Oracle® Fusion Middleware

Oracle CQL Language Reference

12*c* Release (12.2.1.3.0)
E98671-01
August 2018

Oracle Fusion Middleware Oracle CQL Language Reference, 12*c* Release (12.2.1.3.0)

E98671-01

Primary Author: Oracle® Corporation

Contents

11 Colt Aggregate Functions

Preface

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

Audience

This document is intended for all users of Oracle CQL.

Documentation Accessibility

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

Access to Oracle Support

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

Related Documents

For more information, see the following:

- Known Issues for Oracle SOA and BPM Products at: http://www.oracle.com/technetwork/middleware/soasuite/documentation/soaknownissues122120-3111966.html.
- *Oracle Fusion Middleware Administering Oracle Stream Analytics*
- *Oracle Fusion Middleware Developing Applications for Event Processing with Oracle Stream Analytics*
- *Oracle Fusion Middleware Getting Started with Event Processing for Oracle Stream Analytics*
- *Oracle Fusion Middleware Schema Reference for Oracle Stream Analytics*
- *Oracle Fusion Middleware Using Visualizer for Oracle Stream Analytics*
- *Oracle Fusion Middleware Customizing Event Processing for Oracle Stream Analytics*

- *Oracle Fusion Middleware Developing Applications with Oracle CQL Data Cartridges*
- *Oracle Fusion Middleware Java API Reference for Oracle Stream Analytics*
- *Oracle Fusion Middleware Using Oracle Stream Analytics*
- *Oracle Fusion Middleware Getting Started with Oracle Stream Analytics*
- SQL99 Specifications (ISO/IEC 9075-1:1999, ISO/IEC 9075-2:1999, ISO/IEC 9075-3:1999, and ISO/IEC 9075-4:1999)
- Oracle Event Processing Forum: https://community.oracle.com/community/fusion_middleware/soa_and_process_management/complex_event_processing.

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).

What's New in This Guide

The product has been renamed from Oracle Stream Explorer to Oracle Stream Analytics in the 12*c* (12.2.1.0.0) release.

Screens shown in this guide may differ from your implementation, depending on the skin used. Any differences are cosmetic.

1

Introduction to Oracle CQL

Oracle Continuous Query Language (Oracle CQL), a query language based on SQL with added constructs that support streaming data is introduced. Using Oracle CQL, you can express queries on data streams with Oracle Stream Analytics.

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 Event Processing, a data management infrastructure that supports the notion of streams of structured data records together with stored relations.

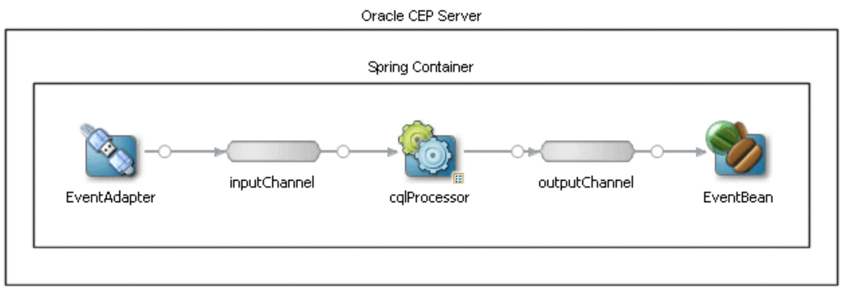
To provide a uniform declarative framework, Oracle offers Oracle Continuous Query Language (Oracle CQL), a query language based on SQL with added constructs that support streaming data.

Oracle CQL is designed to be:

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

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

Figure 1-1 Oracle Event Processing Architecture



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

Using Oracle JDeveloper and Oracle Event Processing Visualizer, you:

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

Consider the typical Oracle CQL statements in the following example.

```
<?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:

- Streams and Relations
- Relation-to-Relation Operators
- Stream-to-Relation Operators (Windows)
- Relation-to-Stream Operators
- Stream-to-Stream Operators
- Queries, Views, and Joins
- Pattern Recognition
- Event Sources and Event Sinks
- Functions
- Time
- Oracle CQL Statements
- Lexical Conventions
- Documentation Conventions

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

1.1.1 Streams and Relations

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

- Streams
- Relations

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

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

- Stream-to-Stream Operators: to produce a stream from one or more other streams.

1.1.1.1 Streams

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

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

Stream elements are tuple-timestamp pairs, which can be represented as a sequence of timestamped tuple insertions. In other words, a stream is a sequence of timestamped tuples. There could be more than one tuple with the same timestamp. The tuples of an input stream are required to arrive at the system in the order of increasing timestamps. For more information, see 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>`). The following example shows how a stock ticker data stream might appear, where each stream element is made up of `<timestamp value>`, `<stock symbol>`, and `<stock price>`:

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

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

By definition, a stream is unbounded.

This section describes:

- Streams and Channels
- Channel Schema
- Querying a Channel
- Controlling Which Queries Output to a Downstream Channel.

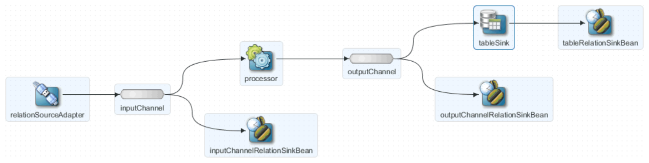
For more information, see:

- Event Sources and Event Sinks
- Introduction to Oracle CQL Queries, Subqueries, Views, and Joins

1.1.1.1.1 Streams and Channels

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

Figure 1-2 Stream in the Event Processing Network



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

1.1.1.1.2 Channel Schema

The event source you connect to a stream determines the stream's schema. The `PriceAdapter` adapter determines the `priceStream` stream's schema. The following example shows the `PriceAdapter` Event Processing Network (EPN) assembly file: the `wlevs:event-type` element defines event type `PriceEvent`. The `wlevs:property` element defines the property names and types for each property in this event type.

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

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

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

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

...
```

1.1.1.1.3 Querying a Channel

Once the event source, channel, and processor are connected, you can write Oracle CQL statements that make use of the stream. The following example shows the component configuration file that defines the Oracle CQL statements for the `filterFanoutProcessr`.

```
<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>
```

1.1.1.1.4 Controlling Which Queries Output to a Downstream Channel

If you specify more than one query for a processor, then all query results are output to the processor's out-bound channel (`filteredStream`).

Optionally, in the component configuration file, you can use the `channel` element `selector` attribute to control which query's results are output. In this example, query results for query `Yr3Sector` and `Yr2Sector` are output to filteredStream but not query results for query `Yr1Sector`.

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

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

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

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

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

A base relation is an input relation.

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

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

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

For more information, see Time.

1.1.1.3 Relations and Oracle Event Processing Tuple Kind Indicator

By default, Oracle Event Processing includes time stamp and an Oracle Event Processing tuple kind indicator in the relations it generates.

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

The Oracle Event Processing tuple kind indicators are:

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

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 the following example, 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.

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

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

For more information on `distinct`, see:

- Built-In Aggregate Functions
- Select Clause.

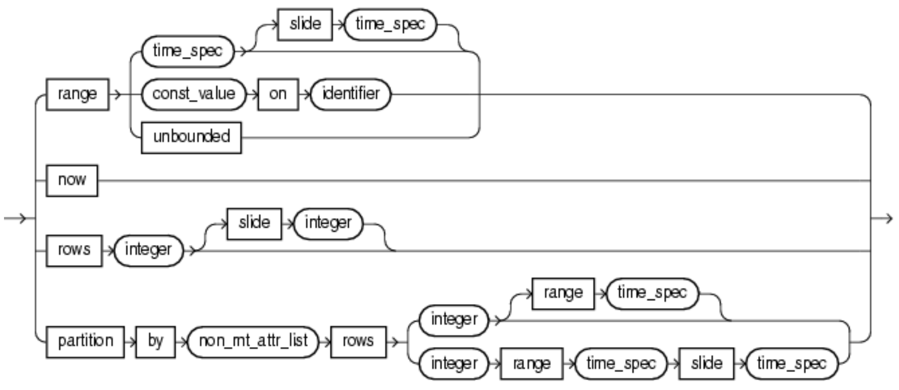
1.1.3 Stream-to-Relation Operators (Windows)

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`.

Queries that have the same source (stream) and window specifications are optimized by the system to share common memory space. When a new query is added with these parameters, it automatically receives the content (events) of this shared window. This optimization can cause the query to output initial events even though it might not have received newly added events.

***window_type*::=**

Figure 1-3 window_type



Oracle CQL supports the following built-in window types:

- Range: time-based

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

  `S[Range T1 Slide T2]`
- Range: time-based unbounded

  `S[Range Unbounded]`
- Range: time-based now

  `S[Now]`
- Range: constant value

  `S[Range C on ID]`
- Tuple-based:

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

  `S[Rows N1 Slide N2]`
- Partitioned:

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

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

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

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

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

For more information, see:

- Range-Based Stream-to-Relation Window Operators
- Tuple-Based Stream-to-Relation Window Operators
- Partitioned Stream-to-Relation Window Operators.

1.1.3.1 Range, Rows, and Slide

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

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

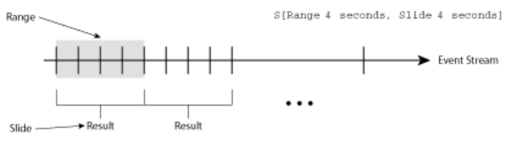
The Slide keyword specifies how frequently you want to see output from the query, while the Range keyword specifies the time range from which to query events. Using Range and Slide together results in a set of events from which to query, with that set changing based on where the query window slides to.

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

Another way to express this is the following formula: `timeInterval = actualTime / slideSpecification if((actualTime % slideSpecification) == 0) // No remainder setTime = timeInterval * slideSpecification else setTime = (timeInterval + 1) * slideSpecification`.

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

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



In Figure 1-4, the `Range` specification indicates "I want to look at 8 seconds worth of data" and the `Slide` specification indicates "I want a result every 4 seconds". In this case, the query returns a result twice during each `Range` specification (except for certain conditions, as Range, Rows, and Slide at Query Start-Up and for Empty Relations describes)

Figure 1-5 Range and Slide: Different (Steady-State Condition)

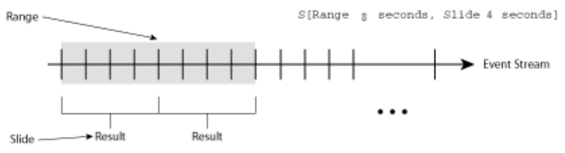


Table 1-1 lists the default `Range`, `Range` unit, and `Slide` (where applicable) for range-based and tuple-based stream-to-relation window operators:

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

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

1.1.3.1.1 Range, Rows, and Slide at Query Start-Up and for Empty Relations

Table 1-2 lists the behavior of `Range`, `Rows`, and `Slide` for special cases such as query start-up time and for an empty relation.

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

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

For more information and detailed examples, see:

- Range-Based Stream-to-Relation Window Operators
- Tuple-Based Stream-to-Relation Window Operators
- Partitioned Stream-to-Relation Window Operators
- Functions
- Using count With *, *identifier*.*, and *identifier*.*attr*.

1.1.3.2 Partition

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

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

1.1.3.3 Default Stream-to-Relation Operator

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

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

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

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

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 the following example, 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.

```
<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.
- `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.
- `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.

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

1.1.4.1 Default Relation-to-Stream Operator

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

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

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

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

1.1.5 Stream-to-Stream Operators

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

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

However, some relation-relation operators (like filter and project) can also act as stream-stream operators. Consider the query 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.

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

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

For more information, see:

- Default Stream-to-Relation Operator
- Default Relation-to-Stream Operator

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

- `MATCH_RECOGNIZE`: use this clause to write various types of pattern recognition queries on the input stream. For more information, see Pattern Recognition.
- `XMLTABLE`: use this clause to parse data from the `xmltype` stream elements using XPath expressions. For more information, see 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 Event Processing CQL processor to retrieve data from one or more streams, relations, or views. A top-level `SELECT` statement that you create in a `<query>` element is called a **query**. For more information, see 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 Views.

Each query and view must have an identifier unique to the processor that contains it. The following example shows a query with an `id` of `q0`. The `id` value must conform with the specification given.

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

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

For more information, see 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 Pattern Recognition With MATCH_RECOGNIZE.

1.1.8 Event Sources and Event Sinks

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

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

1.1.8.1 Event Sources

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

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

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

Note:

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

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

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

For more information, see:

- Table Event Sources
- Cache Event Sources.

1.1.8.2 Event Sinks

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

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

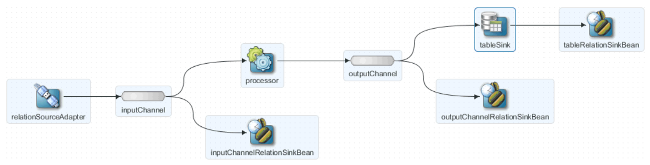
- adapter (JMS, HTTP, and file)
- channel
- processor
- cache
- table.

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 Event Processing, you define event sources and event sinks using Oracle JDeveloper to create the Event Processing Network (EPN) as Figure 1-6 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 1-6 Event Sources and Event Sinks in the Event Processing Network



For more information, see:

- Streams and Relations
- Introduction to Oracle CQL Queries, Subqueries, Views, and Joins

1.1.9 Table Event Sources

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

- Relational Database Table Event Sources
- XML Table Event Sources

  Function Table Event Sources

For more information, see Event Sources and Event Sinks.

1.1.9.1 Relational Database Table Event Sources

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

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

1.1.9.2 XML Table Event Sources

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

For more information, see XMLTABLE Query.

1.1.9.3 Function Table Event Sources

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

For more information, see:

- Function TABLE Query
- Functions.

1.1.10 Table Event Sink

The table event sink feature supports the insert, delete or update events from the EPN upstream, and send the events to the downstream connected to the table.

Spring Assembly File

By default, the adapter sends the `SampleEvent` type event and all other stages receive this type event. If the event type changes to other types, it must follow the configuration given below:

Example 1-1 Event Type

```
<wlevs:event-type type-name="SampleEvent">
    <wlevs:properties>
        <wlevs:property name="eventId" type="int"/>
        <wlevs:property name="msg" type="char[]" length="64"/>
    </wlevs:properties>
</wlevs:event-type>
```

Example 1-2 Table Tag

```
<wlevs:table id="tableSink" event-type="SampleEvent" data-source="test-ds" key-
properties="eventId" table-name="TTest">
    <wlevs:listener ref="tableRelationSinkBean"/>
</wlevs:table>
```

For other channels, the attributes are the same.

Example 1-3 Channel

The `outputChannel` is defined as below.

```
<wlevs:channel id="outputChannel" event-type="SampleEvent" is-relation="true"
primary-key="eventId>
    <wlevs:listener ref="outputChannelRelationSinkBean"/>
    <wlevs:listener ref="tableSink"/>
    <wlevs:source ref="processor"/>
</wlevs:channel>
```

Example 1-4 Application Configuration File

To support `table-sink`, you must configure the `processor`.

```
<processor>
    <name>processor</name>
    <rules>
    <query id="getAllEventsRule">
        <![CDATA[ select * from inputChannel ]]>
    </query>
    </rules>
</processor>
```

During the Adapter initialization phase, it connects to the data source defined in the server configuration file and create the table with the SQL statement:

```
CREATE TABLE TTest (eventId INTEGER,msg VARCHAR(64))
```

As an adapter implements `RunnableBean` and `RelationSource`, inside the `run()` method, it sends *insert*, *delete*, or *update* events by case.

The three event sink beans implement the same Java class `BatchRelationSink` which receives both non-batch and batch events.

Example 1-5   Server Configuration File

The data source `test-ds` needs to be defined in the configuration file:

```
<data-source>
    <name>test-ds</name>
    <connection-pool-params>
        <initial-capacity>15</initial-capacity>
        <max-capacity>50</max-capacity>
    </connection-pool-params>
        <driver-params>
            <url>jdbc:derby:testTableSinkDB;create=true</url>
        <driver-name>
            org.apache.derby.jdbc.EmbeddedDriver
        </driver-name>
    </driver-params>
</data-source>
```

1.1.10.1 Spring Assembly File

By default, the adapter sends the `SampleEvent` type event and all other stages receive this type event.

If the event type changes to other types, it must follow the configuration given below:

Example 1-6   Event Type

```
<wlevs:event-type type-name="SampleEvent">
    <wlevs:properties>
        <wlevs:property name="eventId" type="int"/>
<wlevs:property name="msg" type="char[]" length="64"/>
    </wlevs:properties>
</wlevs:event-type>
```

Example 1-7   Table Tag

The `outputChannel` is as defined below:

```
<wlevs:table id="tableSink" event-type="SampleEvent" data-source="test-ds" key-
properties="eventId" table-name="TTest">
<wlevs:listener ref="tableRelationSinkBean"/>
</wlevs:table>
```

For other channels, the attributes are the same.

Example 1-8   Channel

```
<wlevs:channel id="outputChannel" event-type="SampleEvent" is-relation="true"
primary-key="eventId">
    <wlevs:listener ref="outputChannelRelationSinkBean"/>
    <wlevs:listener ref="tableSink"/>
<     <wlevs:source ref="processor"/>
</wlevs:channel>
```

1.1.10.2 Application Configuration File

To support the table sink, you must configure the processor.

Example 1-9 Application Configuration

The configuration must be as follows:

```
<processor>
    <name>processor</name>
    <rules>
      <query id="getAllEventsRule">
        <![CDATA[ select * from inputChannel ]]>
      </query>
    </rules>
</processor>
```

During the Adapter initialization phase, it will connect to the data source defined in the server configuration file and create the table with the SQL statement:

```
CREATE TABLE TTest (eventId INTEGER,msg VARCHAR(64))
```

As an adapter implements `RunnableBean` and `RelationSource`, inside the `run()` method, it sends insert, delete or update events by case.

The three event sink beans implement the same Java class `BatchRelationSink` which receives both non-batch and batch events. The data source named `test-ds` is defined in the following configuration file:

Example 1-10 Server-configuration File

```
<data-source>
    <name>test-ds</name>
    <connection-pool-params>
      <initial-capacity>15</initial-capacity>
      <max-capacity>50</max-capacity>
    </connection-pool-params>
    <driver-params>
      <url>jdbc:derby:testTableSinkDB;create=true</url>
      <driver-name>
        org.apache.derby.jdbc.EmbeddedDriver
      </driver-name>
    </driver-params>
</data-source>
```

1.1.11 Cache Event Sources

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

For more information, see:

- Event Sources and Event Sinks
- Cache Query
- S[now].

1.1.12 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 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 data type other than the data type expected by the Oracle CQL function, then Oracle Event Processing attempts to convert the argument to the expected data type before performing the Oracle CQL function.

Oracle CQL provides a variety of built-in single-row functions and aggregate functions based on the Colt open source libraries for high performance scientific and technical computing. The functions which are available as part of Colt library will not support Big Decimal data type and NULL input values. Also the value computation of the functions are not incremental. See the COLT website for details.

Note:

Function names are case sensitive:

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

For more information, see:

- Built-In Single-Row Functions
- Built-In Aggregate Functions
- Colt Single-Row Functions
- Colt Aggregate Functions

- java.lang.Math Functions
- User-Defined Functions
- Data Type Conversion.

1.1.13 Time

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

Note that the timestamp ordering requirement is specific to one stream or a relation. For example, tuples of different streams could be arbitrarily interleaved. The order of processing tuples with the same time-stamps is not guaranteed in the case where multiple streams are processing. In addition, there is no defined behavior for negative timestamps. For t = 0, the event will be outputted immediately, assuming total order.

Oracle Event Processing can observe application time or system time.

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

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

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

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

1.2 Oracle CQL Statements

Oracle CQL provides statements for creating queries and views.

This section describes:

- Lexical Conventions
- Syntactic Shortcuts and Defaults
- Documentation Conventions.

For more information, see:

- Oracle CQL Queries, Views, and Joins
- Oracle CQL Statements.

1.2.1 Lexical Conventions

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

The configuration source must conform with the `wlevs_application_config.xsd` schema and may contain only `rule`, `view`, or `query` elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<n1:config xsi:schemaLocation="http://www.bea.com/ns/wlevs/config/application
wlevs_application_config.xsd"
    xmlns:n1="http://www.bea.com/ns/wlevs/config/application"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<processor>
    <name>cqlProcessor</name>
    <rules>
        <view id="lastEvents" schema="cusip bid srcId bidQty ask askQty seq"><!
[CDATA[
            select cusip, bid, srcId, bidQty, ask, askQty, seq
            from inputChannel[partition by srcId, cusip rows 1]
        ]]></view>
        <view id="bidask" schema="cusip bid ask"><![CDATA[
            select cusip, max(bid), min(ask)
            from lastEvents
            group by cusip
        ]]></view>
            <view ...><![CDATA[
                ...
        ]]></view>
        ...
        <view id="MAXBIDMINASK" schema="cusip bidseq bidSrcId bid askseq askSrcId
ask bidQty askQty"><![CDATA[
            select bid.cusip, bid.seq, bid.srcId as bidSrcId, bid.bid, ask.seq,
ask.srcId as askSrcId, ask.ask, bid.bidQty, ask.askQty
            from BIDMAX as bid, ASKMIN as ask
            where bid.cusip = ask.cusip
        ]]></view>
        <query id="BBAQuery"><![CDATA[
            ISTREAM(select bba.cusip, bba.bidseq, bba.bidSrcId, bba.bid, bba.askseq,
                bba.askSrcId, bba.ask, bba.bidQty, bba.askQty, "BBAStrategy" as
intermediateStrategy,
                p.seq as correlationId, 1 as priority
            from MAXBIDMINASK as bba, inputChannel[rows 1] as p where bba.cusip =
p.cusip)
        ]]></query>
    </rules>
</processor>
```

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

- You may specify one Oracle CQL statement per `view` or `query` element.
- You must *not* terminate Oracle CQL statements with a semicolon (`;`).
- You must enclose each Oracle CQL statement in `<![CDATA[` and `]]>`.

- 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. pcbpel/cep/test/sql/tklinroadbm3hrs_5000000.cqlx.new

  ```
  <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>

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

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

  For more information, see:

  – Functions

  – Literals

  – Schema Object Names and Qualifiers.

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

1.2.2 Syntactic Shortcuts and Defaults

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

For more information, see:

- Default Stream-to-Relation Operator

- Default Relation-to-Stream Operator

1.2.3 Documentation Conventions

All Oracle CQL statements in this reference (see 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 Event Processing Server

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

2 Basic Elements of Oracle CQL

A reference for fundamental parts of Oracle Continuous Query Language (Oracle CQL), including data types, literals, nulls, and more. Oracle CQL is the query language used in Oracle Stream Analytics applications is provided.

2.1 Data Types

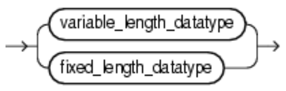
Each value manipulated by Oracle Event Processing has a data type. The data type of a value associates a fixed set of properties with the value. These properties cause Oracle Event Processing to treat values of one data type differently from values of another. For example, you can add values of `INTEGER` data type, but not values of `CHAR` data type.When you create a stream, you must specify a data type for each of its elements. When you create a user-defined function, you must specify a data type for each of its arguments. These data types define the domain of values that each element can contain or each argument can have. For example, attributes with `TIMESTAMP` as data type 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 data types that you can use. The syntax of Oracle CQL data types appears in the diagrams that follow.

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

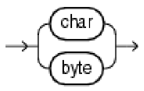
For more information, see:

- Oracle CQL Built-in Data Types
- Handling Other Data Types Using Oracle CQL Data Cartridges
- Handling Other Data Types Using a User-Defined Function
- Data Type Comparison Rules
- Literals
- Format Models
- How to Define a Data Type Alias Using the Aliases Element.

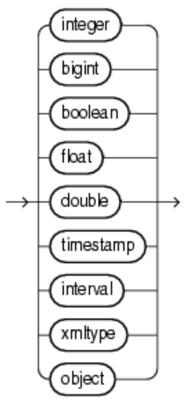
***datatype*::=**



***variable_length_datatype*::=**



***fixed_length_datatype*::=**



2.1.1 Oracle CQL Built-in Data Types

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

Consider these data type and data type literal restrictions when defining event types.

Table 2-1 Oracle CQL Built-in Data Type Summary

Oracle CQL Data Type	Description
`BIGINT`	Fixed-length number equivalent to a Java `Long` type. For more information, see Numeric LiteralsNumeric LiteralsNumeric LiteralsNumeric LiteralsNumeric 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 Numeric Literals.
`CHAR[(`*`size`*`)]` Oracle CQL supports single-dimension arrays only.	Variable-length character data of length *`size`* characters. Maximum *`size`* is 4096 characters. Default and minimum *`size`* is 1 character. For more information, see Text Literals.
`DOUBLE`	Fixed-length number equivalent to a Java `double` type. For more information, see Numeric Literals.
`FLOAT`	Fixed-length number equivalent to a Java `float` type. For more information, see Numeric Literals.
`INTEGER`	Fixed-length number equivalent to a Java `int` type. For more information, see Numeric Literals.

Table 2-1 (Cont.) Oracle CQL Built-in Data Type Summary

Oracle CQL Data Type	Description
`INTERVAL`	Fixed-length `INTERVAL` data type specifies a period of time. Oracle Event Processing supports `DAY TO SECOND` and `YEAR TO MONTH`. Maximum length is 64 bytes. This corresponds to a Java `long` type. For more information, see Interval Literals.
`TIMESTAMP`	Fixed-length `TIMESTAMP` data type 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 Datetime Literals.
`XMLTYPE`	Use this data type 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 data type of a stream attribute. This corresponds to a Java `java.lang.String` type. For more information, see SQL/XML (SQLX).
`OBJECT`	This stands for any Java object (that is, any subclass of `java.lang.Object`). We refer to this as opaque type support in Oracle Event Processing since the Oracle Event Processing engine does not understand the contents of an `OBJECT` field. You typically use this type to pass values, from an adapter to its destination, as-is; these values need not be interpreted by the Oracle Event Processing engine (such as `Collection` types or any other user-specific Java type) but that are associated with the event whose other fields are referenced in a query.

2.1.2 Handling Other Data Types Using Oracle CQL Data Cartridges

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

2.1.3 Handling Other Data Types Using a User-Defined Function

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

```
package com.oracle.app;

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

package com.oracle.app;

import com.oracle.capp.ProcessStatus;

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

```
...
}

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;
    }
}

<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>

<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 User-Defined Functions.

2.2 Data Type Comparison Rules

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

2.2.1 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.

2.2.2 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'`.

2.2.3 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'
```

2.2.4 Data Type Conversion

Generally an expression cannot contain values of different data types. For example, an arithmetic expression cannot multiply 5 by 10 and then add `'JAMES'`. However, Oracle Event Processing supports both implicit and explicit conversion of values from one data type 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 data type conversion functions.
- Implicit data type conversion can have a negative impact on performance.
- Implicit conversion depends on the context in which it occurs and may not work the same way in every case.
- Algorithms for implicit conversion are subject to change across software releases and among Oracle products. Behavior of explicit conversions is more predictable.

This section describes:

- Implicit Data Type Conversion
- Explicit Data Type Conversion
- SQL Data Type Conversion
- Oracle Data Cartridge Data Type Conversion
- User-Defined Function Data Type Conversion.

2.2.4.1 Implicit Data Type Conversion

Oracle Event Processing automatically converts a value from one data type 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

`from/to`	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	--	--	--	--	--	--

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

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

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

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

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

2.2.4.2 Explicit Data Type Conversion

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

Table 2-3 Explicit Type Conversion Matrix

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

2.2.4.3 SQL Data Type Conversion

.

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

For more information, see Relational Database Table Query.

2.2.4.4 Oracle Data Cartridge Data Type Conversion

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

2.2.4.5 User-Defined Function Data Type Conversion

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

For more information, see User-Defined Function Data Types.

2.3 Literals

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

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

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

2.3.1 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 Event Processing can distinguish them from schema object names.

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

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

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

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

For more information, see:

- Lexical Conventions
- Schema Object Names and Qualifiers
- *const_string*.

2.3.2 Numeric Literals

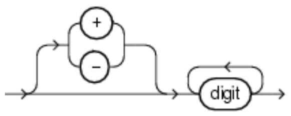
Use numeric literal notation to specify fixed and floating-point numbers.

2.3.2.1 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:

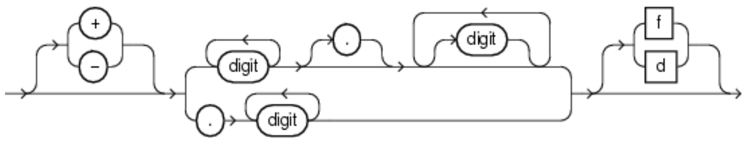
```
7
+255
```

2.3.2.2 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 32-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`. pcbpel/cep/src/oracle/cep/common/Constants.BIGINT_LENGTH

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

  The suffixes `f` or `F` and `d` or `D` are supported only in floating-point number literals, not in character strings that are to be converted to `INTEGER`. For example, if Oracle

Event Processing is expecting an `INTEGER` and it encounters the string `'9'`, then it converts the string to the Java `Integer` 9. However, if Oracle Event Processing encounters the string `'9f'`, then conversion fails and an error is returned.

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

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

Note:

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

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

```
'5,123'
```

Here are some valid `NUMBER` literals:

```
25
+6.34
0.5
-1
```

Here are some valid floating-point number literals:

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

2.3.3 Datetime Literals

Oracle Event Processing supports datetime data type `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 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, Oracle CQL cannot parse the `MyTimestamp` element.

```
<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`.

```
<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, Oracle CQL can process events 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 Datetime Format Models.

```
<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).

2.3.4 Interval Literals

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

Interval literals must not exceed 64 bytes.

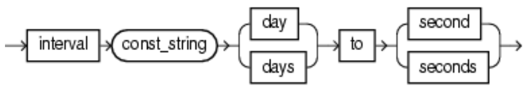
`pcbpel/cep/src/oracle/cep/common/Constants.BIGINT_LENGTH`

2.3.4.1 INTERVAL DAY TO SECOND

Stores time in terms of days, hours, minutes, and seconds.

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 Datetime Format Models).

Restriction on the Leading Field:

If you specify a trailing field, then it must be less significant than the leading field. For example, `INTERVAL MINUTE TO DAY` is not valid. As a result of this restriction, if `SECOND` is the leading field, the interval literal cannot have any trailing field.

The valid range of values for the trailing field are as follows:

- `SECOND`: 0 to 59.999999999

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. In this example, stream `tkdata2_SIn1` has schema `(c1 integer, c2 interval)`.

```
<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>
```

Using `INTERVAL DAY TO SECOND` in the define clause of pattern match:

```
query 'select its.itemId from ch0
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 "00:00:05.00" HOUR TO SECOND)),
C AS (to_timestamp(C.element_time) - to_timestamp(A.element_time)
>= INTERVAL "00:05.00" MINUTE TO SECOND)
) as its'
```

Input:

```
send [itemId=2 temp=30]
send [itemId=2 temp=55]
thread:sleep 5000
send [itemId=2 temp=125]
```

Output:

```
-> insert event: {itemId=2}
```

2.3.4.2 INTERVAL YEAR TO MONTH

Stores time in terms of years and months.

Examples of the various forms of `INTERVALYEARTOMONTH` literals follow:

Form of Interval Literal	Interpretation
`INTERVAL "12-10" YEAR TO MONTH`	12 years and 10 months.

Note:

If used in the query DDL, the `INTERVAL YEAR TO MONTH` notation is used to specify constant in the CQL query. Specify a constant interval value instead of a variable.

The code sample given below uses the tuples listed below:

```
<event-type type-name="IntervalInputTupleEvent">
    <properties>
        <property name="interval_inputevent" type="interval year to month"/>
    </properties>
</event-type>

<event-type type-name="IntervalOutputTupleEvent">
    <properties>
        <property name="interval_outputevent" type="interval year to month"/>
    </properties>
</event-type>
```

The following is a sample query for `INTERVAL YEARS TO MONTH`:

Query:

```
select interval_inputevent as interval_outputevent from inputChannel
```

Input:

```
send [interval_inputevent='INTERVAL "212-10" YEAR(3) TO MONTH']
```

Output:

```
-> insert event: {interval_outputevent=+212-10}
```

2.4 Format Models

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

- Number Format Models
- Datetime Format Models.

2.4.1 Number Format Models

You can use number format models in the following functions:

- In the function to translate a value of `int` data type to `bigint` data type.

- In the to_float function to translate a value of `int` or `bigint` data type to `float` data type

2.4.2 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 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` data type to a `TIMESTAMP` data type.

2.5 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 data type that are not restricted by primary key integrity constraints. Use a null when the actual value is not known or when a value would not be meaningful.

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

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.
- *out_of_line_constraint*.

2.5.1 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`.

2.5.2 Nulls with Comparison Conditions

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

2.5.3 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 Null Conditions .

2.6 Comments

Oracle CQL does not support comments.

2.7 Aliases

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

This section describes:

- Defining Aliases Using the AS Operator
- Defining Aliases Using the Aliases Element.

2.7.1 Defining Aliases Using the AS Operator

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

This section describes:

- Aliases in the relation_variable Clause
- Aliases in Window Operators.

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

2.7.1.1 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 Sorting Query Results), but not other clauses in the query.

The following example shows how to define alias badItem for a stream element its.itemId in a SELECT list and alias its for a MATCH_RECOGNIZE clause.

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

For more information, see From Clause.

2.7.1.2 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.

The following example shows how to define aliases bid and ask after partitioned range window operators.

```
<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 Stream-to-Relation Operators (Windows).

2.7.2 Defining Aliases Using the Aliases Element

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

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

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

Note the following:

- If the alias already exists then, Oracle Event Processing will throw an exception.
- If a query or view definition references an alias, then the alias must already exist.

This section describes:

- How to Define a Data Type Alias Using the Aliases Element.

2.7.2.1 How to Define a Data Type Alias Using the Aliases Element

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

For more information, see Data Types.

To define a type alias using the aliases element:

1. Edit the component configuration file of a processor.
2. Add an aliases element.

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

3. Add a `type-alias` child element to the aliases element.

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

4. Add a `source` and `target` child element to the `type-alias` element, where:
   - `source` specifies the alias.

     You can use any valid schema name. For more information, see Schema Object Names and Qualifiers.
   - `target` specifies the data type the alias refers to.

     For Oracle CQL data cartridge types, use the fully qualified type name.

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

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

   You can use the alias in exactly the same way you would use the data type it refers to. As shown in the following example, you can access methods and fields of the aliased type.

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

2.8 Schema Object Names and Qualifiers

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

- Schema Object Naming Rules
- Schema Object Naming Guidelines
- Schema Object Naming Examples

For more information, see Lexical Conventions.

2.8.1 Schema Object Naming Rules

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

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

The following list of rules applies to identifiers:

- Identifiers cannot be Oracle Event Processing reserved words.

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

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

For more information, see

  – *identifier*.

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

  For more information, see:

  – *const_string*

  – *identifier*.

- In general, you should choose names that are unique across an application for the following objects:

  – Streams

  – Queries

  – Views

  – User-defined functions.

  Specifically, a query and view cannot have the same name.
- Identifier names are case sensitive.
- Stream elements in the same stream or view cannot have the same name. However, stream elements in different streams or views can have the same name.
- Functions can have the same name, if their arguments are not of the same number and data types (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:

  – User-Defined Functions.

2.8.2 Schema Object Naming Guidelines

Here are several guidelines for naming objects and their parts:

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

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

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

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

2.8.3 Schema Object Naming Examples

The following examples are valid schema object names:

```
last_name
horse
a_very_long_and_valid_name
```

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

3

Pseudocolumns

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

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 Functions).

Oracle CQL supports the following pseudocolumns:

- ELEMENT_TIME Pseudocolumn.

3.2 ELEMENT_TIME Pseudocolumn

In CQL, every stream event is associated with a timestamp. The `ELEMENT_TIME` pseudo column returns the timestamp of the stream event. The datatype of `ELEMENT_TIME` pseudo column is Oracle CQL native `bigint` type. The unit of timestamp value returned by `ELEMENT_TIME` is in nanoseconds.

Note:

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

This section describes:

- Understanding the Value of the ELEMENT_TIME Pseudocolumn
- Using the ELEMENT_TIME Pseudocolumn in Oracle CQL Queries.

For more information, see:

- to_timestamp.

3.2.1 Understanding the Value of the ELEMENT_TIME Pseudocolumn

The value of `ELEMENT_TIME` for each stream event is the timestamp of that event. The timestamp of stream event depends on the stream definition and source.

3.2.1.1 ELEMENT_TIME for a System-Timestamped Stream

If source stream is a system timestamped stream, then the timestamp for a stream event is assigned by computing `System.nanoTime()`. For each event, `ELEMENT_TIME` pseudo column outputs the event's timestamp.

