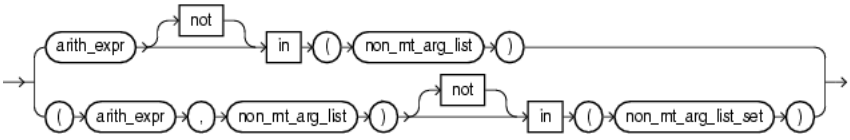


See Also: Section 12.3, "Logical Conditions" for more information about `NOT`, `AND`, and `OR` conditions

12.8 IN Condition

An *`in_condition`* is a membership condition. It tests a value for membership in a list of values or view.

***in_condition*::=**



(*arith_expr::=* on page 11-6, *non_mt_arg_list::=* on page 13-13, *non_mt_arg_list_set::=* on page 13-17)

If you use the upper form of the *`in_condition`* condition (with a single expression to the left of the operator), then you must use a *`non_mt_arg_list`*. If you use the lower

form of this condition (with multiple expressions to the left of the operator), then you must use a *non_mt_arg_list_set*, and the expressions in each list of expressions must match in number and datatype.

Table 12–11 lists the form of IN condition.

Table 12–11 IN Conditions

Type of Condition	Operation	Example
IN	Equal-to-any-member-of test. Equivalent to =ANY.	<query id="Q1"><![CDATA[SELECT * FROM S0 WHERE job_id IN ('PU_CLERK','SH_CLERK')]]></query> <view id="V1" schema="salary"><![CDATA[SELECT salary FROM S0 WHERE department_id = 30]]></view> <query id="Q2"><![CDATA[SELECT * FROM S0 WHERE salary IN (V1)]]></query>
NOT IN	Equivalent to !=ALL. Evaluates to FALSE if any member of the set is NULL.	<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> <query id="Q2"><![CDATA[SELECT * FROM S0 WHERE salary NOT IN (V1)]]></query>

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 the c1 and c2 if c1 is neither 50 nor 30:

```
create query check_notin1 as
    select c1,c2 from S0[range 1]
    where
        c1 not in (50, 30);
```

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:

```
<view id="V1" schema="c1"><![CDATA[
    IStream(select * from S1[range 10 slide 10] where 1=2)
]]></view>
<query id="Q1"><![CDATA[
    select 'True' from S0 where department_id not in (V1)
]]></query>
```

13

Common Oracle CQL DDL Clauses

This chapter describes the Oracle CQL data definition language (DDL) clauses that appear in multiple Oracle CQL statements.

13.1 Introduction to Common Oracle CQL DDL Clauses

Oracle CQL supports the following common DDL clauses:

For more information on Oracle CQL statements, see Chapter 16, "Oracle CQL Statements".

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:

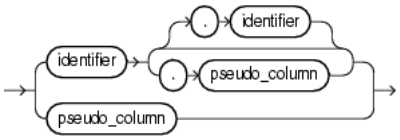
- *arith_expr::=* on page 11-6
- *order_expr::=* on page 11-19
- *non_mt_attr_list::=* on page 13-14

Prerequisites

None.

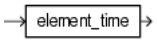
Syntax

attr::=



(*identifier::=* on page 13-11, *pseudo_column::=* on page 13-2)

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 13-3.

For syntax, see *identifier::=* on page 13-11 (parent: *attr::=* on page 13-2).

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 13-3.

For more information, see Chapter 3, "Pseudocolumns".

For syntax, see *pseudo_column::=* on page 13-2 (parent: *attr::=* on page 13-2).

Examples

Given the stream that Example 13–1 shows, valid attribute clauses are:

- `ItemTempStream.temp`
- `temp`
- `B.element_time`
- `element_time`

Example 13–1 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:

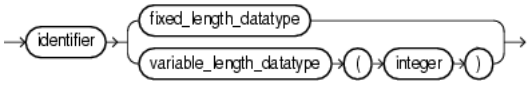
- *non_mt_attrspec_list::=* on page 13-16

Prerequisites

None.

Syntax

attrspec::=



(*identifier::=* on page 13-11, *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 13-11 (parent: *attrspec::=* on page 13-4).

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 13-4).

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 13-4).

integer

Specify the length of the variable-length datatype.

For syntax, see *attrspec::=* on page 13-4.

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 11-15
- *const_value::=* on page 13-8

For more information, see Section 2.4.2, "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 11-15
- *order_expr::=* on page 11-19
- *const_value::=* on page 13-8

For more information, see Section 2.4.2, "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 11-15
- *order_expr::=* on page 11-19
- *condition::=* on page 12-4
- *const_value::=* on page 13-8
- *interval_value::=* on page 13-8
- *xmltable_clause::=* on page 16-6
- *xtbl_col::=* on page 16-6

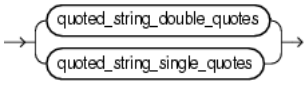
For more information, see Section 2.4.1, "Text Literals".

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 11-6
- *condition::=* on page 12-4

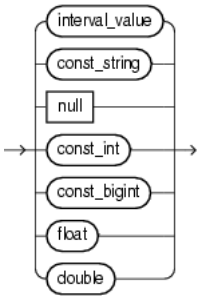
For more information, see Section 2.4, "Literals".

Prerequisites

None.

Syntax

const_value::=



(*interval_value::=* on page 13-8, *const_string::=* on page 13-7, *const_int::=* on page 13-6, *const_bigint::=* on page 13-5)

interval_value::=

(*const_string::=* on page 13-7)

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 2.4.4, "Interval Literals".

For syntax, see *interval_value::=* on page 13-8 (parent: *const_value::=* on page 13-8).

const_string

Specify a quoted `String` constant value.

For more information, see Section 2.4.1, "Text Literals".

For syntax, see *const_string*::= on page 13-7 (parent: *interval_value*::= on page 13-8 and *const_value*::= on page 13-8).

null

Specify a null constant value.

For more information, see Section 2.6, "Nulls".

const_int

Specify an `int` constant value.

For more information, see Section 2.4.2, "Numeric Literals".

bigint

Specify a `bigint` constant value.

For more information, see Section 2.4.2, "Numeric Literals".

float

Specify a float constant value.

For more information, see Section 2.4.2, "Numeric Literals".

identifier

Purpose

Use the `identifier` clause to reference an existing Oracle CQL schema object.

You can reference a Oracle CQL query in the following Oracle CQL statements:

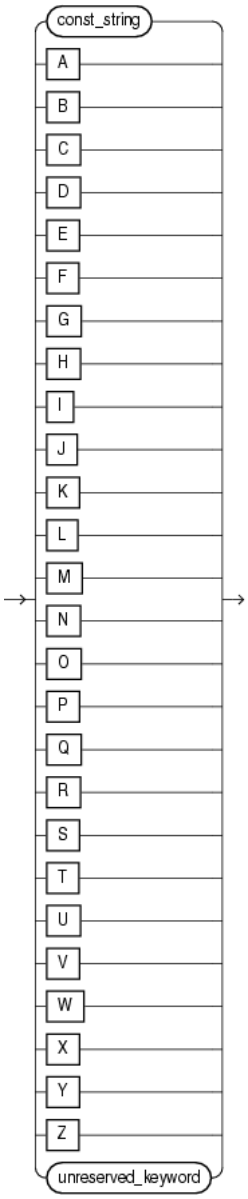
- *binary::=* on page 16-5
- *aggr_expr::=* on page 11-4
- *func_expr::=* on page 11-15
- *attr::=* on page 13-2
- *attrspec::=* on page 13-4
- *query_ref::=* on page 13-20
- *non_mt_attrname_list::=* on page 13-15
- *relation_variable::=* on page 16-4
- *measure_column::=* on page 15-9
- "Query" on page 16-2
- *projterm::=* on page 16-3
- "View" on page 16-18

Prerequisites

The schema object must already exist.

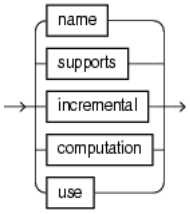
Syntax

identifier::=



(*const_string*::= on page 13-7, *unreserved_keyword*::= on page 13-11)

unreserved_keyword::=



Symantics

const_string

Specify the identifier as a String.

For more information, see Section 2.9.1, "Schema Object Naming Rules".

For syntax, see *identifier::=* on page 13-11.

[A-Z]

Specify the identifier as a single uppercase letter.

For syntax, see *identifier::=* on page 13-11.

unreserved_keyword

These are names that you may use as identifiers.

For more information, see:

- "reserved_keyword" on page 13-12
- Section 2.9.1, "Schema Object Naming Rules"

For syntax, see *unreserved_keyword::=* on page 13-11 (parent: *identifier::=* on page 13-11).

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`.

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 13-14.

You can use the `non_mt_arg_list` clause in the following Oracle CQL statements:

- *decode::=* on page 11-13
- *func_expr::=* on page 11-15
- *condition::=* on page 12-4
- *non_mt_arg_list_set::=* on page 13-17

Prerequisites

If any stream elements are referenced, the stream must already exist.

Syntax

non_mt_arg_list::=

(*arith_expr::=* on page 11-6)

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 13-13.

You can use the `non_mt_attr_list` clause in the following Oracle CQL statements:

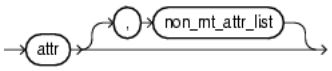
- *pattern_partition_clause::=* on page 15-10
- *window_type::=* on page 16-4
- *opt_group_by_clause::=* on page 16-4

Prerequisites

If any stream elements are referenced, the stream must already exist.

Syntax

non_mt_attr_list::=



(*attr::=* on page 13-2)

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:

- *out_of_line_constraint::=* on page 13-19
- "View" on page 16-18

Prerequisites

If any stream elements are referenced, the stream must already exist.

Syntax

non_mt_attrname_list::=

(*identifier::=* on page 13-11)

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:

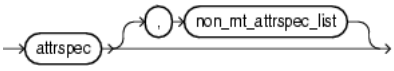
- "View" on page 16-18

Prerequisites

If any stream elements are referenced, the stream must already exist.

Syntax

non_mt_attrspec_list::=



(*attrspec::=* on page 13-4)

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 15-6
- *searched_case::=* on page 11-9
- *opt_where_clause::=* on page 16-4
- *opt_having_clause::=* on page 16-5

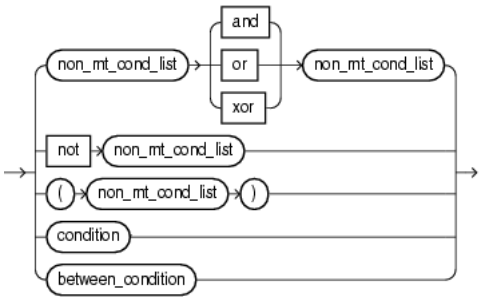
For more information, see Chapter 12, "Conditions".

Prerequisites

None.

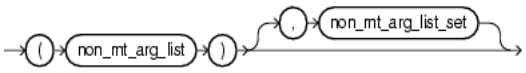
Syntax

non_mt_cond_list::=



(*non_mt_cond_list::=* on page 13-17, *condition::=* on page 12-4, *between_condition::=* on page 12-7)

non_mt_arg_list_set::=



(*non_mt_arg_list::=* on page 13-13)

Semantics

condition

Specify a comparison condition.

For more information, see Section 12.2, "Comparison Conditions".

For syntax, see *condition::=* on page 12-4 (parent: *non_mt_cond_list::=* on page 13-17).

between_condition

Specify a condition that tests for inclusion in a range.

For more information, see Section 12.5, "Range Conditions".

For syntax, see *between_condition::=* on page 12-7 (parent: *non_mt_cond_list::=* on page 13-17).

non_mt_arg_list_set

If you use the form of the `IN` condition with multiple expressions to the left of the operator, then you must use a *`non_mt_arg_list_set`*, and the expressions in each list of expressions must match in number and datatype.

For more information, see Section 12.8, "IN Condition".

For syntax, see *non_mt_arg_list_set::=* on page 13-17 (parent: *non_mt_cond_list::=* on page 13-17).

out_of_line_constraint

Purpose

Use this *out_of_line_constraint* clause to restrict a tuple of any datatype by a primary key integrity constraint.

If you plan to configure a query on a relation with `USE UPDATE SEMANTICS`, you must declare one or more stream elements as a primary key. Use this constraint to specify a compound primary key made up of one or more stream element values.

You can use the *out_of_line_constraint* clause in the following Oracle CQL statements:

- "Query" on page 16-2

For more information, see:

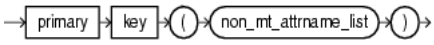
- Section 2.6, "Nulls"

Prerequisites

A tuple that you specify with an *out_of_line_constraint* may not contain a null value.

Syntax

out_of_line_constraint::=



(*non_mt_attrname_list*::= on page 13-15)

Semantics

non_mt_attrname_list

Specify one or more tuples to restrict by a primary key integrity constraint.

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:

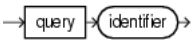
- "View" on page 16-18

Prerequisites

The query must already exist (see "Query" on page 16-2).

Syntax

query_ref::=



(*identifier::=* on page 13-11)

Symantics

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 CEP assumes `[second|seconds]`.

You can use the *time_spec* clause in the following Oracle CQL statements:

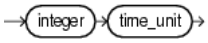
- *window_type::=* on page 16-4
- *duration_clause::=* on page 15-6

Prerequisites

None.

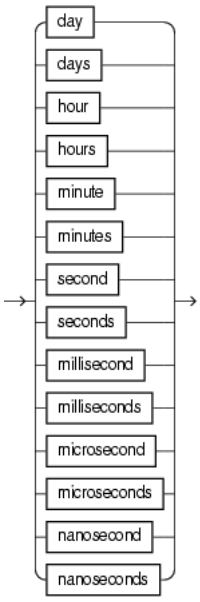
Syntax

time_spec::=



(*time_unit::=* on page 13-21)

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 11-24

Prerequisites

If any stream elements are referenced, the stream must already exist.

Syntax

xml_attribute_list::=

(*xml_attr_list::=* on page 13-24)

Semantics

xml_attr_list

Specify one or more XML attributes as Example 13–2 shows.

Example 13–2 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 13-24 (parent: *xml_attribute_list::=* on page 13-23).

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:

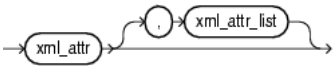
- "xml_attribute_list" on page 13-23
- "xmlforest_expr" on page 11-26
- "xml_agg_expr" on page 11-20

Prerequisites

If any stream elements are referenced, the stream must already exist.

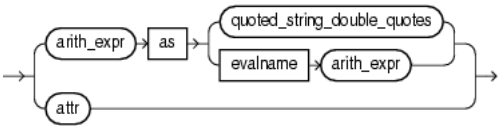
Syntax

xml_attr_list::=



(*xml_attr*::= on page 13-24)

xml_attr::=



(*const_string*::= on page 13-7, *arith_expr*::= on page 11-6, *attr*::= on page 13-2)

Semantics

xml_attr

Specify an XML attribute.

For syntax, see *xml_attr*::= on page 13-24 (parent: *xml_attr_list*::= on page 13-24).

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 5-25
- "xmlquery" on page 5-27
- *func_expr::=* on page 11-15
- *xmltable_clause::=* on page 16-6

Prerequisites

If any stream elements are referenced, the stream must already exist.

Syntax

xqryargs_list::=

(*xqryarg::=* on page 13-25)

xqryarg::=

(*const_string::=* on page 13-7, *arith_expr::=* on page 11-6)

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 11-6) to either a variable in a given XQuery or an XPath operator such as `"."` as a quoted string.

For syntax, see *xqryarg::=* on page 13-25 (parent: *xqryargs_list::=* on page 13-25).

14

Oracle CQL Queries, Views, and Joins

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.

For more information, see:

- Section 1.2.1, "Oracle CQL Statement Lexical Conventions"
- Section 1.2.2, "Oracle CQL Statement Documentation Conventions"
- Chapter 2, "Basic Elements of Oracle CQL"
- Chapter 13, "Common Oracle CQL DDL Clauses"
- Chapter 16, "Oracle CQL Statements"

14.1 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 CEP CQL Processor to process data from one or more streams or views. For more information, see Section 14.2, "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 14.3, "Views".

Oracle CEP performs a join whenever multiple streams appear in the `FROM` clause of the query. For more information, see Section 14.4, "Joins".

Example 14–1 shows typical Oracle CQL queries defined in an Oracle CQL processor component configuration file for the processor named `proc`.