For example, consider a system timestamped stream defined as: `tktest_S1(c1 integer)`.

```
select ELEMENT_TIME, to_timestamp(ELEMENT_TIME) from tktest_S1

Input (c1)                   Output (timestamp: element_time, 
to_timestamp(element_time))
10                                   1261967187839275O:+ 
1261967187839275O,05/26/1970 18:27:51
20                                   12619671889193750:+ 
12619671889193750,05/26/1970 18:27:51
30                                   12619671890093750:+ 
12619671890093750,05/26/1970 18:27:51
40                                   12619671891399750:+ 
12619671891399750,05/26/1970 18:27:51
50                                   12619671896472750:+ 
12619671896472750,05/26/1970 18:27:51
```

Note:

The output may vary for each execution and also depends on the machine as timestamp is computed by calculating `System.nanoTime()`.

3.2.1.2 ELEMENT_TIME for an Application-Timestamped Stream

If source stream is an application timestamped stream, then timestamp for stream event is assigned by computing the application timestamp expression. The unit of computed timestamp value is always in nanoseconds. `ELEMENT_TIME` pseudo column outputs the event's timestamp.

For example, consider an application timestamped stream defined as `tktest_S1(C1 integer, c2 bigint)` and application timestamp expression as: `c2*1000000000L`.

```
select ELEMENT_TIME, to_timestamp(ELEMENT_TIME) from tktest_S1

Input(c1,c2)     Output(timestamp: element_time, to_timestamp(element_time))
10, 10               10000000000:+ 10000000000,12/31/1969 17:00:10
20, 20               20000000000:+ 20000000000,12/31/1969 17:00:20
30, 30               30000000000:+ 30000000000,12/31/1969 17:00:30
40, 40               40000000000:+ 40000000000,12/31/1969 17:00:40
50, 50               50000000000:+ 50000000000,12/31/1969 17:00:50
```

In the above query, the timestamp of each event is computed by computing `c2*1000000000L` for each event. You can see that `ELEMENT_TIME` is same as timestamp of the event.

3.2.1.2.1 Derived Timestamp Expression Evaluates to int or bigint

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

3.2.1.2.2 Derived Timestamp Expression Evaluates to timestamp

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

3.2.1.3 ELEMENT_TIME for an Inline CQL View

If source stream is received from an inline CQL view, then the timestamp of stream event is computed by the view query. `ELEMENT_TIME` outputs the timestamp of event. The unit of `ELEMENT_TIME` is always in nanosecond and the datatype is a CQL native `bigint` type.

For example, consider an application timestamped stream defined as `tktest_S1(C1 integer, c2 bigint)` and application timestamp expression as: `c2*1000000000L`.

The view `V` is defined using the query `ISTREAM(SELECT * FROM SYSTS_STREAM[RANGE 1 MINUTE SLIDE 15 SECONDS]`.

```
select ELEMENT_TIME, to_timestamp(ELEMENT_TIME) from V

Input(c1,c2)        Output(timestamp: element_time, to_timestamp(element_time))
10, 10                15000000000:+ 15000000000,12/31/1969 17:00:15
20, 20                30000000000:+ 30000000000,12/31/1969 17:00:30
30, 30                30000000000:+ 30000000000,12/31/1969 17:00:30
40, 40                45000000000:+ 45000000000,12/31/1969 17:00:45
50, 50                50000000000:+ 50000000000,12/31/1969 17:00:55
```

3.2.1.4 ELEMENT_TIME for a Subquery

If source stream is received from a subquery, then CQL does not support `ELEMENT_TIME` on the subquery results.

The following example depicts the scenario which is not supported.

```
SELECT ELEMENT_TIME FROM ( ISTREAM(SELECT * FROM SYSTS_STREAM[RANGE 1 HOUR SLIDE 5
MINUTES])
```

3.2.2 Using the ELEMENT_TIME Pseudocolumn in Oracle CQL Queries

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

- Using ELEMENT_TIME With SELECT
- Using ELEMENT_TIME With GROUP BY
- Using ELEMENT_TIME With PATTERN.

3.2.2.1 Using ELEMENT_TIME With SELECT

The following example shows how you can use the `ELEMENT_TIME` pseudocolumn in a select statement. Stream `S1` has schema `(c1 integer)`. Note that the function `to_timestamp` is used to convert the `Long` values to timestamp values.

```
<query id="q4"><![CDATA[
    select
        c1,
        to_timestamp(element_time)
    from
```

```
        S1[range 10000000 nanoseconds slide 10000000 nanoseconds]
]]></query>

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

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

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

3.2.2.2 Using ELEMENT_TIME With GROUP BY

You cannot use `ELEMENT_TIME` in the `SELECT` statement of the query because of the `GROUP BY` clause.

```
<query id="Q1"><![CDATA[

    SELECT
        R.queryText AS queryText,
        COUNT(*) AS queryCount
    FROM
        queryEventChannel [range 30 seconds] AS R
    GROUP BY
        queryText

]]></query>
```

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

```
<view id="V1"><![CDATA[

    ISTREAM (
        SELECT
            R.queryText AS queryText,
            COUNT(*) AS queryCount,
            MAX(R.ELEMENT_TIME) as maxTime
        FROM
            queryEventChannel [range 30 seconds] AS R
        GROUP BY
            queryText
    )

]]></view>
```

Note:

The element time associated with an output element of view V1 need not be the same as the value of the attribute `maxTime` for that output event.

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

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

To select the `ELEMENT_TIME` of the output events of view V1, create a query.

```
<query id="Q1"><![CDATA[

    SELECT
        queryText,
        queryCount,
        ELEMENT_TIME as eventTime
    FROM
        V1

]]></query>
```

3.2.2.3 Using ELEMENT_TIME With PATTERN

The following example 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)`.

```
...
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
...
```

3.3 ORA_QUERY_ID Pseudocolumn

To partition the incoming events, you need to have the information of the query name or identifier in the output. The query name or identifier should be part of tuple attribute and it should be accessed by calling TupleValue's getter APIs.

For this purpose, Oracle CQL provides a new pseudo column ORA_QUERY_ID to access the query name in the output of a query.

You can get the query name in the output tuples by using the above pseudo column in the CQL query's `SELECT` list as follows:

```
CREATE QUERY Q1 AS SELECT ORA_QUERY_ID from STREAM;
```

Each output event of the above query `Q1` has only one attribute whose value is equal to query's name or identifier. In the above query, for each incoming event to `STREAM`, the application sends an output tuple with one attribute that has the value `Q1`.

```
CREATE QUERY Q1 AS
SELECT ORA_QUERY_ID, stock_quote, stock_price FROM StockStream;
```

The input and output values are listed below:

```
Input(stock_quote, stock_price)            Output(ORA_QUERY_ID, stock_quote,
stock_price)
ORCL, 34                                                 Q1, ORCL, 34
MSFT, 38                                                 Q1, MSFT, 38
CSCO, 21                                                 Q1, CSCO, 21
INTC, 24                                                 Q1, INTC, 24
FB, 48                                                     Q1, FB, 48
```

4

Operators

A reference for operators in Oracle Continuous Query Language (Oracle CQL). An operator manipulates data items and returns a result is provided. 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:

- Arithmetic Operators
- Concatenation Operator
- Alternation Operator
- Range-Based Stream-to-Relation Window Operators
- Tuple-Based Stream-to-Relation Window Operators
- Partitioned Stream-to-Relation Window Operators
- User-Defined Stream-to-Relation Window Operators
- IStream Relation-to-Stream Operator
- DStream Relation-to-Stream Operator
- RStream Relation-to-Stream Operator.

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

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

Table 4-1 Oracle CQL Operator Precedence

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

4.2 Arithmetic Operators

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

In certain cases, Oracle Event Processing converts the arguments to the data type as required by the operation. For example, when an integer and a float are added, the integer argument is converted to a float. The data type of the resulting expression is a float. For more information, see Implicit Data Type Conversion.

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>`

Table 4-2 (Cont.) Arithmetic Operators

Operator	Purpose	Example

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

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

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

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

For more information, see:

- Numeric Literals
- Datetime Literals
- Interval Literals.

4.3 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 data type `CHAR`, then the result has data type `CHAR` and is limited to 2000 characters. Trailing blanks in character strings are preserved by concatenation, regardless of the data types of the string.

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

Processing. To concatenate an expression that might be null, use the `NVL` function to explicitly convert the expression to a zero-length string.

See Also:

- Data Types
- concat
- xmlconcat
- nvl.

The following example shows how to use the concatenation operator to append the String "xyz" to the value of `c2` in a select statement.

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

4.4 Alternation Operator

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

Table 4-4 Alternation Operator

Operator	Purpose	Example

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

The alternation operator is applicable only within a `PATTERN` clause.

The following example shows how to use the alternation operator to change the sense of the `PATTERN` clause to mean "A one or more times followed by either B one or more times or C one or more times, whichever comes first".

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

```
        A as A.c1 = 10,
        B as B.c1 = 20
        C as C.c1 = 30
) as T
]]></query>
```

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

4.5 Range-Based Stream-to-Relation Window Operators

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

Note:

Very large numbers must be suffixed. Without the suffix, Java treats very large numbers like an integer and the value might be out of range for an integer, which throws an error.

Add a suffix as follows:

l or L for Long

f or F for float

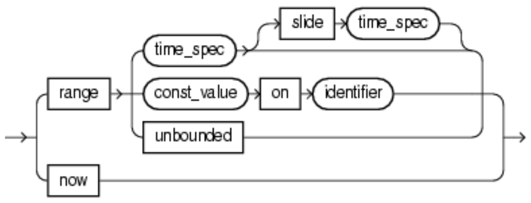
d or D for double

n or N for big decimal

For example:

```
SELECT * FROM channel0[RANGE 1368430107027000000l nanoseconds]
```

window_type_range::=



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

For more information, see:

- Query
- Stream-to-Relation Operators (Windows)

- Aliases in Window Operators.

4.5.1 S[now]

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

For an example, see S [now] Example.

4.5.1.1 Examples

S [now] Example

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

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

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

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

4.5.2 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.

4.5.2.1 Examples

S [range T] Example

Consider the query `q1`. Given the data stream `S`, the query returns the relation. By default, the range time unit is `second`, so `S[range 1]` is equivalent to `S[range 1 second]`. Timestamps are shown in milliseconds (`1 s = 1000 ms`). As many elements as there are in the first `1000 ms` interval enter the window, namely tuple `(10,0.1)`. At time `1002 ms`, tuple `(15,0.14)` enters the window. At time `2000 ms`, any tuples that have been in the window longer than the range interval are subject to deletion from the relation, namely tuple `(10,0.1)`. Tuple `(15,0.14)` is still in the relation at this time. At time `2002 ms`, tuple

`(15,0.14)` is subject to deletion because by that time, it has been in the window longer than `1000 ms`.

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

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

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

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
```

4.5.3 S[range T1 slide T2]

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

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

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

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

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

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

For an example, seeS [range T1 slide T2] Example.

4.5.3.1 Examples

S [range T1 slide T2] Example

Consider the query `q1`. Given the data stream `S`, the query returns the relation. By default, the range time unit is `second`, 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`.

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

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

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
```

4.5.4 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.

4.5.4.1 Examples

S [range unbounded] Example

Consider the query `q1` and the data stream . Timestamps are shown in milliseconds (`1 s = 1000 ms`). Elements are inserted into the relation as they arrive. No elements are subject to deletion. The following example shows the relation that the query returns at time `5000 ms` and the relation that the query returns at time `205000 ms`.

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

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

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

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
```

4.5.5 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
- S [range C on E] Example: INTERVAL and TIMESTAMP.

4.5.5.1 Examples

S [range C on E] Example: Constant Value

Consider the query `tkdata56_q0` and the data stream `tkdata56_S0` . Stream `tkdata56_S0` has schema `(c1 integer, c2 float)`. The following example 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.

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

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

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

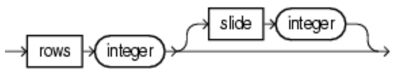
Similarly, you can use the `S[range C on ID]` window with `INTERVAL` and `TIMESTAMP`. Consider the query `tkdata56_q2` in and the data stream `tkdata56_S1`. Stream `tkdata56_S1` has schema `(c1 timestamp, c2 double)`. The following example shows the relation that the query returns.

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

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

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
```

4.6 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]

For more information, see:

- Range-Based Stream-to-Relation Window Operators
- Query
- Stream-to-Relation Operators (Windows)
- Aliases in Window Operators.

4.6.1 S [rows N]

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

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

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

By default, the slide is 1.

For examples, see S [rows N] Example.

4.6.1.1 Examples

S [rows N] Example

Consider the query `q1` and the data stream `S`. 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).

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

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

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

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

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

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

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

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

4.6.2 S [rows N1 slide N2]

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

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

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

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

For examples, see S [rows N] Example.

4.6.2.1 Examples

S [rows N1 slide N2] Example

Consider the query `tkdata55_q0` and the data stream `tkdata55_S55`. Stream `tkdata55_S55` has schema `(c1 integer, c2 float)`.

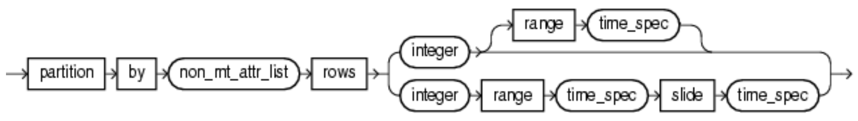
At time 100000, the output relation is empty because only one tuple `(20,0.1)` has arrived on the stream. By time 150000, the number of tuples that the `slide` value specifies (2) have arrived: at that time, the output relation contains tuples `(20,0.1)` and `(15,0.14)`. By time 250000, another `slide` 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)`.

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

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

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
```

4.7 Partitioned Stream-to-Relation Window Operators

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

window_type_partition::=



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

For more information, see:

- Tuple-Based Stream-to-Relation Window Operators
- Query
- Stream-to-Relation Operators (Windows)
- Aliases in Window Operators.

4.7.1 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.

4.7.1.1 Examples

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

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

Note:

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

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

```
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
```

```
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
```

4.7.2 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.

4.7.2.1 Examples

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

Consider the query `qPart_range2` and the data stream `SP5`. Stream `SP5` has schema `(c1 integer, name char(10))`. The query returns the relation. By default, the range time unit is `second`, so `range 2` is equivalent to `range 2 seconds`. Timestamps are shown in milliseconds (`1 s = 1000 ms`).

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

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

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
```

4.7.3 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.

4.7.3.1 Examples

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

Consider the query `qPart_rangeslide` and the data stream `SP1`. Stream `SP1` has schema `(c1 integer, name char(10))`. The query returns the relation. By default, the range and slide time unit is `second` so `range 1 slide 1` is equivalent to `range 1 second slide 1 second`. Timestamps are shown in milliseconds (`1 s = 1000 ms`).

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

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

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
```

4.8 User-Defined Stream-to-Relation Window Operators

You can write user-defined (or extensible) windows in Java to create stream-to-relation operators that are more advanced or application-specific than the built-in stream-to-relation operators that Oracle Event Processing provides. User-defined windows can appear in Oracle CQL statements wherever a built-in stream-to-relation window operator can occur.

Note:

You can also create user-defined functions (see User-Defined Functions).

4.8.1 Implementing a User-Defined Window

Using the classes in the `oracle.cep.extensibility.windows` package you can create the following types of user-defined windows:

- generic time window.

4.8.1.1 How to Implement a User-Defined Generic Time Window

You implement a user-defined generic time window by implementing a Java class that implements the `GenericTimeWindow` interface.

To implement a user-defined generic time window:

1. Implement a Java class.

```
import java.io.IOException;
import java.sql.Timestamp;

import oracle.cep.extensibility.windows.GenericTimeWindow;

public class MyRangeSlideWindow implements GenericTimeWindow {
    private long range;
    private long slide;

    public void setInputParams(Object[] obj) throws IOException{
        if(obj.length != 2)
            throw new IOException("inappropriate number of arguments");
        range = (((Integer)obj[0]).intValue())*1000;
        slide = (((Integer)obj[1]).intValue())*1000;
    }

    public boolean visibleW(Timestamp t, Timestamp visTs) {
        long actual = t.getTime();
        if(getVisibleTs(actual) < actual)
            return false;
        visTs.setTime(getVisibleTs(actual));
        return true;
    }

    private long getVisibleTs(long time) {
        if(slide > 1) {
            long t = time / slide;
            if((time % slide) == 0)
                return(t*slide);
            else
                return((t+1)*slide);
        } else
            return time;
    }

    public boolean canOutputTsGTInputTs() {
        if(slide > 1)
            return true;
        return false;
    }

    public boolean expiredW(Timestamp ts, Timestamp expTs) {
        long actual = ts.getTime();
        long visibleTs = getVisibleTs(actual);
        long expiredTs = visibleTs + range;
        expTs.setTime(expiredTs);
        // This is the border line case, when range > slide and visibleTs < range
            if(visibleTs < range)
                return false;
        return true;
    }
}
```

2. Register the window in Oracle CQL.

```
...
<rule id="range_slide"><![CDATA[
    register window range_slide(winrange int, winslide int) implement using
"MyRangeSlideWindow"
]]></rule>
...
<query id="q79"><![CDATA[
    select * from S12 [range_slide(10,5)]
]]></query>
...
```

3. Use the user-defined window in the `FROM` clause.

4.9 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.

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.

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

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

For more information, see:

- Detecting Differences in Query Results.

4.10 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`. If a tuple happens to be both inserted and deleted with the same timestamp, then IStream does not output the insertion.

In the following example, 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.

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

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

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

Table 4-5   DStream Example Output

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

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

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

For more information, see:

- Detecting Differences in Query Results.

4.11 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 the following example, `Rstream` outputs the entire state of the relation at time `Now` and filtered by the `where` clause.

```
<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>
```

5
Expressions

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

Every expression maps to a data type. This simple expression evaluates to 4 and has data type `NUMBER` (the same data type 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` data type:

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

5.1 Introduction to Expressions

Oracle Event Processing provides the following expressions:

- Aggregate distinct expressions: *aggr_distinct_expr*.
- Aggregate expressions: *aggr_expr*.
- Arithmetic expressions: *arith_expr*.
- Arithmetic expression list: *arith_expr_list*
- Case expressions: *case_expr*.
- Decode expressions: *decode*.
- Function expressions: *func_expr*.
- Object expressions: *object_expr*
- Order expressions: *order_expr*.
- XML aggregate expressions: *xml_agg_expr*
- XML column attribute value expressions: *xmlcolattval_expr*.
- XML element expressions: *xmlelement_expr*.
- XML forest expressions: *xmlforest_expr*
- XML parse expressions: *xml_parse_expr*.

You can use expressions in:

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

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

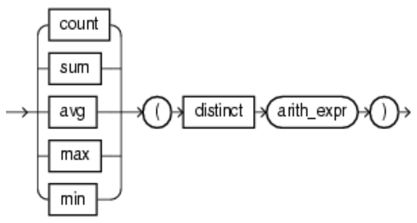
You must use appropriate expression notation whenever *expr* appears in conditions, Oracle CQL functions, or Oracle CQL statements in other parts of this reference. The sections that follow describe and provide examples of the various forms of expressions.

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

5.2 *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*.

***aggr_distinct_expr*::=**



(*arith_expr*::=)

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*::=

For more information, see Built-In Aggregate Functions.

5.2.1 Examples

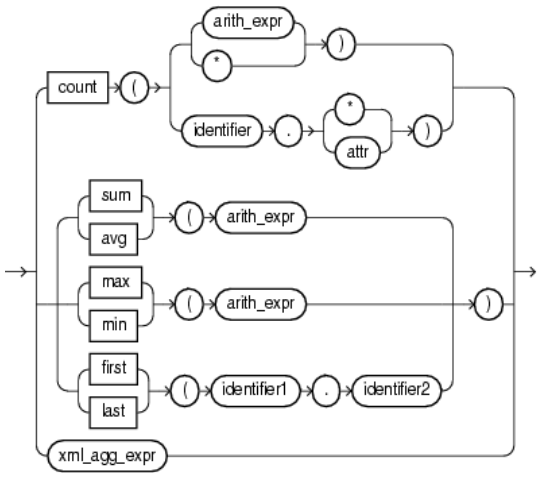
The following example shows how to use a `COUNT` aggregate distinct expression.

```
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)))
```

5.3 *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*.

aggr_expr::=



(*arith_expr*::= and *xml_agg_expr*::=)

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

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

Table 5-1   Return Values for COUNT Aggregate Function

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

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

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

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

- *arith_expr*::=

For more information, see:

- Built-In Aggregate Functions
- Using count With *, *identifier*.*, and *identifier*.*attr*
- first
- last.

5.3.1 Examples

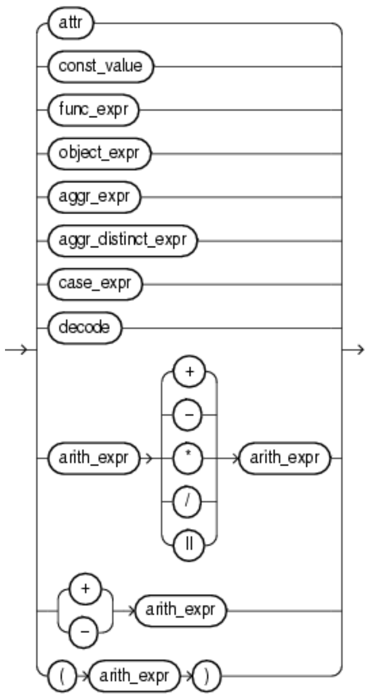
The following example shows how to use a `COUNT` aggregate expression.

```
<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>
```

5.4 *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*::=**



(*func_expr*::=, *aggr_expr*::=, *aggr_distinct_expr*::=, *case_expr*::=, *decode*::=, *arith_expr*::=)

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

- *aggr_distinct_expr*::=
- *aggr_expr*::=
- *arith_expr*::=
- *case_expr*::=
- *searched_case*::=
- *simple_case*::=
- *condition*::=
- *between_condition*::=
- *param_list*
- *measure_column*::=.

For more information, see Arithmetic Operators.

5.4.1 Examples

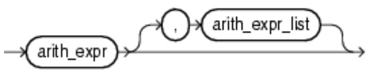
The following example shows how to use an `arith_expr` expression.

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

5.5 *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*::=)

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

- *xmlelement_expr*::=

For more information, see Arithmetic Operators.

5.5.1 Examples

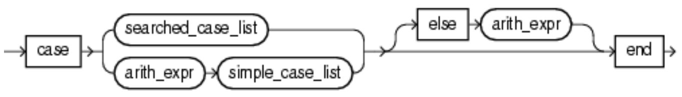
The following example shows how to use a `arith_expr_list` expression.

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

5.6 *case_expr*

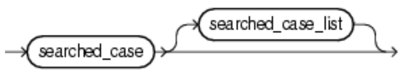
Use a `case_expr` case expression to evaluate stream elements against multiple conditions.

***case_expr*::=**



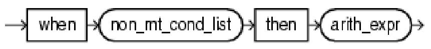
(*searched_case_list*::=, *arith_expr*::=, *simple_case_list*::=)

***searched_case_list*::=**



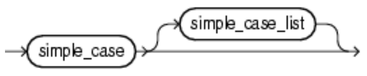
(*searched_case*::=)

***searched_case*::=**



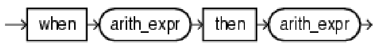
(*arith_expr*::=)

***simple_case_list*::=**



(*simple_case*::=)

***simple_case*::=**



(*arith_expr*::=)

The `case_expr` is similar to the `DECODE` clause of an arithmetic expression (see *decode*).

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 a `case_expr` in the following Oracle CQL statements:

- *arith_expr*::=.

5.6.1 Examples

This section describes the following `case_expr` examples:

- case_expr with SELECT *
- case_expr with SELECT.

case_expr with SELECT *

Consider the query `q97` and the data stream `S0`. Stream `S1` has schema `(c1 integer, c2 float)`. The query returns the relation.

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

```
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
```

```
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` and the data streams `S0` and `S1`. Stream `S0` has schema `(c1 float, c2 integer)` and stream `S1` has schema `(c1 float, c2 integer)`. The query returns the relation.

```
<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>
```

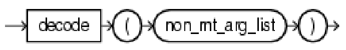
```
Timestamp   Tuple
     1000    0.1,10
     1002   0.14,15
   200000    0.2,20
   400000    0.3,30
   500000    0.3,35
   600000       ,35
h 800000
100000000   4.04,40
h 200000000
```

```
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

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
```

5.7 *decode*

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

***decode*::=**



```
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`) data types. For more information, see Data Types).

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

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

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

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

The *`decode`* expression is similar to the `case_expr` (see *case_expr::=*).

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

- *arith_expr::=*.

5.7.1 Examples

Consider the query `q` and the input relation `R`. Relation `R` has schema `(c1 float, c2 integer)`. The query returns the relation.

```
<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>
```

```
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:    +     0.3,35
   601000:    -     0.3,35
100000000:    +     4.04,40
100001000:    -     4.04,40
```

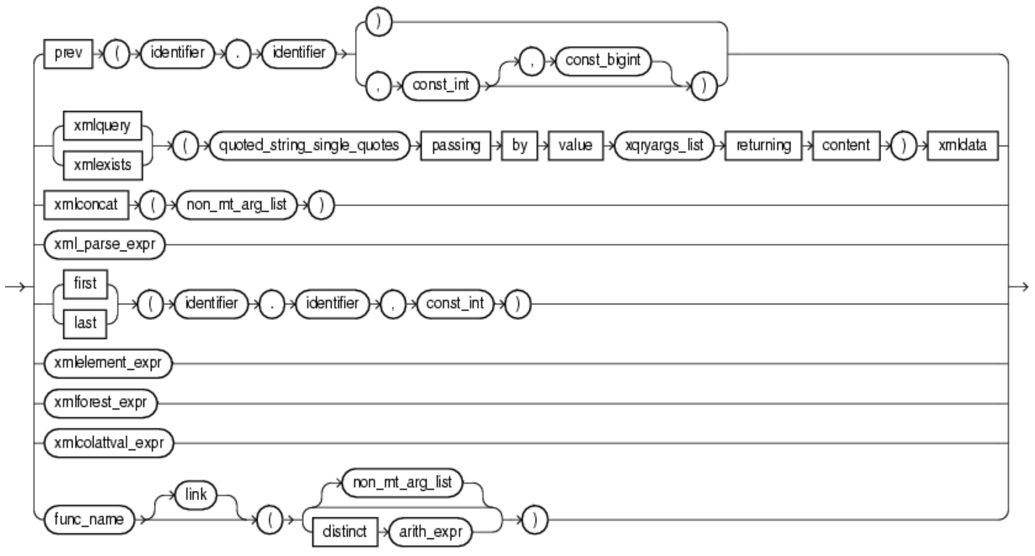
```
Timestamp   Tuple Kind  Tuple
   1000:    +     0.6
   1002:    +     0.44
   2000:    -     0.1,10
   2002:    -     0.14,15
 200000:    +     0.3
 201000:    -     0.2,20
 400000:    +     0.5
 401000:    -     0.3,30
 500000:    +     0.6
 501000:    -     0.3,35
```

```
100000000:  +    4.34
100001000:  -    4.34
```

5.8 *func_expr*

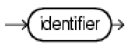
Use the `func_expr` function expression to define a function invocation using any Oracle CQL built-in, user-defined, or Oracle data cartridge function.

***func_expr*::=**



(*xml_parse_expr*::=, *xmlelement_expr*::=, *xmlforest_expr*::=, *xmlcolattval_expr*::=, *func_name*:=, *link*::=, *arith_expr*::=)

***func_name*:=**



func_name

You can specify the identifier of a function explicitly:

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

  For more information, see *link*::=.
- with an empty argument list.
- with an argument list of one or more arguments.
- with a distinct arithmetic expression.

  For more information, see *aggr_distinct_expr*.

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 ELEMENT_TIME Pseudocolumn).

For more information, see prev. For an example, see func_expr PREV Function Example.

XQuery: XMLEXISTS and XMLQUERY

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

- xmlexists
- xmlquery.

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 an example, see func_expr XMLQUERY Function Example.

For more information, see SQL/XML (SQLX).

XMLCONCAT

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

For more information, see:

- xmlconcat
- SQL/XML (SQLX).

SQL/XML (SQLX)

The SQLX specification extends SQL to support XML data.

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

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

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

For more information on Oracle CQL SQLX expressions, see:

- *xml_agg_expr*
- *xmlcolattval_expr*
- *xmlelement_expr*
- *xmlforest_expr*
- *xml_parse_expr*.

For more information on Oracle CQL SQLX functions, see:

- XQuery: XMLEXISTS and XMLQUERY
- xmlcomment
- xmlconcat
- xmlagg.

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

- Datetime Literals

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
- last

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

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

- *arith_expr*::=

For more information, see Functions.

5.8.1 Examples

This section describes the following func_expr examples:

- func_expr PREV Function Example
- func_expr XMLQUERY Function Example

func_expr PREV Function Example

The following example shows how to compose a *func_expr* to invoke the PREV function.

```
<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

The following example shows how to compose a *func_expr* to invoke the XMLQUERY function.

```
<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>
```

The following example shows how to compose a *func_expr* to invoke the SUM function.

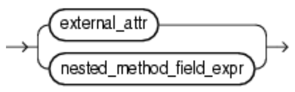
```
<query id="q3"><![CDATA[
    select sum(c2) from S1[range 5]
]]></query>
```

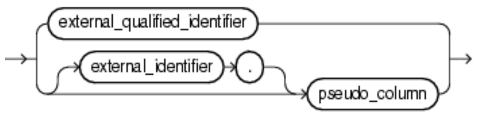
5.9 *object_expr*

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

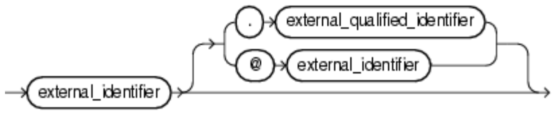
You can use an *object_expr* anywhere an arithmetic expression can be used. For more information, see *arith_expr*.

object_expr::=



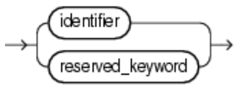


***external_qualified_identifier*::=**

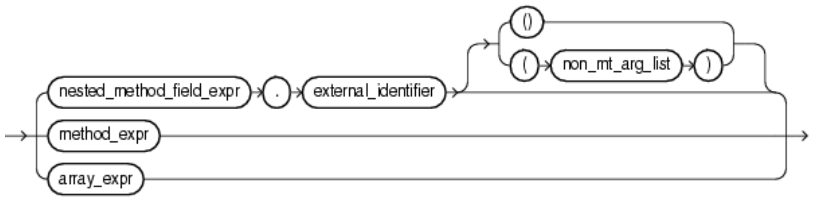


()

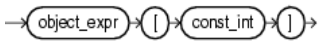
***external_identifier*::=**



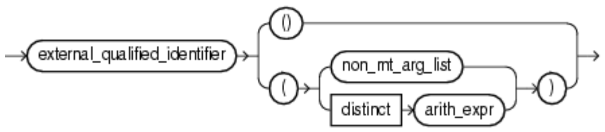
***nested_method_field_expr*::=**



***array_expr*::=**



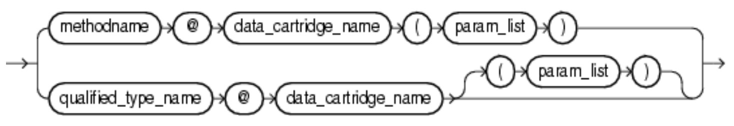
***method_expr*::=**



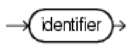
Optionally, you can use a link (`@`) in the *`object_expr`* to specify the data cartridge name. Use a *`link`* to specify the location of an Oracle CQL data cartridge complex type class, method, field, or constructor to disambiguate the reference, if necessary. The location must reference a data cartridge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

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

***link*::=**



data_cartridge_name::=



data_cartridge_name

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

Data cartridge names include:

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

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

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

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

For syntax, see data_cartridge_name::= (parent: *link*::=).

Type Declaration

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

For examples, see Type Declaration Example: *link*

Field Access

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

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

For examples, see Field Access Example: *link*.

Method Access

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

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

For examples, see Method Access Example: *link*.

Constructor Invocation

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

For examples, see Constructor Invocation Example: *link*.

5.9.1 Examples

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

- Object Expression Example
- Type Declaration Example: *link*
- Field Access Example: *link*
- Method Access Example: *link*.
- Constructor Invocation Example: *link*

Object Expression Example

The following example shows *`object_expr`*:

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

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

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

Type Declaration Example: *link*

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

```
package com.myapplication.event;

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

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

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

Field Access Example: *link*

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

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

Method Access Example: *link*

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

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

Constructor Invocation Example: *link*

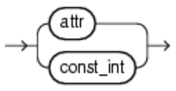
The following example shows how to instantiate complex type `MyType`. The link clause explicitly references the `com.myPackage.MyType` class that belongs to the Oracle CQL data cartridge `myCartridge`.

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

5.10 *order_expr*

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

order_expr::=



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.

5.10.1 Examples

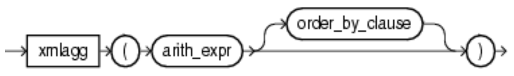
Stream `S3` has schema `(c1 bigint, c2 interval, c3 byte(10), c4 float)`. This example shows how to order the results of query `q210` by `c1` and then `c2` and how to order the results of query `q211` by `c2`, then by the stream element at index 3 (`c3`) and then by the stream element at index 4 (`c4`).

```
<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>
```

5.11 *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*)

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*::=

For more information, see:

- Built-In Aggregate Functions
- xmlagg

5.11.1 Examples

Consider the query `tkdata67_q1` and the input relation `tkdata67_S0`. Relation `tkdata67_S0` has schema `(c1 integer, c2 float)`. The query returns the relation.

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

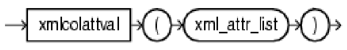
```
Timestamp           Tuple
 1000               15, 0.1
 1000               20, 0.14
 1000               15, 0.2
 4000               20, 0.3
10000               15, 0.04
h 12000
```

```
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>
```

5.12 *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*::=**



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).

You can use an *`xmlcolattval_expr`* in the following Oracle CQL statements *func_expr*::=.

5.12.1 Examples

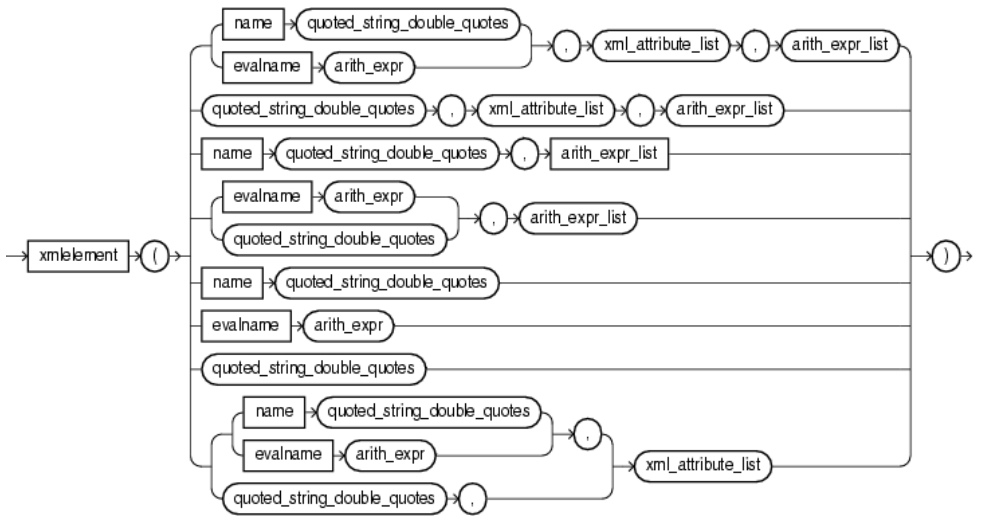
Consider the query `tkdata53_q1` and the input relation `tkdata53_S0`. Relation `tkdata53_S0` has schema `(c1 integer, c2 float)`. The query returns the relation.

```
<query id="tkdata53_q1"><![CDATA[
    select
        XMLELEMENT("tkdata53_S0", XMLCOLATTVAL( tkdata53_S0.c1, tkdata53_S0.c2))
    from
        tkdata53_S0 [range 1]
]]></query>

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

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>
```

5.13 *xmlelement_expr*

Use an `xmlelement_expr` expression when you want to construct a well-formed XML element from stream elements.

xmlelement_expr::=



(*arith_expr*::= and *arith_expr_list*::=)

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)".