Example 14–1 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 [Now]
                WHERE  price > 10000
            ]]></query>
        </rules>
    </processor>
</n1:config>
```

As Example 14–1 shows, the rules element contains each Oracle CQL statement in a `rule`, `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 14–1, 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 14–1, 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 "How to Configure a Channel in the Default Component Configuration File Using Oracle CEP IDE for Eclipse" in the *Oracle CEP IDE Developer's Guide for Eclipse*.

Each Oracle CQL statement is contained in a `<![CDATA[ ... ]]>` tag and does *not* end in a semicolon (`;`).

For more information, see:

- Section 1.2.1, "Oracle CQL Statement Lexical Conventions"
- Chapter 16, "Oracle CQL Statements"

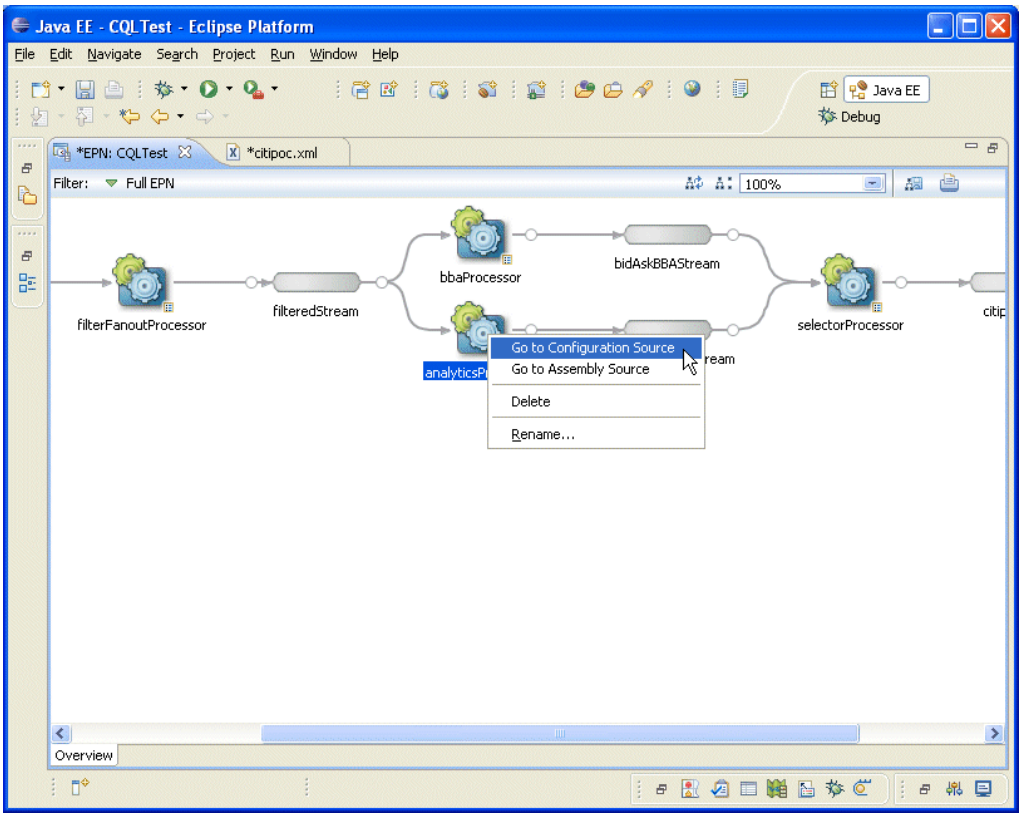
To create an Oracle CQL query:

1. Using Oracle CEP IDE for Eclipse, create an Oracle CEP application and Event Processing Network (EPN).

   For more information, see *Oracle CEP IDE Developer's Guide for Eclipse*.

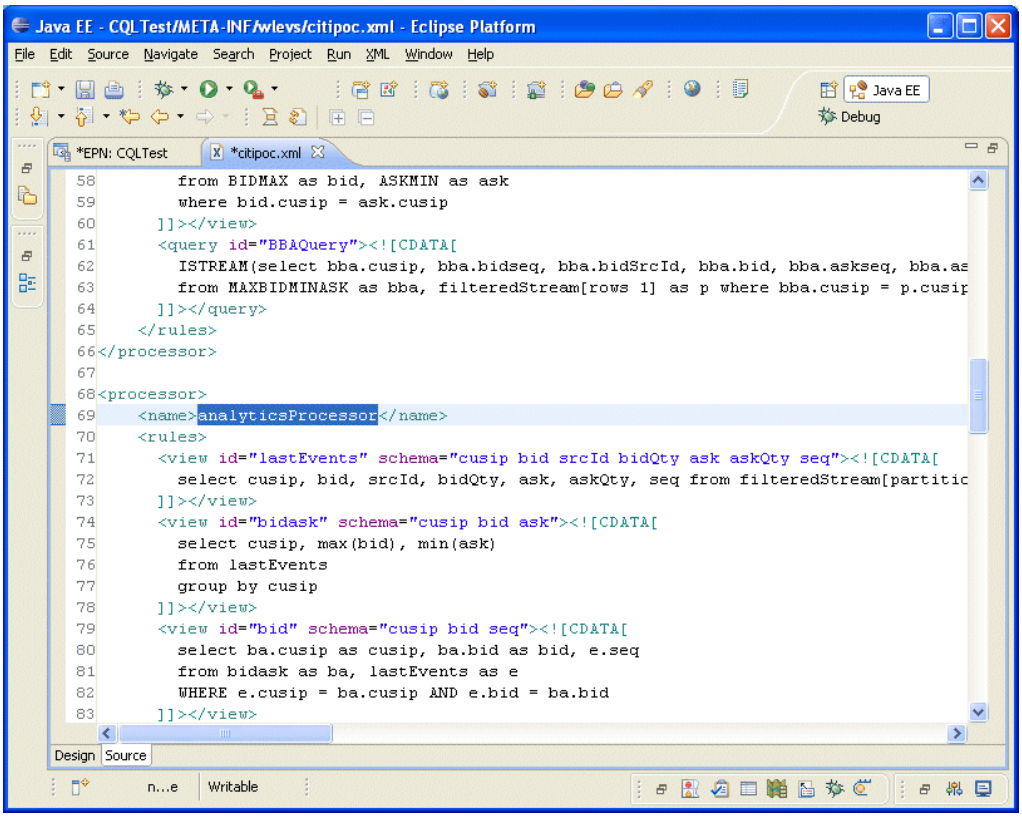
2. In the EPN Editor, right-click an Oracle CQL processor and select **Go to Configuration Source** as Figure 14–1 shows.

Figure 14–1 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 14–2 shows.

Figure 14–2 Editing the Configuration Source for a Processor



3. Create queries and views and register user-defined functions and windows.

   For examples, see

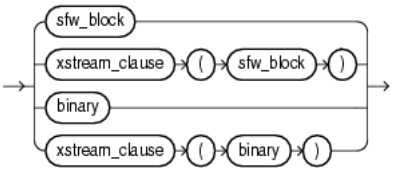
   - Section 14.2, "Queries"
   - Section 14.3, "Views"
   - Section 14.4, "Joins"
   - Section 14.5, "Oracle CQL Queries and the Oracle CEP Server Cache"
   - Chapter 4, "Operators"
   - Chapter 5, "Functions: Single-Row"
   - Chapter 6, "Functions: Aggregate"
   - Chapter 7, "Functions: Colt Single-Row"
   - Chapter 9, "Functions: java.lang.Math"
   - Chapter 10, "Functions: User-Defined"
   - Chapter 11, "Expressions"
   - Chapter 12, "Conditions"
   - Chapter 15, "Pattern Recognition With MATCH_RECOGNIZE"
   - *Oracle CEP Getting Started*

4. Using Oracle CEP IDE for Eclipse, package your Oracle CEP application and deploy to the Oracle CEP server.

14.2 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 14.2.1.1, "Select, From, Where Block".

- `binary`: use this clause to perform set operations on the tuples that two queries or views return.

  For more information, see Section 14.2.1.8, "Binary Clause"

- `xstream_clause`: use this clause to specify a relation-to-stream operator that applies to the query.

  For more information, see Section 14.2.1.9, "Xstream Clause"

The following sections discuss the basic query types that you can create:

- Section 14.2.2, "Simple Query"
- Section 14.2.3, "Built-In Window Query"
- Section 14.2.4, "MATCH_RECOGNIZE Query"
- Section 14.2.5, "XMLTable Query"

14.2.1 Query Building Blocks

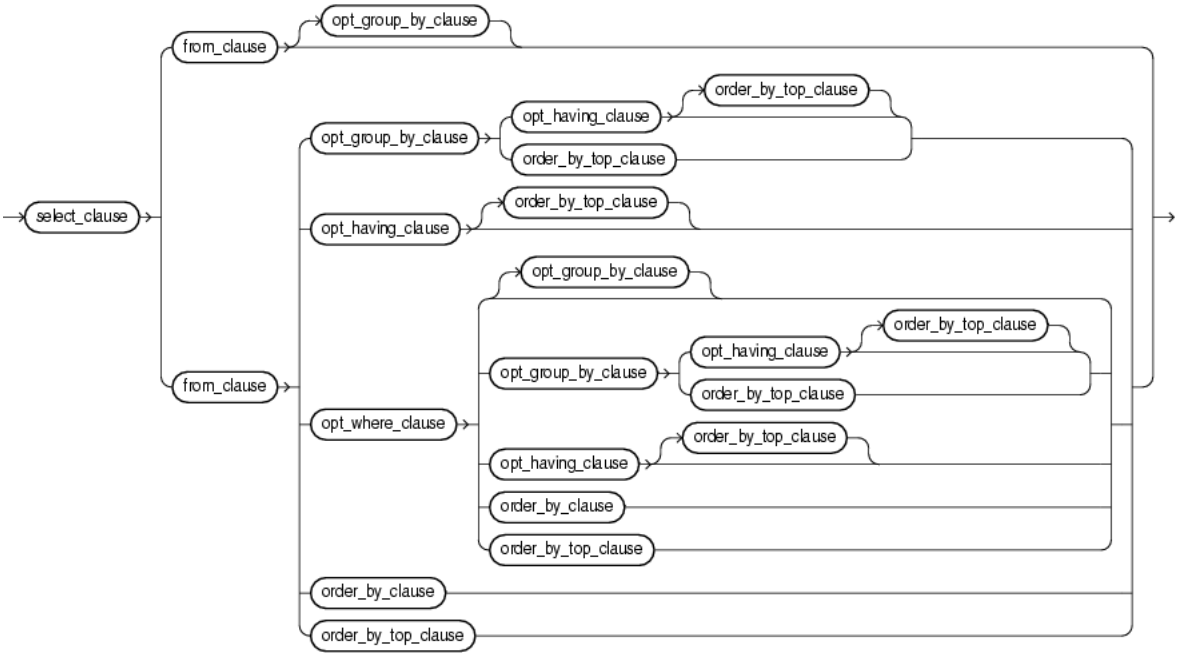
This section summarizes the basic building blocks that you use to construct an Oracle CQL query, including:

- Section 14.2.1.1, "Select, From, Where Block"
- Section 14.2.1.2, "Select Clause"
- Section 14.2.1.3, "From Clause"
- Section 14.2.1.4, "Where Clause"
- Section 14.2.1.5, "Group By Clause"
- Section 14.2.1.6, "Order By Clause"
- Section 14.2.1.7, "Having Clause"
- Section 14.2.1.8, "Binary Clause"
- Section 14.2.1.9, "Xstream Clause"

14.2.1.1 Select, From, Where Block

Use the `sfw_block` to specify the select, from, and optional where clauses of your Oracle CQL query.

***sfw_block*::=**



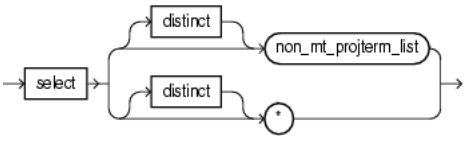
The `sfw_block` is made up of the following parts:

- Section 14.2.1.2, "Select Clause"
- Section 14.2.1.3, "From Clause"
- Section 14.2.1.4, "Where Clause"
- Section 14.2.1.5, "Group By Clause"
- Section 14.2.1.6, "Order By Clause"
- Section 14.2.1.7, "Having Clause"

14.2.1.2 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 CEP 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 CEP 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.

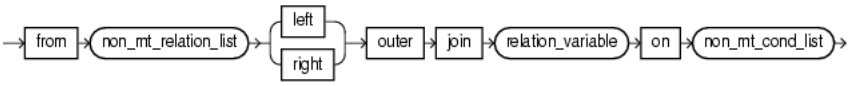
For more information, see *select_clause::=* on page 16-3

14.2.1.3 From Clause

Use this clause to specify the streams and views that provide the stream elements you specify in the `select_clause` (see Section 14.2.1.2, "Select Clause").

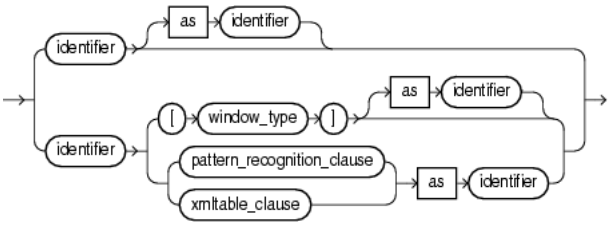
The `from_clause` may specify one or more comma-delimited `relation_variable` clauses.

***from_clause*::=**



For more information, see *from_clause::=* on page 16-3

***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 2.8.1, "Aliases in the relation_variable Clause").

If you create a join (see Section 14.4, "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 14–2 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 14–2 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 CEP often does less work with fully qualified stream element names.

14.2.1.4 Where Clause

Use this optional clause to specify conditions that determine when the `select_clause` returns results (see Section 14.2.1.2, "Select Clause").

For more information, see *opt_where_clause::=* on page 16-4.

14.2.1.5 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.

For more information, see:

- *opt_group_by_clause::=* on page 16-4
- Section 14.2.1.5, "Group By Clause"

14.2.1.6 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 16-4
- *order_by_top_clause::=* on page 16-5
- Section 14.2.6, "Sorting Query Results"

14.2.1.7 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 CEP returns summary results for all groups.

For more information, see *opt_having_clause::=* on page 16-5.

14.2.1.8 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 16-5.

14.2.1.9 Xstream Clause

Use this clause to take either a select-from-where clause or binary clause and return its results as one of IStream, DStream, or Rstream relation-to-stream operators.

For more information, see:

- *xstream_clause::=* on page 16-6
- "IStream Relation-to-Stream Operator" on page 4-24
- "DStream Relation-to-Stream Operator" on page 4-25
- "RStream Relation-to-Stream Operator" on page 4-26

14.2.2 Simple Query

Example 14–3 shows a simple query that selects all stream elements from a single stream.

Example 14–3 Simple Query