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

- *func_expr*::=

5.13.1 Examples

Consider the query `tkdata51_q0` and the input relation `tkdata51_S0`. Relation `tkdata51_S0` has schema `(c1 integer, c2 float)`. The query returns the relation.

```
<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>
```

```
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
```

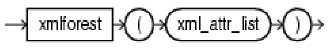
```
Timestamp   Tuple Kind  Tuple
     1000:  +           <S0>
                            <c1>10</c1>
                            <c2>0.1</c2>
```

```
                        </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>
```

5.14 *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*::=**



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).

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

- *func_expr*::=

5.14.1 Examples

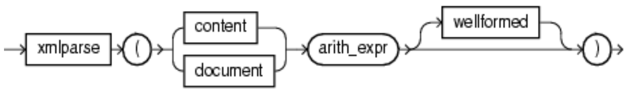
Consider the query `tkdata52_q0`and the input relation `tkdata52_S0`. Relation `tkdata52_S0` has schema `(c1 integer, c2 float)`. The query returns the relation.

```
<query id="tkdata52_q0"><![CDATA[
    select
        XMLFOREST( tkdata52_S0.c1, tkdata52_S0.c2)
    from
        tkdata52_S0 [range 1]
]]></query>

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

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>
```

5.15 *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*::=)

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
- If you specify `document`, then *arith_expr* must resolve to a singly rooted XML document. For an example, see xml_parse_expr Content Example.
- 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.

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).

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

- *func_expr*::=

5.15.1 Examples

This section describes the following xml_parse_expr examples:

- xml_parse_expr Content Example
- xml_parse_expr Document Example
- xml_parse_expr Wellformed Example

xml_parse_expr Content Example

Consider the query tkdata62_q3 and the input relation tkdata62_S1. Relation tkdata62_S1 has schema (c1 char(30)). The query returns the relation.

```
<query id="tkdata62_q3"><![CDATA[
    select XMLPARSE(CONTENT c1) from tkdata62_S1
]]></query>

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>"

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 and the input relation tkdata62_S1. Relation tkdata62_S1 has schema (c1 char(30)). The query returns the relation.

```
<query id="tkdata62_q4"><![CDATA[
    select XMLPARSE(DOCUMENT c1) from tkdata62_S1
]]></query>

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>"

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` and the input relation `tkdata62_S`. Relation `tkdata62_S` has schema `(c char(30))`. The query returns the relation.

```
<query id="tkdata62_q2"><![CDATA[
    select XMLPARSE(DOCUMENT c WELLFORMED) from tkdata62_S
]]></query>

Timestamp               Tuple
1000                    "<a>3</a>"
1000                    "<e3>blaaaaa</e3>"
1000                    "<r>4</r>"
1000                    "<a/>"
1003                    "<a>s3</a>"
1004                    "<d>b6</d>"

Timestamp   Tuple Kind  Tuple
1000:       +           <a>3</a>
1000:       +           <e3>blaaaaa</e3>
1000:       +           <r>4</r>
1000:       +           <a/>
1003:       +           <a>s3</a>
1004:       +           <d>b6</d>
```

6
Conditions

A reference to conditions in Oracle Continuous Query Language (Oracle CQL) is provided. A condition specifies a combination of one or more expressions and logical operators and returns a value of `TRUE`, `FALSE`, or `UNKNOWN`.

6.1 Introduction to Conditions

You must use appropriate condition syntax whenever *`condition`* appears in Oracle CQL statements.

You can use a condition in the `WHERE` clause of these statements:

- `SELECT`

You can use a condition in any of these clauses of the `SELECT` statement:

- `WHERE`
- `HAVING`

See Also:

Query.

A condition could be said to be of a logical data type.

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
```

6.1.1 Condition Precedence

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

Table 6-1 lists the levels of precedence among Oracle CQL condition from high to low. Conditions listed on the same line have the same precedence. As the table indicates, Oracle evaluates operators before conditions.

Table 6-1 Oracle CQL Condition Precedence

Type of Condition	Purpose
Oracle CQL operators are evaluated before Oracle CQL conditions	See What You May Need to Know About Operator Precedence .
`=, <>, <, >, <=, >=`	comparison
`IS NULL`, `IS NOT NULL`, `LIKE`, `BETWEEN`, `IN`, `NOT IN`	comparison
`NOT`	exponentiation, logical negation
`AND`	conjunction
`OR`	disjunction
`XOR`	disjunction

6.2 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 Event Processing uses numeric precedence to determine whether the condition compares `INTEGER`, `FLOAT`, or `BIGINT` values.

Two objects of nonscalar type are comparable if they are of the same named type and there is a one-to-one correspondence between their elements.

A comparison condition specifies a comparison with expressions or view results.

Table 6-2 lists comparison conditions.

Table 6-2 Comparison Conditions

Type of Condition	Purpose	Example

=	Equality test.	`<query id="Q1"><![CDATA[`
		`    SELECT *`
		`    FROM S0`
		`    WHERE salary = 2500`
		`]]></query>`

Table 6-2 (Cont.) Comparison Conditions

Type of Condition	Purpose	Example

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

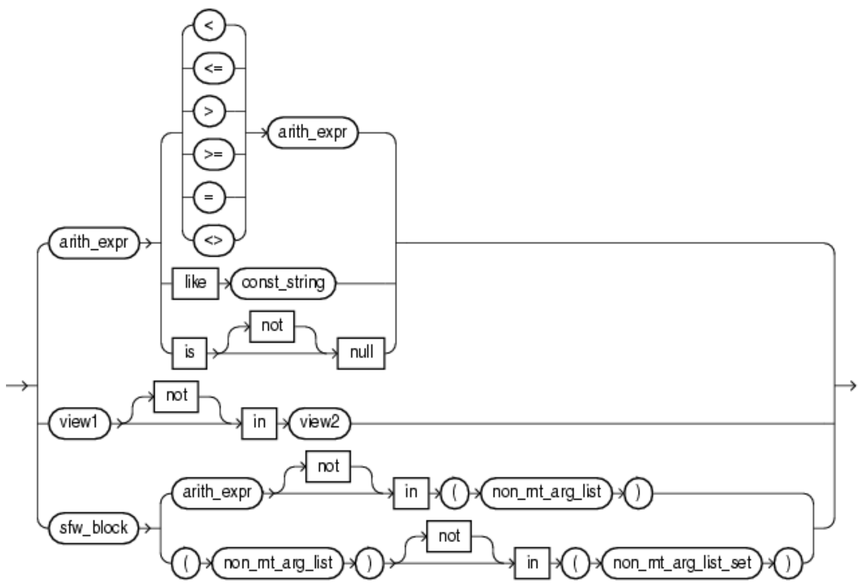
`is [not] null`	Null tests.
	
	For more information, see Null Conditions .
		`<query id="Q1"><![CDATA[`
		`    SELECT last_name`
		`  FROM S0`
		`  WHERE commission_pct`
		`  IS NULL`
		`]]></query>`
		
		`<query id="Q2"><![CDATA[`
		`    SELECT last_name`
		`  FROM S0`
		`  WHERE commission_pct`
		`  IS NOT NULL`
		`]]></query>`

Table 6-2 (Cont.) Comparison Conditions

Type of Condition	Purpose	Example

`[not] in`	Set and membership tests.
	
	For more information, see IN Condition.
		`<query id="Q1"><![CDATA[`
		`  SELECT * FROM S0`
		`  WHERE job_id NOT IN`
		`  ('PU_CLERK','SH_CLERK')`
		`]]></query>`
		`<view id="V1" schema="salary"><![CDATA[`
		`   SELECT salary`
		`   FROM S0`
		`   WHERE department_id = 30`
		`]]></view>`
		`<view id="V2" schema="salary"><![CDATA[`
		`   SELECT salary`
		`   FROM S0`
		`   WHERE department_id = 20`
		`]]></view>`
		`<query id="Q2"><![CDATA[`
		`  V1 IN V2`
		`]]></query>`

***condition*::=**



(*aggr_expr::=* and *non_mt_arg_list_set::=*.

6.3 Logical Conditions

A logical condition combines the results of two component conditions to produce a single result based on them or to invert the result of a single condition. Table 6-3 lists logical conditions.

Table 6-3 Logical Conditions

Type of Condition	Operation	Examples

`NOT`	Returns `TRUE` if the following condition is `FALSE`. Returns `FALSE` if it is `TRUE`. If it is `UNKNOWN`, then it remains `UNKNOWN`.	`<query id="Q1"><![CDATA[`
		`    SELECT *`
		`    FROM S0`
		`    WHERE NOT (job_id IS NULL)`
		`]]></query>`

`AND`	Returns `TRUE` if both component conditions are `TRUE`. Returns `FALSE` if either is `FALSE`. Otherwise returns `UNKNOWN`.	`<query id="Q1"><![CDATA[`
		`    SELECT *`
		`    FROM S0`
		`    WHERE job_id = 'PU_CLERK'`
		`    AND dept_id = 30`
		`]]></query>`

`OR`	Returns `TRUE` if either component condition is `TRUE`. Returns `FALSE` if both are `FALSE`. Otherwise returns `UNKNOWN`.	`<query id="Q1"><![CDATA[`
		`    SELECT *`
		`    FROM S0`
		`    WHERE job_id = 'PU_CLERK'`
		`    OR department_id = 10`
		`]]></query>`

`XOR`	Returns `TRUE` if either component condition is `TRUE`. Returns `FALSE` if both are `FALSE`. Otherwise returns `UNKNOWN`.	`<query id="Q1"><![CDATA[`
		`    SELECT *`
		`    FROM S0`
		`    WHERE job_id = 'PU_CLERK'`
		`    XOR department_id = 10`
		`]]></query>`

Table 6-4 shows the result of applying the `NOT` condition to an expression.

Table 6-4 NOT Truth Table

--	TRUE	FALSE	UNKNOWN
NOT	FALSE	TRUE	UNKNOWN

Table 6-5 shows the results of combining the `AND` condition to two expressions.

Table 6-5 AND Truth Table

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

For example, in the `WHERE` clause of the following `SELECT` statement, the `AND` logical condition returns values only when both `product.levelx is BRAND` and `v1.prodkey` equals `product.prodkey`:

```
<view id="v2" schema="region, dollars, month_"><![CDATA[
    select
```

```
        v1.region,
        v1.dollars,
        v1.month_
    from
        v1,
        product
    where
        product.levelx = "BRAND" and v1.prodkey = product.prodkey
]]></view>
```

Table 6-6 shows the results of applying `OR` to two expressions.

Table 6-6   OR Truth Table

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

For example, the following query returns the internal account identifier for `RBK` or `RBR` accounts with a risk of type 2:

```
<view id="ValidAccounts" schema="ACCT_INTRL_ID"><![CDATA[
    select ACCT_INTRL_ID from Acct
    where (
        ((MANTAS_ACCT_BUS_TYPE_CD = "RBK") OR (MANTAS_ACCT_BUS_TYPE_CD = "RBR")) AND
        (ACCT_EFCTV_RISK_NB != 2)
    )
]]></view>
```

Table 6-7 shows the results of applying `XOR` to two expressions.

Table 6-7   XOR Truth Table

XOR	TRUE	FALSE	UNKNOWN
TRUE	FALSE	TRUE	UNKNOWN
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN

For example, the following query returns `c1` and `c2` when `c1` is 15 and `c2` is 0.14 or when `c1` is 20 and `c2` is 100.1, but not both:

```
<query id="q6"><![CDATA[
    select
        S2.c1,
        S3.c2
    from
        S2[range 1000], S3[range 1000]
    where
        (S2.c1 = 15 and S3.c2 = 0.14) xor (S2.c1 = 20 and S3.c2 = 100.1)
]]></query>
```

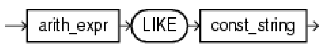
6.4 LIKE Condition

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.

The LIKE condition with the syntax of the comparison String supports % for 0 or more characters and - for any single character in coherence.

***like_condition*::=**



(*arith_expr*::=)

In this syntax:

- *arith_expr* is an arithmetic expression whose value is compared to *const_string*.
- *const_string* is a constant value regular expression to be compared against the *arith_expr*.

If any of *arith_expr* or *const_string* is null, then the result is unknown.

The *const_string* can contain any of the regular expression assertions and quantifiers that java.util.regex supports: that is, a regular expression that is specified in string form in a syntax similar to that used by Perl.

Table 6-8 describes the LIKE conditions.

Table 6-8 LIKE Conditions

Type of Condition	Operation	Example

x LIKE y	TRUE if *x* does match the pattern *y*, FALSE otherwise.	`<query id="q291"><![CDATA[`
		`    select * from SLk1 where first1`
		`like "^Ste(v|ph)en$"`
		`]]></query>`
		
		`<query id="q292"><![CDATA[`
		`    select * from SLk1 where first1`
		`like ".*intl.*"`
		`]]></query>`

See Also:

lk

For more information on Perl regular expressions, see http://perldoc.perl.org/perlre.html.

6.4.1 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.

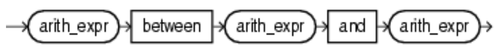
```
SMITHSTONIAN, SMITHY, SMITHS
```

For more information, see `http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html`.

6.5 Range Conditions

A range condition tests for inclusion in a range.

***between_condition*::=**



(*arith_expr*::=)

Table 6-9 describes the range conditions.

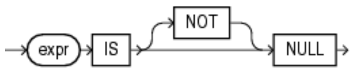
Table 6-9 Range Conditions

Type of Condition	Operation	Example
`BETWEEN x AND y`	Greater than or equal to *x* and less than or equal to *y*.	`<query id="Q1"><![CDATA[ SELECT * FROM S0 WHERE salary BETWEEN 2000 AND 3000 ]]></query>`

6.6 Null Conditions

A NULL condition tests for nulls. This is the only condition that you should use to test for nulls.

***null_conditions*::=**



(Expressions).

Table 6-10 lists the null conditions.

Table 6-10 Null Conditions

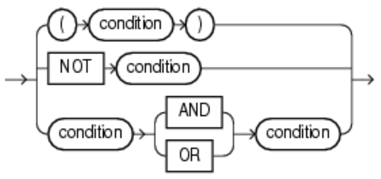
Type of Condition	Operation	Example

`IS [NOT] NULL`	Tests for nulls.
	See Also: 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>`

6.7 Compound Conditions

A compound condition specifies a combination of other conditions.

***compound_conditions*::=**



See Also:

Logical Conditions for more information about NOT, AND, and OR conditions.

6.8 IN Condition

You can use the `IN` and `NOT IN` condition in the following ways:

- *`in_condition_set`*: Using IN and NOT IN as a Membership Condition
- *`in_condition_membership`*: Using IN and NOT IN as a Membership Condition.

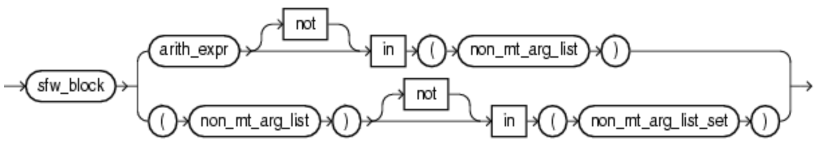
> **Note:**
>
> You cannot combine these two usages.

When using the `NOT IN` condition, be aware of the effect of null values as NOT IN and Null Values describes.

6.8.1 Using IN and NOT IN as a Membership Condition

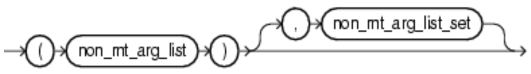
In this usage, the query will be a `SELECT-FROM-WHERE` query that either tests whether or not one argument is a member of a list of arguments of the same type or tests whether or not a list of arguments is a member of a set of similar lists.

***in_condition_membership*::=**



(*arith_expr*::= and *non_mt_arg_list_set*::=)

***non_mt_arg_list_set*::=**



When you use `IN` or `NOT IN` to test whether or not a *`non_mt_arg_list`* is a member of a set of similar lists, then you must use a *`non_mt_arg_list_set`*. Each *`non_mt_arg_list`* in the *`non_mt_arg_list_set`* must match the *`non_mt_arg_list`* to the left of the condition in number and type of arguments.

Consider the query `Q1` and the data stream `S0`. Stream `S0` has schema `(c1 integer, c2 integer)`. The following example shows the relation that the query returns. In Q1, the *`non_mt_arg_list_set`* is `((50,4),(4,5))`. Note that each *`non_mt_arg_list`* that it contains matches the number and type of arguments in the *`non_mt_arg_list`* to the left of the condition, `(c1, c2)`.

```
<query id="Q1"><![CDATA[
    select c1,c2 from S0[range 1] where (c1,c2) in ((50,4),(4,5))
]]></query>

Timestamp   Tuple
1000        50, 4
2000        30, 6
3000          , 5
4000        22,
h 200000000

Timestamp   Tuple Kind  Tuple
1000:       +           50,4
2000:       -           50,4
```

6.8.2 NOT IN and Null Values

If any item in the list following a `NOT IN` operation evaluates to null, then all stream elements evaluate to `FALSE` or `UNKNOWN`, and no rows are returned. For example, the following statement returns `c1` and `c2` if `c1` is neither 50 nor 30:

```
<query id="check_notin1"><![CDATA[
    select c1,c2 from S0[range 1]
    where
        c1 not in (50, 30)
]]></query>
```

However, the following statement returns no stream elements:

```
<query id="check_notin1"><![CDATA[
    select c1,c2 from S0[range 1]
    where
        c1 not in (50, 30, NULL)
]]></query>
```

The preceding example returns no stream elements because the `WHERE` clause condition evaluates to:

```
c1 != 50 AND c1 != 30 AND c1 != null
```

Because the third condition compares `c1` with a null, it results in an `UNKNOWN`, so the entire expression results in `FALSE` (for stream elements with `c1` equal to 50 or 30). This behavior can easily be overlooked, especially when the `NOT IN` operator references a view.

Moreover, if a `NOT IN` condition references a view that returns no stream elements at all, then all stream elements will be returned, as shown in the following example. Since V1 returns no stream elements at all, Q1 will return all V2 stream elements.

```
<view id="V1" schema="c1"><![CDATA[
    select * from S1[range 10 slide 10] where 1=2
]]></view>
<view id="V2" schema="c1"><![CDATA[
    select * from S1[range 10 slide 10] where c1=2
]]></view>
<query id="Q1"><![CDATA[
    V2 not in V1
]]></query>
```

7 Common Oracle CQL DDL Clauses

A reference to clauses in the data definition language (DDL) in Oracle Continuous Query Language (Oracle CQL) is provided.

7.1 Introduction to Common Oracle CQL DDL Clauses

Oracle CQL supports the following common DDL clauses:

- *array_type*
- *attr*
- *attrspec*
- *complex_type*
- *const_int*
- *const_string*
- *const_value*
- *identifier*
- *l-value*
- *methodname*
- *non_mt_arg_list*
- *non_mt_attr_list*
- *non_mt_attrname_list*
- *non_mt_attrspec_list*
- *non_mt_cond_list*
- *param_list*
- *qualified_type_name*
- *query_ref*
- *time_spec*
- *xml_attribute_list*
- *xml_attr_list*
- *xqryargs_list*.

For more information on Oracle CQL statements, see Oracle CQL Statements.

7.2 array_type

Purpose

Use the *`array_type`* clause to specify an Oracle CQL data cartridge type composed of a sequence of *complex_type* components, all of the same type.

Note:

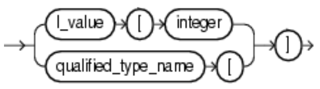
Oracle CQL supports single-dimension arrays only. That is, you can use `java.lang.String[]` but not `java.lang.String[][]`.

Prerequisites

None.

Syntax

array_type::=



Semantics

Array Declaration

You declare an array type using the *qualified_type_name* of the Oracle CQL data cartridge *complex_type*. Only arrays of *`complextype`* are supported: you cannot declare an array of Oracle CQL simple types unless there is an equivalent type defined in the Oracle CQL Java data cartridge.

Array Access

You access a *`complex_type`* array element by integer index. The index begins at 0 or 1 depending on the data cartridge implementation.

There is no support for the instantiation of new array type instances directly in Oracle CQL at the time you access an array. For example, the following is *not* allowed:

```
SELECT java.lang.String[10] ...
```

Examples

The following example shows how to create an event type as a Java class that specifies an event property as an array of Oracle CQL data cartridge complex type `MyClass` defined in package `com.mypackage`.

```
package com.myapplication.event;

import java.util.Date;

public final class MarketEvent {
    private final String symbol;
    private final Double price;
    private final com.mypackage.MyClass[] a1;

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

Array Declaration Example: Oracle CQL Simple Type

Only arrays of Oracle CQL data cartridge types are supported: you *cannot* declare an array of Oracle CQL simple types.

```
int[] a1
```

However, you can work around this by using the Oracle CQL Java data cartridge and referencing the Java equivalent of the simple type, if one exists:

```
int@java[] a1
```

For more information on the `@` syntax, see *link*::=.

Array Access Examples

The following example shows how to register the following queries that use Oracle CQL data cartridge complex type array access:

- View `v1` accesses the third element of the array `a1`. This array contains instances of Oracle CQL data cartridge complex type `com.mypackage.MyClass`.
- Query `q1` accesses the first element of the array `field1`. This array is defined on Oracle CQL data cartridge complex type `a1`.

```
<view id="v1" schema="symbol price a1"><![CDATA[
    IStream(select symbol, price, a1[3] from S1[range 10 slide 10])
]]></view>
<query id="q1"><![CDATA[
    SELECT a1.field1[1] …
]]></query>
```

7.3 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:

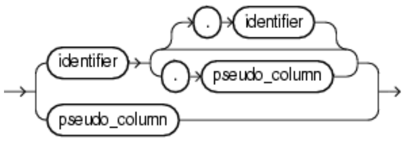
- *arith_expr*::=
- *order_expr*::=.

Prerequisites

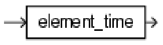
None.

Syntax

Figure 7-1   *attr*::=



identifier::= and Example 7-1.



Semantics

identifier

Specify the identifier of the stream element.

You can specify

- *`StreamOrViewName.ElementName`*
- *`ElementName`*
- *`CorrelationName.PseudoColumn`*
- *`PseudoColumn`*.

For examples, see Examples.

Example 7-1   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
- Using ELEMENT_TIME With SELECT
- Using ELEMENT_TIME With GROUP BY
- Using ELEMENT_TIME With PATTERN.

For more information, see Pseudocolumns.

Examples

Given the stream, valid attribute clauses are:

- `ItemTempStream.temp`
- `temp`
- `B.element_time`

- `element_time`

```
<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>
```

7.4 attrspec

Purpose

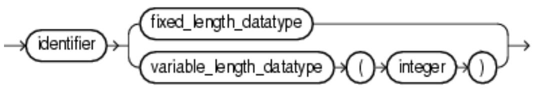
Use the `attrspec` clause to define the identifier and data type of a stream element.

Prerequisites

None.

Syntax

Figure 7-2 ***attrspec*::=**



(*fixed_length_datatype*::= and *variable_length_datatype*::=).

Semantics

identifier

Specify the identifier of the stream element.

fixed_length_datatype

Specify the stream element data type as a fixed-length data type.

For syntax, see *fixed_length_datatype*::= .

variable_length_datatype

Specify the stream element data type as a variable-length data type.

For syntax, see *variable_length_datatype*::=.

integer

Specify the length of the variable-length data type.

7.5 complex_type

Purpose

Use the `complex_type` clause to specify an Oracle CQL data cartridge type that defines:

- member fields (static or instance)
- member methods (static or instance)
- constructors.

The type of a field, and the return type and parameter list of a method may be complex types or simple types.

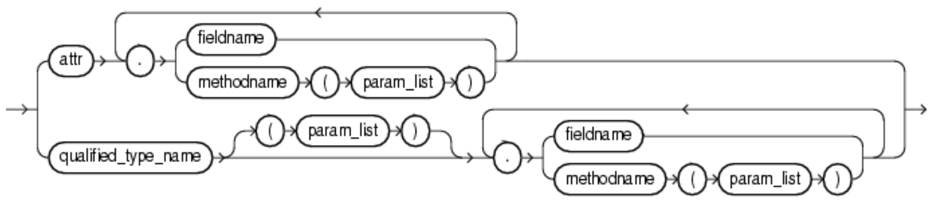
A complex type is identified by its qualified type name (set of identifiers separated by a period ".") and the optional name of the data cartridge to which it belongs (see *link*::=). If you do not specify a link name, then Oracle Event Processing assumes that the complex type is a Java class (that is, Oracle Event Processing assumes that the complex type belongs to the Java data cartridge).

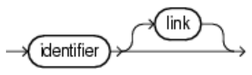
Prerequisites

The Oracle CQL data cartridge that provides the complextype must be loaded by Oracle Event Processing server at runtime.

Syntax

complex_type::=





(*link*::=)

Semantics

fieldname

Use the `fieldname` clause to specify a static field of an Oracle CQL data cartridge complex type.

Field Access

You cannot use a complex type `l-value` generated in expressions within an `ORDER BY` clause. Currently, only expressions within a `SELECT` clause and a `WHERE` clause may generate a complex type `l-value`.

You may access only a static field using *qualified_type_name*. To access a non-static field, you must first instantiate the complex type.

Method Access

Accessing complex type setter methods may cause side effects. Side effects decrease the opportunities for concurrency and sharing. For example, if you invoke a setter method and change the value of a view attribute (such as an event property) shared by different queries that depend on the view, then the query results may change as a side effect of your method invocation.

You may access only a static method using *qualified_type_name*. To access a non-static field, you must first instantiate the complex type.

Constructor Invocation

You may access only a static fields and static methods using *qualified_type_name*. To access a non-static field or non-static method, you must first instantiate the complex type by invoking one of its constructors.

Examples

Field Access Examples: *complex_type*

The following example shows how to register the following queries that use Oracle CQL data cartridge complex type field access:

- Query `q1` accesses field `myField` from Oracle CQL data cartridge complex type `a1`.
- Query `q2` accesses field `myField` defined on the Oracle CQL data cartridge complex type returned by the method `function-returning-object`.

  Query `q3` accesses field `myNestedField` defined on the Oracle CQL data cartridge complex type `myField` which is defined on Oracle CQL data cartridge complex type `a1`.
- Query `q4` accesses the static field `myStaticField` defined in the class `MyType` in package `com.myPackage`. Note that a link (`@myCartridge`) is necessary in the case of a static field.

```
<query id="q1"><![CDATA[
    SELECT a1.myField …
]]></query>
<query id="q2"><![CDATA[
    SELECT function-returning-object().myField …
]]></query>
<query id="q3"><![CDATA[
    SELECT a1.myField.myNestedField …
]]></query>
<query id="q4"><![CDATA[
    SELECT com.myPackage.MyType.myStaticField@myCartridge …
]]></query>
```

Method Access Examples: *complex_type*

The following example shows how to register the following queries that use Oracle CQL data cartridge complex type method access:

- Query `q1` accesses method `myMethod` defined on Oracle CQL data cartridge complex type `a1`. This query accesses the method with an empty parameter list.

- Query `q2` accesses method `myMethod` defined on Oracle CQL data cartridge complex type `a1` with a different signature than in query `q1`. In this case, the query accesses the method with a three-argument parameter list.
- Query `q3` accesses static method `myStaticMethod` defined on Oracle CQL data cartridge complex type `MyType`. This query accesses the method with a single parameter. Note that a link (`@myCartridge`) is necessary in the case of a static method.

```
<query id="q1"><![CDATA[
    SELECT a1.myMethod() …
]]></query>
<query id="q2"><![CDATA[
    SELECT a1.myMethod(a2, “foo", 10) …
]]></query>
<query id="q3"><![CDATA[
    SELECT myPackage.MyType.myStaticMethod@myCartridge("foo") …
]]></query>
```

Constructor Invocation Examples: *complex_type*

The following example shows how to register the following queries that use Oracle CQL data cartridge complex type constructor invocation:

- Query `q1` invokes the constructor `String` defined in package `java.lang`. In this case, the query invokes the constructor with an empty argument list.
- Query `q2` invokes the constructor `String` defined in package `java.lang`. In this case, the query invokes the constructor with a single argument parameter list and invokes the non-static method `substring` defined on the returned `String` instance.

```
<query id="q1"><![CDATA[
    SELECT java.lang.String() …
]]></query>
<query id="q2"><![CDATA[
    SELECT java.lang.String(“food").substring(0,1) …
]]></query>
```

7.6 *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*::=

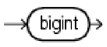
For more information, see Numeric Literals.

Prerequisites

None.

Syntax

const_bigint::=



7.7 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*::=
- *order_expr*::=

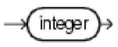
For more information, see Numeric Literals.

Prerequisites

None.

Syntax

const_int::=



7.8 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*::=
- *order_expr*::=
- *condition*::=
- Figure 7-4
- Figure 16-26
- Figure 16-31.

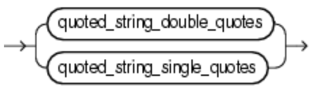
For more information, see Text Literals.

Prerequisites

None.

Syntax

Figure 7-3 *const_string*::=



7.9 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*::=
- *condition*::=

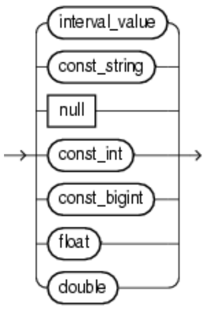
For more information, see Literals.

Prerequisites

None.

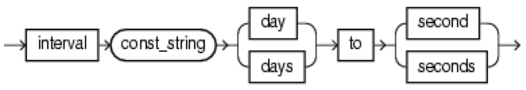
Syntax

Figure 7-4 ***const_value*::=**



Example 7-2.

Figure 7-5 interval_value



Example 7-2 ***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 Interval Literals.

const_string

Specify a quoted `String` constant value.

For more information, see Text Literals.

null

Specify a null constant value.

For more information, see Nulls.

const_int

Specify an `int` constant value.

For more information, see Numeric Literals.

bigint

Specify a `bigint` constant value.

For more information, see Numeric Literals.

float

Specify a float constant value.

For more information, see Numeric Literals.

7.10 *identifier*

Purpose

Use the *`identifier`* clause to reference an existing Oracle CQL schema object.

You can use the *`identifier`* clause in the following Oracle CQL statements:

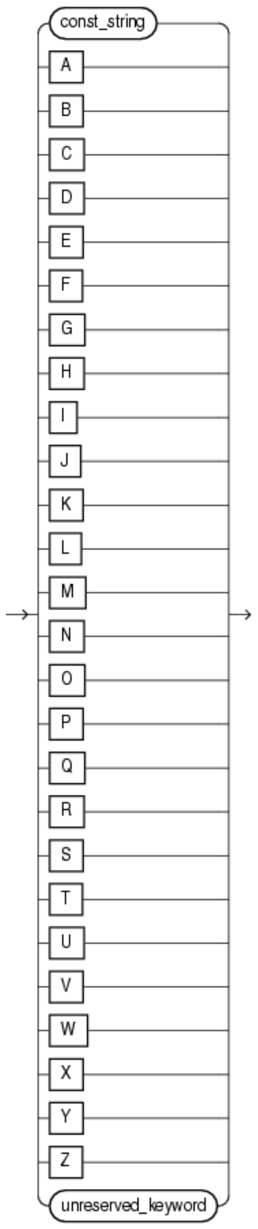
- Figure 16-20
- *aggr_expr*::=
- *func_expr*::=
- Figure 7-1
- Figure 7-2
- Figure 7-9
- Figure 7-8
- Figure 16-8
- *measure_column*::=
- Query
- Figure 16-5
- View.

Prerequisites

The schema object must already exist.

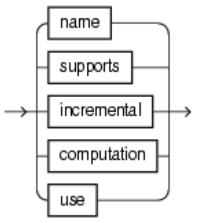
Syntax

Figure 7-6 ***identifier*::=**



const_string and Example 7-3.

Example 7-3 unreserved_keyword::=



Semantics

const_string

Specify the identifier as a String.

For more information, see Schema Object Naming Rules.

[A-Z]

Specify the identifier as a single uppercase letter.

unreserved_keyword

These are names that you may use as identifiers.

For more information, see:

- Schema Object Naming Rules.

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`.

7.11 *l-value*

Purpose

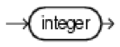
Use the `l-value` clause to specify an integer literal.

Prerequisites

None.

Syntax

l-value::=



7.12 *methodname*

Purpose

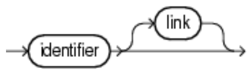
Use the `methodname` clause to specify a method of an Oracle CQL data cartridge complex type.

Prerequisites

None.

Syntax

methodname::=



(*link*::=)

7.13 *non_mt_arg_list*

Purpose

Use the `non_mt_arg_list` clause to specify one or more arguments as arithmetic expressions involving stream elements.

You can use the `non_mt_arg_list` clause in the following Oracle CQL statements:

- *decode*::=
- *func_expr*::=
- *condition*::=

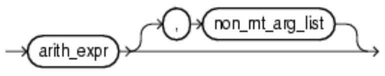
- *non_mt_arg_list_set*::=.

Prerequisites

If any stream elements are referenced, the stream must already exist.

Syntax

non_mt_arg_list::=



(*arith_expr*::=)

Semantics

arith_expr

Specify the arithmetic expression that resolves to the argument value.

7.14 *non_mt_attr_list*

Purpose

Use the `non_mt_attr_list` clause to specify one or more arguments as stream elements directly.

You can use the `non_mt_attr_list` clause in the following Oracle CQL statements:

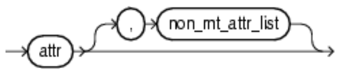
- Figure 15-6
- Figure 16-9
- Figure 16-12.

Prerequisites

If any stream elements are referenced, the stream must already exist.

Syntax

Figure 7-7 *non_mt_attr_list*::=



Semantics

attr

Specify the argument as a stream element directly.

7.15 *non_mt_attrname_list*

Purpose

Use the `non_mt_attrname_list` clause to one or more stream elements by name.

You can use the `non_mt_attrname_list` clause in the following Oracle CQL statements:

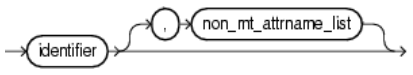
- View

Prerequisites

If any stream elements are referenced, the stream must already exist.

Syntax

Figure 7-8 *non_mt_attrname_list*::=



Semantics

identifier

Specify the stream element by name.

7.16 *non_mt_attrspec_list*

Purpose

Use the `non_mt_attrspec_list` clause to specify one or more attribute specifications that define the identifier and data type of stream elements.

You can use the `non_mt_attrspec_list` clause in the following Oracle CQL statements:

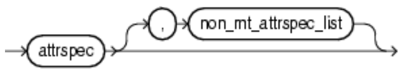
- View.

Prerequisites

If any stream elements are referenced, the stream must already exist.

Syntax

non_mt_attrspec_list::=



Semantics

attrspec

Specify the attribute identifier and data type.

7.17 *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:

- Figure 15-5
- *searched_case*::=
- Figure 16-11
- Figure 16-19.

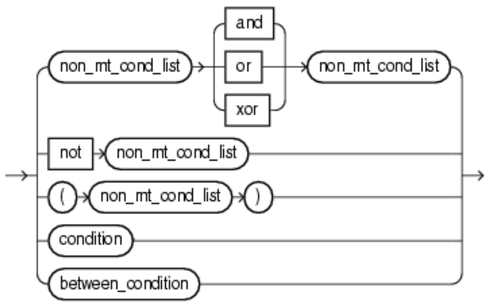
For more information, see Conditions.

Prerequisites

None.

Syntax

non_mt_cond_list::=



(*condition*::=, *between_condition*::=)

Semantics

condition

Specify a comparison condition.

For more information, see Comparison Conditions .

For syntax, see *condition*::=.

between_condition

Specify a condition that tests for inclusion in a range.

For more information, see Range Conditions .

For syntax, see *between_condition*::=.

7.18 *out_of_line_constraint*

Purpose

Use this *`out_of_line_constraint`* clause to restrict a tuple of any data type 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.

For more information, see:

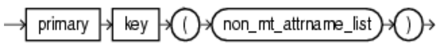
- Nulls.

Prerequisites

A tuple that you specify with an *`out_of_line_constraint`* may not contain a null value.

Syntax

out_of_line_constraint::=



Semantics

non_mt_attrname_list

Specify one or more tuples to restrict by a primary key integrity constraint.

7.19 *param_list*

Purpose

Use the *`param_list`* clause to specify a comma-separated list of zero or more parameters, similar to a function parameter list, for an Oracle CQL data cartridge complex type method or constructor.

You can use the *`param_list`* clause in the following Oracle CQL data cartridge statements:

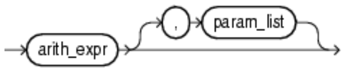
- *link*::=

Prerequisites

None.

Syntax

param_list::=



(*arith_expr*::=).

7.20 qualified_type_name

Purpose

Use the `qualified_type_name` clause to specify a fully specified type name of an Oracle CQL data cartridge complex type, for example `java.lang.String`. Use the `qualified_type_name` when invoking Oracle CQL data cartridge static fields, static methods, or constructors.

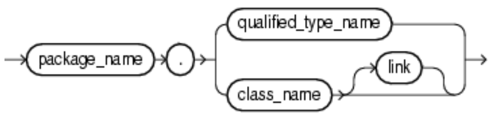
There is no default package. For example, using the Java data cartridge, you must specify `java.lang` when referencing the class `String`. To be able to distinguish a reserved word from a qualified type, all qualified types must have at least two identifiers, that is, there must be at least one period (`.`) in a qualified name.

Prerequisites

None.

Syntax

qualified_type_name::=



(*arith_expr*::= and *link*::=)

identifier

identifier

Semantics

package_name

Use the `package_name` clause to specify the name of an Oracle CQL data cartridge package.

class_name

Use the *class_name* clause to specify the name of an Oracle CQL data cartridge `Class`.

7.21 *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:

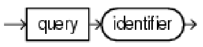
- View

Prerequisites

The query must already exist (see Query).

Syntax

Figure 7-9 *query_ref*::=



Semantics

identifier

Specify the name of the query. This is the name you use to reference the query in subsequent Oracle CQL statements.

7.22 *time_spec*

Purpose

Use the *time_spec* clause to define a time duration in days, hours, minutes, seconds, milliseconds, or nanoseconds.

Default: if units are not specified, Oracle Event Processing assumes `[second|seconds]`.

You can use the *time_spec* clause in the following Oracle CQL statements:

- Figure 15-9
- *windows_type* in Query Semantics

Prerequisites

None.

Syntax

Figure 7-10 *time_spec*::=

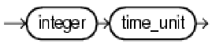
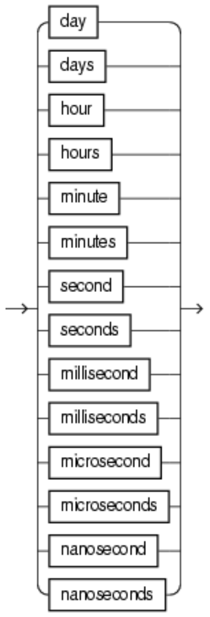


Figure 7-11 *time_unit*::=



Semantics

integer

Specify the number of time units.

time_unit

Specify the unit of time.

7.23 *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:

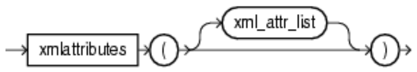
- *xmlelement_expr*.

Prerequisites

If any stream elements are referenced, the stream must already exist.

Syntax

xml_attribute_list::=



Semantics

xml_attr_list

Specify one or more XML attributes.

```
<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>
```

7.24 *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:

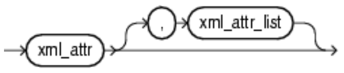
- *xml_attribute_list*
- *xmlforest_expr*
- *xml_agg_expr*.

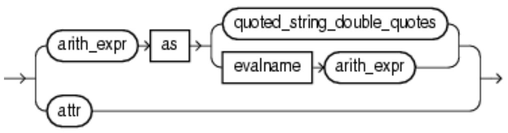
Prerequisites

If any stream elements are referenced, the stream must already exist.

Syntax

xml_attr_list::=





(*arith_expr*::=)

Semantics

xml_attr

Specify an XML attribute.

7.25 *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:

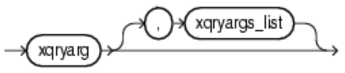
- xmlexists
- xmlquery.
- *func_expr*::=

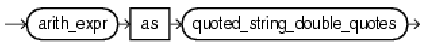
Prerequisites

If any stream elements are referenced, the stream must already exist.

Syntax

xqryargs_list::=





(*arith_expr*::=)

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*::=) to either a variable in a given XQuery or an XPath operator such as "." as a quoted string.

8

Built-In Single-Row Functions

A reference to single-row functions in Oracle Continuous Query Language (Oracle CQL) is provided. Single-row functions return a single result row for every row of a queried stream or view.

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

Table 8-1 lists the built-in single-row functions that Oracle CQL provides.

Table 8-1 Oracle CQL Built-in Single-Row Functions

Type	Function
Character (returning character values)	• concat
Character (returning numeric values)	• length
Datetime	• systimestamp
Conversion	• to_bigint • to_boolean • to_char • to_double • to_float • to_timestamp
XML and SQLX	• xmlcomment • xmlconcat • xmlexists • xmlquery
Encoding and Decoding	• hextoraw • rawtohex
Null-related	• nvl
Pattern Matching	• lk • prev

Note:

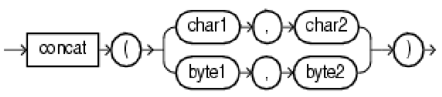
Built-in function names are case sensitive and you must use them in the case shown (in lower case).

Note:

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

8.2.1 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 data type depends on the data types of the arguments.

Using `concat`, you can concatenate any combination of character, byte, and numeric data types. The `concat` performs automatic numeric to string conversion.

This function is equivalent to the concatenation operator (||).

To concatenate `xmltype` arguments, use `xmlconcat`. For more information, see xmlconcat.

Examples

concat Function

Consider the query `chr_concat` in concat and data stream `S4` in concat. Stream `S4` has schema `(c1 char(10))`. The query returns the relation in concat.

Example 8-1 concat Function Query

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

Example 8-2 concat Function Stream Input

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

Example 8-3 concat Function Relation Output

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

Concatenation Operator (||)

Consider the query `q264` in Example 8–4 and the data stream `S10` in Example 8–5. Stream `S10` has schema `(c1 integer, c2 char(10))`. The query returns the relation in Example 8–6.

Example 8-4 Concatenation Operator (||) Query

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

Example 8-5 Concatenation Operator (||) Stream Input

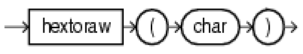
```
Timestamp Tuple
1 1,abc
2 2,ab
3 3,abc
4 4,a
h 200000000
```

Example 8-6 Concatenation Operator (||) Relation Output

```
Timestamp Tuple Kind Tuple
1: + abcxyz
2: + abxyz
3: + abcxyz
4: + axyz
```

8.2.2 hextoraw

Syntax



Purpose

`hextoraw` converts *`char`* containing hexadecimal digits in the `char` character set to a raw value.

See Also:

rawtohex.

Examples

Consider the query `q6` and the data stream `SByt`. Stream `SByt` has schema `(c1 integer, c2 char(10))`. The query returns the relation.

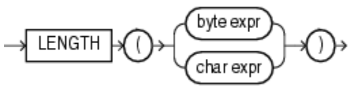
```
<query id="q6"><![CDATA[
    select * from SByt[range 2]
    where
        hextoraw(c2) between and hextoraw("5600")
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
3000        +           3,"53aa"
5000        -           3,"53aa"
5000        +            ,"55ef"
7000        -            ,"55ef"
```

8.2.3 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

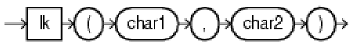
Consider the query `chr_len` and the data stream `S2`. Stream `S2` has schema `(c1 char(10), c2 integer)`. The query returns the relation.

```
<query id="chr_len"><![CDATA[
    select length(c1) from S2[range 5]
]]></query>
```

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

Timestamp   Tuple Kind  Tuple
 1000:      +           0
 2000:      +           2
 6000:      -           0
 7000:      -           2
 8000:      +           3
 9000:      +           0
13000:      -           3
14000:      -           0
15000:      +           3
20000:      -           3
```

8.2.4 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 .

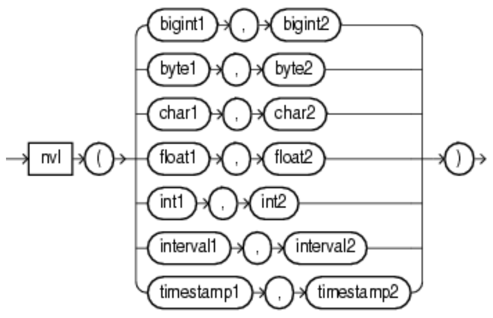
Examples

Consider the query `q291` and the data stream `SLk1`. Stream `SLk1` has schema `(first1 char(20), last1 char(20))`. The query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
1:          +           Steven,King
3:          +           Stephen,Stiles
4:          +           Steven,Markles
```

8.2.5 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 data type. If their data types are different, then Oracle Event Processing implicitly converts one to the other. If they cannot be converted implicitly, Oracle Event Processing returns an error. The implicit conversion is implemented as follows:

- If *`expr1`* is character data, then Oracle Event Processing converts *`expr2`* to character data before comparing them and returns `VARCHAR2` in the character set of *`expr1`*.
- If *`expr1`* is numeric, then Oracle Event Processing determines which argument has the highest numeric precedence, implicitly converts the other argument to that data type, and returns that data type.

Examples

Consider the query `q281` and the data stream `SNVL`. Stream `SNVL` has schema `(c1 char(20), c2 integer)`. The query returns the relation.

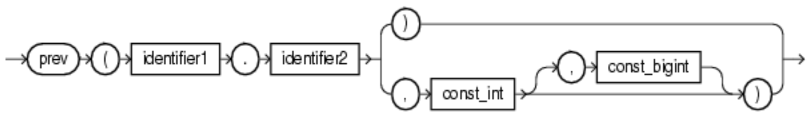
```
<query id="q281"><![CDATA[
    select nvl(c1,"abcd") from SNVL
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
1:          +           abcd
2:          +           ab
3:          +           abc
4:          +           abcd
```

8.2.6 prev

Syntax



Purpose

`prev` returns the value of the stream attribute (function argument `identifier2`) of the event that occurred previous to the current event and which belongs to the partition to which the current event belongs. It evaluates to `NULL` if there is no such previous event.

The type of the specified stream element may be any of:

- `integer`
- `bigint`
- `float`
- `double`
- `byte`
- `char`
- `interval`
- `timestamp`.

The return type of this function depends on the type of the specified stream attribute (function argument `identifier2`).

Where:

- `identifier1.identifier2` : `identifier1` is the name of a correlation variable used in the `PATTERN` clause and defined in the `DEFINE` clause and `identifier2` is the name of a stream attribute whose value in the previous event should be returned by `prev`.
- `const_int`: if this argument has a value *n*, then it specifies the *n*th previous event in the partition to which the current event belongs. The value of the attribute (specified in argument `identifier2`) in the *n*th previous event will be returned if such an event exists, `NULL` otherwise.
- `const_bigint`: specifies a time range duration in nanoseconds and should be used if you are interested in previous events that occurred only within a certain range of time before the current event.

If the query uses `PARTITION BY` with the `prev` function and input data will include many different partition key values (meaning many partitions), then total memory consumed for storing the previous event(s) per partition could be large. In such cases, consider using the time range duration (the third argument, possibly with a large range value) so that this memory can be reclaimed wherever possible.

Examples

prev(identifier1.identifier2)

Consider query `q2` and the data stream `S1`. Stream `S1` has schema `(c1 integer)`. This example defines pattern `A` as `A.c1 = prev(A.c1)`. In other words, pattern `A` matches when the value of `c1` in the current stream element matches the value of `c1` in the stream element immediately before the current stream element. The query returns the stream.

```
<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>
```

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

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

prev(identifier1.identifier2, const_int)

Consider query `q35` and the data stream `S15`. Stream `S15` has schema `(c1 integer, c2 integer)`. This example defines pattern `A` as `A.c1 = prev(A.c1,3)`. In other words, pattern `A` matches when the value of `c1` in the current stream element matches the value of `c1` in the third stream element before the current stream element. The query returns the stream.

```
<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>
```

```
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

Timestamp   Tuple Kind  Tuple
 4000:       +          45
 5000:       +          45
 6000:       +          45
 7000:       +          45
 8000:       +          45
12000:       +          43
13000:       +          52
15000:       +          43
```

prev(identifier1.identifier2, const_int, const_bigint)

Consider query `q36` and the data stream `S15`. Stream `S15` has schema `(c1 integer, c2 integer)`. This example defines pattern `A` as `A.c1 = prev(A.c1,3,5000000000L)`. In other words, pattern `A` matches when:

- the value of `c1` in the current event equals the value of `c1` in the third previous event of the partition to which the current event belongs, and
- the difference between the timestamp of the current event and that third previous event is less than or equal to `5000000000L` nanoseconds.

The query returns the output stream. Notice that in the output stream, there is no output at `8000`. The following example shows the contents of the partition (partitioned by the value of the `c2` attribute) to which the event at `8000` belongs.

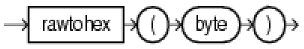
```
Timestamp   Tuple
1000        45,20
6000        45,20
7000        45,20
8000        45,20
```

As the following example shows, even though the value of `c1` in the third previous event (the event at `1000`) is the same as the value `c1` in the current event (the event at `8000`), the range condition is not satisfied. This is because the difference in the timestamps of these two events is more than `5000000000` nanoseconds. So it is treated as if there is no previous tuple and `prev` returns `NULL` so the condition fails to match.

```
<query id="q36"><![CDATA[
    select T.Ac1 from S15
    MATCH_RECOGNIZE (
        PARTITION BY
            c2
        MEASURES
            A.c1 as Ac1
        PATTERN(A)
        DEFINE
            A as (A.c1 = prev(A.c1,3,5000000000L) )
    ) as T
]]></query>

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

Timestamp   Tuple Kind  Tuple
5000:       +           45
```

8.2.7 rawtohex

Syntax



Purpose

`rawtohex` converts *`byte`* containing a raw value to hexadecimal digits in the `CHAR` character set.

See Also:

hextoraw.

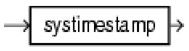
Examples

Consider the query `byte_to_hex` and the data stream `S5`. 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 hexadecimal digits in the character set of your current locale. The query returns the relation.

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

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

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:      -
```

8.2.8 systimestamp

Syntax



Purpose

`systimestamp` returns the system date, including fractional seconds and time zone, of the system on which the Oracle Event Processing server resides. The return type is `TIMESTAMP WITH TIME ZONE`.

Examples

Consider the query `q106` and the data stream `S0`. Stream `S0` has schema `(c1 float, c2 integer)`. The query returns the relation.

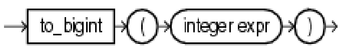
```
<query id="q106"><![CDATA[
    select * from S0
    where
        case c2
            when 10 then null
            when 20 then null
            else systimestamp()
        end > "07/06/2007 14:13:33"
]]></query>
```

```
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
```

```
Timestamp   Tuple Kind  Tuple
     1002:  +           0.14,15
   400000:  +           0.3 ,30
   500000:  +           0.3 ,35
   600000:  +               ,35
100000000:  +           4.04,40
```

8.2.9 to_bigint

Syntax



Purpose

Input/Output Types

The input/output types for this function are as follows:

Input Type	Output Type
INTEGER	BIGINT
TIMESTAMP	BIGINT
CHAR	BIGINT

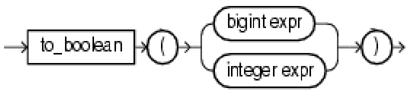
Examples

Consider the query `q282` and the data stream `S11`. Stream `S11` has schema `(c1 integer, name char(10))`. The query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           1
2000:       +           5.2
3400:       +           3
4700:       +           5.2
```

8.2.10 to_boolean

Syntax



Purpose

`to_boolean` returns a value of `true` or `false` for its `bigint` or `integer` expression argument.

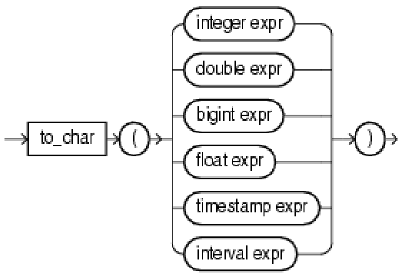
Examples

Consider the query `q282` and the data stream `S11`. Stream `S11` has schema `(c1 integer, name char(10))`. The query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
1000        +         true,-2147483648
1000        -         true,-2147483648
4000        +         true,-12345678901
4000        -         true,-12345678901
7000        +         ,
7000        -         ,
```

8.2.11 to_char

Syntax



Purpose

`to_char` returns a `char` value for its `integer`, `double`, `bigint`, `float`, `timestamp`, or `interval` expression argument. If the `bigint` argument exceeds the `char` precision, Oracle Event Processing returns an error.

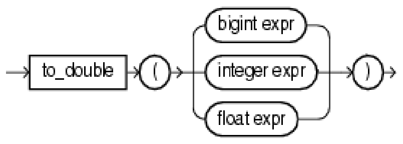
Examples

Consider the query `q282` and the data stream `S11`. Stream `S11` has schema `(c1 integer, name char(10))`. The query returns the relation.

```
<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>

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

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

8.2.12 to_double

Syntax



Purpose

`to_double` returns a `double` value for its `bigint`, `integer`, or `float` expression argument. If the `bigint` argument exceeds the `double` precision, Oracle Event Processing returns an error.

Examples

Consider the query `q282` and the data stream `S11`. Stream `S11` has schema `(c1 integer, name char(10))`. The query returns the relation.

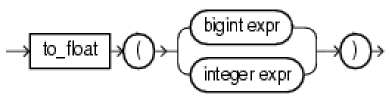
```
<query id="q282"><![CDATA[
    select nvl(to_double(c1), 5.2) from S11
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           1
2000:       +           5.2
3400:       +           3
4700:       +           5.2
```

8.2.13 to_float

Syntax



Purpose

`to_float` returns a `float` number equivalent of its `bigint` or `integer` argument. If the `bigint` argument exceeds the `float` precision, Oracle Event Processing returns an error.

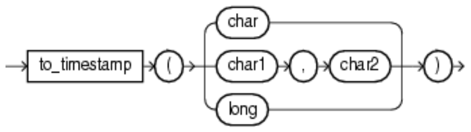
Examples

Consider the query q1 and the data stream S11. Stream S1 has schema (c1 integer, name char(10)). The query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
10:+ 1.02000:+ 5.23400:+ 3.04700:+ 5.2
```

8.2.14 to_timestamp

Syntax



Purpose

to_timestamp converts char literals that conform to java.text.SimpleDateFormat format models to timestamp data types. 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 data type.
- 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 data type.
- long: this form of the to_timestamp function converts a single long argument that represents the number of nanoseconds since the standard base time known as "the epoch", namely January 1, 1970, 00:00:00 GMT, into the corresponding timestamp data type represented as a number in milliseconds since "the epoch" with a date format that conforms to the default java.text.SimpleDateFormat format model (MM/dd/yyyy HH:mm:ss).

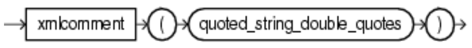
Examples

Consider the query q277 and the data stream STs2. Stream STs2 has schema (c1 integer, c2 char(20)). The query returns the relation.

```
<query id="q277"><![CDATA[
    select * from STs2
    where
        to_timestamp(c2,"yyMMddHHmmss") = to_timestamp("09/07/2005 10:13:48")
]]></query>

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

Timestamp   Tuple Kind  Tuple
2:          +           2,050907101348
```

8.2.15 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.

Examples

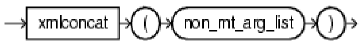
Consider the query `tkdata64_q1` and data stream `tkdata64_S`. Stream `tkdata64_S` has schema `(c1 char(30))`. The query returns the relation.

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

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

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-->
```

8.2.16 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 .

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 .

To concatenate data types other than `xmltype`, use `CONCAT`. For more information, see concat.

Examples

Consider the query `tkdata64_q1` and the data stream `tkdata64_S`. Stream `tkdata64_S` has schema `(c1 char(30))`. The query returns the relation.

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

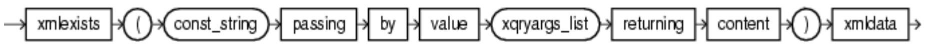
```
Timestamp   Tuple
c 30
1000        "san jose"
1000        "mountain view"
1000
1000        "sunnyvale"
1003
1004        "belmont"
```

```
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-->
```

8.2.17 xmlexists

Syntax



Purpose

`xmlexists` creates a continuous query against a stream of XML data to return a `boolean` that indicates whether or not the XML data satisfies the XQuery you specify.

This function takes the following arguments:

- `const_string`: An XQuery that Oracle Event Processing applies to the XML stream element data that you bind in `xqryargs_list`. For more information, see .
- `xqryargs_list`: A list of one or more bindings between stream elements and XQuery variables or XPath operators. For more information, see .

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 .

See Also:

- xmlquery.

Examples

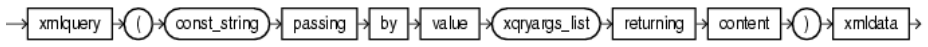
Consider the query `q1` and the XML data stream `S`. Stream `S` has schema `(c1 integer, c2 xmltype)`. In this example, the value of stream element `c2` is bound to the current node (`"."`) and the value of stream element `c1 + 1` is bound to XQuery variable `x`. The query returns the relation.

```
<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>

Timestamp   Tuple
3           1, "<PDRecord><PDName>hello</PDName></PDRecord>"
4           2, "<PDRecord><PDName>hello</PDName><PDName>hello1</PDName></PDRecord>"
```

```
5           3, "<PDRecord><PDId>4</PDId><PDName>hello1</PDName></PDRecord>"
6           4, "<PDRecord><PDId>46</PDId><PDName>hello2</PDName></PDRecord>"

Timestamp   Tuple Kind  Tuple
3:          +           false
4:          +           false
5:          +           true
6:          +           false
```

8.2.18 xmlquery

Syntax



Purpose

xmlquery creates a continuous query against a stream of XML data to return the XML data that satisfies the XQuery you specify.

This function takes the following arguments:

- const_string: An XQuery that Oracle Event Processing applies to the XML stream element data that you bind in xqryargs_list. For more information, see .
- xqryargs_list: A list of one or more bindings between stream elements and XQuery variables or XPath operators. For more information, see .

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 .

See Also:

- xmlexists

Examples

Consider the query and the XML data stream S. 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.

```
<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>

Timestamp   Tuple
3           1, "<PDRecord><PDName>hello</PDName></PDRecord>"
4           2, "<PDRecord><PDName>hello</PDName><PDName>hello1</PDName></PDRecord>"
5           3, "<PDRecord><PDId>4</PDId><PDName>hello1</PDName></PDRecord>"
6           4, "<PDRecord><PDId>46</PDId><PDName>hello0</PDName></PDRecord>"
7           5, "<PDRecord><PDId>5</PDId><PDName>hello2</PDName></PDRecord>"

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

9

Built-In Aggregate Functions

A reference to built-in aggregate functions included in Oracle Continuous Query Language (Oracle CQL) is provided. Built-in aggregate functions perform a summary operation on all the values that a query returns.

9.1 Introduction to Oracle CQL Built-In Aggregate Functions

Table 9-1 lists the built-in aggregate functions that Oracle CQL provides:

Table 9-1 Oracle CQL Built-in Aggregate Functions

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

Specify `distinct` if you want Oracle Event Processing to return only one copy of each set of duplicate tuples selected. Duplicate tuples are those with matching values for each expression in the select list. For more information, see .

Oracle Event Processing does not support nested aggregations.

Note:

Built-in function names are case sensitive and you must use them in the case shown (in lower case).

Note:

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

For more information, see:

- Built-In Aggregate Functions and the Where_ Group By_ and Having Clauses

9.1.1 Built-In Aggregate Functions and the Where, Group By, and Having Clauses

In Oracle CQL, the `where` clause is applied before the `group by` and `having` clauses. This means the Oracle CQL statement is invalid:

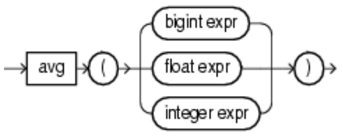
```
<query id="q1"><![CDATA[
    select * from InputChanel[rows 4 slide 4] as ic where count(*) = 4
]]></query>
```

Instead, you must use the Oracle CQL statement:

```
<query id="q1"><![CDATA[
    select * from InputChanel[rows 4 slide 4] as ic where count(*) = 4
]]></query>
```

9.2.1 avg

Syntax



Purpose

`avg` returns average value of *`expr`*.

This function takes as an argument any `bigint`, `float`, or `int` data type. The function returns a `float` regardless of the numeric data type of the argument.

Input/Output Types

The following tables lists the input types and the corresponding output types:

Input Type	Output Type
INT	FLOAT
BIGINT	FLOAT
FLOAT	FLOAT
DOUBLE	DOUBLE
BIGDECIMAL	BIGDECIMAL

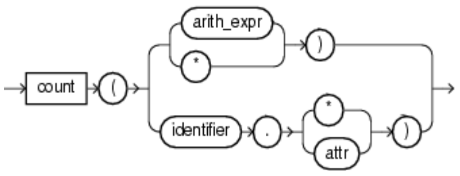
Examples

Consider the query `float_avg` and the data stream `S3`. Stream `S3` has schema `(c1 float)`. The query returns the relation. Note that the `avg` function returns a result of `NaN` if the average value is not a number.

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

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

Timestamp   Tuple Kind  Tuple
 1000:      -
 1000:      +            0.0
 2000:      -            0.0
 2000:      +            5.5
 6000:      -            5.5
 6000:      +            5.5
 7000:      -            5.5
 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:      +
```

9.2.2 count

Syntax



Purpose

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

The return value depends on the argument as Table 9-2 shows.

Table 9-2 Return Values for COUNT Aggregate Function

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

`count` never returns null.

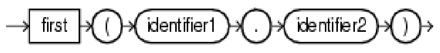
Example

Consider the query `q2` and the data stream `S2`. Stream `S2` has schema `(c1 integer, c2 integer)`. The query returns the relation.

```
<query id="q2"><![CDATA[
    SELECT COUNT(c2), COUNT(*) FROM  S2 [RANGE 10]
]]></query>

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

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
6000:                    +           3,4
```

9.2.3 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.

Examples

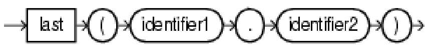
Consider the query `q9` and the data stream `S0`. Stream `S0` has schema `(c1 integer, c2 float)`. This example defines pattern `C` as `C.c1 = 7`. It defines `firstc` as `first(C.c2)`. In other words, `firstc` will equal the value of `c2` the first time `c1 = 7`. The query returns the relation.

```
<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>
```

```
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

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
```

9.2.4 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.

Examples

Consider the query `q9` and the data stream `S0`. 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.

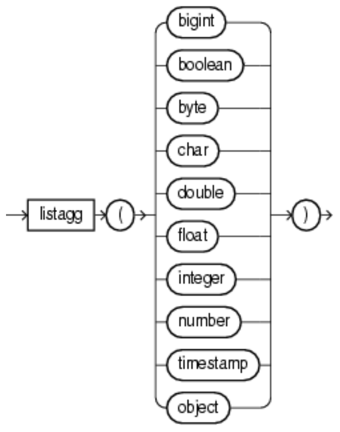
```
<query id="q9"><![CDATA[
    select
        T.firstc,
        T.lastc,
        T.Ac1,
        T.Bc1,
        T.avgCc1,
        T.Dc1
    from
        S0
    MATCH_RECOGNIZE (
        MEASURES
            first(C.c2) as firstc,
            last(C.c2) as lastc,
            avg(C.c1) as avgCc1,
            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>
```

```
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
```

```
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
```

9.2.5 listagg

Syntax



Purpose

`listagg` returns a `java.util.List` containing the Java equivalent of the function's argument.

Note that when a user-defined class is used as the function argument, the class must implement the `equals` method.

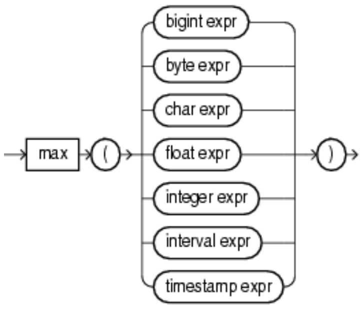
Examples

```
<view id="v1"><![CDATA[
    ISTREAM(select c1, listAgg(c3) as l1,
        java.util.LinkedHashSet(listAgg(c3)) as set1
    from S1
    group by c1)
]]></view>

<query id="q1"><![CDATA[
    select v1.l1.size(), v1.set1.size()
    from v1
]]></query>

Timestamp     Tuple
1000          orcl, 1, 15, 400
1000          msft, 1, 15, 400
2000          orcl, 2, 20, 300
2000          msft, 2, 20, 300
5000          orcl, 4, 5, 200
5000          msft, 4, 5, 200
7000          orcl, 3, 10, 100
7000          msft, 3, 20, 100
h 20000000

Timestamp   Tuple Kind   Tuple
 1000:          +        1,1
 1000:          +        1,1
 2000:          +        2,2
```

```
2000:        +       2,2
5000:        +       3,3
5000:        +       3,3
7000:        +       4,4
7000:        +       4,3
```

9.2.6 max

Syntax



Purpose

`max` returns maximum value of *`expr`*. Its data type depends on the data type of the argument.

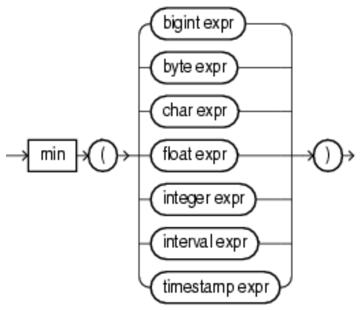
Examples

Consider the query `test_max_timestamp`and the data stream `S15` . Stream `S15` has schema `(c1 int, c2 timestamp)`. The query returns the relation.

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

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

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:       +
```

9.2.7 min

Syntax



Purpose

`min` returns minimum value of *`expr`*. Its data type depends on the data type of its argument.

Examples

Consider the query `test_min_timestamp` and the data stream `S15`. Stream `S15` has schema `(c1 int, c2 timestamp)`. The query returns the relation.

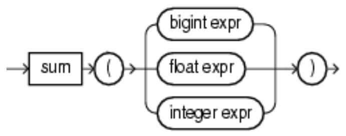
```
<query id="test_min_timestamp"><![CDATA[
    select min(c2) from S15[range 2]
]]></query>
```

```
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
```

```
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:       +
```

9.2.8 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 data type as the numeric data type of the argument.

Examples

Consider the query `q3` and the data stream `S1`. Stream `S1` has schema `(c1 integer, c2 bigint)`. The query returns the relation. For more information on `range`, see .

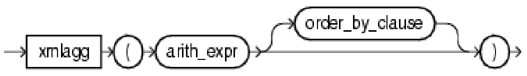
```
<query id="q3"><![CDATA[
    select sum(c2) from S1[range 5]
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
 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:      +
```

9.2.9 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.

Examples

xmlagg Function and the xmlelement Function

Consider the query tkdata67_q1 and the input relation. 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.

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

Timestamp           Tuple
 1000               15, 0.1
 1000               20, 0.14
 1000               15, 0.2
 4000               20, 0.3
10000               15, 0.04
h 12000

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 and the input relation. 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.

```
<query id="tkxmlAgg_q5"><![CDATA[
    select
        xmlagg(c2),
        xmlagg(c2 order by c1)
    from
        tkxmlAgg_S1[range 2]
]]></query>

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>"

Timestamp   Tuple Kind  Tuple
    0:      +
1000:       -
1000:       +           <a>hello</a>
                       ,<a>hello</a>
2000:       -           <a>hello</a>
                       ,<a>hello</a>
2000:       +           <a>hello</a>
                        <b>hello1</b>
                       ,<a>hello</a>
                        <b>hello1</b>
3000:       -           <a>hello</a>
                        <b>hello1</b>
                       ,<a>hello</a>
                        <b>hello1</b>
3000:       +           <b>hello1</b>
                        <PDRecord>
                          <PDName>hello</PDName>
                        </PDRecord>
                       ,<b>hello1</b>
                        <PDRecord>
                          <PDName>hello</PDName>
                        </PDRecord>
4000:       -           <b>hello1</b>
                        <PDRecord>
                          <PDName>hello</PDName>
                        </PDRecord>
                       ,<b>hello1</b>
                        <PDRecord>
                          <PDName>hello</PDName>
                        </PDRecord>
4000:       +           <PDRecord>
                          <PDName>hello</PDName>
                        </PDRecord>
                        <PDRecord>
                          <PDName>hello</PDName>
                          <PDName>hello1</PDName>
                        </PDRecord>
                       ,<PDRecord>
                          <PDName>hello</PDName>
                          <PDName>hello1</PDName>
                        </PDRecord>
                        <PDRecord>
                          <PDName>hello</PDName>
                        </PDRecord>
5000:       -           <PDRecord>
                          <PDName>hello</PDName>
                        </PDRecord>
                        <PDRecord>
                          <PDName>hello</PDName>
                          <PDName>hello1</PDName>
                        </PDRecord>
                       ,<PDRecord>
                          <PDName>hello</PDName>
                          <PDName>hello1</PDName>
                        </PDRecord>
                        <PDRecord>
                          <PDName>hello</PDName>
                        </PDRecord>
5000:       +           <PDRecord>
                          <PDName>hello</PDName>
                          <PDName>hello1</PDName>
                        </PDRecord>
                        <PDRecord>
                          <PDId>6</PDId>
                          <PDName>hello1</PDName>
                        </PDRecord>
                       ,<PDRecord>
                          <PDName>hello</PDName>
                          <PDName>hello1</PDName>
                        </PDRecord>
                        <PDRecord>
                          <PDId>6</PDId>
                          <PDName>hello1</PDName>
                        </PDRecord>
6000:       -           <PDRecord>
                          <PDName>hello</PDName>
                          <PDName>hello1</PDName>
                        </PDRecord>
                        <PDRecord>
                          <PDId>6</PDId>
                          <PDName>hello1</PDName>
                        </PDRecord>
                       ,<PDRecord>
                          <PDName>hello</PDName>
                          <PDName>hello1</PDName>
                        </PDRecord>
                        <PDRecord>
                          <PDId>6</PDId>
                          <PDName>hello1</PDName>
                        </PDRecord>
6000:       +           <PDRecord>
                          <PDId>6</PDId>
                          <PDName>hello1</PDName>
                        </PDRecord>
                        <PDRecord>
                          <PDId>46</PDId>
                          <PDName>hello2</PDName>
                        </PDRecord>
                       ,<PDRecord>
                          <PDId>46</PDId>
                          <PDName>hello2</PDName>
                        </PDRecord>
                        <PDRecord>
                          <PDId>6</PDId>
                          <PDName>hello1</PDName>
                        </PDRecord>
7000:       -           <PDRecord>
                          <PDId>6</PDId>
                          <PDName>hello1</PDName>
                        </PDRecord>
                        <PDRecord>
                          <PDId>46</PDId>
                          <PDName>hello2</PDName>
                        </PDRecord>
                       ,<PDRecord>
                          <PDId>46</PDId>
                          <PDName>hello2</PDName>
                        </PDRecord>
                        <PDRecord>
```

```
  <PDId>6</PDId>
  <PDName>hello1</PDName>
</PDRecord>
```

10
Colt Single-Row Functions

A reference to Colt single-row functions included in Oracle Continuous Query Language (Oracle CQL) is provided. Colt single-row functions are based on the Colt open source libraries for high performance scientific and technical computing.

For more information, see Functions.

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

Table 10-1 lists the built-in single-row Colt functions that Oracle CQL provides.

Table 10-1 Oracle CQL Built-in Single-Row Colt-Based Functions

Colt Package	Function
`cern.jet.math.Arithmetic` A set of basic polynomials, rounding, and calculus functions.	• binomial • binomial1 • ceil • factorial • floor • log • log2 • log10 • logFactorial • longFactorial • stirlingCorrection
`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 10-1 (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:

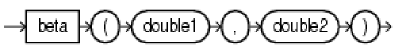
In stream input examples, lines beginning with `h` (such as `h 3800`) are heartbeat input tuples. These inform Oracle Event Processing that no further input will have a timestamp lesser than the heartbeat value.

For more information, see:

- Functions
- Data Types
- `http://dsd.lbl.gov/~hoschek/colt/`.

10.2.1 beta

Syntax



Purpose

`beta` is based on `cern.jet.stat.Gamma`. It returns the beta function (see Figure 10-1) of the input arguments as a `double`.

Figure 10-1   cern.jet.stat.Gamma beta

$$B(x, y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x + y)}$$

This function takes the following arguments:

- `double1`: the `x` value.
- `double2`: the `y` value.

For more information, see https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/random/Beta.html#Beta(double, double, cern.jet.random.engine.RandomEngine).

Examples

Consider the query `qColt28`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation .

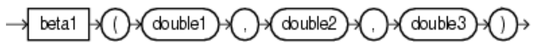
```
<query id="qColt28"><![CDATA[
    select beta(c2,c2) from SColtFunc
]]></query>
```

```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
```

```
1200        3,0.89,12
2000        8,0.4,4

Timestamp   Tuple Kind  Tuple
  10:       +           3.1415927
1000:       +           1.899038
1200:       +           1.251922
2000:       +           4.226169
```

10.2.2 beta1

Syntax



Purpose

`beta1` is based on `cern.jet.stat.Probability`. It returns the area `P(x)` from `0` to `x` under the beta density function (see Figure 10-2) as a `double`.

Figure 10-2 cern.jet.stat.Probability beta1

$$P(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \int_0^x t^{a-1} (1-t)^{b-1}\, dt$$

This function takes the following arguments:

- `double1`: the alpha parameter of the beta distribution `a`.
- `double2`: the beta parameter of the beta distribution `b`.
- `double3`: the integration end point `x`.

For more information, see https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/random/Beta.html#Beta(double, double, cern.jet.random.engine.RandomEngine).

Examples

Consider the query `qColt35`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

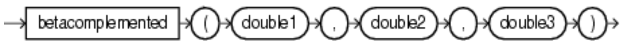
```
<query id="qColt35"><![CDATA[
    select beta1(c2,c2,c2) from SColtFunc
]]></query>
```

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.5
1000:       +           0.66235894
```

```
1200:       +           0.873397
2000:       +           0.44519535
```

10.2.3 betaComplemented

Syntax



Purpose

`betaComplemented` is based on `cern.jet.stat.Probability`. It returns the area under the right hand tail (from `x` to infinity) of the beta density function (see Figure 10-2) as a `double`.