```
<query id="q0"><![CDATA[
    select * from OrderStream where orderAmount > 10000.0
]]></query>
```

For more information, see "Query" on page 16-2.

14.2.3 Built-In Window Query

Example 14–4 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 14–4 Built-In Window Query

```
<query id="BBAQuery"><![CDATA[
    create query q209 as select * from S2 [range 5 minutes] where S2.c1 > 10
]]></query>
```

For more information, see:

- Section 1.1.3, "Stream-to-Relation Operators (Windows)"
- *window_type::=* on page 16-4

14.2.4 MATCH_RECOGNIZE Query

Example 14–5 shows a query that uses the `MATCH_RECOGNIZE` clause to express complex relationships among the stream elements of `ItemTempStream`.

Example 14–5 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 15, "Pattern Recognition With MATCH_RECOGNIZE"
- *pattern_recognition_clause::=* on page 15-1

14.2.5 XMLTable Query

Example 14–6 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.

Example 14–6 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>
```

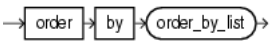
For more information, see:

- Section 1.1.5, "Stream-to-Stream Operators"
- *xmltable_clause::=* on page 16-6
- "SQL/XML (SQLX)" on page 11-16

14.2.6 Sorting Query Results

Use the ORDER BY clause to order the rows selected by a query.

***order_by_clause*::=**



(*order_by_list::=* on page 16-5)

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 CEP sorts values for the ORDER BY clause is specified by your Java locale.

14.3 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.

In Example 14–7, 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 14–7 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.

If you create a join (see Section 14.4, "Joins") 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 14–8 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 14–8 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 always a good idea to qualify stream element references explicitly. Oracle CEP often does less work with fully qualified stream element names.

You can specify any query type in the definition of your view. For more information, see Section 14.2, "Queries".

For complete details on the view statement, see "View" on page 16-18.

14.4 Joins

A **join** is a query that combines rows from two or more streams, views, or relations. Oracle CEP 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 14–9 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 14–9 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 CEP often does less work with fully qualified stream element names.

Oracle CEP supports the following types of joins:

- Inner Joins
- Outer Joins

14.4.1 Inner Joins

By default, Oracle CEP 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 14–10 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 14–10 Inner Joins

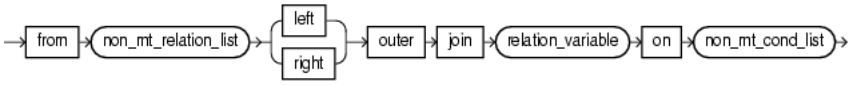
```
<query id="q4"><![CDATA[
    select *
        from
            S0[range 5] as a,
            S1[range 3] as b
        where
            a.c1+a.c2+4.9 = b.c1 + 10
]]></query>
```

14.4.2 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 16-4, *relation_variable::=* on page 16-4, *non_mt_cond_list::=* on page 13-17)

Example 14–11 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 14–11 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 14.4.2.1, "Left Outer Join"

- Section 14.4.2.2, "Right Outer Join"
- Section 14.4.2.3, "Outer Join Look-Back"

14.4.2.1 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 14–12 shows. For all stream elements in A that have no matching stream elements in B, Oracle CEP returns null for any select list expressions containing stream elements of B.

Example 14–12 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>
```

14.4.2.2 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 14–13 shows. For all stream elements in B that have no matching stream elements in A, Oracle CEP returns null for any select list expressions containing stream elements of A.

Example 14–13 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>
```

14.4.2.3 Outer Join Look-Back

You can create an outer join that refers or looks-back to a previous outer join as Example 14–14 shows.

Example 14–14 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>
```

14.5 Oracle CQL Queries and the Oracle CEP Server Cache

You can access an Oracle CEP cache from an Oracle CQL statement or user-defined function.

For more information, see:

- "Configuring Oracle CEP Caching" in the *Oracle CEP IDE Developer's Guide for Eclipse*

- "Accessing a Cache From an Oracle CQL Statement" in the *Oracle CEP IDE Developer's Guide for Eclipse*
- "Accessing a Cache From an Oracle CQL User-Defined Function" in the *Oracle CEP IDE Developer's Guide for Eclipse*

15

Pattern Recognition With MATCH_RECOGNIZE

The Oracle CQL `MATCH_RECOGNIZE` construct and its sub-clauses perform pattern recognition in Oracle CQL queries.

15.1 Understanding Pattern Recognition With MATCH_RECOGNIZE

Pattern recognition functionality is provided in Oracle CQL using the `MATCH_RECOGNIZE` construct.

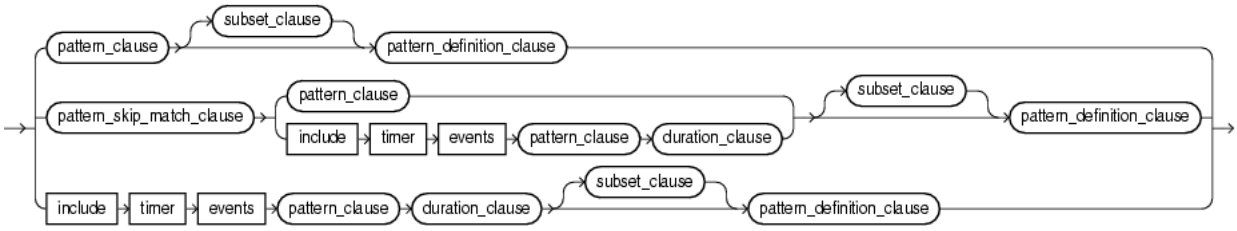
A sequence of consecutive events or tuples in the input stream, each satisfying certain conditions constitutes a pattern. The pattern recognition functionality in Oracle CQL allows you to define conditions on the attributes of incoming events or tuples and to identify these conditions by using `String` names called correlation variables (Section 15.3, "DEFINE Clause"). 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 (see Section 15.7, "PATTERN Clause"). The use of regular expressions lends increased expressibility while specifying the pattern to be recognized. You can also perform computations over the attributes of the tuples that match the pattern specification and use them in the `SELECT` clause of the query of which `MATCH_RECOGNIZE` is a part (see Section 15.5, "MEASURES Clause"). Additional clauses such as `ALL MATCHES`, `PARTITION BY`, and `DURATION` give you more control over the way the pattern recognition is performed over the input stream.

The output of a `MATCH_RECOGNIZE` query is always a stream.

***pattern_recognition_clause*::=**

(*pattern_partition_clause::=* on page 15-10, *pattern_measures_clause::=* on page 15-9, *pattern_def_dur_clause::=* on page 15-2)

***pattern_def_dur_clause*::=**



(*pattern_clause::=* on page 15-11, *pattern_skip_match_clause::=* on page 15-2, *pattern_definition_clause::=* on page 15-5, *duration_clause::=* on page 15-6, *subset_clause::=* on page 15-12)

Example 15–1 shows a typical `MATCH_RECOGNIZE` condition in a query. The query will return the `MEASURES` clause values in its select statement when `DEFINE` clause conditions are satisfied as constrained by the `PATTERN` clause.

Example 15–1 Pattern Matching Conditions