This function takes the following arguments:

- `double1`: the alpha parameter of the beta distribution `a`.
- `double2`: the beta parameter of the beta distribution `b`.
- `double3`: the integration end point `x`.

For more information, see:

- incompleteBeta
- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#betaComplemented(double, double, double).

Examples

Consider the query `qColt37`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

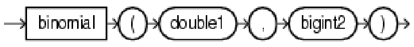
```
<query id="qColt37"><![CDATA[
    select betaComplemented(c2,c2,c2) from SColtFunc
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.5
1000:       +           0.66235894
1200:       +           0.873397
2000:       +           0.44519535
```

10.2.4 binomial

Syntax



Purpose

`binomial` is based on `cern.jet.math.Arithmetic`. It returns the binomial coefficient `n` over `k` (see Figure 10-3) as a `double`.

Figure 10-3 Definition of binomial coefficient

$$\frac{(n * n-1 \ldots * n-k+1)}{(1 * 2 * \ldots * k)}$$

This function takes the following arguments:

- `double1`: the `n` value.
- `long2`: the `k` value.

Table 10-2 lists the `binomial` function return values for various values of `k`.

Table 10-2 cern.jet.math.Arithmetic binomial Return Values

Arguments	Return Value
`k` < 0	0
`k` = 0	1
`k` = 1	`n`
Any other value of `k`	Computed binomial coefficient as given in Figure 10-3.

For more information, see https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#binomial(double, long).

Examples

Consider the query `qColt6`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 long)`, the query returns the relation.

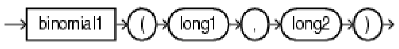
```
<query id="qColt6"><![CDATA[
    select binomial(c2,c3) from SColtFunc
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           -0.013092041
1000:       +           -0.012374863
1200:       +           -0.0010145549
2000:       +           -0.0416
```

10.2.5 binomial1

Syntax



Purpose

`binomial1` is based on `cern.jet.math.Arithmetic`. It returns the binomial coefficient `n` over `k` (see Figure 10-3) as a `double`.

This function takes the following arguments:

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

Table 10-3 lists the `BINOMIAL` function return values for various values of `k`.

Table 10-3 cern.jet.math.Arithmetic Binomial1 Return Values

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 10-3.

For more information, see https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/random/Binomial.html#Binomial(int, double, cern.jet.random.engine.RandomEngine).

Examples

Consider the query `qColt7`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 float, c3 long)`, the query returns the relation.

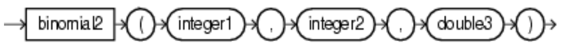
```
<query id="qColt7"><![CDATA[
    select binomial1(c3,c3) from SColtFunc
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           1.0
1000:       +           1.0
1200:       +           1.0
2000:       +           1.0
```

10.2.6 binomial2

Syntax



Purpose

`binomial2` is based on `cern.jet.stat.Probability`. It returns the sum of the terms 0 through `k` of the binomial probability density (see Figure 10-4) as a `double`.

Figure 10-4 cern.jet.stat.Probability binomial2

$$\sum_{j=0}^{k} \binom{n}{j} p^{j} (1-p)^{n-j}$$

This function takes the following arguments (all arguments must be positive):

- `integer1`: the end term `k`.
- `integer2`: the number of trials `n`.
- `double3`: the probability of success `p` in (0.0, 1.0).

For more information, see https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#binomial(int, int, double).

Examples

Consider the query `qColt34`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

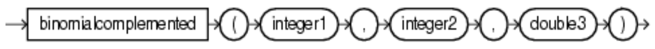
```
<query id="qColt34"><![CDATA[
    select binomial2(c1,c1,c2) from SColtFunc
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           1.0
1000:       +           1.0
1200:       +           1.0
2000:       +           1.0
```

10.2.7 binomialComplemented

Syntax



Purpose

`binomialComplemented` is based on `cern.jet.stat.Probability`. It returns the sum of the terms `k+1` through `n` of the binomial probability density (see Figure 10-5) as a `double`.

Figure 10-5 cern.jet.stat.Probability binomialComplemented

$$\sum_{j=k+1}^{n} \binom{n}{j} p^{j} (1-p)^{n-j}$$

This function takes the following arguments (all arguments must be positive):

- `integer1`: the end term `k`.
- `integer2`: the number of trials `n`.
- `double3`: the probability of success `p` in (0.0, 1.0).

For more information, see https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#binomialComplemented(int, int, double).

Examples

Consider the query `qColt38`. Given the data stream `SColtFunc` with schema `(integer, c2 double, c3 bigint)`, the query returns the relation.

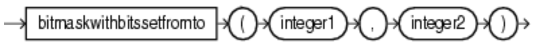
```
<query id="qColt38"><![CDATA[
    select binomialComplemented(c1,c1,c2) from SColtFunc
]]></query>
```

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

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

10.2.8 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:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/colt/bitvector/QuickBitVector.html#bitMaskWithBitsSetFromTo(int, int)
- leastSignificantBit
- mostSignificantBit.

Examples

Consider the query `qColt53`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 float, c3 bigint)`, the query returns the relation.

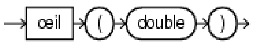
```
query id="qColt53"><![CDATA[
    select bitMaskWithBitsSetFromTo(c1,c1) from SColtFunc
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           2
1000:       +           16
1200:       +           8
2000:       +           256
```

10.2.9 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:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#ceil(double)
- ceil1.

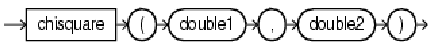
Examples

Consider the query `qColt1`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           1
1000:       +           1
1200:       +           1
2000:       +           1
```

10.2.10 chiSquare

Syntax



Purpose

`chiSquare` is based on `cern.jet.stat.Probability`. It returns the area under the left hand tail (from 0 to `x`) of the Chi square probability density function with `v` degrees of freedom (see Figure 10-6) as a `double`.

Figure 10-6 cern.jet.stat.Probability chiSquare

$$P(x \mid v) = \frac{1}{2^{\frac{v}{2}}\Gamma(\frac{v}{2})}\int_x^{\infty} t^{(\frac{v}{2})-1} e^{\frac{-t}{2}}\,dt$$

This function takes the following arguments (all arguments must be positive):

- `double1`: the degrees of freedom `v`.
- `double2`: the integration end point `x`.

For more information, see https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#chiSquare(double, double).

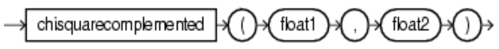
Examples

Consider the query `qColt39`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation .

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

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

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

10.2.11 chiSquareComplemented

Syntax



Purpose

`chiSquareComplemented` is based on `cern.jet.stat.Probability`. It returns the area under the right hand tail (from `x` to infinity) of the Chi square probability density function with `v` degrees of freedom (see Figure 10-6) as a `double`.

This function takes the following arguments (all arguments must be positive):

- `double1`: the degrees of freedom `v`.
- `double2`: the integration end point `x`.

For more information, see https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#chiSquareComplemented(double, double).

Examples

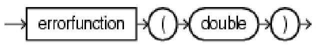
Consider the query `qColt40`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation .

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

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

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

10.2.12 errorFunction

Syntax



Purpose

`errorFunction` is based on `cern.jet.stat.Probability`. It returns the error function of the normal distribution of the `double` argument as a `double`, using the integral that Figure 10-7 shows.

Figure 10-7 cern.jet.stat.Probability errorFunction

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

For more information, see https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#errorFunction(double).

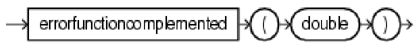
Examples

Consider the query `qColt41`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.5204999
1000:       +           0.6778012
1200:       +           0.79184324
2000:       +           0.42839235
```

10.2.13 errorFunctionComplemented

Syntax



Purpose

`errorFunctionComplemented` is based on `cern.jet.stat.Probability`. It returns the complementary error function of the normal distribution of the `double` argument as a `double`, using the integral that Figure 10-8 shows.

Figure 10-8 cern.jet.stat.Probability errorfunctioncompelemented

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

For more information, see https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#errorFunctionComplemented(double).

Examples

Consider the query `qColt42`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

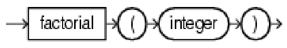
```
<query id="qColt42"><![CDATA[
    select errorFunctionComplemented(c2) from SColtFunc
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.47950011
1000:       +           0.3221988
1200:       +           0.20815676
2000:       +           0.57160765
```

10.2.14 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#factorial(int).

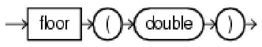
Examples

Consider the query `qColt8`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 float, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +             1.0
1000:       +            24.0
1200:       +             6.0
2000:       +         40320.0
```

10.2.15 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:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#floor(double)
- floor1

Examples

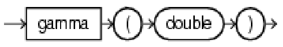
Consider the query `qColt2`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

Timestamp   Tuple
  10        1,0.5,8
```

```
1000        4,0.7,6
1200        3,0.89,12
2000        8,0.4,4

Timestamp   Tuple Kind  Tuple
  10:       +           0
1000:       +           0
1200:       +           0
2000:       +           0
```

10.2.16 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#gamma(double, double, double).

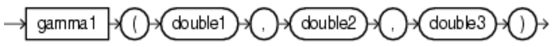
Examples

Consider the query `qColt29`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           1.7724539
1000:       +           1.2980554
1200:       +           1.0768307
2000:       +           2.2181594
```

10.2.17 gamma1

Syntax



Purpose

`gamma1` is based on `cern.jet.stat.Probability`. It returns the integral from zero to `x` of the gamma probability density function (see Figure 10-9) as a `double`.

Figure 10-9 cern.jet.stat.Probability gamma1

$$y = \frac{a^b}{\Gamma(b)} \int_0^x t^{b-1} e^{-at}\, dt$$

This function takes the following arguments:

- `double1`: the gamma distribution alpha value `a`
- `double2`: the gamma distribution beta or lambda value `b`
- `double3`: the integration end point `x`.

For more information, see https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/random/Gamma.html#Gamma(double, double, cern.jet.random.engine.RandomEngine).

Examples

Consider the query `qColt36`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

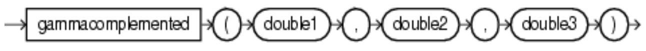
```
<query id="qColt36"><![CDATA[
    select gamma1(c2,c2,c2) from SColtFunc
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.5204999
1000:       +           0.55171627
1200:       +           0.59975785
2000:       +           0.51785487
```

10.2.18 gammaComplemented

Syntax



Purpose

`gammaComplemented` is based on `cern.jet.stat.Probability`. It returns the integral from `x` to infinity of the gamma probability density function (see Figure 10-10) as a `double`.

Figure 10-10 cern.jet.stat.Probability gammaComplemented

$$y = \frac{a^b}{\Gamma(b)} \int_x^\infty t^{b-1} e^{-at}\, dt$$

This function takes the following arguments:

- `double1`: the gamma distribution alpha value `a`
- `double2`: the gamma distribution beta or lambda value `b`
- `double3`: the integration end point `x`.

For more information, see https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#gammaComplemented(double, double, double).

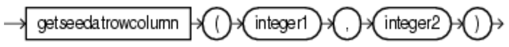
Examples

Consider the query `qColt43`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.47950011
1000:       +           0.44828376
1200:       +           0.40024218
2000:       +           0.48214513
```

10.2.19 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/random/engine/RandomSeedTable.html#getSeedAtRowColumn(int, int).

Examples

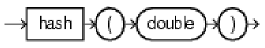
Consider the query `qColt27`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           253987020
1000:       +           1289741558
1200:       +           417696270
2000:       +           350557787
```

10.2.20 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/colt/map/HashFunctions.html#hash(double).

Examples

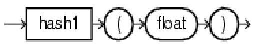
Consider the query `qColt56`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +            1071644672
1000:       +            1608935014
1200:       +            2146204385
2000:       +           -1613129319
```

10.2.21 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/colt/map/HashFunctions.html#hash(float).

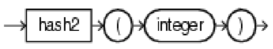
Examples

Consider the query `qColt57`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           1302214522
1000:       +           1306362078
1200:       +           1309462552
2000:       +           1300047248
```

10.2.22 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/colt/map/HashFunctions.html#hash(int).

Examples

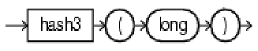
Consider the query `qColt58`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           1
```

```
1000:      +          4
1200:      +          3
2000:      +          8
```

10.2.23 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/colt/map/HashFunctions.html#hash(long).

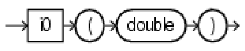
Examples

Consider the query `qColt59`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:        +          8
1000:        +          6
1200:        +          12
2000:        +          4
```

10.2.24 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:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#i0(double)
- j0.

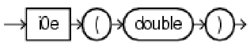
Examples

Consider the query `qColt12`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           1.0634834
1000:       +           1.126303
1200:       +           1.2080469
2000:       +           1.0404018
```

10.2.25 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:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#i0e(double)
- j0

Examples

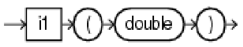
Consider the query `qColt13`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.64503527
```

```
1000:       +           0.55930555
1200:       +           0.4960914
2000:       +           0.6974022
```

10.2.26 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:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#i1(double)
- j1.

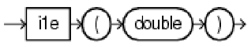
Examples

Consider the query `qColt14`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.2578943
1000:       +           0.37187967
1200:       +           0.49053898
2000:       +           0.20402676
```

10.2.27 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

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#i1e(double)
- j1.

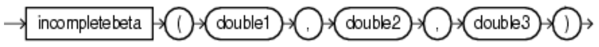
Examples

Consider the query `qColt15`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.1564208
1000:       +           0.18466999
1200:       +           0.20144266
2000:       +           0.13676323
```

10.2.28 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Gamma.html#incompleteBeta(double, double, double).

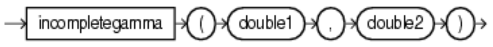
Examples

Consider the query `qColt30`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.5
1000:       +           0.66235894
1200:       +           0.873397
2000:       +           0.44519535
```

10.2.29 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Gamma.html#incompleteGamma(double, double).

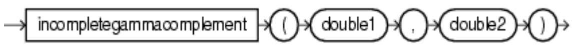
Examples

Consider the query `qColt31`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.6826895
1000:       +           0.6565891
1200:       +           0.6397422
2000:       +           0.7014413
```

10.2.30 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Gamma.html#incompleteGammaComplement(double, double).

Examples

Consider the query `qColt32`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

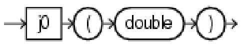
```
<query id="qColt32"><![CDATA[
    select incompleteGammaComplement(c2,c2) from SColtFunc
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.3173105
1000:       +           0.34341094
1200:       +           0.3602578
2000:       +           0.29855874
```

10.2.31 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#j0(double).

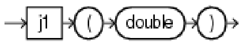
Examples

Consider the query `qColt16`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.9384698
1000:       +           0.8812009
1200:       +           0.8115654
2000:       +           0.9603982
```

10.2.32 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#j1(double).

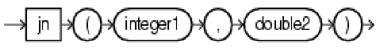
Examples

Consider the query `qColt17`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.24226846
1000:       +           0.32899573
1200:       +           0.40236986
2000:       +           0.19602658
```

10.2.33 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#jn(int, double).

Examples

Consider the query `qColt18`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

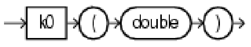
```
<query id="qColt18"><![CDATA[
    select jn(c1,c2) from SColtFunc
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.24226846
1000:       +           6.1009696E-4
1200:       +           0.0139740035
2000:       +           6.321045E-11
```

10.2.34 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#k0(double).

Examples

Consider the query `qColt19`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

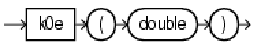
```
<query id="qColt19"><![CDATA[
    select k0(c2) from SColtFunc
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.92441905
1000:       +           0.6605199
1200:       +           0.49396032
2000:       +           1.1145291
```

10.2.35 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#k0e(double).

Examples

Consider the query `qColt20`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

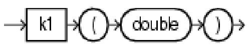
```
<query id="qColt20"><![CDATA[
    select k0e(c2) from SColtFunc
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           1.5241094
1000:       +           1.3301237
1200:       +           1.2028574
2000:       +           1.662682
```

10.2.36 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#k1(double).

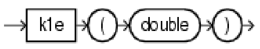
Examples

Consider the query `qColt21`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           1.6564411
1000:       +           1.0502836
1200:       +           0.7295154
2000:       +           2.1843543
```

10.2.37 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:

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#k1e(double)`
- k1.

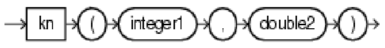
Examples

Consider the query `qColt22`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           2.7310097
1000:       +           2.1150115
1200:       +           1.7764645
2000:       +           3.258674
```

10.2.38 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#kn(int, double).

Examples

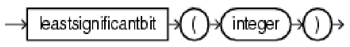
Consider the query `qColt23`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
```

```
1200       3,0.89,12
2000       8,0.4,4

Timestamp   Tuple Kind  Tuple
  10:       +             1.6564411
1000:       +           191.99422
1200:       +            10.317473
2000:       +             9.7876858E8
```

10.2.39 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:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/colt/bitvector/QuickBitVector.html#leastSignificantBit(int)
- bitMaskWithBitsSetFromTo
- mostSignificantBit.

Examples

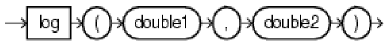
Consider the query `qColt54`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0
1000:       +           2
1200:       +           0
2000:       +           3
```

10.2.40 log

Syntax



Purpose

`log` is based on `cern.jet.math.Arithmetic`. It returns the computation that Figure 10-11 shows as a `double`.

Figure 10-11 cern.jet.math.Arithmetic log

$$\log_{base} value$$

This function takes the following arguments:

- `double1`: the `base`.
- `double2`: the `value`.

For more information, see https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#log(double, double).

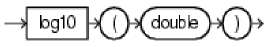
Examples

Consider the query `qColt3`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           1.0
1000:       +           1.0
1200:       +           1.0
2000:       +           1.0
```

10.2.41 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#log10(double).

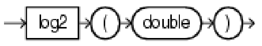
Examples

Consider the query `qColt4`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           -0.30103
1000:       +           -0.15490197
1200:       +           -0.050610002
2000:       +           -0.39794
```

10.2.42 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#log2(double).

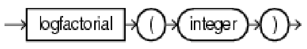
Examples

Consider the query `qColt9`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           -1.0
1000:       +           -0.5145732
1200:       +           -0.16812278
2000:       +           -1.321928
```

10.2.43 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#logFactorial(int).

Examples

Consider the query `qColt10`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

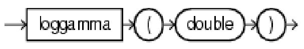
```
<query id="qColt10"><![CDATA[
    select logFactorial(c1) from SColtFunc
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
  10:       +            0.0
1000:       +            3.1780539
1200:       +            1.7917595
2000:       +           10.604603
```

10.2.44 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Gamma.html#logGamma(double).

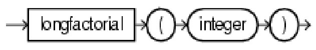
Examples

Consider the query `qColt33`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.5723649
1000:       +           0.26086727
1200:       +           0.07402218
2000:       +           0.7966778
```

10.2.45 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#longFactorial(int).

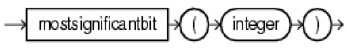
Examples

Consider the query `qColt11`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           1
1000:       +           24
1200:       +           6
2000:       +           40320
```

10.2.46 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:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/colt/bitvector/QuickBitVector.html#mostSignificantBit(int)
- bitMaskWithBitsSetFromTo
- leastSignificantBit.

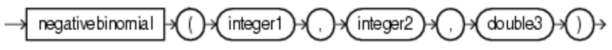
Examples

Consider the query `qColt55`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0
1000:       +           2
1200:       +           1
2000:       +           3
```

10.2.47 negativeBinomial

Syntax



Purpose

`negativeBinomial` is based on `cern.jet.stat.Probability`. It returns the sum of the terms 0 through `k` of the Negative Binomial Distribution (see Figure 10-12) as a `double`.

Figure 10-12 cern.jet.stat.Probability negativeBinomial

$$\sum_{j=0}^{k} \binom{n+j-1}{j} p^{n}(1-p)^{j}$$

This function takes the following arguments:

- `integer1`: the end term `k`.
- `integer2`: the number of trials `n`.
- `double3`: the probability of success `p` in (0.0,1.0).

For more information, see https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#negativeBinomial(int, int, double).

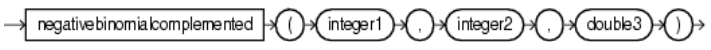
Examples

Consider the query `qColt44`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.75
1000:       +           0.94203234
1200:       +           0.99817264
2000:       +           0.28393665
```

10.2.48 negativeBinomialComplemented

Syntax



Purpose

`negativeBinomialComplemented` is based on `cern.jet.stat.Probability`. It returns the sum of the terms `k+1` to infinity of the Negative Binomial distribution (see Figure 10-13) as a `double`.

Figure 10-13 cern.jet.stat.Probability negativeBinomialComplemented

$$\sum_{j=k+1}^{\infty} \binom{n+j-1}{j} p^{n}(1-p)^{j}$$

This function takes the following arguments:

- `integer1`: the end term `k`.
- `integer2`: the number of trials `n`.
- `double3`: the probability of success `p` in (0.0,1.0).

For more information, see https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#negativeBinomialComplemented(int, int, double).

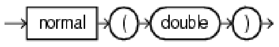
Examples

Consider the query `qColt45`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.25
1000:       +           0.05796766
1200:       +           0.0018273441
2000:       +           0.7160633
```

10.2.49 normal

Syntax



Purpose

`normal` is based on `cern.jet.stat.Probability`. It returns the area under the Normal (Gaussian) probability density function, integrated from minus infinity to the `double` argument `x` (see Figure 10-14) as a `double`.

Figure 10-14 cern.jet.stat.Probability normal

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

For more information, see `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#normal(double)`.

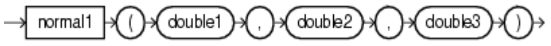
Examples

Consider the query `qColt46`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.69146246
1000:       +           0.7580363
1200:       +           0.81326705
2000:       +           0.65542173
```

10.2.50 normal1

Syntax



Purpose

`normal1` is based on `cern.jet.stat.Probability`. It returns the area under the Normal (Gaussian) probability density function, integrated from minus infinity to `x` (see Figure 10-15) as a `double`.

Figure 10-15 cern.jet.stat.Probability normal1

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

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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#normal(double, double, double).

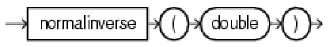
Examples

Consider the query `qColt47`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.5
1000:       +           0.5
1200:       +           0.5
2000:       +           0.5
```

10.2.51 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#normalInverse(double).

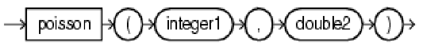
Examples

Consider the query `qColt48`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.0
1000:       +           0.5244005
1200:       +           1.226528
2000:       +           0.2533471
```

10.2.52 poisson

Syntax



Purpose

`poisson` is based on `cern.jet.stat.Probability`. It returns the sum of the first `k` terms of the Poisson distribution (see Figure 10-16) as a `double`.

Figure 10-16 cern.jet.stat.Probability poisson

$$\sum_{j=0}^{k} e^{-m} \frac{m^j}{j!}$$

This function takes the following arguments:

- `integer1`: the number of terms `k`.
- `double2`: the mean of the Poisson distribution `m`.

For more information, see https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#poisson(int, double).

Examples

Consider the query `qColt49`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

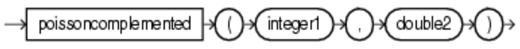
```
<query id="qColt49"><![CDATA[
    select poisson(c1,c2) from SColtFunc
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.909796
1000:       +           0.9992145
1200:       +           0.9870295
2000:       +           1.0
```

10.2.53 poissonComplemented

Syntax



Purpose

`poissonComplemented` is based on `cern.jet.stat.Probability`. It returns the sum of the terms `k+1` to Infinity of the Poisson distribution (see Figure 10-17) as a `double`.

Figure 10-17 cern.jet.stat.Probability poissonComplemented

$$\sum_{j=k+1}^{\infty} e^{-m} \frac{m^j}{j!}$$

This function takes the following arguments:

- `integer1`: the start term `k`.
- `double2`: the mean of the Poisson distribution `m`.

For more information, see https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#poissonComplemented(int, double).

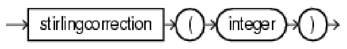
Examples

Consider the query `qColt50`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.09020401
1000:       +           7.855354E-4
1200:       +           0.012970487
2000:       +           5.043364E-10
```

10.2.54 stirlingCorrection

Syntax



Purpose

`stirlingCorrection` is based on `cern.jet.math.Arithmetic`. It returns the correction term of the Stirling approximation of the natural logarithm (base *e*) of the factorial of the `integer` argument (see Figure 10-18) as a `double`.

Figure 10-18   cern.jet.math.Arithmetic stirlingCorrection

$$\log k! = (k + \frac{1}{2})\log(k+1) - (k+1) + (\frac{1}{2})\log(2\pi) + STIRLINGCORRECTION(k+1)$$

$$\log k! = (k + \frac{1}{2})\log(k) - k + (\frac{1}{2})\log(2\pi) + STIRLINGCORRECTION(k)$$

For more information, see https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Arithmetic.html#stirlingCorrection(int).

Examples

Consider the query `qColt5`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

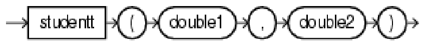
```
<query id="qColt5"><![CDATA[
    select stirlingCorrection(c1) from SColtFunc
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.08106147
1000:       +           0.020790672
1200:       +           0.027677925
2000:       +           0.010411265
```

10.2.55 studentT

Syntax



Purpose

`studentT` is based on `cern.jet.stat.Probability`. It returns the integral from minus infinity to `t` of the Student-t distribution with `k` > 0 degrees of freedom (see Figure 10-19) as a `double`.

Figure 10-19 cern.jet.stat.Probability studentT

$$\frac{\Gamma(\frac{k+1}{2})}{\sqrt{k\pi}\,\Gamma(\frac{k}{2})}\int_{-\infty}^{t}\left(1+\frac{x^2}{k}\right)^{\frac{-(K+1)}{2}} dx$$

This function takes the following arguments:

- `double1`: the degrees of freedom `k`.
- `double2`: the integration end point `t`.

For more information, see https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#studentT(double, double).

Examples

Consider the query `qColt51`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

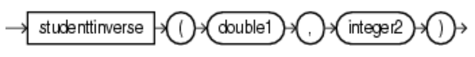
```
<query id="qColt51"><![CDATA[
    select studentT(c2,c2) from SColtFunc
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.621341
1000:       +           0.67624015
1200:       +           0.7243568
2000:       +           0.5930112
```

10.2.56 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:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Probability.html#studentTInverse(double, int)
- studentT.

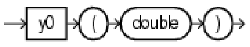
Examples

Consider the query qColt52. Given the data stream SColtFunc with schema (c1 integer, c2 double, c3 bigint), the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           1.0
1000:       +           0.4141633
1200:       +           0.15038916
2000:       +           0.8888911
```

10.2.57 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#y0(double).

Examples

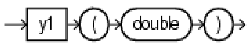
Consider the query qColt24. Given the data stream SColtFunc with schema (c1 integer, c2 double, c3 bigint), the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           -0.44451874
1000:       +           -0.19066493
1200:       +           -0.0031519707
2000:       +           -0.60602456
```

10.2.58 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#y1(double).

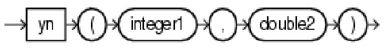
Examples

Consider the query `qColt25`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           -1.4714724
1000:       +           -1.1032499
1200:       +           -0.88294965
2000:       +           -1.780872
```

10.2.59 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 https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/math/Bessel.html#yn(int, double).

Examples

Consider the query `qColt26`. Given the data stream `SColtFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

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

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

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

11 Colt Aggregate Functions

A reference to Colt aggregate functions provided in Oracle Continuous Query Language (Oracle CQL) is provided. Colt aggregate functions are based on the Colt open source libraries for high performance scientific and technical computing.

For more information, see Functions.

11.1 Introduction to Oracle CQL Built-In Aggregate Colt Functions

Table 11-1 lists the built-in aggregate Colt functions that Oracle CQL provides.

Table 11-1 Oracle CQL Built-in Aggregate Colt-Based Functions

Colt Package	Function
`cern.jet.stat.Descriptive` A set of basic descriptive statistics functions.	• autoCorrelation • correlation • covariance • geometricMean • geometricMean1 • harmonicMean • kurtosis • lag1 • mean • meanDeviation • median • moment • pooledMean • pooledVariance • product • quantile • quantileInverse • rankInterpolated • rms • sampleKurtosis • sampleKurtosisStandardError • sampleSkew • sampleSkewStandardError • sampleVariance • skew • standardDeviation • standardError • sumOfInversions • sumOfLogarithms • sumOfPowerDeviations • sumOfPowers • sumOfSquaredDeviations • sumOfSquares • trimmedMean • variance • weightedMean • winsorizedMean

Note:

Built-in function names are case sensitive and you must use them in the case shown (in lower case).

Note:

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

In relation output examples, the first tuple output is:

```
-9223372036854775808:+
```

This value is `-Long.MIN_VALUE()` and represents the largest negative timestamp possible.

For more information, see:

- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments
- Colt Aggregate Functions and the Where, Group By, and Having Clauses
- Functions
- Data Types
- http://dsd.lbl.gov/~hoschek/colt/.

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

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

Since the function expects a stream of tuples having a `double data` value and an `integer lag` value only, the signature of the Oracle CQL `autoCorrelation` function is:

```
double autoCorrelation (double data, int lag)
```

11.1.2 Colt Aggregate Functions and the Where, Group By, and Having Clauses

In Oracle CQL, the `where` clause is applied before the `group by` and `having` clauses. This means the Oracle CQL statement is invalid:

```
<query id="q1"><![CDATA[
    select * from InputChannel[rows 4 slide 4] as ic where geometricMean(c3) > 4
]]></query>
```

Instead, you must use the Oracle CQL statement shown in the following example:

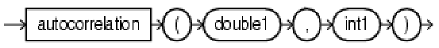
```
<query id="q1"><![CDATA[
    select * from InputChannel[rows 4 slide 4] as ic, myGeoMean = geometricMean(c3)
where myGeoMean > 4
]]></query>
```

For more information, see:

- Figure 16-11
- Figure 16-12
- Figure 16-19.

11.2.1 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.

This function takes the following tuple arguments:

- `double1`: data value.
- `int1`: lag.

For more information, see

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#autoCorrelation(cern.colt.list.DoubleArrayList, int, double, double)

- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

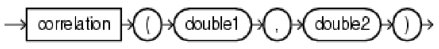
Examples

Consider the query `qColtAggr1`. Given the data stream `SColtAggrFunc` with schema `(c3 double)`, the query returns the relation.

```
<query id="qColtAggr1"><![CDATA[
     select autoCorrelation(c3, 0) from SColtAggrFunc
]]></query>

Timestamp   Tuple
  10        5.441341838866902
1000        6.1593756700951054
1200        3.7269733222923676
1400        4.625160266213489
1600        3.490061774090248
1800        3.6354484064421917
2000        5.635401664977703
2200        5.006087562207967
2400        3.632574304861612
2600        7.618087248962962
h 8000
h 200000000

Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           NaN
1000:       -           NaN
1000:       +           1.0
1200:       -           1.0
1200:       +           1.0
1400:       -           1.0
1400:       +           1.0
1600:       -           1.0
1600:       +           1.000000000000002
1800:       -           1.000000000000002
1800:       +           1.0
2000:       -           1.0
2000:       +           0.9999999999999989
2200:       -           0.9999999999999989
2200:       +           0.999999999999999
2400:       -           0.999999999999999
2400:       +           0.9999999999999991
2600:       -           0.9999999999999991
2600:       +           1.0000000000000013
```

11.2.2 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

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#correlation(cern.colt.list.DoubleArrayList, double, cern.colt.list.DoubleArrayList, double)
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

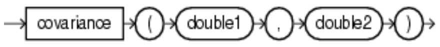
Examples

Consider the query `qColtAggr2`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           NaN
1000:       -           NaN
1000:       +           2.0
1200:       -           2.0
1200:       +           1.5
2000:       -           1.5
2000:       +           1.3333333333333333
```

11.2.3 covariance

Syntax



Purpose

`covariance` is based on `cern.jet.stat.Descriptive.covariance(DoubleArrayList data1, DoubleArrayList data2)`. It returns the correlation of two data sequences (see Figure 11-1) of the input arguments as a `double`.

Figure 11-1 cern.jet.stat.Descriptive.covariance

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

This function takes the following tuple arguments:

- `double1`: data value 1.
- `double2`: data value 2.

For more information, see:

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#covariance(cern.colt.list.DoubleArrayList, cern.colt.list.DoubleArrayList)`
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

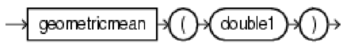
Examples

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

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

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

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
```

11.2.4 geometricMean

Syntax



Purpose

`geometricMean` is based on `cern.jet.stat.Descriptive.geometricMean(DoubleArrayList data)`. It returns the geometric mean of a data sequence (see Figure 11-2) of the input argument as a `double`.

Figure 11-2 cern.jet.stat.Descriptive.geometricMean(DoubleArrayList data)

$$pow(product(data[i]), \frac{1}{data.size()})$$

This function takes the following tuple arguments:

- `double1`: data value.

Note that for a geometric mean to be meaningful, the minimum of the data values must not be less than or equal to zero.

For more information, see:

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#geometricMean(cern.colt.list.DoubleArrayList)`
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

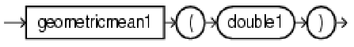
Examples

Consider the query `qColtAggr6`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.5 geometricMean1

Syntax



Purpose

`geometricMean1` is based on `cern.jet.stat.Descriptive.geometricMean(double sumOfLogarithms)`. It returns the geometric mean of a data sequence (see Figure 11-3) of the input arguments as a `double`.

Figure 11-3 cern.jet.stat.Descriptive.geometricMean1(int size, double sumOfLogarithms)

$$pow(product(data[i]), \frac{1}{size})$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#geometricMean(cern.colt.list.DoubleArrayList)
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

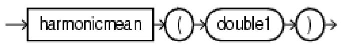
Examples

Consider the query `qColtAggr7`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

```
<query id="qColtAggr7"><![CDATA[
    select geometricMean1(c3) from SColtAggrFunc
]]></query>

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

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

11.2.6 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:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#harmonicMean(int, double)
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

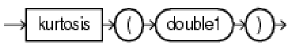
Examples

Consider the query `qColtAggr8`. Given the data stream `SColtAggrFunc` with schema `(c3 double)`, the query returns the relation.

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

Timestamp   Tuple
  10         5.441341838866902
1000        6.1593756700951054
1200        3.7269733222923676
1400        4.625160266213489
1600        3.490061774090248
1800        3.6354484064421917
2000        5.635401664977703
2200        5.006087562207967
2400        3.632574304861612
2600        7.618087248962962
h 8000
h 200000000

Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           5.441341876983643
1000:       -           5.441341876983643
1000:       +           5.778137193205395
1200:       -           5.778137193205395
1200:       +           4.882442561720335
1400:       -           4.882442561720335
1400:       +           4.815475325819701
1600:       -           4.815475325819701
1600:       +           4.475541862878903
1800:       -           4.475541862878903
1800:       +           4.309563447664887
2000:       -           4.309563447664887
2000:       +           4.45944509362759
2200:       -           4.45944509362759
2200:       +           4.5211563834502515
2400:       -           4.5211563834502515
2400:       +           4.401525382790638
2600:       -           4.401525382790638
2600:       +           4.595562422157167
```

11.2.7 kurtosis

Syntax



Purpose

`kurtosis` is based on `cern.jet.stat.Descriptive.kurtosis(DoubleArrayList data, double mean, double standardDeviation)`. It returns the kurtosis or excess (see Figure 11-4) of a data sequence as a `double`.

Figure 11-4 cern.jet.stat.Descriptive.kurtosis(DoubleArrayList data, double mean, double standardDeviation)

$$-3 + \frac{moment(data, 4, mean)}{\text{StandardDeviation}^4}$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#kurtosis(cern.colt.list.DoubleArrayList, double, double)`
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

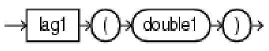
Examples

Consider the query `qColtAggr12`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.8 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.

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#lag1(cern.colt.list.DoubleArrayList, double)
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

Examples

Consider the query `qColtAggr14`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

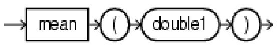
```
<query id="qColtAggr14"><![CDATA[
    select lag1(c3) from SColtAggrFunc
]]></query>
```

```
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
```

```
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
```

11.2.9 mean

Syntax



Purpose

`mean` is based on `cern.jet.stat.Descriptive.mean(DoubleArrayList data)`. It returns the arithmetic mean of a data sequence (see Figure 11-5) as a `double`.

Figure 11-5 cern.jet.stat.Descriptive.mean(DoubleArrayList data)

$$\frac{sum(data[i])}{data.size()}$$

The following table lists the input types and the corresponding output types:

Input Types	Output Types
INT	DOUBLE
BIGINT	DOUBLE
FLOAT	DOUBLE
DOUBLE	DOUBLE

For more information, see:

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#mean(cern.colt.list.DoubleArrayList)`
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

Examples

Consider the query `qColtAggr16`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

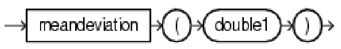
```
<query id="qColtAggr16"><![CDATA[
    select mean(c3) from SColtAggrFunc
]]></query>
```

```
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
```

```
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
```

11.2.10 meanDeviation

Syntax



Purpose

`meanDeviation` is based on `cern.jet.stat.Descriptive.meanDeviation(DoubleArrayList data, double mean)`. It returns the mean deviation of a dataset (see Figure 11-6) as a `double`.

Figure 11-6 cern.jet.stat.Descriptive.meanDeviation(DoubleArrayList data, double mean)

$$\frac{sum(Math.abs(data[i]-mean))}{data.size()}$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#meanDeviation(cern.colt.list.DoubleArrayList, double)`
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

Examples

Consider the query `qColtAggr17`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

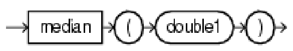
```
<query id="qColtAggr17"><![CDATA[
    select meanDeviation(c3) from SColtAggrFunc
]]></query>
```

```
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
```

```
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
```

11.2.11 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`.

The following table lists the input types and the corresponding output types:

Table 11-2 Input and Output Types

Input Types	Output Types
INT	DOUBLE
BIGINT	DOUBLE
FLOAT	DOUBLE
DOUBLE	DOUBLE

Note:

If the input type is `INT`, then return type will also be `INT` and it will be floor of the divided value.

For more information, see:

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#median(cern.colt.list.DoubleArrayList)`
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

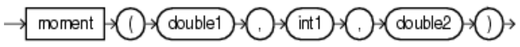
Examples

Consider the query `qColtAggr18`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.12 moment

Syntax



Purpose

`moment` is based on `cern.jet.stat.Descriptive.moment(DoubleArrayList data, int k, double c)`. It returns the moment of the `k`-th order with constant `c` of a data sequence (see Figure 11-7) as a `double`.

Figure 11-7 cern.jet.stat.Descriptive.moment(DoubleArrayList data, int k, double c)

$$\frac{sum((data[i]-c)^k)}{data.size()}$$

This function takes the following tuple arguments:

- `double1`: data value.
- `int1`: `k`.
- `double2`: `c`.

For more information, see:

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#moment(cern.colt.list.DoubleArrayList, int, double)`
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

Examples

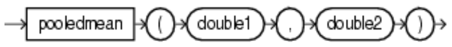
Consider the query `qColtAggr21`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.13 pooledMean

Syntax



Purpose

`pooledMean` is based on `cern.jet.stat.Descriptive.pooledMean(int size1, double mean1, int size2, double mean2)`. It returns the pooled mean of two data sequences (see Figure 11-8) as a `double`.

Figure 11-8 cern.jet.stat.Descriptive.pooledMean(int size1, double mean1, int size2, double mean2)

$$\frac{(size1 * mean1 + size2 * mean2)}{(size1 + size2)}$$

This function takes the following tuple arguments:

- `double1`: mean 1.
- `double2`: mean 2.

For more information, see

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#pooledMean(int, double, int, double)`
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

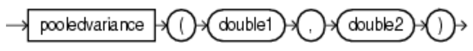
Examples

Consider the query `qColtAggr22`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.14 pooledVariance

Syntax



Purpose

`pooledVariance` is based on `cern.jet.stat.Descriptive.pooledVariance(int size1, double variance1, int size2, double variance2)`. It returns the pooled variance of two data sequences (see Figure 11-9) as a `double`.

Figure 11-9 cern.jet.stat.Descriptive.pooledVariance(int size1, double variance1, int size2, double variance2)

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

This function takes the following tuple arguments:

- `double1`: variance 1.
- `double2`: variance 2.

For more information, see

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#pooledVariance(int, double, int, double)`
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

Examples

Consider the query `qColtAggr23`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

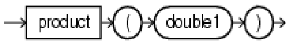
```
<query id="qColtAggr23"><![CDATA[
    select pooledVariance(c3, c3) from SColtAggrFunc
]]></query>
```

```
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
```

```
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
```

11.2.15 product

Syntax



Purpose

`product` is based on `cern.jet.stat.Descriptive.product(DoubleArrayList data)`. It returns the product of a data sequence (see Figure 11-10) as a `double`.

Figure 11-10   cern.jet.stat.Descriptive.product(DoubleArrayList data)

$$data[0] * data[1] * \ldots * data[data.size() - 1]$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#product(cern.colt.list.DoubleArrayList)`
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

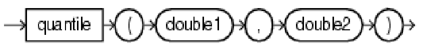
Examples

Consider the query `qColtAggr24`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.16 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:

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#quantile(cern.colt.list.DoubleArrayList, double)`
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

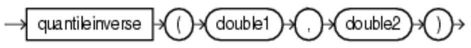
Examples

Consider the query `qColtAggr26`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.17 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:

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#quantileInverse(cern.colt.list.DoubleArrayList, double)`
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

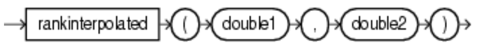
Examples

Consider the query `qColtAggr27`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.18 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:

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#rankInterpolated(cern.colt.list.DoubleArrayList, double)`
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

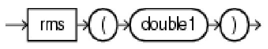
Examples

Consider the query `qColtAggr29`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.19 rms

Syntax



Purpose

`rms` is based on `cern.jet.stat.Descriptive.rms(int size, double sumOfSquares)`. It returns the Root-Mean-Square (RMS) of a data sequence (see Figure 11-11) as a `double`.

Figure 11-11 cern.jet.stat.Descriptive.rms(int size, double sumOfSquares)

$$Math.sqrt(\frac{Sum(data[i] * data[i])}{data.size()})$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#rms(int, double)`
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

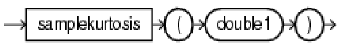
Examples

Consider the query `qColtAggr30`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.20 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:

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#sampleKurtosis(cern.colt.list.DoubleArrayList, double, double)`
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

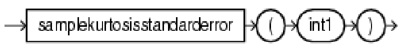
Examples

Consider the query `qColtAggr31`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           NaN
1000:       -           NaN
1000:       +           NaN
1200:       -           NaN
1200:       +           NaN
2000:       -           NaN
2000:       +           -1.1999999999999993
```

11.2.21 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:

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#sampleKurtosisStandardError(int)`
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

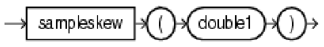
Examples

Consider the query `qColtAggr33`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:       -
  10:       +           0.0
1000:       -           0.0
1000:       +           Infinity
1200:       -           Infinity
1200:       +           Infinity
2000:       -           Infinity
2000:       +           2.6186146828319083
```

11.2.22 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:

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#sampleSkew(cern.colt.list.DoubleArrayList, double, double)`
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

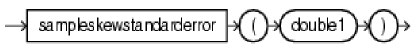
Examples

Consider the query `qColtAggr34`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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

11.2.23 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:

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#sampleSkewStandardError(int)`
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

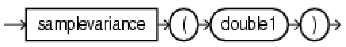
Examples

Consider the query `qColtAggr36`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.24 sampleVariance

Syntax



Purpose

`sampleVariance` is based on `cern.jet.stat.Descriptive.sampleVariance(DoubleArrayList data, double mean)`. It returns the sample variance of a data sequence (see Figure 11-12) as a `double`.

Figure 11-12 cern.jet.stat.Descriptive.sampleVariance(DoubleArrayList data, double mean)

$$\frac{Sum((data[i]-mean)^2)}{(data.size()-1)}$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#sampleVariance(cern.colt.list.DoubleArrayList, double)
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

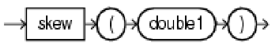
Examples

Consider the query `qColtAggr38`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.25 skew

Syntax



Purpose

`skew` is based on `cern.jet.stat.Descriptive.skew(DoubleArrayList data, double mean, double standardDeviation)`. It returns the skew of a data sequence of a data sequence (see Figure 11-13) as a `double`.

Figure 11-13 cern.jet.stat.Descriptive.skew(DoubleArrayList data, double mean, double standardDeviation)

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

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#skew(cern.colt.list.DoubleArrayList, double, double)
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

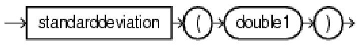
Examples

Consider the query `qColtAggr41`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.26 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`.

The following table lists the input types and the corresponding output types:

Input Types	Output Types
INT	DOUBLE
BIGINT	DOUBLE
FLOAT	DOUBLE

DOUBLE	DOUBLE

For more information, see

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#standardDeviation(double)
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

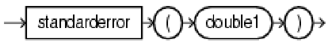
Examples

Consider the query `qColtAggr44`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.27 standardError

Syntax



Purpose

`standardError` is based on `cern.jet.stat.Descriptive.standardError(int size, double variance)`. It returns the standard error of a data sequence (see Figure 11-14) as a `double`.

Figure 11-14 cern.jet.stat.Descriptive.cern.jet.stat.Descriptive.standardError(int size, double variance)

$$Math.sqrt(\frac{\text{variance}}{size})$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#standardError(int, double)
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

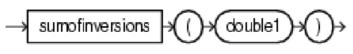
Examples

Consider the query `qColtAggr45`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.28 sumOfInversions

Syntax



Purpose

`sumOfInversions` is based on `cern.jet.stat.Descriptive.sumOfInversions(DoubleArrayList data, int from, int to)`. It returns the sum of inversions of a data sequence (see Figure 11-15) as a `double`.

Figure 11-15 cern.jet.stat.Descriptive.sumOfInversions(DoubleArrayList data, int from, int to)

$$Sum(\frac{1.0}{data[i]})$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#sumOfInversions(cern.colt.list.DoubleArrayList, int, int)
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

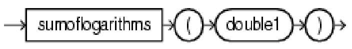
Examples

Consider the query `qColtAggr48`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:        -
  10:        +           0.025
1000:       -           0.025
1000:       +           0.058333333333333334
1200:       -           0.058333333333333334
1200:       +           0.10833333333333334
2000:       -           0.10833333333333334
2000:       +           0.20833333333333334
```

11.2.29 sumOfLogarithms

Syntax



Purpose

`sumOfLogarithms` is based on `cern.jet.stat.Descriptive.sumOfLogarithms(DoubleArrayList data, int from, int to)`. It returns the sum of logarithms of a data sequence (see Figure 11-16) as a `double`.

Figure 11-16 cern.jet.stat.Descriptive.sumOfLogarithms(DoubleArrayList data, int from, int to)

$$Sum(Log(data[i]))$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#sumOfLogarithms(cern.colt.list.DoubleArrayList, int, int)
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

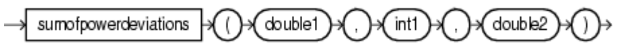
Examples

Consider the query `qColtAggr49`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
  10:        -
  10:        +           3.6888794541139363
1000:       -           3.6888794541139363
1000:       +           7.090076835776092
1200:       -           7.090076835776092
1200:       +           10.085809109330082
2000:       -           10.085809109330082
2000:       +           12.388394202324129
```

11.2.30 sumOfPowerDeviations

Syntax



Purpose

`sumOfPowerDeviations` is based on `cern.jet.stat.Descriptive.sumOfPowerDeviations(DoubleArrayList data, int k, double c)`. It returns sum of power deviations of a data sequence (see Figure 11-17) as a `double`.

Figure 11-17 cern.jet.stat.Descriptive.sumOfPowerDeviations(DoubleArrayList data, int k, double c)

$$Sum((data[i]-c)^{k})$$

This function is optimized for common parameters like `c == 0.0`, `k == -2 .. 4`, or both.

This function takes the following tuple arguments:

- `double1`: data value.
- `int1`: `k`.
- `double2`: `c`.

For more information, see:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#sumOfPowerDeviations(cern.colt.list.DoubleArrayList, int, double)
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

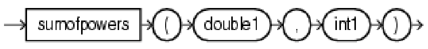
Examples

Consider the query `qColtAggr50`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.31 sumOfPowers

Syntax



Purpose

`sumOfPowers` is based on `cern.jet.stat.Descriptive.sumOfPowers(DoubleArrayList data, int k)`. It returns the sum of powers of a data sequence (see Figure 11-18) as a `double`.

Figure 11-18   cern.jet.stat.Descriptive.sumOfPowers(DoubleArrayList data, int k)

$$Sum(data[i]^k)$$

This function takes the following tuple arguments:

- `double1`: data value.
- `int1`: `k`.

For more information, see:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#sumOfPowers(cern.colt.list.DoubleArrayList, int)
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

Examples

Consider the query `qColtAggr52`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

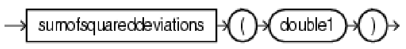
```
<query id="qColtAggr52"><![CDATA[
    select sumOfPowers(c3, c1) from SColtAggrFunc
]]></query>
```

```
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
```

```
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
```

11.2.32 sumOfSquaredDeviations

Syntax



Purpose

`sumOfSquaredDeviations` is based on `cern.jet.stat.Descriptive.sumOfSquaredDeviations(int size, double variance)`. It returns the sum of squared mean deviation of a data sequence (see Figure 11-19) as a `double`.

Figure 11-19 cern.jet.stat.Descriptive.sumOfSquaredDeviations(int size, double variance)

$$variance * (size - 1) == Sum((data[i] - mean)^2)$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see

- `https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#sumOfSquaredDeviations(int, double)`
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

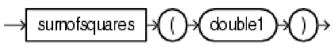
Examples

Consider the query `qColtAggr53`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.33 sumOfSquares

Syntax



Purpose

`sumOfSquares` is based on `cern.jet.stat.Descriptive.sumOfSquares(DoubleArrayList data)`. It returns the sum of squares of a data sequence (see Figure 11-20) as a `double`.

Figure 11-20 cern.jet.stat.Descriptive.sumOfSquares(DoubleArrayList data)

$$Sum(data[i] * data[i])$$

This function takes the following tuple arguments:

- `double1`: data value.

For more information, see:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#sumOfSquares(cern.colt.list.DoubleArrayList)
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

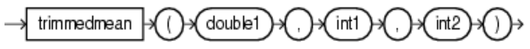
Examples

Consider the query `qColtAggr54`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.34 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:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#trimmedMean(cern.colt.list.DoubleArrayList, double, int, int)

- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

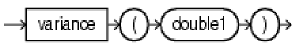
Examples

Consider the query `qColtAggr55`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

Timestamp   Tuple
  10        0, 0.5, 40.0, 8
1000        0, 0.7, 30.0, 6
1200        0, 0.89, 20.0, 12
2000        1, 0.4, 10.0, 4
h 8000
h 200000000

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
```

11.2.35 variance

Syntax



Purpose

`variance` is based on `cern.jet.stat.Descriptive.variance(int size, double sum, double sumOfSquares)`. It returns the variance of a data sequence (see Figure 11-21) as a `double`.

Figure 11-21 cern.jet.stat.Descriptive.variance(int size, double sum, double sumOfSquares)

$$\frac{(SumofSquares - mean * sum)}{\text{size with mean}} = \frac{sum}{size}$$

The following table lists the input types and the corresponding output types:

Input Types	Output Types
INT	DOUBLE

BIGINT	DOUBLE
FLOAT	DOUBLE
DOUBLE	DOUBLE

For more information, see:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#variance(int, double, double)
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

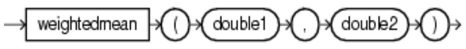
Examples

Consider the query `qColtAggr57`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.36 weightedMean

Syntax



Purpose

`weightedMean` is based on `cern.jet.stat.Descriptive.weightedMean(DoubleArrayList data, DoubleArrayList weights)`. It returns the weighted mean of a data sequence (see Figure 11-22) as a `double`.

Figure 11-22 cern.jet.stat.Descriptive.weightedMean(DoubleArrayList data, DoubleArrayList weights)

$$\frac{Sum(data[i]*weights[i])}{Sum(weights[i])}$$

This function takes the following tuple arguments:

- `double1`: data value.
- `double2`: weight value.

For more information, see:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#weightedMean(cern.colt.list.DoubleArrayList, cern.colt.list.DoubleArrayList)
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

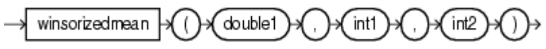
Examples

Consider the query `qColtAggr58`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

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

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
```

11.2.37 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:

- https://dst.lbl.gov/ACSSoftware/colt/api/cern/jet/stat/Descriptive.html#winsorizedMean(cern.colt.list.DoubleArrayList, double, int, int)
- Oracle CQL Colt Aggregate Function Signatures and Tuple Arguments.

Examples

Consider the query `qColtAggr60`. Given the data stream `SColtAggrFunc` with schema `(c1 integer, c2 float, c3 double, c4 bigint)`, the query returns the relation.

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

Timestamp   Tuple
  10        1, 0.5, 40.0, 8
1000        0, 0.7, 30.0, 6
1200        1, 0.89, 20.0, 12
2000        1, 0.4, 10.0, 4
h 8000

Timestamp   Tuple Kind  Tuple
-9223372036854775808:+
10:              -
10:          +          40.0
1000:        -          40.0
1000:        +          35.0
1200:        -          35.0
1200:        +          30.000000000000004
2000:        -          30.000000000000004
2000:        +          25
```

12 java.lang.Math Functions

A reference to the `java.lang.Math` functions provided in Oracle Continuous Query Language (Oracle CQL) is provided.

For more information, see .

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

Table 12-1 lists the built-in `java.lang.Math` functions that Oracle CQL provides.

Table 12-1 Oracle CQL Built-in `java.lang.Math` Functions

Type	Function
Trigonometric	• sin • cos • tan • asin • acos • atan • atan2 • cosh • sinh • tanh
Logarithmic	• log1 • log101 • log1p
Euler's Number	• exp • expm1
Roots	• cbrt • sqrt • hypot
Signum Function	• signum • signum1
Unit of Least Precision	• ulp • ulp1

Table 12-1 (Cont.) Oracle CQL Built-in `java.lang.Math` Functions

Type	Function
Other	• abs • abs1 • abs2 • abs3 • ceil1 • floor1 • IEEEremainder • pow • rint • round • round1 • todegrees • toradians

Note:

Built-in function names are case sensitive and you must use them in the case shown (in lower case).

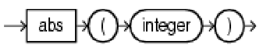
Note:

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

For more information, see `http://java.sun.com/javase/6/docs/api/java/lang/Math.html`.

12.2.1 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)`.

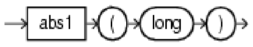
Examples

Consider the query `q66`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           1
1000:       +           4
1200:       +           3
2000:       +           8
```

12.2.2 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)`.

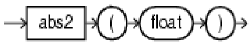
Examples

Consider the query `q67`. Given the data stream `SFunc` with schema `(c1 integer, c2 float, c3 long)`, the query returns the stream.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           8
1000:       +           6
1200:       +           12
2000:       +           4
```

12.2.3 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).

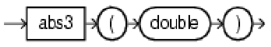
Examples

Consider the query q68. Given the data stream SFunc with schema (c1 integer, c2 float, c3 bigint), the query returns the stream.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.5
1000:       +           0.7
1200:       +           0.89
2000:       +           0.4
```

12.2.4 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).

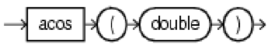
Examples

Consider the query q69. Given the data stream SFunc with schema (c1 integer, c2 float, c3 bigint, c4 double), the query returns the stream.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.25334
1000:       +           4.64322
1200:       +           1.4672272
2000:       +           2.66777
```

12.2.5 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).

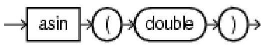
Examples

Consider the query q73. Given the data stream SFunc with schema (c1 integer, c2 double, c3 bigint), the query returns the stream.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           1.0471976
1000:       +           0.79539883
1200:       +           0.4734512
2000:       +           1.1592795
```

12.2.6 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)`.

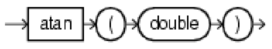
Examples

Consider the query `q74`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.5235988
1000:       +           0.7753975
1200:       +           1.0973451
2000:       +           0.41151685
```

12.2.7 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

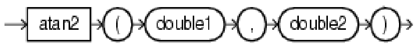
Consider the query `q75`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

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

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.4636476
1000:       +           0.61072594
1200:       +           0.7272627
2000:       +           0.3805064
```

12.2.8 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`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

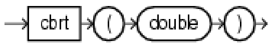
```
<query id="q63"><![CDATA[
    select atan2(c2,c2) from SFunc
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.7853982
1000:       +           0.7853982
1200:       +           0.7853982
2000:       +           0.7853982
```

12.2.9 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)`.

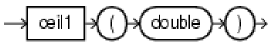
Examples

Consider the query `q76`. Given the data stream `SFunc` with schema `(c1 integer, c2 float, c3 bigint)`, the query returns the stream.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.7937005
1000:       +           0.887904
1200:       +           0.9619002
2000:       +           0.73680633
```

12.2.10 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)`.

Examples

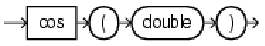
Consider the query `q77`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           1.0
1000:       +           1.0
1200:       +           1.0
2000:       +           1.0
```

12.2.11 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)`.

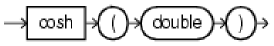
Examples

Consider the query `q61`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.87758255
1000:       +           0.7648422
1200:       +           0.62941206
2000:       +           0.921061
```

12.2.12 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

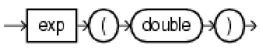
Consider the query q78. Given the data stream SFunc with schema (c1 integer, c2 double, c3 bigint), the query returns the stream.

tkdata140.cqlx, data/inpSColtFunc.txt, log/outSColtcosh.txt

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           1.127626
1000:       +           1.255169
1200:       +           1.4228927
2000:       +           1.0810723
```

12.2.13 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.

Examples

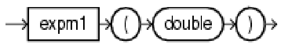
Consider the query q79. Given the data stream SFunc with schema (c1 integer, c2 double, c3 bigint), the query returns the stream.

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

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           1.6487212
1000:       +           2.0137527
1200:       +           2.4351296
2000:       +           1.4918247
```

12.2.14 expm1

Syntax



Purpose

`expm1` returns the computation that Figure 12-1 shows as a `double`, where `x` is the `double` argument and *e* is Euler's number.

Figure 12-1 java.lang.Math Expm1

$$e^x - 1$$

Note that for values of `x` near 0, the exact sum of `expm1(x) + 1` is much closer to the true result of Euler's number *e* raised to the power of `x` than `exp(x)`.

For more information, see:

- `http://java.sun.com/javase/6/docs/api/java/lang/Math.html#expm1(double)`
- exp.

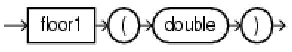
Examples

Consider the query `q80`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.6487213
1000:       +           1.0137527
1200:       +           1.4351296
2000:       +           0.49182472
```

12.2.15 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).

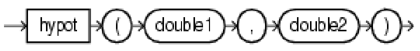
Examples

Consider the query q81. Given the data stream SFunc with schema (c1 integer, c2 double, c3 bigint), the query returns the stream.

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

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

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

12.2.16 hypot

Syntax



Purpose

hypot returns the hypotenuse (see Figure 12-2) of the double arguments as a double.

Figure 12-2 java.lang.Math hypot

$$\sqrt{(x^2 + y^2)}$$

This function takes the following arguments:

- `double1`: the `x` value.
- `double2`: the `y` value.

The hypotenuse is computed without intermediate overflow or underflow.

For more information, see `http://java.sun.com/javase/6/docs/api/java/lang/Math.html#hypot(double,%20double)`.

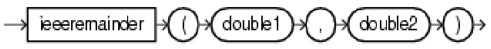
Examples

Consider the query `q82`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.70710677
1000:       +           0.98994946
1200:       +           1.2586501
2000:       +           0.56568545
```

12.2.17 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)`.

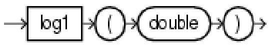
Examples

Consider the query `q72`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

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

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

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

12.2.18 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.

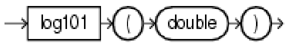
Examples

Consider the query `q83`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           -0.6931472
1000:       +           -0.35667497
1200:       +           -0.11653383
2000:       +           -0.9162907
```

12.2.19 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)`.

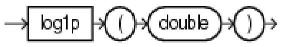
Examples

Consider the query `q84`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

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

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

Timestamp   Tuple Kind  Tuple
  10:           +           -0.30103
1000:       +           -0.15490197
1200:       +           -0.050610002
2000:       +           -0.39794
```

12.2.20 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.

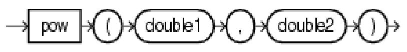
Examples

Consider the query `q85`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.4054651
1000:       +           0.53062826
1200:       +           0.63657683
2000:       +           0.33647224
```

12.2.21 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

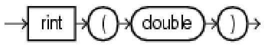
Consider the query `q65`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.70710677
1000:       +           0.7790559
```

```
1200:       +           0.9014821
2000:       +           0.69314486
```

12.2.22 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)`.

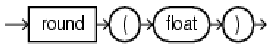
Examples

Consider the query `q86`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.0
1000:       +           1.0
1200:       +           1.0
2000:       +           0.0
```

12.2.23 round

Syntax



Purpose

`round` returns the closest `integer` to the argument.

For more information, see `http://java.sun.com/javase/6/docs/api/java/lang/Math.html#round(float)`.

Input/Output Types

The input/output types for this function are as follows:

Input Type	Output Type
DOUBLE	DOUBLE
INTEGER	INTEGER
FLOAT	FLOAT
BIGINT	BIGINT
BIGDECIMAL	BIGDECIMAL

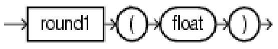
Examples

Consider the query `q87`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           1
1000:       +           1
1200:       +           1
2000:       +           0
```

12.2.24 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

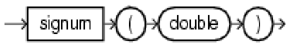
Consider the query `q88`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

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

Timestamp   Tuple
  10        1,0.5,8
```

```
1000       4,0.7,6
1200       3,0.89,12
2000       8,0.4,4

Timestamp   Tuple Kind  Tuple
  10:       +           1
1000:       +           1
1200:       +           1
2000:       +           0
```

12.2.25 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).

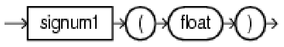
Examples

Consider the query q70. Given the data stream SFunc with schema (c1 integer, c2 double, c3 bigint), the query returns the stream.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           1.0
1000:       +           -1.0
1200:       +           -1.0
2000:       +           1.0
```

12.2.26 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`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

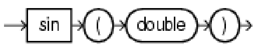
```
<query id="q71"><![CDATA[
    select signum1(c2) from SFunc
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           1.0
1000:       +           -1.0
1200:       +           -1.0
2000:       +           1.0
```

12.2.27 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`. Given the data stream `SFunc` with schema (`c1 integer`, `c2 float`, `c3 bigint`), the query returns the stream.

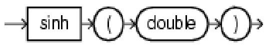
```
<query id="q60"><![CDATA[
    select sin(c2) from SFunc
]]></query>
```

```
Timestamp   Tuple
  10        1,0.5,8
1000        4,0.7,6
```

```
1200       3,0.89,12
2000       8,0.4,4

Timestamp   Tuple Kind  Tuple
  10:       +           0.47942555
1000:       +           0.64421767
1200:       +           0.7770717
2000:       +           0.38941833
```

12.2.28 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)`.

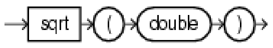
Examples

Consider the query `q89`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

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

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.5210953
1000:       +           0.75858366
1200:       +           1.012237
2000:       +           0.41075233
```

12.2.29 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)`.

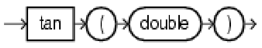
Examples

Consider the query `q64`. Given the data stream `SFunc` with schema (`c1 integer`, `c2 float`, `c3 bigint`), the query returns the stream.

```
<query id="q64"><![CDATA[
    select sqrt(c2) from SFunc
]]></query>

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.70710677
1000:       +           0.83666
1200:       +           0.9433981
2000:       +           0.6324555
```

12.2.30 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)`.

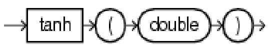
Examples

Consider the query `q62`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

```
<query id="q62"><![CDATA[
    select tan(c2) from SFunc
]]></query>

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.5463025
1000:       +           0.8422884
1200:       +           1.2345995
2000:       +           0.42279324
```

12.2.31 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`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

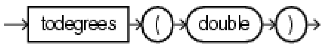
```
<query id="q90"><![CDATA[
    select tanh(c2) from SFunc
]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
  10:       +           0.46211717
1000:       +           0.6043678
1200:       +           0.7113937
2000:       +           0.37994897
```

12.2.32 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.
- cos.

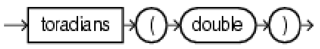
Examples

Consider the query `q91`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

```
<query id="q91"><![CDATA[
    select todegrees(c2) from SFunc
]]></query>

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

Timestamp   Tuple Kind  Tuple
  10:       +           28.64789
1000:       +           40.107044
1200:       +           50.993244
2000:       +           22.918312
```

12.2.33 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
- cos.

Examples

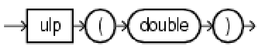
Consider the query `q92`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

```
<query id="q92"><![CDATA[
    select toradians(c2) from SFunc
]]></query>

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

Timestamp   Tuple Kind  Tuple
  10:       +           0.008726646
```

```
1000:       +           0.012217305
1200:       +           0.0155334305
2000:       +           0.006981317
```

12.2.34 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)`.

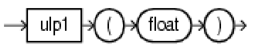
Examples

Consider the query `q93`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the stream.

```
<query id="q93"><![CDATA[
    select ulp(c2) from SFunc
]]></query>

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

Timestamp   Tuple Kind  Tuple
  10:       +           1.110223E-16
1000:       +           1.110223E-16
1200:       +           1.110223E-16
2000:       +           5.551115E-17
```

12.2.35 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`. Given the data stream `SFunc` with schema `(c1 integer, c2 double, c3 bigint)`, the query returns the relation.

```
<query id="q94"><![CDATA[
    select ulp1(c2) from SFunc
]]></query>

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

Timestamp   Tuple Kind  Tuple
  10:       +           5.9604645E-8
1000:       +           5.9604645E-8
1200:       +           5.9604645E-8
2000:       +           2.9802322E-8
```

13 User-Defined Functions

How you can write user-defined functions for use in Oracle Continuous Query Language (Oracle CQL) to perform more advanced or application-specific operations on stream data than is possible using built-in functions is described.

For more information, see Functions.

13.1 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

Note:

You can also create user-defined windows (see User-Defined Stream-to-Relation Window Operators).

To make your user-defined function available for use in Oracle CQL queries, the JAR file that contains the user-defined function implementation class must be in the Oracle Event Processing server class path or the Oracle Event Processing server class path must be modified to include the JAR file.

For more information, see:

- Types of User-Defined Functions
- User-Defined Function Data Types
- User-Defined Functions and the Oracle Event Processing Server Cache
- Implementing a User-Defined Function
- Functions.

13.1.1 Types of User-Defined Functions

You can create the following types of user-defined functions:

- User-Defined Single-Row Functions
- User-Defined Aggregate Functions.

You can create overloaded functions and you can override built-in functions.

13.1.1.1 User-Defined Single-Row Functions

A user-defined single-row function is a function that returns a single result row for every row of a queried stream or view (for example, like the `concat` built-in function does).

For more information, see How to Implement a User-Defined Single-Row Function.

13.1.1.2 User-Defined Aggregate Functions

A user-defined aggregate is a function that implements `com.bea.wlevs.processor.AggregationFunctionFactory` and returns a single aggregate result based on group of tuples, rather than on a single tuple (for example, like the `sum` built-in function does).

Consider implementing your aggregate function so that it performs incremental processing, if possible. This will improve scalability and performance because the cost of (re)computation on arrival of new events will be proportional to the number of new events as opposed to the total number of events seen thus far.

For more information, see How to Implement a User-Defined Aggregate Function.

13.1.2 User-Defined Function Data Types

User-defined functions support any of the built-in Oracle CQL data types listed in Oracle CQL Built-in Data Types. See the table in that section for a list of Oracle CQL data types and their Java equivalents.

The Oracle CQL data types shown there list the data types you can specify in the Oracle CQL statement you use to register your user-defined function. The Java equivalents are the Java data types you can use in your user-defined function implementation.

At run time, Oracle Event Processing maps between the Oracle CQL data type and the Java data type. If your user-defined function returns a data type that is not in this list, Oracle Event Processing will throw a `ClassCastException`.

For more information about data conversion, see Data Type Conversion.

13.1.3 User-Defined Functions and the Oracle Event Processing Server Cache

You can access an Oracle Event Processing cache from an Oracle CQL statement or user-defined function.

13.2 Implementing a User-Defined Function

This section describes:

- How to Implement a User-Defined Single-Row Function
- How to Implement a User-Defined Aggregate Function.

For more information, see Introduction to Oracle CQL User-Defined Functions.

13.2.1 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.

   Ensure that the data type of the return value corresponds to a supported data type as User-Defined Function Data Types describes.

   For more information on accessing the Oracle Event Processing cache from a user-defined function, see User-Defined Functions and the Oracle Event Processing Server Cache.

   ```
   package com.bea.wlevs.example.function;

   public class MyMod {
       public Object execute(int arg0, int arg1) {
           return new Integer(arg0 % arg1);
       }
   }
   ```

2. Compile the user-defined function Java implementation class and register the class in your Oracle Event Processing application assembly file.

   ```
   <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).

3. Invoke your user-defined function in the select list of a `SELECT` statement or the condition of a `WHERE` clause.

   ```
   ...
   <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>
   ...
   ```

13.2.2 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 shown in the below example.

   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 supports incremental processing.

   Ensure that the data type of the return value corresponds to a supported data type as User-Defined Function Data Types describes.

   For more information on accessing the Oracle Event Processing cache from a user-defined function, see User-Defined Functions and the Oracle Event Processing Server Cache.

```
package com.bea.wlevs.test.functions;

import com.bea.wlevs.processor.AggregationFunction;
import com.bea.wlevs.processor.AggregationFunctionFactory;

public class Variance implements AggregationFunctionFactory, AggregationFunction {

    private int count;
    private float sum;
    private float sumSquare;

    public Class<?>[] getArgumentTypes() {
        return new Class<?>[] {Integer.class};
    }

    public Class<?> getReturnType() {
        return Float.class;
    }

    public AggregationFunction newAggregationFunction() {
        return new Variance();
    }

    public void releaseAggregationFunction(AggregationFunction function) {
    }

    public Object handleMinus(Object[] params) {
        if (params != null && params.length == 1) {
            Integer param = (Integer) params[0];
            count--;
            sum -= param;
            sumSquare -= (param * param);
        }

        if (count == 0) {
            return null;
        } else {
            return getVariance();
        }
    }

    public Object handlePlus(Object[] params) {
        if (params != null && params.length == 1) {
            Integer param = (Integer) params[0];
            count++;
            sum += param;
            sumSquare += (param * param);
        }
```

```
            if (count == 0) {
                return null;
            } else {
                return getVariance();
            }
        }

        public Float getVariance() {
            float avg = sum / (float) count;
            float avgSqr = avg * avg;
            float var = sumSquare / (float)count - avgSqr;
            return var;
        }

        public void initialize() {
            count = 0;
            sum = 0.0F;
            sumSquare = 0.0F;
        }

    }
```

2. Compile the user-defined function Java implementation class and register the class in your Oracle Event Processing application assembly file.