```
<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>
```

The `MATCH_RECOGNIZE` construct provides the following sub-clauses:

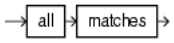
- Section 15.2, "ALL MATCHES Clause"
- Section 15.3, "DEFINE Clause"
- Section 15.4, "DURATION Clause"
- Section 15.5, "MEASURES Clause"
- Section 15.6, "PARTITION BY Clause"
- Section 15.7, "PATTERN Clause"
- Section 15.8, "SUBSET Clause"

For more examples, see Section 15.9, "Examples".

15.2 ALL MATCHES Clause

Use this optional clause to configure Oracle CEP to match overlapping patterns. Omitting the `ALL MATCHES` clause configures Oracle CEP to match only one pattern.

***pattern_skip_match_clause*::=**



Consider the query `tkpattern_q41` in Example 15–2 that uses `ALL MATCHES` and the data stream `tkpattern_S11` in Example 15–3. Stream `tkpattern_S11` has schema `(c1 integer, c2 integer)`. The query returns the stream in Example 15–4.

The query `tkpattern_q41` in Example 15–2 will report a match when the input stream values, when plotted, form the shape of the English letter **W**. The relation in Example 15–4 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 15–2 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 15–3 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 15–4 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 15–4 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.

15.3 DEFINE Clause

Use this clause to define one or more conditions on the tuples of the underlying base stream. You refer to the conditions by using correlation names and variables such as `A`, `B`, and `C` in Example 15–1. You specify the pattern to be recognized as a regular expression over these correlation variables in the `PATTERN` clause of `MATCH_RECOGNIZE` condition(Section 15.7, "PATTERN Clause").

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. It is straight forward to see that 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.

You can refer to the attributes of a base stream directly, such as `c1 < 20`, or using a correlation variable (condition), such as `A.c1 < 20`, while defining the correlation variables.

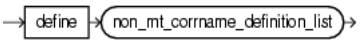
When you refer to the attributes directly, a tuple that last matched any of the correlation variables is consulted for evaluation. Note that the definitions `DEFINE A as c1 < 20` and `DEFINE A as A.c1 < 20` both refer to `c1` in the same tuple which is the latest input tuple. This is because on receiving an input we evaluate the condition of a correlation variable assuming that the latest input matches that correlation variable.

When you refer to the attributes using a correlation variable, such as `A`, the tuple which matched `A` last is consulted for evaluation.

A definition of one correlation variable can refer to another correlation variable, such as `DEFINE B as A.c1 > 20`. Here `A.c1` refers the value of `c1` in the tuple that last matched `A`. Using the `PREV` function, you can refer to the earlier tuples that matched the same correlation variable. For more information, see "prev" on page 5-9.

You can also use functions over the correlation variables while defining them.

***pattern_definition_clause*::=**

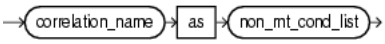


(*non_mt_corrname_definition_list::=* on page 15-5)

***non_mt_corrname_definition_list*::=**

(*correlation_name_definition::=* on page 15-6)

***correlation_name_definition*::=**



(*correlation_name*::= on page 15-11, *non_mt_cond_list*::= on page 13-17)

In Example 15–1, the *pattern_definition_clause* is:

```
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)
```

If you specify a correlation name that is not defined in the DEFINE clause, it is considered to be true for every input. As Example 15–5 shows, because correlation name A is true for every input, it is not defined in the DEFINE clause. It is an error to define a correlation name which is not used in a PATTERN clause.

Example 15–5 Undefined Correlation Name

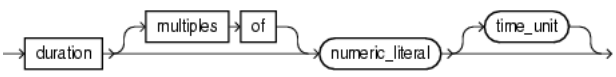
```
<query id="q"><![CDATA[
    SELECT
        T.firstW,
        T.lastZ
    FROM
        S2
    MATCH_RECOGNIZE (
        MEASURES
            A.c1 as firstW,
            last(Z) as lastZ
        PATTERN(A W+ X+ Y+ Z+)
        DEFINE
            W as W.c2 < prev(W.c2),
            X as X.c2 > prev(X.c2),
            Y as Y.c2 < prev(Y.c2),
            Z as Z.c2 > prev(Z.c2)
    ) as T
]]></query>
```

For more information, see Section 15.7, "PATTERN Clause".

15.4 DURATION Clause

The DURATION clause is an optional clause that you should use only when you are 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 13-21)

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.

This section describes:

- Section 15.4.1, "Using the DURATION Clause for Fixed Duration Non-Event Detection"
- Section 15.4.2, "Using the DURATION Clause for Recurring Non-Event Detection"

15.4.1 Using the DURATION Clause for 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 13-21); the default time unit is seconds.

Consider the query `tkpattern_q59` in Example 15–6 that uses `DURATION 10` to specify a delay of 10 s (10000 ms) and the data stream `tkpattern_S19` in Example 15–7. Stream `tkpattern_S19` has schema `(c1 integer)`. The query returns the stream in Example 15–8.

Example 15–6 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 15–7 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 15–8 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 15-18.

15.4.2 Using the DURATION Clause for 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 15–9 that uses `DURATION MULTIPLES OF 10` to specify a delay of 10 s (10000 ms) and the data stream `tkpattern_S23` in Example 15–10. Stream `tkpattern_S23` has schema `(c1 integer)`. The query returns the stream in Example 15–11.

tkpattern.cqlx, /data/inpPattern23.txt, log/patternout75.txt

Example 15–9 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 15–10 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 15–11 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,,
```

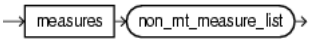
The execution here follows similar logic to that of the example above for just the `DURATION` clause (see "Using the DURATION Clause for Fixed Duration Non-Event Detection" on page 15-7). The difference comes for the later outputs. The tuple at 72000 matches `A` and then there is nothing else after that. 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.

15.5 MEASURES Clause

Use this clause to define expressions over attributes of the tuples in the base stream that match the conditions (correlation variables) in the `DEFINE` clause and to alias these expressions so that they can suitably be used in the `SELECT` clause of the main query of which this `MATCH_RECOGNIZE` condition is a part. You can refer to the attributes of a base stream either directly or via a correlation variable.

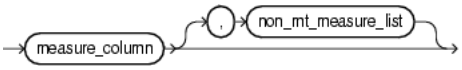
You can use any of the Oracle CQL built-in or user-defined functions (see Section 1.1.9, "Functions").

***pattern_measures_clause*::=**



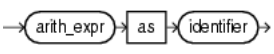
(*non_mt_measure_list*::= on page 15-9)

***non_mt_measure_list*::=**



(*measure_column*::= on page 15-9)

***measure_column*::=**



(*arith_expr*::= on page 11-6, *identifier*::= on page 13-11)

In Example 15–1, the *`pattern_measures_clause`* is:

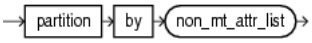
```
MEASURES
    A.itemId as itemId
```

When an attribute of an underlying stream or view that evaluates to a stream is referred to using a correlation variable, such as `A.c1`, the value of `c1` is the value in the tuple that last matched the condition corresponding to correlation variable `A`. In case the definition of `A` is not provided in the `DEFINE` clause, it is considered as `TRUE` always. So effectively all the tuples in the input match to `A`. Therefore the value of `A.c1` is the value of `c1` in the last processed tuple.

15.6 PARTITION BY Clause

Use this optional clause to specify the stream attributes by which a `MATCH_RECOGNIZE` clause should partition its results.

***pattern_partition_clause*::=**



(*non_mt_attr_list*::= on page 13-14)

In Example 15–1, 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 15–12 shows.

Example 15–12 Input Stream S1

```
     c1  c2
1000 10, 1
2000 10, 2
3000 20, 2
4000 20, 1
```

Consider the `MATCH_RECOGNIZE` query that Example 15–13 shows.

Example 15–13 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 15–13 shows, then the output would change to:

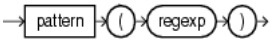
```
3000:+ 2000, 3000, 2
4000:+ 1000, 4000, 1
```

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.

15.7 PATTERN Clause

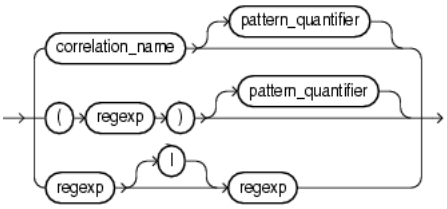
Use this clause to specify the pattern to be matched as a regular expression over correlation variables defined in the DEFINE clause (see Section 15.3, "DEFINE Clause"). However such a regular expression can contain some correlation variables that are not defined in the DEFINE clause and they are considered as always TRUE meaning they match every input.

***pattern_clause*::=**



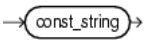
(*regexp::=* on page 15-11)

***regexp*::=**



(*correlation_name::=* on page 15-11, *pattern_quantifier::=* on page 15-11)

***correlation_name*::=**



(*const_string::=* on page 13-7)

***pattern_quantifier*::=**

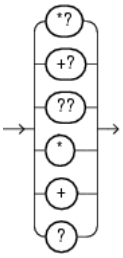


Table 15–1 lists the pattern quantifiers (*pattern_quantifier::=* on page 15-11) Oracle CQL supports. Use the pattern quantifiers to specify the behavior of pattern matches. The one-character pattern quantifiers are maximal or "greedy"; they will attempt to match as many instances of the regular expression 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.

Table 15–1 MATCH_RECOGNIZE Pattern Quantifiers

Maximal	Minimal	Description
*	*?	0 or more times
+	+?	1 or more times.
?	??	0 or 1 time.

An unquantified pattern (such as A) is assumed to have a quantifier that requires exactly one match.

In Example 15–1, the *pattern_clause* is:

```
PATTERN (A B* C)
```

This pattern clause means a pattern match will be recognized and reported when the following conditions are met by consecutive incoming input tuples:

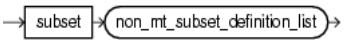
1. A tuple matches the condition that defines correlation variable A followed by
2. Zero or more tuples that match the correlation variable B followed by
3. A tuple that matches correlation variable C.

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.

15.8 SUBSET Clause

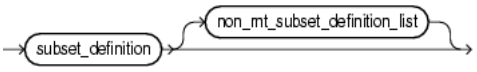
Using this clause, you can group together one or more correlation variables that are defined in the DEFINE clause. You can use this named subset in the MEASURES (see Section 15.5, "MEASURES Clause") and DEFINE (Section 15.3, "DEFINE Clause") clauses just like any other correlation variable.

***subset_clause*::=**



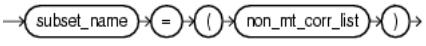
(*non_mt_subset_definition_list*::= on page 15-12)

***non_mt_subset_definition_list*::=**



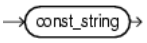
(*subset_definition*::= on page 15-12)

***subset_definition*::=**



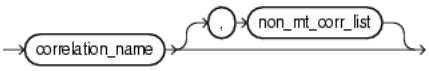
(*subset_name::=* on page 15-13, *non_mt_corr_list::=* on page 15-13)

***subset_name*::=**



(*const_string::=* on page 13-7)

***non_mt_corr_list*::=**



(*correlation_name::=* on page 15-11)

Consider the query `q55` in Example 15–14 and the data stream `S11` in Example 15–15. 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`. 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 15–16.

Example 15–14 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 15–15 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 15–16 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 15.7, "PATTERN Clause"
- Section 15.5, "MEASURES Clause"
- Section 15.3, "DEFINE Clause"

15.9 Examples

The following examples illustrate basic `MATCH_RECOGNIZE` practices:

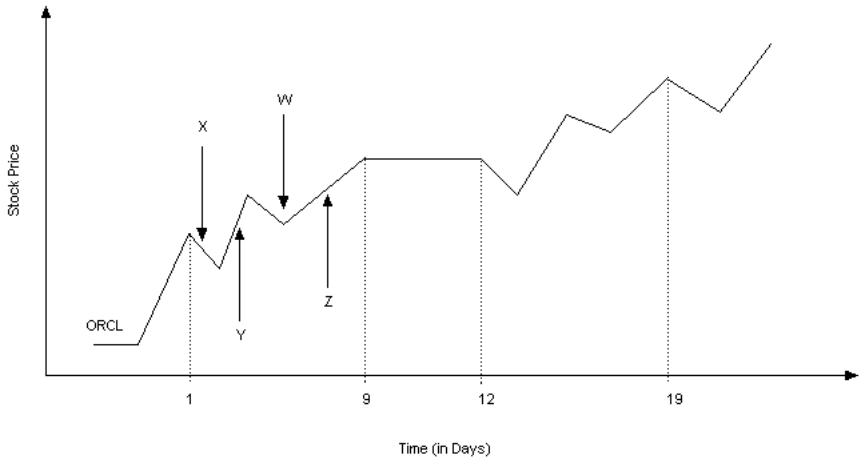
- "Pattern Detection" on page 15-15
- "Pattern Detection With Partition By" on page 15-16
- "Pattern Detection With Aggregates" on page 15-18
- "Fixed Duration Non-Event Detection" on page 15-18

For more examples, see *Oracle CEP Getting Started*.

15.9.1 Pattern Detection

Consider the stock fluctuations that Figure 15–1 shows. This data can be represented as a stream of index number (or time) and stock price. Figure 15–1 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 15–1 Pattern Detection: Double Bottom Stock Fluctuations



Example 15–17 shows a query `q` on stream `S2` with schema `c1` index number (or time) and `c2` stock 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.stockprice < Prev(W.stockprice). 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.

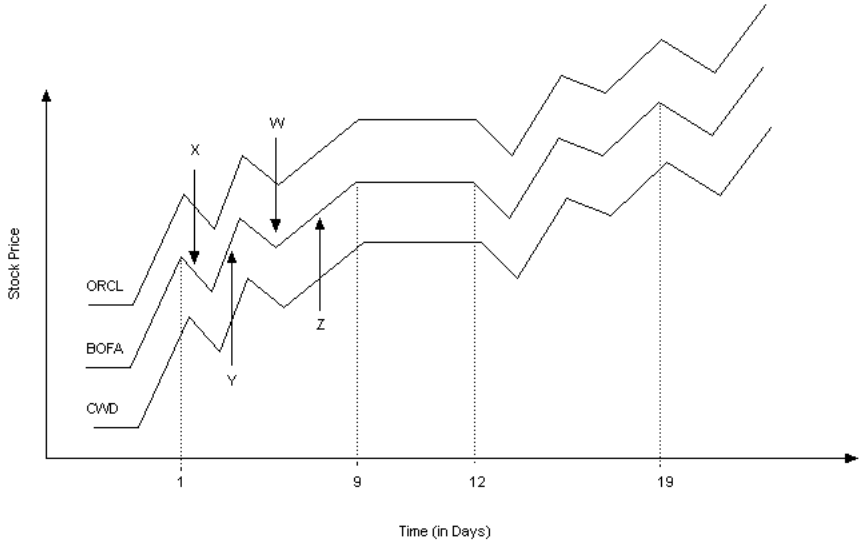
Example 15–17 Simple Pattern Detection: Query

```
<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>
```

15.9.2 Pattern Detection With Partition By

Consider the stock fluctuations that Figure 15–2 shows. This data can be represented as a stream of index number (or time) and stock price. In this case, the stream contains data for more than one stock ticker symbol. Figure 15–2 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 15–2 Pattern Detection With Partition By: Stock Fluctuations



Example 15–18 shows a query `q` on stream `S2` with schema `c1` index number (or time) and `c2` stock 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.stockprice < Prev(W.stockprice)`, 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 15–18 Pattern Detection With Partition By: Query

```
<query id="q"><![CDATA[
    SELECT
        T.firstW,
        T.lastZ
    FROM
        S2
    MATCH_RECOGNIZE (
        PARTITION BY
            A.ticker
        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>
```

15.9.3 Pattern Detection With Aggregates

Consider the query `q1` in Example 15–19 and the data stream `S` in Example 15–20. Stream `S` has schema `(c1 integer)`. The query returns the stream in Example 15–21.

Example 15–19 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 15–20 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 15–21 Pattern Detection With Aggregates: Stream Output

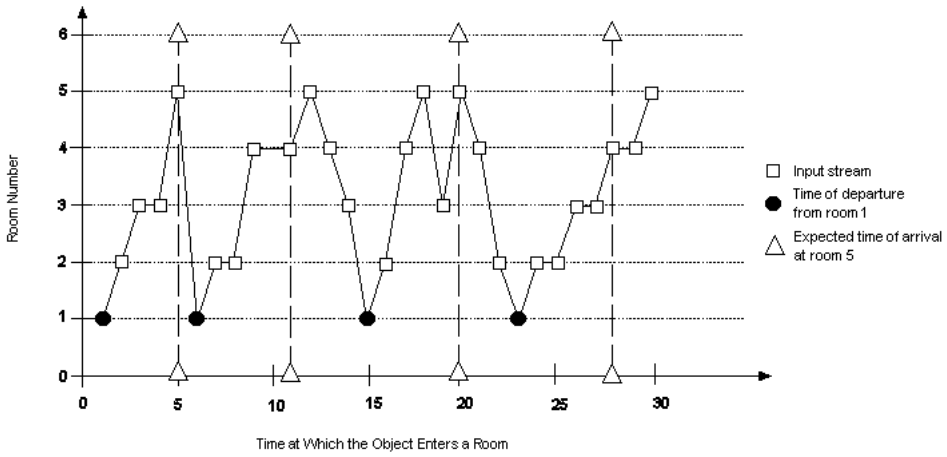
```
Timestamp   Tuple
8000        338
12000       52
```

15.9.4 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 15–3 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 devents: 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 15–3 Fixed Duration Non-Event Detection



Example 15–22 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 15–22 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 15.4, "DURATION Clause".

16

Oracle CQL Statements

This chapter describes the various Oracle CQL data definition language (DDL) and data modification language (DML) statements that Oracle CEP supports.

16.1 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 CEP that no further input will have a timestamp lesser than the heartbeat value.

For more information, see:

- Section 1.2.1, "Oracle CQL Statement Lexical Conventions"
- Section 1.2.2, "Oracle CQL Statement Documentation Conventions"
- Chapter 2, "Basic Elements of Oracle CQL"
- Chapter 13, "Common Oracle CQL DDL Clauses"
- Chapter 14, "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 CEP server throws an exception.

For more information, see:

- "View" on page 16-18
- Chapter 14, "Oracle CQL Queries, Views, and Joins"

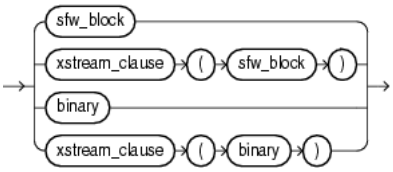
Syntax

You express a query in a `<query></query>` element as Example 16–1 shows. Specify the `identifier` as the `query` element `id` attribute.

Example 16–1 Query in a <query></query> Element

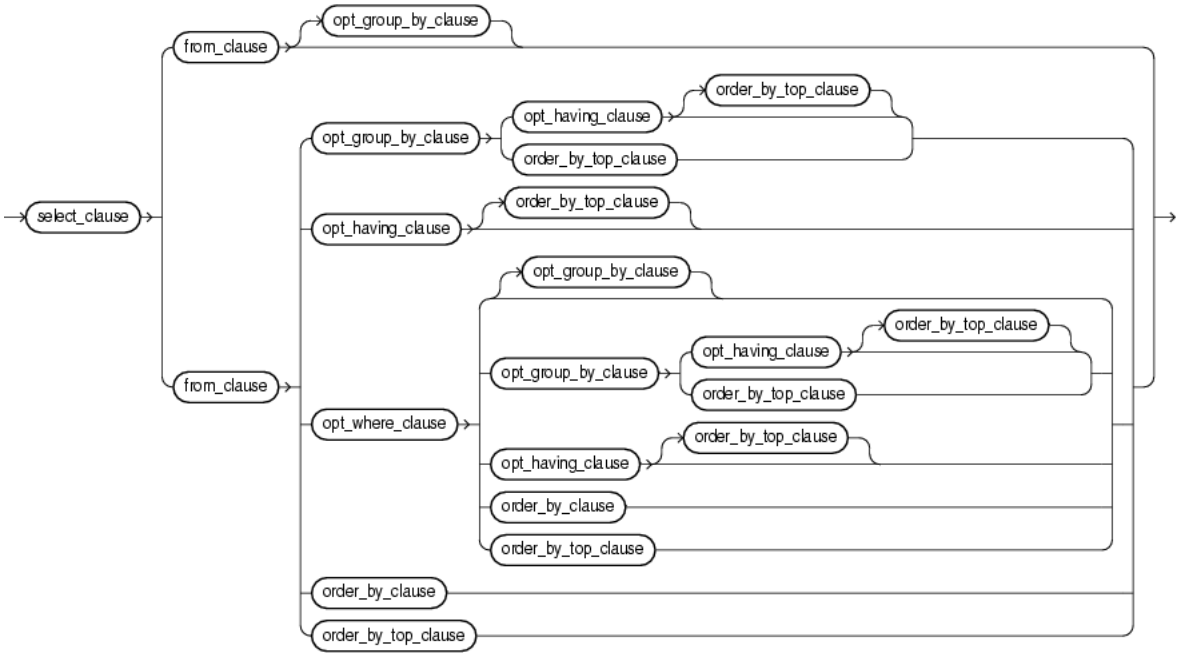
```
<query id="q0"><![CDATA[
    select * from OrderStream where orderAmount > 10000.0
]]></query>
```