```
  <wlevs:processor id="testProcessor">
     <wlevs:listener ref="providerCache"/>
     <wlevs:listener ref="outputCache"/>
     <wlevs:cache-source ref="testCache"/>
     <wlevs:function function-name="var" is-incremental="true">
       <bean class="com.bea.wlevs.test.functions.Variance"/>
     </wlevs:function>
   </wlevs:processor>
```

   You must set the `is-incremental` attribute of the function element to `true` to indicate that the user-defined function, `var`, has an incremental implementation. Setting the `is-incremental` function to `true` guarantees that Oracle Event Processing calls the `handleMinus` method when it purges events from the current processing window. If this attribute is set to `false` (default), then the `handleMinus` function is never called and instead Oracle Event Processing provides the full set of events of the current window with every call to the `handlePlus` method.

3. Invoke your user-defined function in the select list of a `SELECT` statement or the condition of a `WHERE` clause.

```
...
<query id="uda6"><![CDATA[
    select var(c2) from S4[range 3]
]]></query>
...
```

   At run-time, when the user-defined aggregate is executed, and a new event becomes active in the window of interest, the aggregations will have to be recomputed (since the set over which the aggregations are defined has a new member). To do so, Oracle Event Processing passes only the new event (rather than the entire active set) to the appropriate handler context by invoking the appropriate `handlePlus*` method. This state can now be updated to include the new event. Thus, the aggregations have been recomputed in an incremental fashion.

   Similarly, when an event expires from the window of interest, the aggregations will have to be recomputed (since the set over which the aggregations are defined has lost a member). To do so, Oracle Event Processing passes only the expired event (rather than the entire active set) to the appropriate handler context by invoking the appropriate `handleMinus` method. As before, the state in the handler context can be incrementally updated to accommodate expiry of the event in an incremental fashion.

14
Oracle CQL Queries, Views, and Joins

A reference and usage guidelines for queries, views, and joins in Oracle Continuous Query Language (Oracle CQL) is provided. 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 `subquery` is a nested or embedded query inside another, via the mechanism of views.

A top-level `VIEW` statement that you create using the `VIEW` statement is called a **view**.

A `join` is a query that combines rows from two or more streams, views, or relations.

For more information, see:

- Lexical Conventions
- Documentation Conventions
- Basic Elements of Oracle CQL
- Common Oracle CQL DDL Clauses
- Oracle CQL Statements.

14.1 Introduction to Oracle CQL Queries, Subqueries, Views, and Joins

An Oracle CQL query is an operation that you express in Oracle CQL syntax and execute on an Oracle Event Processing CQL Processor to process data from one or more streams or views. For more information, see Queries.

An Oracle subquery is the nesting or embeddeding of one query inside another via the mechanism of views. For more information, see Views.

An Oracle CQL view represents an alternative selection on a stream or relation. For more information, see Views.

Oracle Event Processing performs a join whenever multiple streams appear in the `FROM` clause of the query. For more information, see Joins.

The following example shows typical Oracle CQL queries defined in an Oracle CQL processor component configuration file for the processor named `proc`.

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

```
                <view id="lastEvents" schema="cusip mbid srcId bidQty ask askQty seq"><![CDATA[
                    select cusip, mod(bid) as mbid, srcId, bidQty, ask, askQty, seq
                    from filteredStream[partition by srcId, cusip rows 1]
                ]]></view>
                <query id="q1"><![CDATA[
                    SELECT *
                    FROM   lastEvents [Range Unbounded]
                    WHERE  price > 10000
                ]]></query>
            </rules>
        </processor>
</n1:config>
```

The rules element contains each Oracle CQL statement in a `view` or `query` child element:

- `rule`: contains Oracle CQL statements that register or create user-defined windows. The `rule` element `id` attribute must match the name of the window.

  The `rule` element specifies an Oracle CQL statement that registers a user-defined window named `range_slide`. The `rule` element `id` must match the name of the window.
- `view`: contains Oracle CQL view statements (the Oracle CQL equivalent of subqueries). The `view` element `id` attribute defines the name of the view.

  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.

  The `query` element specifies an Oracle CQL query statement. The query `q1` selects from the view `lastEvents`. 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.

Each Oracle CQL statement is contained in a `<![CDATA[` ... `]]>` tag and does *not* end in a semicolon (`;`).

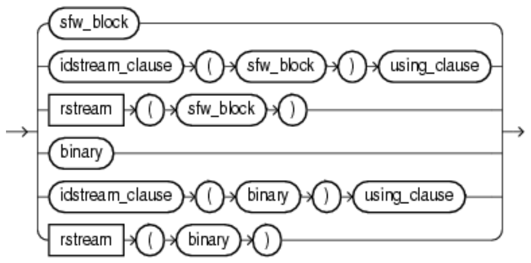
For more information, see:

- Lexical Conventions
- Oracle CQL Statements.

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 Select, From, Where Block.
- `idstream_clause`: use this clause to specify an input `IStream` or delete `DStream` relation-to-stream operator that applies to the query.

  For more information, see IDStream Clause.
- `rstream`: use this clause to specify an `RStream` relation-to-stream operator that applies to the query.

  For more information, see RStream Relation-to-Stream Operator.
- `binary`: use this clause to perform set operations on the tuples that two queries or views return.

  For more information, see Binary Clause.

The following sections discuss the basic query types that you can create:

- Simple Query
- Built-In Window Query
- User-Defined Window Query
- MATCH_RECOGNIZE Query
- Relational Database Table Query
- XMLTABLE Query
- Function TABLE Query
- Cache Query.

For more information, see:

- Sorting Query Results
- Detecting Differences in Query Results
- Parameterized Queries.

14.2.1 Query Building Blocks

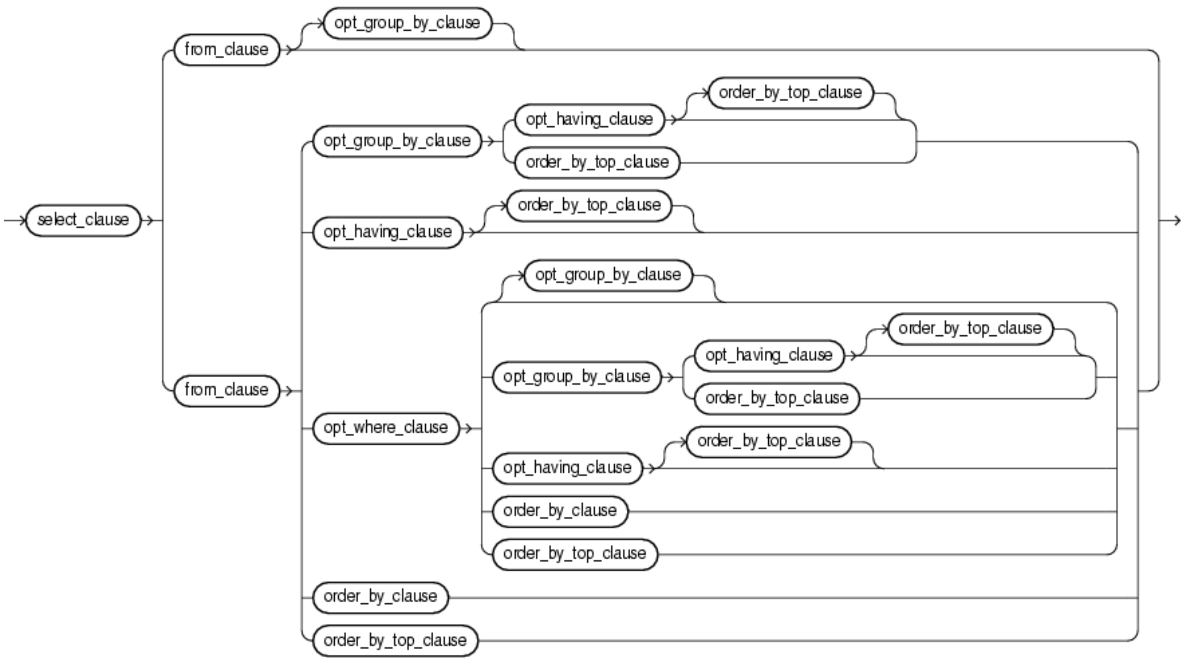
This section summarizes the basic building blocks that you use to construct an Oracle CQL query, including:

- Select, From, Where Block
- Select Clause
- From Clause
- Where Clause
- Group By Clause
- Order By Clause
- Having Clause
- Binary Clause
- IDStream 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*::=**



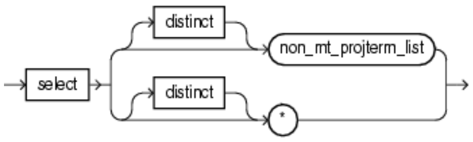
The `sfw_block` is made up of the following parts:

- Select Clause
- From Clause
- Where Clause
- Group By Clause
- Order By Clause
- 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 Event Processing to return from one or more streams or views. The number of stream elements, and their data type and length, are determined by the elements of the select list.

Optionally, specify distinct if you want Oracle Event Processing to return only one copy of each set of duplicate tuples selected. Duplicate tuples are those with matching values for each expression in the select list.

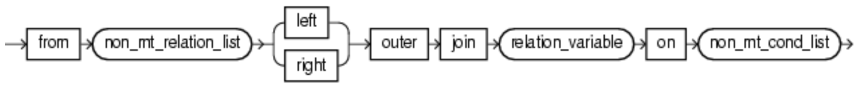
For more information, see Figure 16-3.

14.2.1.3 From Clause

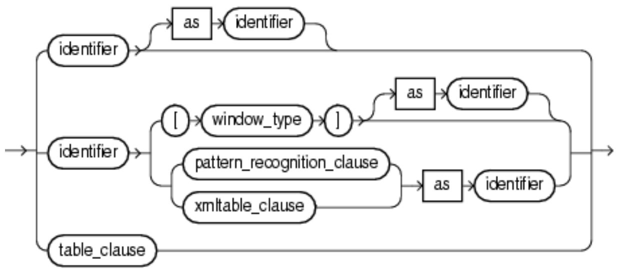
Use this clause to specify the streams and views that provide the stream elements you specify in the select_clause (see Select Clause).

The from_clause may specify one or more comma-delimited relation_variable clauses.

***from_clause*::=**



***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 Aliases in the relation_variable Clause).

If you create a join (see 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. The following example shows how to use stream names to distinguish between the customerID stream element in the OrderStream and the customerID stream element in the CustomerStream.

```
<query id="q0"><![CDATA[
    select * from OrderStream, CustomerStream
    where
        OrderStream.customerID = CustomerStream.customerID
]]></query>
```

Otherwise, fully qualified stream element names are optional. However, Oracle recommends that you always qualify stream element references explicitly. Oracle Event Processing often does less work with fully qualified stream element names.

For more information, see:

- MATCH_RECOGNIZE Query
- XMLTABLE Query
- Function TABLE Query.

14.2.1.4 Where Clause

Use this optional clause to specify conditions that determine when the `select_clause` returns results (see Select Clause).

Because Oracle CQL applies the `WHERE` clause before `GROUP BY` or `HAVING`, if you specify an aggregate function in the `SELECT` clause, you must test the aggregate function result in a `HAVING` clause, not the `WHERE` clause.

For more information, see:

- Built-In Aggregate Functions and the Where, Group By, and Having Clauses
- Colt Aggregate Functions and the Where, Group By, and Having Clauses.

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.

Because Oracle CQL applies the `WHERE` clause before `GROUP BY` or `HAVING`, if you specify an aggregate function in the `SELECT` clause, you must test the aggregate function result in a `HAVING` clause, not the `WHERE` clause.

For more information, see:

- Group By Clause
- Built-In Aggregate Functions and the Where, Group By, and Having Clauses
- Colt Aggregate Functions and the Where, Group By, and Having Clauses.

14.2.1.6 Order By Clause

Use this optional clause to order all results or the top-`n` results.

For more information, see 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 Event Processing returns summary results for all groups.

Because Oracle CQL applies the `WHERE` clause before `GROUP BY` or `HAVING`, if you specify an aggregate function in the `SELECT` clause, you must test the aggregate function result in a `HAVING` clause, not the `WHERE` clause.

For more information, see:

- Built-In Aggregate Functions and the Where, Group By, and Having Clauses
- Colt Aggregate Functions and the Where, Group By, and Having Clauses.

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`.

14.2.1.9 IDStream Clause

Use this clause to take either a select-from-where clause or binary clause and return its results as one of `IStream` or `DStream` relation-to-stream operators.

You can succinctly detect differences in query output by combining an `IStream` or `Dstream` operator with the `using_clause`.

For more information, see:

- IStream Relation-to-Stream Operator
- DStream Relation-to-Stream Operator
- Detecting Differences in Query Results.

14.2.2 Simple Query

The following example shows a simple query that selects all stream elements from a single stream.

```
<query id="q0"><![CDATA[
    select * from OrderStream where orderAmount > 10000.0
]]></query>
```

For more information, see Query.

14.2.3 Built-In Window Query

The following example 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.

tkdata2.cqlx

```
<query id="BBAQuery"><![CDATA[
    select * from S2 [range 5 minutes] where S2.c1 > 10
]]></query>
```

For more information, see:

- Stream-to-Relation Operators (Windows).

14.2.4 User-Defined Window Query

The following example shows a query that selects all stream elements from stream `S12`, with schema `(c1 integer, c2 float)`, using a user-defined window of type `range_slide` based on user-defined Java class `MyRangeSlideWindow.java`.

```
<rule id="range_slide"><![CDATA[
    register window range_slide(winrange int, winslide int) implement using "MyRangeSlideWindow"
]]></rule>
<query id="q79"><![CDATA[
    select * from S12 [range_slide(10,5)]
]]></query>
```

For more information, see User-Defined Stream-to-Relation Window Operators.

14.2.5 MATCH_RECOGNIZE Query

The following example shows a query that uses the `MATCH_RECOGNIZE` clause to express complex relationships among the stream elements of `ItemTempStream`.

```
<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:

- Pattern Recognition With MATCH_RECOGNIZE.

14.2.6 Relational Database Table Query

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

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

14.2.7 XMLTABLE Query

Use this query to map the results of an XPath or XQuery expression into tuples.

XMLTABLE has the following sub-clauses:

- XMLNAMESPACES -- Optional. A string with a set of XML namespace declarations that can be used in the query expression.
- XQuery_string -- A string with the XQuery or XPath string to use to query the XML.
- PASSING BY VALUE -- Points to the XML that is being used for input.

- COLUMNS -- Optional. Defines the output properties of the result.
  - PATH -- Optional. A subclause of COLUMNS that specifies an XPath expression that points to where values for that property should be drawn from the XML.

The following example 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 data types 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.

```
<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>
```

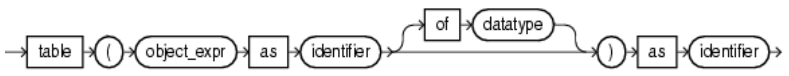
For more information, see:

- Stream-to-Stream Operators
- SQL/XML (SQLX).

14.2.8 Function TABLE Query

Use the `TABLE` clause to access the multiple rows returned by a built-in or user-defined function in the `FROM` clause of an Oracle CQL query. The `TABLE` clause converts the set of returned rows into an Oracle CQL relation. Because this is an external relation, you must join the `TABLE` function clause with a stream.

***table_clause*::=**



(*object_expr*, *identifier*, *datatype*::=)

Note the following:

- The function must return an array type or `Collection` type.
- You must join the `TABLE` function clause with a stream.

The following example shows a data cartridge `TABLE` clause that invokes the Oracle Spatial method `getContainingGeometries`, passing in one parameter (`InputPoints.point`). The return value of this method, a `Collection`, is aliased as `validGeometries`. The relation that the `TABLE` clause returns is aliased as `R2`.

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

The following example shows an invalid data cartridge `TABLE` query that fails to join the data cartridge `TABLE` clause with another stream because the function `getAllGeometries@spatial` was called without any parameters. Oracle Event Processing invokes the data cartridge method only on the arrival of elements on the joined stream.

```
<query id="q2"><![CDATA[
RSTREAM (
    SELECT
        R2.validGeometries.shape as containingGeometry
    FROM
        TABLE (getAllGeometries@spatial () as validGeometries) AS R2
)
]]></query>
```

For more examples, see Functions.

14.2.9 Cache Query

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

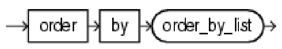
Oracle Event Processing cache event sources are pull data sources: that is, Oracle Event Processing polls the event source on arrival of an event on the data stream.

For more information, see Oracle CQL Queries and the Oracle Event Processing Server Cache.

14.2.10 Sorting Query Results

Use the `ORDER BY` clause to order the rows selected by a query.

***order_by_clause*::=**



Sorting by position is useful in the following cases:

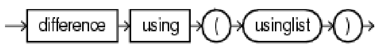
- To order by a lengthy select list expression, you can specify its position in the `ORDER BY` clause rather than duplicate the entire expression.
- For compound queries containing set operators `UNION`, `INTERSECT`, `MINUS`, or `UNION ALL`, the `ORDER BY` clause must specify positions or aliases rather than explicit expressions. Also, the `ORDER BY` clause can appear only in the last component query. The `ORDER BY` clause orders all rows returned by the entire compound query.

The mechanism by which Oracle Event Processing sorts values for the `ORDER BY` clause is specified by your Java locale.

14.2.11 Detecting Differences in Query Results

Use the `DIFFERENCE USING` clause to succinctly detect differences in the `IStream` or `DStream` of a query.

***using_clause*::=**



Consider the query.

```
<query id="q0">
    ISTREAM (
        SELECT c1 FROM S [RANGE 1 NANOSECONDS]
    )    DIFFERENCE USING (c1)
</query>
```

Table 14-1 shows sample input for this query. The Relation column shows the contents of the relation `S [RANGE 1 NANOSECONDS]` and the Output column shows the query results after the `DIFFERENCE USING` clause is applied. This clause allows you to succinctly detect only differences in the `IStream` output.

Table 14-1 DIFFERENCE USING Clause Affect on IStream

Input	Relation	Output
1000: +5	{5}	+5
1000: +6	{5, 6}	+6
1000: +7	{5, 6, 7}	+7
1001: +5	{5, 6, 7, 5}	
1001: +6	{5, 6, 7, 5, 6}	
1001: +7	{5, 6, 7, 5, 6, 7}	
1001: +8	{5, 6, 7, 5, 6, 7, 8}	+8
1002: +5	{5, 6, 7, 5, 6, 7, 8, 5}	
1003: -5	{5, 6, 7, 5, 6, 7, 8}	
1003: -5	{5, 6, 7, 6, 7, 8}	
1003: -5	{6, 7, 6, 7, 8}	
1003: -6	{6, 7, 7, 8}	

Table 14-1 (Cont.) DIFFERENCE USING Clause Affect on IStream

Input	Relation	Output
`1003:    -6`	`{7, 7, 8}`	
`1003:    -7`	`{7, 8}`	
`1003:    -7`	`{8}`	
`1003:    -8`	`{}`	
`1004:    +5`	`{5}`	`+5`

When you specify the `usinglist` in the `DIFFERENCE USING` clause, you may specify columns by:

- attribute name: use this option when you are selecting by attribute name.
- alias: use this option when you want to include the results of an expression where an alias is specified.
- position: use this option when you want to include the results of an expression where no alias is specified.

  Specify position as a constant, positive integer starting at 1, reading from left to right.

  The following example specifies the result of expression `funct(c2, c3)` by its position (3) in the `DIFFERENCE USING` clause `usinglist`.

```
<query id="q1">
    ISTREAM (
        SELECT c1, log(c4) as logval, funct(c2, c3) FROM S [RANGE 1 NANOSECONDS]
    )    DIFFERENCE USING (c1, logval, 3)
</query>
```

You can use the `DIFFERENCE USING` clause with both `IStream` and `DStream` operators.

For more information, see:

- IStream Relation-to-Stream Operator
- DStream Relation-to-Stream Operator.

14.2.12 Parameterized Queries

You can parameterize an Oracle CQL query and bind parameter values at run time using the `:n` character string as a placeholder, where *n* is a positive integer that corresponds to the position of the replacement value in a `params` element.

Note:

You cannot parameterize a view.

The following example shows a parameterized Oracle CQL query.

```
<n1:config xmlns:n1="http://www.bea.com/ns/wlevs/config/application"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    ...
    <processor>
        <name>myProcessor</name>
        <rules>
            <query id="MarketRule"><![CDATA[
                 SELECT symbol, AVG(price) AS average, :1 AS market
                 FROM StockTick [RANGE 5 SECONDS]
                 WHERE symbol = :2
            ]]></query>
        </rules>
        <bindings>
            <binding id="MarketRule">
                <params id="nasORCL">NASDAQ, ORCL</params>
                <params id="nyJPM">NYSE, JPM</params>
                <params id="nyWFC">NYSE, WFC</params>
            </binding>
        </bindings>
    </processor>
    <processor>
        <name>summarizeResults</name>
        <rules>
            <query id="SummarizeResultsRule"><![CDATA[
                select
                    crossRate1 || crossRate2 as crossRatePair,
                    count(*) as totalCount,
                    :1 as averageInternalPrice
                from CrossRateStream
                group by crossRate1,crossRate2
                having :2
            ]]></query>
        </rules>
        <bindings>
            <binding id="SummarizeResultsRule">
                <params id="avgcount">avg(internalPrice), count(*) > 0</params>
            </binding>
        </bindings>
    </processor>
</n1:config>
```

In this example, the:

- `MarketRule` query specifies two parameters: the third term in the `SELECT` and the value of `symbol` in the `WHERE` clause
- `SummarizeResultsRule` query specifies two parameters: the third term in the `SELECT` and the value of the `HAVING` clause.

This section describes:

- Parameterized Queries in Oracle CQL Statements
- The `bindings` Element
- Run-Time Query Naming
- Lexical Conventions for Parameter Values
- Parameterized Queries at Runtime
- Replacing Parameters Programmatically.

14.2.12.1 Parameterized Queries in Oracle CQL Statements

You may specify a placeholder anywhere an arithmetic expression or a `String` literal is legal in an Oracle CQL statement. For example:

- `SELECT` list items

- `WHERE` clause predicates
- `WINDOW` constructs (such as `RANGE`, `SLIDE`, `ROWS`, and `PARTITION BY`)
- `PATTERN` duration clause.

For more information, see:

- *arith_expr*
- Literals.

14.2.12.2 The `bindings` Element

Parameter values are contained by a `bindings` element. There may be one `bindings` element per `processor` element.

For each parameterized query, the `bindings` element must contain a `binding` element with the same `id` as the query.

The `binding` element must contain one or more `params` elements. Each `params` element must have a unique `id` and must contain a comma separated list of parameter values equal in number to the number of placeholder characters (`:`*n*) in the corresponding query.

The order of the parameter values corresponds to placeholder characters (`:`*n*) in the parameterized query, such that `:1` corresponds to the first parameter value, `:2` corresponds to the second parameter value, and so on. You may use placeholder characters (`:`*n*) in any order. That is, `:1` corresponds to the first parameter value whether it precedes or follows `:2` in a query. A placeholder number can be used only once in a query.

For more information, see:

- Lexical Conventions for Parameter Values
- Parameterized Queries at Runtime.

14.2.12.3 Run-Time Query Naming

When a binding instantiates a parameterized query, Oracle Event Processing creates a new query at run time with the name queryId_paramId. For example, the run-time name of the first query instantiated by the MarketRule binding is `MarketRule_nasORCL`.

To avoid run-time naming conflicts, be sure query ID and parameter ID combinations are unique.

14.2.12.4 Lexical Conventions for Parameter Values

Each `params` element must have a unique `id` and must contain a comma separated list of parameter values equal in number to the number of placeholder characters (`:`*n*) in the corresponding query.

Table 14-2 Parameterized Query Parameter Value Lexical Conventions

Convention	Example	Replacement Value

Primitive type literals	`<params id="p1">NASDAQ, 200.0</params>`	`:1 = NASDAQ`
		`:2 = 200.0`

Oracle CQL fragments	`<params id="p1">count(*), avg(val)</params>`	`:1 = count(*)`
		`:2 = avg(val)`

Quotes	`<params id="p1">'alert', "Seattle, WA", 'fun' || "house", one "two" 3</params>`	`:1 = 'alert'`
		`:2 = "Seattle, WA"`
		`:3 = 'fun' || "house"`
		`:4 = one "two" 3`

In an Oracle CQL query, a placeholder within single or double quotes is a `String` literal. The following query is not a parameterized query:

```
SELECT ":1" as symbol, price FROM StockTick [RANGE 5 SECONDS]
```

Oracle Event Processing parses this query as assigning the `String` literal `":1"` to alias `symbol`. To make this query into a parameterized query, use:

```
SELECT :1 as symbol, price FROM StockTick [RANGE :2 SECONDS]
```

And define a `params` element like this:

```
<params id="p1">"ORCL", 5</params>
```

Because the parameter value (`ORCL`) does not contain a comma, the quotes are not required. You could specify a `params` element like this:

```
<params id="p1">ORCL, 5</params>
```

However, if the parameter value does contain a comma, then you must use quotes around the parameter value. Consider this parameterized query:

```
SELECT :1 = cityAndState AS cityOfInterest FROM channel1 [RANGE :2 SECONDS]
```

Where `cityAndState` has values like `"Seattle, WA"` or `"Ottawa, ON"`. In this case, you must specify a `params` element like this:

```
<params id="p1">"Seattle, WA", 5</params>
<params id="p1">"Ottawa, ON", 5</params>
```

Commas are allowed only in quoted parameter values that signify string values. Commas are not allowed as a separator character in unquoted parameter values. For example:

`"Seattle, WA"` is valid, because the comma is part of the string.

`PARTITION BY fromRate,toRate ROWS 10` is invalid. Create the following two parameters instead:

```
PARTITION BY fromRate ROWS 10
PARTITION BY toRate ROWS 10
```

14.2.12.5 Parameterized Queries at Runtime

Each `params` element effectively causes a new Oracle CQL query to execute with the new parameters. At rule execution time, Oracle CQL substitutes parameter values for placeholder characters, from left to right.

```
SELECT symbol, AVG(price) AS average, NASDAQ AS market
FROM StockTick [RANGE 5 SECONDS]
WHERE symbol = ORCL

SELECT symbol, AVG(price) AS average, NYSE AS market
FROM StockTick [RANGE 5 SECONDS]
WHERE symbol = JPM

SELECT symbol, AVG(price) AS average, NYSE AS market
FROM StockTick [RANGE 5 SECONDS]
WHERE symbol = WFC
```

You can dynamically modify or delete single parameter sets by the `id` attribute of the `params` element using JMX or `wlevs.Admin`.

14.2.12.6 Replacing Parameters Programmatically

If you use the `CQLProcessorMBean.replaceAllBoundParameters()` method to programmatically replace parameters in a parameterized query, any existing parameters not replaced by the method are automatically removed from the query.

14.2.13 Subqueries

A subquery can be specified in the FROM clauses of a queries where sources relations/streams are specified. Subquery support will also be extended to SET queries.

```
CREATE QUERY q0 AS
SELECT  prodid, sum(sales)
FROM (SELECT prodid AS prodid, sales AS sales FROM sales_stream [RANGE 24 HOURS]) AS
foo
GROUP BY prodid;
```

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.

You can specify any query type in the definition of your view. For more information, see Queries.

For complete details on the view statement, see View.

Note:

Subqueries are used with binary set operators such as union, union all, and minus. Use parentheses in the subqueries to ensure the right precedence is applied to the query.

The 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.

```
<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.

14.3.1 Views and Joins

If you create a join between two or more views that have some stream element names in common, then you must qualify stream element names with names of streams. The following example 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.

```
<view id="MAXBIDMINASK" schema="cusip bidseq bidSrcId bid askseq askSrcId ask bidQty askQty"><!
[CDATA[
    select bid.cusip, bid.seq, bid.srcId as bidSrcId, bid.bid, ask.seq, ask.srcId as askSrcId,
ask.ask, bid.bidQty, ask.askQty
    from BIDMAX as bid, ASKMIN as ask
    where bid.cusip = ask.cusip
]]></view>
```

Otherwise, fully qualified stream element names are optional. However, it is a best practice to always qualify stream element references explicitly. Oracle Event Processing often does less work with fully qualified stream element names.

For more information, see Joins.

14.3.2 Views and Schemas

You may define the optional schema of the view using a space delimited list of event attribute names.

```
<view id="MAXBIDMINASK" schema="cusip bidseq"><![CDATA[
    select ...
]]></view>
```

14.4 Joins

A **join** is a query that combines rows from two or more streams, views, or relations. Oracle Event Processing performs a join whenever multiple streams appear in the `FROM` clause of the query. The select list of the query can select any stream elements from any of these streams. If any two of these streams have a stream element name in common, then you must qualify all references to these stream elements throughout the query with stream names to avoid ambiguity.

If you create a join between two or more streams, view, or relations that have some stream element names in common, then you must qualify stream element names with the name of their stream, view, or relation. The following example 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.

```
<query id="q0"><![CDATA[
    select * from OrderStream[range 5] as orders, CustomerStream[range 3] as customers where
        orders.customerID = customers.customerID
]]></query>
```

Otherwise, fully qualified stream element names are optional. However, Oracle recommends that you always qualify stream element references explicitly. Oracle Event Processing often does less work with fully qualified stream element names.

Oracle Event Processing supports the following types of joins:

- Inner Joins
- Outer Joins.

> **Note:**
>
> When joining against a cache, you must observe additional query restrictions as Creating Joins Against the Cache describes.

14.4.1 Inner Joins

By default, Oracle Event Processing performs an inner join (sometimes called a simple join): a join of two or more streams that returns only those stream elements that satisfy the join condition.

The following example 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)`.

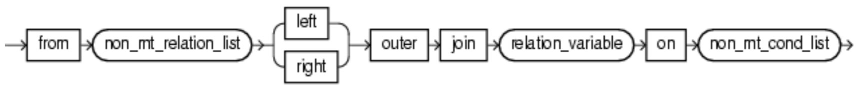
```
<query id="q4"><![CDATA[
    select *
        from
            S0[range 5] as a,
            S1[range 3] as b
        where
            a.c1+a.c2+4.9 = b.c1 + 10
]]></query>
```

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*::=**



The following example 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)`.

```
<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:

- Left Outer Join
- Right Outer Join
- 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. For all stream elements in A that have no matching stream elements in B, Oracle Event Processing returns null for any select list expressions containing stream elements of B.

```
<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. For all stream elements in B that have no matching stream elements in A, Oracle Event Processing returns null for any select list expressions containing stream elements of A.

```
<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.

```
<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 Event Processing Server Cache

You can access an Oracle Event Processing cache from an Oracle CQL statement or user-defined function.

This section describes:

- Creating Joins Against the Cache

14.5.1 Creating Joins Against the Cache

When writing Oracle CQL queries that join against a cache, you must observe the following restrictions:

- Cache Key First and Simple Equality
- No Arithmetic Operations on Cache Keys
- No Full Scans
- Multiple Conditions and Inequality.

For more information, see Joins.

14.5.1.1 Cache Key First and Simple Equality

The complex predicate's first subclause (from the left) with a comparison operation over a cache key attribute may only be a simple equality predicate.

The following predicate is invalid because the predicate is not the first sub-clause (from the left) which refers to cache attributes:

```
... S.c1 = 5 AND CACHE.C2 = S.C2 AND CACHE.C1 = S.C1 ...
```

However, the following predicate is valid:

```
... S.c1 = 5 AND CACHE.C1 = S.C1 AND CACHE.C2 = S.C2 ...
```

14.5.1.2 No Arithmetic Operations on Cache Keys

The subclause may not have any arithmetic operations on a cache key attribute.

The following predicate is invalid because arithmetic operations are not allowed on cache key attributes:

```
... CACHE.C1 + 5 = S.C1 AND CACHE.C2 = S.C2 ...
```

14.5.1.3 No Full Scans

The complex predicate must not require a full scan of the cache.

Assume that your cache has cache key `C1`.

The following predicates are invalid. Because they do not use the cache key attribute in comparisons, they must scan through the whole cache which is not allowed.

```
... CACHE.C2 = S.C1 ...
```

```
... CACHE.C2 > S.C1 ...
```

```
... S.C1 = S.C2 ...
```

```
... S.C1 = CACHE.C2 AND S.C2 = CACHE.C2 ...
```

The following predicates are also invalid. Although they do use the cache key attribute in comparisons, they use inequality operations that must scan through the whole cache which is not allowed.

```
... CACHE.C1 != S.C1 ...
... CACHE.C1 > 5 ...
```

```
... CACHE.C1 + 5 = S.C1 ...
```

The following predicate is also invalid. Although they do use the cache key attribute in comparisons, the first subclause referring to the cache attributes does not refer to the cache key attribute (in this example, `C1`). That is, the first subclause refers to `C2` which is not a cache key and the cache key comparison subclause (`CACHE.C1 = S.C1`) appears after the non-key comparison subclause.

```
 ... CACHE.C2 = S.C2 AND CACHE.C1 = S.C1 ...
```

14.5.1.4 Multiple Conditions and Inequality

To support multiple conditions, inequality, or both, you must make the first sub-clause an equality predicate comparing a cache key value and specify the rest of the predicate subclauses separated by one `AND` operator.

The following are valid predicates:

```
... S.c1 = 5 AND CACHE.C1 = S.C1 AND CACHE.C2 > S.C2 ...
```

```
... CACHE.C1 = S.C1 AND CACHE.C2 = S.C2 ...
```

```
... S.c1 = 5 AND CACHE.C1 = S.C1 AND CACHE.C2 != S.C2 ...
```

14.6 Oracle CQL Queries and Relational Database Tables

You can access a relational database table from an Oracle CQL query using:

- table source: using a table source, you may join a stream only with a `NOW` window and only to a single database table.

  Note:

  Because changes in the table source are not coordinated in time with stream data, you may only join the table source to an event stream using a `Now` window and you may only join to a single database table. For more information, see S[now].

  To integrate arbitrarily complex SQL queries and multiple tables with your Oracle CQL queries, consider using the Oracle JDBC data cartridge instead.

- Oracle JDBC data cartridge: using the Oracle JDBC data cartridge, you may integrate arbitrarily complex SQL queries and multiple tables and datasources with your Oracle CQL queries.

  Note:

  XMLTYPE is not supported for table sources.

In all cases, you must define datasources in the Oracle Event Processing server `config.xml` file.

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

14.7 Oracle CQL Queries and Oracle Data Cartridges

You can access Oracle CQL data cartridge types in Oracle CQL queries just as you would Oracle CQL native types.

15
Pattern Recognition With MATCH_RECOGNIZE

A reference and usage information about the `MATCH_RECOGNIZE` clause in Oracle Continuous Query Language (Oracle CQL) is provided. This clause and its sub-clauses perform pattern recognition in Oracle CQL queries.

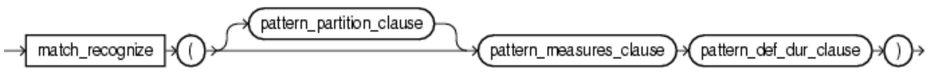
Pattern matching with multiple streams in `FROM` clause is also supported.

15.1 Understanding Pattern Recognition With `MATCH_RECOGNIZE`

The `MATCH_RECOGNIZE` clause performs pattern recognition in an Oracle CQL query. This query will export (make available for inclusion in the `SELECT`) the `MEASURES` clause values for events (tuples) that satisfy the `PATTERN` clause regular expression over the `DEFINE` clause conditions.

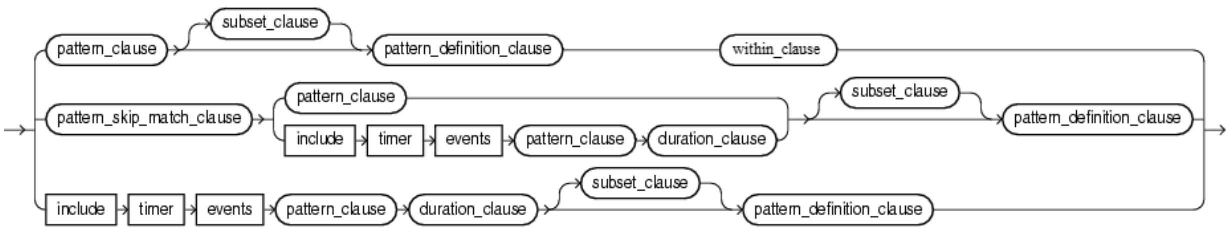
```
<query id="detectPerish"><![CDATA[
  select its.badItemId
  from tkrfid_ItemTempStream
  MATCH_RECOGNIZE (
      PARTITION BY itemId
      MEASURES A.itemId as badItemId
      PATTERN (A B* C)
      DEFINE
          A  AS  (A.temp >= 25),
          B  AS  ((B.temp >= 25) and (to_timestamp(B.element_time)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

***pattern_recognition_clause*::=**



(Figure 15-6, Figure 15-1, *pattern_def_dur_clause::=*)

***pattern_def_dur_clause*::=**



(Figure 15-2, Figure 15-7, Figure 15-3, Figure 15-9, Figure 15-11)

Using `MATCH_RECOGNIZE`, you define conditions on the attributes of incoming events and identify these conditions by using *`identifiers`* called correlation variables. The previous example defines correlation variables `A`, `B`, and `C`. A sequence of consecutive events in the input stream satisfying these conditions constitutes a pattern.