query::=



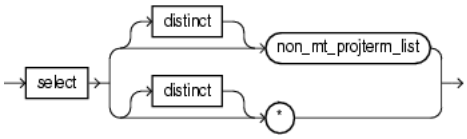
(*sfw_block*::= on page 16-3, *xstream_clause*::= on page 16-6, *binary*::= on page 16-5)

sfw_block::=



(*select_clause::=* on page 16-3, *from_clause::=* on page 16-3, *opt_where_clause::=* on page 16-4, *opt_group_by_clause::=* on page 16-4, *order_by_clause::=* on page 16-4, *order_by_top_clause::=* on page 16-5, *opt_having_clause::=* on page 16-5)

select_clause::=

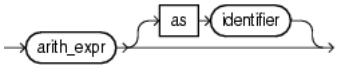


(*non_mt_projterm_list::=* on page 16-3)

non_mt_projterm_list::=

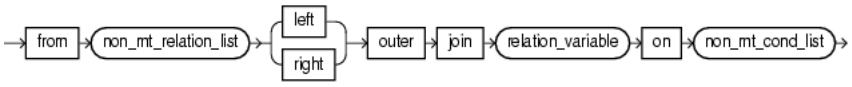
(*projterm::=* on page 16-3)

projterm::=



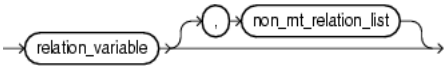
(*identifier::=* on page 13-11)

from_clause::=



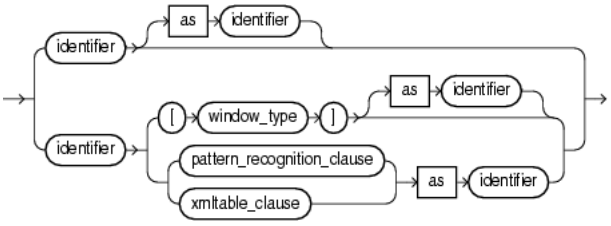
(*non_mt_relation_list::=* on page 16-4, *relation_variable::=* on page 16-4, *non_mt_cond_list::=* on page 13-17)

non_mt_relation_list::=



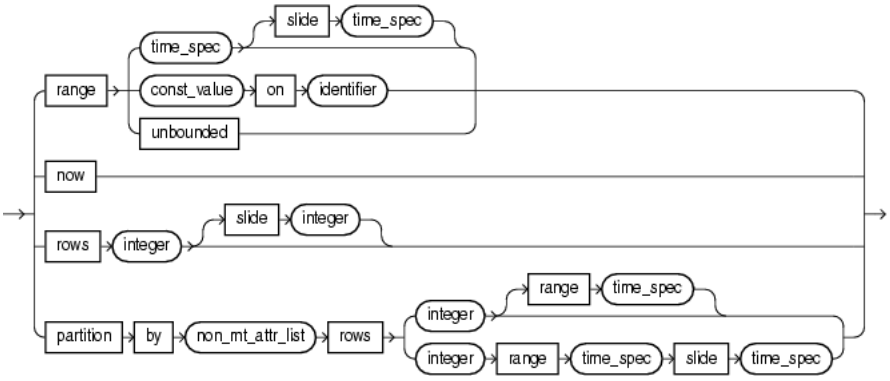
(*relation_variable::=* on page 16-4)

relation_variable::=



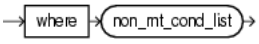
(*identifier::=* on page 13-11, *window_type::=* on page 16-4, *pattern_recognition_clause::=* on page 15-1, *xmltable_clause::=* on page 16-6)

window_type::=



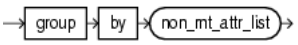
(*identifier::=* on page 13-11, *non_mt_attr_list::=* on page 13-14, *time_spec::=* on page 13-21)

opt_where_clause::=



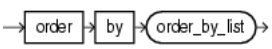
(*non_mt_cond_list::=* on page 13-17)

opt_group_by_clause::=



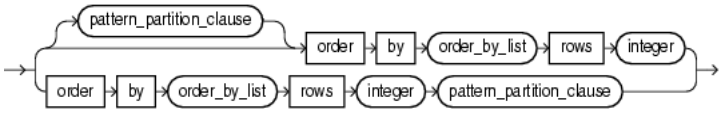
(*non_mt_attr_list::=* on page 13-14)

order_by_clause::=



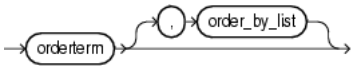
(*order_by_list::=* on page 16-5)

order_by_top_clause::=



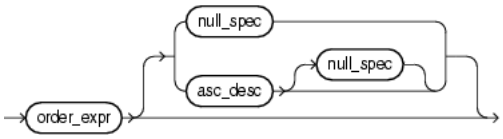
(*pattern_partition_clause::=* on page 15-10, *order_by_list::=* on page 16-5)

order_by_list::=



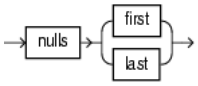
(*orderterm::=* on page 16-5)

orderterm::=

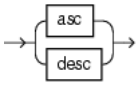


(*order_expr::=* on page 11-19, *null_spec::=* on page 16-5, *asc_desc::=* on page 16-5)

null_spec::=



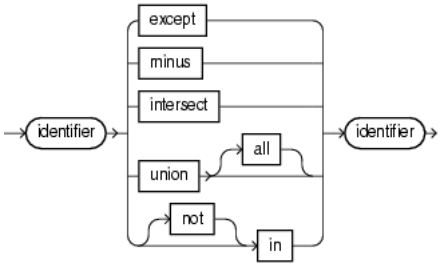
asc_desc::=



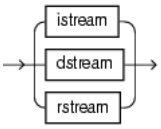
opt_having_clause::=

(*non_mt_cond_list::=* on page 13-17)

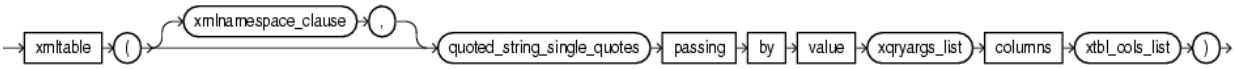
binary::=



xstream_clause::=

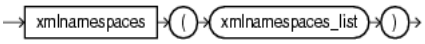


xmltable_clause::=



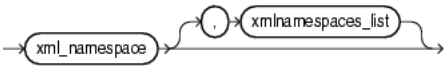
(*xmlnamespace_clause::=* on page 16-6, *const_string::=* on page 13-7, *xqryargs_list::=* on page 13-25, *xtbl_cols_list::=* on page 16-6)

xmlnamespace_clause::=



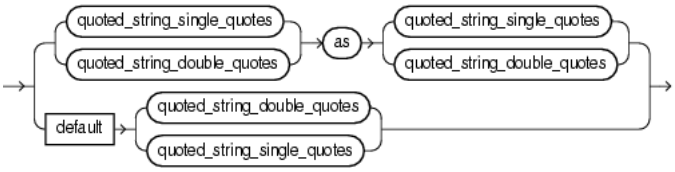
(*xmlnamespaces_list::=* on page 16-6)

xmlnamespaces_list::=



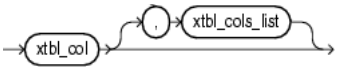
(*xml_namespace::=* on page 16-6)

xml_namespace::=



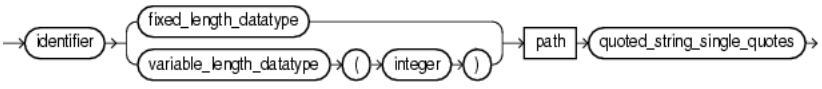
(*const_string::=* on page 13-7)

xtbl_cols_list::=



(*xtbl_col::=* on page 16-6)

xtbl_col::=



Semantics

named_query

Specify the Oracle CQL query statement itself (see "query" on page 16-7).

If you plan to configure the query with `USE UPDATE SEMANTICS`, you must declare one or more stream elements as a primary key (*out_of_line_constraint::=* on page 13-19).

For syntax, see "Query" on page 16-2.

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 16-7).
- *`binary`*: an optional set operation clause (see "binary" on page 16-11).
- *`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 "xstream_clause" on page 16-11).

For syntax, see *query::=* on page 16-2.

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 16-7).
- *`from_clause`*: the stream or view from which your query selects (see "from_clause" on page 16-8).
- *`opt_where_clause`*: optional conditions your query applies to its selection (see "opt_where_clause" on page 16-9)
- *`opt_group_by_clause`*: optional grouping conditions your query applies to its results (see "opt_group_by_clause" on page 16-9)
- *`order_by_clause`*: optional ordering conditions your query applies to its results (see "order_by_clause" on page 16-9)
- *`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 16-10)
- *`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 16-11)

For syntax, see *sfw_block::=* on page 16-3 (parent: *query::=* on page 16-2).

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 16-8).

Optionally, specify `distinct` if you want Oracle CEP 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 16-15.

For syntax, see *select_clause::=* on page 16-3 (parent: *sfw_block::=* on page 16-3).

non_mt_projterm_list

Specify the projection term ("projterm" on page 16-8) 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 16-3 (parent: *select_clause::=* on page 16-3).

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 16-8) 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 16-3 (parent: *non_mt_projterm_list::=* on page 16-3).

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 16-8).

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 16-9
- "Joins" on page 14-12

For syntax, see *from_clause::=* on page 16-3 (parent: *sfw_block::=* on page 16-3).

non_mt_relation_list

Specify the stream or view ("relation_variable" on page 16-8) 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 16-4 (parent: *from_clause::=* on page 16-3).

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 16-9). 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 16-11). 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 16-11). Optionally, use the `AS` keyword to specify an alias for the stream or view instead of using its name as is.

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:

- "View" on page 16-18

For syntax, see *relation_variable::=* on page 16-4 (parent: *non_mt_relation_list::=* on page 16-4).

window_type

Specify a built-in stream-to-relation operator.

For more information, see Section 1.1.3, "Stream-to-Relation Operators (Windows)".

For syntax, see *window_type::=* on page 16-4 (parent: *relation_variable::=* on page 16-4).

time_spec

Specify the time over which a range or partitioned range sliding window should slide.

Default: if units are not specified, Oracle CEP assumes `[second|seconds]`.

For more information, see "Range-Based Stream-to-Relation Window Operators" on page 4-5 and "Partitioned Stream-to-Relation Window Operators" on page 4-18.

For syntax, see *time_spec::=* on page 13-21 (parent: *window_type::=* on page 16-4).

opt_where_clause

Specify the (optional) where clause of the Oracle CQL query statement.

For syntax, see *opt_where_clause::=* on page 16-4 (parent: *sfw_block::=* on page 16-3).

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 CEP 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.

For syntax, see *opt_group_by_clause::=* on page 16-4 (parent: *sfw_block::=* on page 16-3).

order_by_clause

Specify the `ORDER BY` clause of the Oracle CQL query statement as a comma-delimited list ("order_by_list" on page 16-10) of one or more order terms (see "orderterm" on page 16-10). 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.

For more information, see Section 14.2.6, "Sorting Query Results".

For syntax, see *order_by_clause::=* on page 16-4 (parent: *sfw_block::=* on page 16-3).

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 16-10) of one or more order terms (see "orderterm" on page 16-10) 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 always 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
```

For more information, see

- "ORDER BY ROWS Query Example" on page 16-16
- Section 14.2.6, "Sorting Query Results"

For syntax, see *order_by_top_clause::=* on page 16-5 (parent: *sfw_block::=* on page 16-3).

order_by_list

Specify a comma-delimited list of one ore more order terms (see "orderterm" on page 16-10) in an (optional) `ORDER BY` clause.

For syntax, see *order_by_list::=* on page 16-5 (parent: *order_by_clause::=* on page 16-4).

orderterm

A stream element (*attr::=* on page 13-2) 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 16-10).

order_expr (*order_expr::=* on page 11-19) can be an `attr` or `constant_int`. The `attr` (*attr::=* on page 13-2) can be any stream element or pseudo column.

For syntax, see *orderterm::=* on page 16-5 (parent: *order_by_list::=* on page 16-5).

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 16-10).

For syntax, see *null_spec::=* on page 16-5 (parent: *orderterm::=* on page 16-5).

asc_desc

Specify whether an order term is ordered in ascending (`ASC`) or descending (`DESC`) order.

For syntax, see *asc_desc::=* on page 16-5 (parent: *orderterm::=* on page 16-5).

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 CEP 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.

For an example, see "HAVING Example" on page 16-12.

For syntax, see *opt_having_clause::=* on page 16-5 (parent: *sfw_block::=* on page 16-3).

binary

Use the *`binary`* clause to perform set operations on the tuples that two streams or views return.

For examples, see:

- "BINARY Example: UNION and UNION ALL" on page 16-13
- "BINARY Example: INTERSECT" on page 16-14
- "BINARY Example: MINUS" on page 16-14

For syntax, see *binary::=* on page 16-5 (parent: *query::=* on page 16-2).

xstream_clause

Use an *`xstream_clause`* to specify a relation-to-stream operator that applies to the query.

For more information, see Section 1.1.4, "Relation-to-Stream Operators".

For syntax, see *xstream_clause::=* on page 16-6 (parent: *query::=* on page 16-2).

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 16-6) of one or more XML table columns (see *xtbl_col::=* on page 16-6), with or without an XML namespace.

For examples, see:

- "XMLTABLE Query Example" on page 16-15
- "XMLTABLE With XML Namespaces Query Example" on page 16-16

For syntax, see *xmltable_clause::=* on page 16-6 (parent: *relation_variable::=* on page 16-4).

pattern_recognition_clause

Use a *`pattern_recognition_clause`* to perform advanced comparisons optimized for data streams.

For more information and examples, see Chapter 15, "Pattern Recognition With MATCH_RECOGNIZE".

For syntax, see *pattern_recognition_clause::=* on page 15-1 (parent: *relation_variable::=* on page 16-4).

Examples

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

For more examples, see Chapter 14, "Oracle CQL Queries, Views, and Joins".

Simple Query Example

Example 16–2 shows how to register a simple query `q0` that selects all (`*`) tuples from stream `OrderStream` where stream element `orderAmount` is greater than 10000.

Example 16–2 REGISTER QUERY

```
<query id="q0"><![CDATA[
    select * from OrderStream where orderAmount > 10000.0
]]></query>
```

HAVING Example

Consider the query `q4` in Example 16–3 and the data stream `S2` in Example 16–4. Stream `S2` has schema `(c1 integer, c2 integer)`. The query returns the relation in Example 16–5.

Example 16–3 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 16–4 HAVING Stream Input

```
Timestamp   Tuple
1000        ,2
2000        ,4
3000        1,4
5000        1,
6000        1,6
7000        ,9
8000        ,
```

Example 16–5 HAVING Relation Output

```
Timestamp   Tuple Kind  Tuple
5000:       +           1,2
6000:       -           1,2
6000:       +           1,3
```

BINARY Example: UNION and UNION ALL

Given the relations R1 and R2 in Example 16–7 and Example 16–8, respectively, the UNION query q1 in Example 16–6 returns the relation in Example 16–9 and the UNION ALL query q2 in Example 16–6 returns the relation in Example 16–10.

Example 16–6 Set Operators: UNION Query

```
<query id="q1"><![CDATA[
    R1 UNION R2
]]></query>
<query id="q2"><![CDATA[
    R1 UNION ALL R2
]]></query>
```

Example 16–7 Set Operators: 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 16–8 Set Operators: 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 16–9 Set Operators: 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 16–10 Set Operators: UNION ALL Relation Output

```
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
```

BINARY Example: INTERSECT

Given the relations `R1` and `R2` in Example 16–12 and Example 16–13, respectively, the `INTERSECT` query `q1` in Example 16–11 returns the relation in Example 16–14.

Example 16–11 Set Operators: INTERSECT Query

```
<query id="q1"><![CDATA[
    R1 INTERSECT R2
]]></query>
```

Example 16–12 Set Operators: INTERSECT Relation Input R1

```
Timestamp   Tuple Kind  Tuple
1000:        +          10,30
1000:        +          10,40
2000:        +          11,20
3000:        -          10,30
3000:        -          10,40
```

Example 16–13 Set Operators: INTERSECT Relation Input R2

```
Timestamp   Tuple Kind  Tuple
1000:       +           10,40
2000:       +           10,30
2000:       -           10,40
3000:       -           10,30
```

Example 16–14 Set Operators: INTERSECT Relation Output

```
Timestamp   Tuple Kind  Tuple
1000:       +           10,30
1000:       +           10,40
1000:       -           10,30
1000:       -           10,40
1000:       +           10,40
2000:       +           11,20
2000:       -           11,20
2000:       +           10,30
2000:       -           10,40
3000:       -           10,30
```

BINARY Example: MINUS

Given the relations `R1` and `R2` in Example 16–16 and Example 16–17, respectively, the `MINUS` query `q1` in Example 16–15 returns the relation in Example 16–18.

Example 16–15 Set Operators: MINUS Query

```
<query id="q1BBAQuery"><![CDATA[
    R1 MINUS R2
]]></query>
```

Example 16–16 Set Operators: MINUS Relation Input R1

```
Timestamp   Tuple Kind  Tuple
1500:       +           10,40
2000:       +           10,30
2000:       -           10,40
3000:       -           10,30
```

Example 16–17 Set Operators: MINUS Relation Input R2

```
Timestamp   Tuple Kind  Tuple
1000:       +           11,20
2000:       +           10,40
```

```
3000:       -           10,30
```

Example 16–18 Set Operators: MINUS Relation Output

```
Timestamp   Tuple Kind  Tuple
1000:       +           10,40.0
2000:       -           10,40.0
```

Select and Distinct Examples

Consider the query `q1` in Example 16–19. Given the data stream `S` in Example 16–20, the query returns the relation in Example 16–21.

Example 16–19 Select DISTINCT Query

```
<query id="q1"><![CDATA[
    SELECT DISTINCT FROM S WHERE c1 > 10
]]></query>
```

Example 16–20 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 16–21 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 16–22 and the data stream `S` in Example 16–23. Stream `S` has schema `(c1 xmltype)`. The query returns the relation in Example 16–24. For more information, see Section 14.2.5, "XMLTable Query".

Example 16–22 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 16–23 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 16–24 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 16–25 and the data stream `S1` in Example 16–26. Stream `S1` has schema `(c1 xmltype)`. The query returns the relation in Example 16–27. For more information, see Section 14.2.5, "XMLTable Query".

Example 16–25 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 16–26 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 16–27 XMLTABLE With XML Namespaces Relation Output

```
Timestamp   Tuple Kind  Tuple
3000:       +     John,1
3000:       +     Jack,2
3000:       +     Jill,3
```

ORDER BY ROWS Query Example

Consider the query `q1` in Example 16–28. Given the data stream `S0` in Example 16–29, the query returns the relation in Example 16–30.

Example 16–28 ORDER BY ROWS Query

```
<query id="q1"><![CDATA[
    create query q1 as select c1 ,c2 from S0 order by c1,c2 rows 5
]]></query>
```

Example 16–29 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 16–30 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
```

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 16-2
- Chapter 14, "Oracle CQL Queries, Views, and Joins".

Syntax

You express the a view in a `<view></view>` element as Example 16–31 shows. Specify the `identifier` as the `view` element `id` attribute. Optionally, specify the schema as the `view` element `schema` attribute.

Example 16–31 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 16-2.

Examples

The following examples illustrate the various semantics that this statement supports. For more examples, see Chapter 14, "Oracle CQL Queries, Views, and Joins".

Registering a View Example

Example 16–32 shows how to register view `v2`.

Example 16–32 REGISTER VIEW

```
<view id="v2" schema="cusip bid ask"><![CDATA[
    IStream(select * from S1[range 10 slide 10])
]]></view>
```

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y