The output of a `MATCH_RECOGNIZE` query is always a stream.

The principle `MATCH_RECOGNIZE` sub-clauses are:

- `MEASURES`: exports (makes available for inclusion in the `SELECT`) attribute values of events that successfully match the pattern you specify.

  See `MEASURES` Clause.
- `PATTERN`: specifies the pattern to be matched as a regular expression over one ore more correlation variables.

  See PATTERN Clause.
- `DEFINE`: specifies the condition for one or more correlation variables.

  See `DEFINE` Clause.

To refine pattern recognition, you may use the optional `MATCH_RECOGNIZE` sub-clauses, including:

- PARTITION BY Clause
- ALL MATCHES Clause
- WITHIN Clause
- DURATION Clause
- INCLUDE TIMER EVENTS Clause
- SUBSET Clause.

For more information, see:

- MATCH_RECOGNIZE and the WHERE Clause
- Referencing Singleton and Group Matches
- Referencing Aggregates.
  - Using count With *, *identifier*.*, and *identifier*.*attr*
  - Using `first` and `last`.
- Using `prev`
- MATCH_RECOGNIZE Examples.

15.1.1 MATCH_RECOGNIZE and the WHERE Clause

In Oracle CQL (as in SQL), the `FROM` clause is evaluated before the `WHERE` clause.

Consider the following Oracle CQL query:

```
SELECT ... FROM S MATCH_RECOGNIZE ( .... ) as T WHERE ...
```

In this query, the `S MATCH_RECOGNIZE ( .... ) as T` is like a subquery in the `FROM` clause and is evaluated first, before the `WHERE` clause.

Consequently, you rarely use both a `MATCH_RECOGNIZE` clause and a `WHERE` clause in the same Oracle CQL query. Instead, you typically use a view to apply the required `WHERE` clause to a stream and then select from the view in a query that applies the `MATCH_RECOGNIZE` clause.

The following example shows two views, `e1p1` and `e2p2`, each applying a `WHERE` clause to stream `S` to pre-filter the stream for the required events. The query `q` then selects from these two views and applies the `MATCH_RECOGNIZE` on this filtered stream of events.

```
<view id="e1p1">
    SELECT * FROM S WHERE eventName = 'E1' and path = 'P1' and statName = 'countValue'
</view>
<view id="e2p2">
    SELECT * FROM S WHERE eventName = 'E2' and path = 'P2' and statName = 'countValue'
</view>

<query id="q">
    SELECT
        T.e1p1Stat as e1p1Stat, T.e2p2Stat as e2p2Stat
    FROM
        e1p1, e2p2
    MATCH_RECOGNIZE(
        ALL MATCHES
        PATTERN(A+)
        DURATION 60 MINUTES
        DEFINE
            A as (A.e1p1Stat < 1000 and A.e2p2Stat > 2000 and count(A) > 3)
        ) as T
</query>
```

15.1.2 Referencing Singleton and Group Matches

The `MATCH_RECOGNIZE` clause identifies the following types of matches:

- singleton: a correlation variable is a singleton if it occurs exactly once in a pattern, is not defined by a `SUBSET`, is not in the scope of an alternation, and is not quantified by a pattern quantifier.

  References to such a correlation variable refer to this single event.
- group: a correlation variable is a group if it occurs in more than one pattern, is defined by a `SUBSET`, is in the scope of an alternation, or is quantified by a pattern quantifier.

  References to such a correlation variable refer to this group of events.

When you reference singleton and group correlation variables in the `MEASURES` and `DEFINE` clauses, observe the following rules:

- For singleton correlation variables, you may reference individual event attributes only, not aggregates.
- For group correlation variables:
  - If you reference an individual event attribute, then the value of the last event to match the correlation variable is returned.

    If the correlation variable is not yet matched, `NULL` is returned. In the case of `count(A.*)`, if the correlation variable `A` is not yet matched, 0 is returned.

    If the correlation variable is being referenced in a definition of the same variable (such as `DEFINE A as A.balance > 1000`), then the value of the current event is returned.

- If you reference an aggregate, then the aggregate is performed over all events that have matched the correlation variable so far.

For more information, see:

- Using count With *, *identifier*.*, and *identifier.attr*
- Pattern Quantifiers and Regular Expressions
- Referencing Attributes in the DEFINE Clause.

15.1.3 Referencing Aggregates

You can use any built-in, Colt, or user-defined aggregate function in the `MEASURES` and `DEFINE` clause of a `MATCH_RECOGNIZE` query.

When using aggregate functions, consider the following:

- Running Aggregates and Final Aggregates
- Operating on the Same Correlation Variable
- Referencing Variables That Have not Been Matched Yet
- Referencing Attributes not Qualified by Correlation Variable.

For more information, see:

- Using count With *, *identifier*.*, and *identifier.attr*
- Using `first` and `last`
- Introduction to Oracle CQL Built-In Aggregate Functions
- Introduction to Oracle CQL Built-In Aggregate Colt Functions
- User-Defined Aggregate Functions
- `MEASURES` Clause
- `DEFINE` Clause.

15.1.3.1 Running Aggregates and Final Aggregates

In the `DEFINE` clause, any aggregate function on a correlation variable `X` is a running aggregate: that is, the aggregate includes all preceding matches of `X` up to and including the current match. If the correlation variable `X` has been completely matched so far, then the aggregate is final, otherwise it is running.

In the `MEASURES` clause, because it is evaluated after the match has been found, all aggregates are final because they are computed over the final match.

When using a `SUBSET` clause, be aware of the fact that you may inadvertently imply a running aggregate.

```
...
PATTERN (X+ Y+)
SUBSET Z = (X, Y)
DEFINE
    X AS X.price > 100,
    Y AS sum(Z.price) < 1000
...
```

Because correlation variable `Z` involves `Y`, the definition of `Y` involves a running aggregate on `Y`.

For more information, see:

- `MEASURES` Clause
- `DEFINE` Clause
- SUBSET Clause.

15.1.3.2 Operating on the Same Correlation Variable

In both the `MEASURES` and `DEFINE` clause, you may only apply an aggregate function to attributes of the same correlation variable.

For example: the use of aggregate function `correlation`.

```
...
MEASURES xycorr AS correlation(X.price, Y.price)
PATTERN (X+ Y+)
DEFINE
    X AS X.price <= 10,
    Y AS Y.price > 10
...
```

The `correlation` aggregate function may not operate on more than one correlation variable.

15.1.3.3 Referencing Variables That Have not Been Matched Yet

In the `DEFINE` clause, you may reference a correlation variable that has not been matched yet. However, you should use caution when doing so.

```
PATTERN (X+ Y+)
DEFINE
    X AS count(Y.*) >= 3
    Y AS Y.price > 10,
```

Although this syntax is legal, note that in this particular example, the pattern will never match because at the time `X` is matched, `Y` has not yet been matched, and `count(Y.*)` is 0.

To implement the desired behavior ("Match when the price of `Y` has been greater than 10, 3 or more times in a row"), implement this pattern.

```
PATTERN (Y+ X+)
DEFINE
    Y AS Y.price > 10,
    X AS count(Y.*) >= 3
```

For more information, see Using count With *, *identifier*.*, and *identifier*.*attr*.

15.1.3.4 Referencing Attributes not Qualified by Correlation Variable

In the `DEFINE` clause, if you apply an aggregate function to an event attribute not qualified by correlation variable, the aggregate is a running aggregate.

```
PATTERN ((RISE FALL)+)
DEFINE
```

```
RISE AS count(RISE.*) = 1 or RISE.price > FALL.price,
FALL AS FALL.price < RISE.price and count(*) > 1000
```

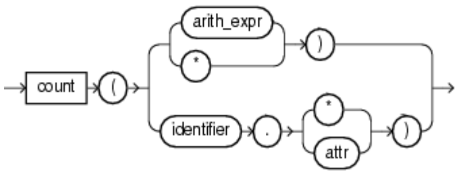
This query detects a pattern in which a price alternately goes up and down, for as long as possible, but for at least more than 1000 matches.

For more information, see:

- Running Aggregates and Final Aggregates
- Using count With *, *identifier*.*, and *identifier*.*attr*.

15.1.3.5 Using count With *, *identifier*.*, and *identifier*.*attr*

The built-in aggregate function `count` has syntax:



(*arith_expr*::=)

The return value of `count` depends on the argument as Table 15-1 shows.

Table 15-1 Return Values for count Aggregate Function

Input Argument	Return Value
`arith_expr`	The number of tuples where *`arith_expr`* is not `NULL`.
`*`	The number of tuples matching all the correlation variables in the pattern, including duplicates and nulls.
`identifier``.*`	The number of all tuples that match the correlation variable *`identifier`*, including duplicates and nulls. Note the following: • `count(A.*) = 1` is true for the first event that matches `A`. • `count(A.*) = 0` is true while `A` has not been matched yet.
`identifier``.`*`attr`*	The number of tuples that match correlation variable *`identifier`*, where *`attr`* is not `NULL`.

Assume that the schema of `S` includes attributes `account` and `balance`. This query returns an event for each `account` that has not received 3 or more events in 60 minutes.

```
select
    T.account,
    T.Atime
FROM S
    MATCH_RECOGNIZE(
        PARTITION BY account
        MEASURES
            A.account has account
            A.ELEMENT_TIME as Atime
        ALL MATCHES
        INCLUDE TIMER EVENTS
        PATTERN (A+)
        DURATION 60 MINUTES
        DEFINE
            A AS count(A.*) < 3
    ) as T
```

The `PATTERN (A+)` specifies the pattern "Match `A` one or more times".

The `DEFINE` clause specifies the condition:

```
A AS count(A.*) < 3
```

This condition for `A` places no restrictions on input tuples (such as `A.balance > 1000`). The only restrictions are imposed by the `PARTITION BY account` and `DURATION 60 MINUTES` clauses. In the `DEFINE` clause, the `A.*` means, "Match all input tuples for the group `A+`". This group includes the one or more input tuples with a particular `account` received in the 60 minutes starting with the first input tuple. The `count(A.*)` is a running aggregate that returns the total number of events in this group.

If the `DEFINE` clause specifies the condition:

```
A AS A.balance > 1000 and count(A.*) < 3
```

Then `A.*` still means "Match all input tuples for the group `A+`". In this case, this group includes the one or more input tuples with a particular `account` received in the 60 minutes starting with the first input tuple and with `balance > 1000`.

In contrast:

- `count(*)` means "The number of tuples matching all the correlation variables in the pattern, including duplicates and nulls."
- `count(A.balance)` means "The number of all tuples that match the correlation variable `A` where the `balance` is not `NULL`".

For more information, see:

- count
- Range, Rows, and Slide at Query Start-Up and for Empty Relations
- Referencing Singleton and Group Matches
- Referencing Aggregates
- Referencing Variables That Have not Been Matched Yet
- Referencing Attributes not Qualified by Correlation Variable.

15.1.3.6 Using `first` and `last`

Use the `first` and `last` built-in aggregate functions to access event attributes of the first or last event match, respectively:

`first` returns the value of the first match of a group in the order defined by the `ORDER BY` clause or the default order.

`last` returns the value of the last match of a group in the order defined by the `ORDER BY` clause or the default order.

The `first` and `last` functions accept an optional non-negative, constant integer argument (`N`) that indicates the offset following the first and the offset preceding the last match of the variable, respectively. If you specify this offset, the `first` function returns the `N`-th matching event following the first match and the `last` function returns the `N`-th matching event preceding the last match. If the offset does not fall within the match of the variable, the `first` and `last` functions return `NULL`.

For more information, see:

- first
- last
- Referencing Aggregates.

15.1.4 Using `prev`

Use the `prev` built-in single-row function to access event attributes of a previous event match. If there is no previous event match, the `prev` function returns `NULL`.

The `prev` function accepts an optional non-negative, constant integer argument (`N`) that indicates the offset to a previous match. If you specify this offset, the `prev` function returns the `N`-th matching event preceding the current match. If there is no such previous match, the `prev` functions returns `NULL`.

When you use the `prev` function in the `DEFINE` clause, this function may only access the currently defined correlation variable.

For example: the correlation variable definition:

```
Y AS Y.price < prev(Y.price, 2)
```

However, the correlation variable definition is invalid because while defining correlation variable `Y`, it references correlation variable `X` inside the `prev` function.

```
Y AS Y.price < prev(X.price, 2)
```

For more information, see:

- prev

  `DEFINE` Clause.

15.2 `MEASURES` Clause

The `MEASURES` clause exports (makes available for inclusion in the `SELECT`) attribute values of events that successfully match the pattern you specify.

You may specify expressions over correlation variables that reference partition attributes, order by attributes, singleton variables and aggregates on group variables, and aggregates on the attributes of the stream that is the source of the `MATCH_RECOGNIZE` clause.

You qualify attribute values by correlation variable to export the value of the attribute for the event that matches the correlation variable's condition. For example, within the `MEASURES` clause, `A.c1` refers to the value of event attribute `c1`:

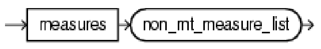
- In the tuple that last matched the condition corresponding to correlation variable `A`, if `A` is specified in the `DEFINE` clause.

- In the last processed tuple, if `A` is not specified in the `DEFINE` clause.

  This is because if `A` is not specified in the `DEFINE` clause, then `A` is considered as `TRUE` always. So effectively all the tuples in the input match to `A`.

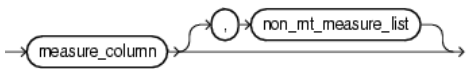
You may include in the `SELECT` statement only attributes you specify in the `MEASURES` clause.

Figure 15-1 *pattern_measures_clause*::=



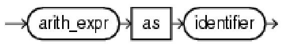
(*non_mt_measure_list*::=)

***non_mt_measure_list*::=**



(*measure_column*::=)

***measure_column*::=**



(*arith_expr*::=)

The *`pattern_measures_clause`* is:

```
MEASURES
    A.itemId as itemId
```

This section describes:

- Functions Over Correlation Variables in the MEASURES Clause.

For more information, see:

- Referencing Singleton and Group Matches
- Referencing Aggregates
- `DEFINE` Clause
- Functions.

15.2.1 Functions Over Correlation Variables in the MEASURES Clause

In the `MEASURES` clause, you may apply any single-row or aggregate function to the attributes of events that match a condition.

The following example applies the `last` function over correlation variable `Z.c1` in the `MEASURES` clause.

```
<query id="tkpattern_q41"><![CDATA[
    select
        T.firstW, T.lastZ
    from
        tkpattern_S11
    MATCH_RECOGNIZE (
        MEASURES A.c1 as firstW, last(Z.c1) as lastZ
        ALL MATCHES
        PATTERN(A W+ X+ Y+ Z+)
        DEFINE
            W as W.c2 < prev(W.c2),
            X as X.c2 > prev(X.c2),
            Y as Y.c2 < prev(Y.c2),
            Z as Z.c2 > prev(Z.c2)
    ) as T
]]></query>
```

Note the following in the `MEASURES` clause:

- `A.c1` will export the value of `c1` in the first and only the first event that the query processes because:
  - `A` is not specified in the `DEFINE` clause, therefor it is always true.
  - `A` has no pattern quantifiers, therefor it will match exactly once.
- The built-in aggregate function `last` will export the value of `c1` in the last event that matched `Z` at the time the `PATTERN` clause was satisfied.

For more information, see:

- Referencing Aggregates
- Using count With *, *identifier*.*, and *identifier*.*attr*
- Using `first` and `last`
- Using `prev`.

15.3 PATTERN Clause

The `PATTERN` clause specifies the pattern to be matched as a regular expression over one ore more correlation variables.

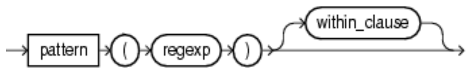
Incoming events must match these conditions in the order given (from left to right).

The regular expression may contain correlation variables that are:

- Defined in the `DEFINE` clause: such correlation variables are considered true only if their condition definition evaluates to `TRUE`.

  See `DEFINE` Clause.
- Not defined in the `DEFINE` clause: such correlation variables are considered as always `TRUE`; that is, they match every input.

Figure 15-2 *pattern_clause*::=



(*regexp*::=, Figure 15-8)

This section describes:

- Pattern Quantifiers and Regular Expressions
- Grouping and Alternation in the PATTERN Clause.

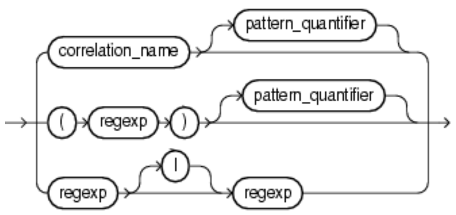
For more information, see:

- Pattern Detection
- Pattern Detection With PARTITION BY
- Pattern Detection With Aggregates.

15.3.1 Pattern Quantifiers and Regular Expressions

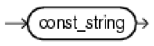
You express the pattern as a regular expression composed of correlation variables and pattern quantifiers.

***regexp*::=**



(*correlation_name*::=, *pattern_quantifier*::=)

***correlation_name*::=**



***pattern_quantifier*::=**

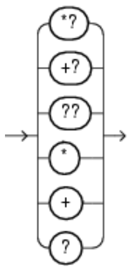


Table 15-2 lists the pattern quantifiers (*pattern_quantifier*::=) Oracle CQL supports.

Table 15-2 MATCH_RECOGNIZE Pattern Quantifiers

Maximal	Minimal	Description
*	*?	0 or more times
+	+?	1 or more times.
?	??	0 or 1 time.
None	None	An unquantified pattern, such as `A`, is assumed to have a quantifier that requires exactly 1 match.

Use the pattern quantifiers to specify the pattern as a regular expression, such as `A*` or `A+?`.

The one-character pattern quantifiers are maximal or "greedy"; they will attempt to match as many instances of the regular expression on which they are applied as possible.

The two-character pattern quantifiers are minimal or "reluctant"; they will attempt to match as few instances of the regular expression on which they are applied as possible.

Consider the following *`pattern_clause`*:

```
PATTERN (A B* C)
```

This pattern clause means a pattern match will be recognized when the following conditions are met by consecutive incoming input tuples:

1. Exactly one tuple matches the condition that defines correlation variable `A`, followed by
2. Zero or more tuples that match the correlation variable `B`, followed by
3. Exactly one tuple that matches correlation variable `C`.

While in state 2, if a tuple arrives that matches both the correlation variables `B` and `C` (since it satisfies the defining conditions of both of them) then as the quantifier `*` for `B` is greedy that tuple will be considered to have matched `B` instead of `C`. Thus due to the greedy property `B` gets preference over `C` and we match a greater number of `B`. Had the pattern expression be `A B*? C`, one that uses a lazy or reluctant quantifier over `B`, then a tuple matching both `B` and `C` will be treated as matching `C` only. Thus `C` would get preference over `B` and we will match fewer `B`.

For more information, see:

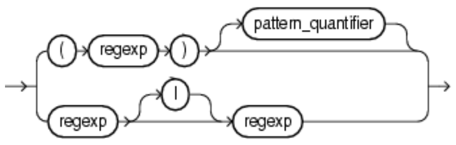
- Referencing Singleton and Group Matches
- Grouping and Alternation in the PATTERN Clause.

15.3.2 Grouping and Alternation in the PATTERN Clause

As shown in the *`regexp_grp_alt`* syntax, you can use:

- open and close round brackets (`(` and `)`) to group correlation variables
- alternation operators (`|`) to match either one correlation variable (or group of correlation variables) or another

***regexp_grp_alt*::=**



(*correlation_name*::=, *pattern_quantifier*::=, *regexp*::=)

Consider the following *`pattern_clause`*:

```
PATTERN (A+ B+)
```

This means "A one or more times followed by B one or more times".

You can group correlation variables. For example:

```
PATTERN (A+ (C+ B+)*)
```

This means "A one or more times followed by zero or more occurrences of C one or more times and B one or more times".

Using the `PATTERN` clause alternation operator (`|`), you can refine the sense of the *`pattern_clause`*. For example:

```
PATTERN (A+ | B+)
```

This means "A one or more times or B one or more times, whichever comes first".

Similarly, you can both group correlation variables and use the alternation operator. For example:

```
PATTERN (A+ (C+ | B+))
```

This means "A one or more times followed by either C one or more times or B one or more times, whichever comes first".

To match every permutation you can use:

```
PATTERN ((A B) | (B A))
```

This means "A followed by B or B followed by A, which ever comes first".

For more information, see:

- Pattern Quantifiers and Regular Expressions
- Alternation Operator.

15.4 `DEFINE` Clause

The `DEFINE` clause specifies the boolean condition for each correlation variable.

You may specify any logical or arithmetic expression and apply any single-row or aggregate function to the attributes of events that match a condition.

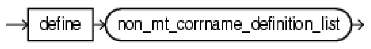
On receiving a new tuple from the base stream, the conditions of the correlation variables that are relevant at that point in time are evaluated. A tuple is said to have matched a correlation variable if it satisfies its defining condition. A particular input can match zero, one, or more correlation variables. The relevant conditions to be evaluated on receiving an input are determined by logic governed by the `PATTERN` clause regular expression and the state in pattern recognition process that we have reached after processing the earlier inputs.

The condition can refer to any of the attributes of the schema of the stream or view that evaluates to a stream on which the `MATCH_RECOGNIZE` clause is being applied.

A correlation variable in the `PATTERN` clause need not be specified in the `DEFINE` clause: the default for such a correlation variable is a predicate that is always true. Such a correlation variable matches every event. It is an error to specify a correlation variable in the `DEFINE` clause which is not used in a `PATTERN` clause

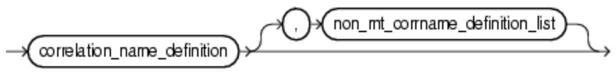
No correlation variable defined by a `SUBSET` clause may be defined in the `DEFINE` clause.

Figure 15-3 *pattern_definition_clause*::=



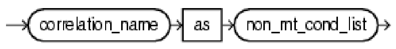
(Figure 15-4)

Figure 15-4 *non_mt_corrname_definition_list*::=



(Figure 15-5)

Figure 15-5 *correlation_name_definition*::=



(*correlation_name*::=, *non_mt_cond_list*)

This section describes:

- Functions Over Correlation Variables in the DEFINE Clause
- Referencing Attributes in the DEFINE Clause
- Referencing One Correlation Variable From Another in the DEFINE Clause.

For more information, see:

- Referencing Singleton and Group Matches
- Referencing Aggregates
- Using `first` and `last`
- Using `prev`
- PATTERN Clause

- SUBSET Clause
- Functions.

15.4.1 Functions Over Correlation Variables in the DEFINE Clause

You can use functions over the correlation variables while defining them.

The following example applies the `to_timestamp` function to correlation variables.

```
...
      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)
...
```

The following example applies the `count` function to correlation variable `B` to count the number of times its definition was satisfied. A match is recognized when `totalCountValue` is less than 1000 two or more times in 30 minutes.

```
...
    MATCH_RECOGNIZE(
        ...
        PATTERN(B*)
        DURATION 30 MINUTES
        DEFINE
            B as (B.totalCountValue < 1000 and count(B.*) >= 2)
...
```

For more information, see:

- Referencing Aggregates
- Using count With *, *identifier*.*, and *identifier*.*attr*
- Using `first` and `last`
- Using `prev`.

15.4.2 Referencing Attributes in the DEFINE Clause

You can refer to the attributes of a base stream:

- Without a correlation variable: `c1 < 20`.
- With a correlation variable: `A.c1 < 20`.

When you refer to the attributes without a correlation variable, a tuple that last matched any of the correlation variables is consulted for evaluation.

Consider the following definitions:

- `DEFINE A as c1 < 20`
- `DEFINE A as A.c1 < 20`

Both refer to `c1` in the same tuple which is the latest input tuple. This is because on receiving an input we evaluate the condition of a correlation variable assuming that the latest input matches that correlation variable.

If you specify a correlation name that is not defined in the DEFINE clause, it is considered to be true for every input.

The correlation variable A appears in the PATTERN clause but is not specified in the DEFINE clause. This means the correlation name A is true for every input. It is an error to define a correlation name which is not used in a PATTERN clause.

```
<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:

- Referencing One Correlation Variable From Another in the DEFINE Clause
- Referencing Singleton and Group Matches
- Referencing Variables That Have not Been Matched Yet
- Referencing Attributes not Qualified by Correlation Variable
- PATTERN Clause.

15.4.3 Referencing One Correlation Variable From Another in the DEFINE Clause

A definition of one correlation variable can refer to another correlation variable. Consider the query:

```
...
Select
    a_firsttime, d_lasttime, b_avgprice, d_avgprice
FROM
    S
MATCH_RECOGNIZE (
    PARTITION BY symbol
    MEASURES
        first(a.time) as a_firsttime,
        last(d.time) as d_lasttime,
        avg(b.price) as b_avgprice,
        avg(d.price) as d_avgprice
    PATTERN (A B+ C+ D)
    DEFINE
        A as A.price > 100,
        B as B.price > A.price,
        C as C.price < avg(B.price),
        D as D.price > prev(D.price)
)
...
```

Note the following:

- Because correlation variable `A` defines a single attribute, `B` can refer to this single attribute.
- Because `B` defines more than one attribute, `C` cannot reference a single attribute of `B`. In this case, `C` may only reference an aggregate of `B`.
- `D` is defined in terms of itself: in this case, you may refer to a single attribute or an aggregate. In this example, the `prev` function is used to access the match of `D` prior to the current match.

For more information, see:

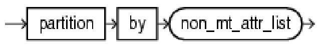
- Referencing Attributes in the DEFINE Clause
- Referencing Singleton and Group Matches
- Referencing Variables That Have not Been Matched Yet
- Referencing Attributes not Qualified by Correlation Variable
- Referencing Attributes in the DEFINE Clause.

15.5 PARTITION BY Clause

Use this optional clause to specify the stream attributes by which a `MATCH_RECOGNIZE` clause should partition its results.

Without a `PARTITION BY` clause, all stream attributes belong to the same partition.

Figure 15-6 ***pattern_partition_clause*::=**



(*non_mt_attr_list*)

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.

```
     c1  c2
1000 10, 1
2000 10, 2
3000 20, 2
4000 20, 1
```

Consider the `MATCH_RECOGNIZE` query.

```
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, 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.

15.6 ALL MATCHES Clause

Use this optional clause to configure Oracle Event Processing to match overlapping patterns.

With the `ALL MATCHES` clause, Oracle Event Processing finds all possible matches. Matches may overlap and may start at the same event. In this case, there is no distinction between greedy and reluctant pattern quantifiers. For example, the following pattern:

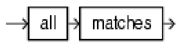
```
ALL MATCHES
PATTERN (A* B)
```

produces the same result as:

```
ALL MATCHES
PATTERN (A*? B)
```

Without the `ALL MATCHES` clause, overlapping matches are not returned, and quantifiers such as the asterisk determine which among a set of candidate (and overlapping) matches is the preferred one for output. The rest of the overlapping matches are discarded.

Figure 15-7 ***pattern_skip_match_clause*::=**



Consider the query `tkpattern_q41` that uses `ALL MATCHES` and the data stream `tkpattern_S11`. Stream `tkpattern_S11` has schema `(c1 integer, c2 integer)`. The query returns the stream.

The query `tkpattern_q41` will report a match when the input stream values, when plotted, form the shape of the English letter **W**. The relation 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.

```
<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>
```

```
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

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
```

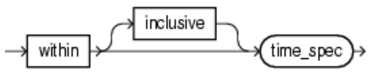
The `ALL MATCHES` clause reports all the matched pattern instances on receiving a particular input. For example, at time 20000, all of the tuples `{12,20}`, `{13,20}`, and `{14,20}` are output.

For more information, see Pattern Quantifiers and Regular Expressions.

15.7 WITHIN Clause

The `WITHIN` clause is an optional clause that outputs a *`pattern_clause`* match if and only if the match occurs within the specified time duration.

Figure 15-8 *within_clause*::=



(*time_spec*)

That is, if and only if:

```
TL - TF < WD
```

Where:

- `TL` - Timestamp of last event matching the pattern.
- `TF` - Timestamp of first event matching the pattern.
- `WD` - Duration specified in the `WITHIN` clause.

The `WITHIN INCLUSIVE` clause tries to match events at the boundary case as well. That is, it outputs a match if and only if:

```
TL - TF <= WD
```

If the match completes within the specified time duration, then the event is output as soon as it happens. That is, if the match can be output, it is output with the timestamp at which it completes. The `WITHIN` clause does not wait for the time duration to expire as the `DURATION` clause does.

When the `WITHIN` clause duration expires, it discards any potential candidate matches which are incomplete.

For more information, see Pattern Detection With the WITHIN Clause.

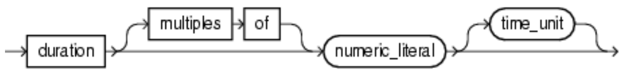
Note:

You cannot use a `WITHIN` clause with a `DURATION` clause. For more information, see DURATION Clause.

15.8 DURATION Clause

The `DURATION` clause is an optional clause that you should use only when writing a query involving non-event detection. Non-event detection is the detection of a situation when a certain event which should have occurred in a particular time limit does not occur in that time frame.

Figure 15-9 *duration_clause*::=



(Figure 7-10)

Using this clause, a match is reported only when the regular expression in the PATTERN clause is matched completely and no other event or input arrives until the duration specified in the DURATION clause expires. The duration is measured from the time of arrival of the first event in the pattern match.

You must use the INCLUDE TIMER EVENTS clause when using the DURATION clause. For more information, see INCLUDE TIMER EVENTS Clause.

This section describes:

- Fixed Duration Non-Event Detection
- Recurring Non-Event Detection.

Note:

You cannot use a DURATION clause with a WITHIN clause. For more information, see WITHIN Clause.

15.8.1 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 Figure 7-11); the default time unit is seconds.

Consider the query tkpattern_q59 that uses DURATION 10 to specify a delay of 10 s (10000 ms) and the data stream tkpattern_S19. Stream tkpattern_S19 has schema (c1 integer). The query returns the stream.

```
<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>
```

```
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

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.

15.8.2 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` that uses `DURATION MULTIPLES OF 10` to specify a delay of 10 s (10000 ms) and the data stream `tkpattern_S23`. Stream `tkpattern_S23` has schema `(c1 integer)`. The query returns the stream.

```
<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>

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

Timestamp   Tuple Kind  Tuple
 11000:     +           10,88,637
 25000:     +           10,,
 38000:     +           10,18,18
 48000:     +           10,19,37
 58000:     +           10,19,37
 68000:     +           10,20,57
 82000:     +           10,,
 92000:     +           10,,
102000:     +           10,,
112000:     +           10,,
```

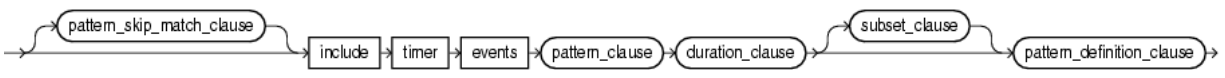
The execution here follows similar logic to that of the example above for just the `DURATION` clause (see Fixed Duration Non-Event Detection). The difference comes for the later outputs. The tuple at 72000 matches `A` and then there is nothing else after that. So the pattern `AB*` is matched and we get output at 82000. Since we have the `MULTIPLES OF` clause and duration 10 we see outputs at time 92000, 102000, and so on.

15.9 INCLUDE TIMER EVENTS Clause

Use this clause in conjunction with the `DURATION` clause for non-event detection queries.

Typically, in most pattern match queries, a pattern match output is always triggered by an input event on the input stream over which pattern is being matched. The only exception is non-event detection queries where there could be an output triggered by a timer expiry event (as opposed to an explicit input event on the input stream).

Figure 15-10 ***pattern_inc_timer_evs_clause*::=**



(Figure 15-2, Figure 15-7, Figure 15-3, Figure 15-9, Figure 15-11)

For more information, see DURATION Clause.

15.10 SUBSET Clause

Using this clause, you can group together one or more correlation variables that are defined in the `DEFINE` clause. You can use this named subset in the `MEASURES` and `DEFINE` clauses just like any other correlation variable.

For example:

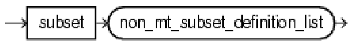
```
SUBSET S1 = (Z,X)
```

The right-hand side of the subset (`(Z,X)`) is a comma-separated list of one or more correlation variables as defined in the `PATTERN` clause.

The left-hand side of the subset (`S1`) is the union of the correlation variables on the right-hand side.

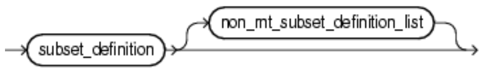
You cannot include a subset variable in the right-hand side of a subset.

Figure 15-11 *subset_clause*::=



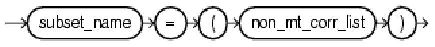
(Figure 15-12)

Figure 15-12 *non_mt_subset_definition_list*::=



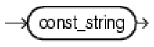
(Figure 15-13)

Figure 15-13 *subset_definition*::=



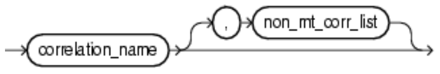
(Figure 15-14, Figure 15-15)

Figure 15-14 *subset_name*::=



(Figure 7-3)

Figure 15-15 *non_mt_corr_list*::=



(*correlation_name*::=)

Consider the query `q55` inExample 15-1 and the data stream `S11` in Example 15-2. Stream `S11` has schema `(c1 integer, c2 integer)`. This example defines subsets `S1` through `S6`. This query outputs a match if the `c2` attribute values in the input stream form the shape of the English letter **W**. Now suppose we want to know the sum of the values of `c2` for those tuples which form the incrementing arms of this **W** shape. The correlation variable `X` represents tuples that are part of the first incrementing arm and `Z` represent the tuples that are part of the second incrementing arm. So we need some way to group the tuples that match both. Such a requirement can be captured by defining a `SUBSET` clause as the example shows.

Subset `S4` is defined as `(X,Z)`. It refers to the tuples in the input stream that match either `X` or `Z`. This subset is used in the `MEASURES` clause statement `sum(S4.c2) as sumIncrArm`. This computes the sum of the value of `c2` attribute in the tuples that match either `X` or `Z`. A reference to `S4.c2` in a `DEFINE` clause like `S4.c2 = 10` will refer to the value of `c2` in the latest among the last tuple that matched `X` and the last tuple that matched `Z`.

Subset `S6` is defined as `(Y)`. It refers to all the tuples that match correlation variable `Y`.

The query returns the stream.

Example 15-1   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 15-2   SUBSET Clause Example

```
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

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:

- Running Aggregates and Final Aggregates
- `MEASURES` Clause
- PATTERN Clause
- `DEFINE` Clause.

15.11 MATCH_RECOGNIZE Examples

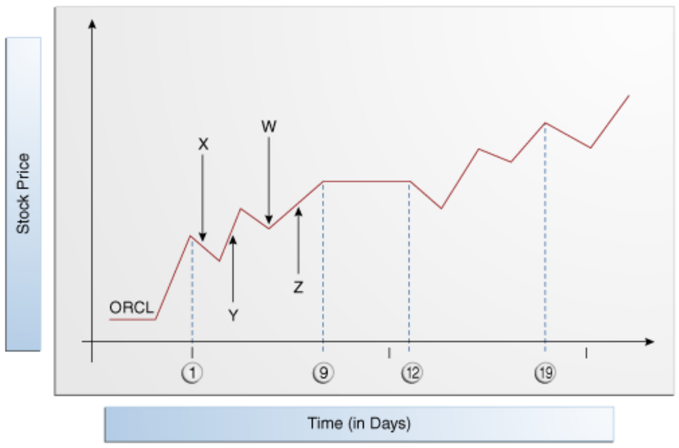
The following examples illustrate basic `MATCH_RECOGNIZE` practices:

- Pattern Detection
- Pattern Detection With PARTITION BY
- Pattern Detection With Aggregates
- Fixed Duration Non-Event Detection.

15.11.1 Pattern Detection

Consider the stock fluctuations that Figure 15-16 shows. This data can be represented as a stream of stock ticks (index number or time) and stock price. Figure 15-16 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 15-16 Pattern Detection: Double Bottom Stock Fluctuations



Example 15-3 shows a query `q` on stream `S2` of stock price events with schema `symbol`, `stockTick`, and `price`. This query detects double bottom patterns on the incoming stock trades using the `PATTERN` clause (`A W+ X+ Y+ Z+`). The correlation names in this clause are:

- `A`: corresponds to the start point of the double bottom pattern.

  Because correlation name `A` is true for every input, it is not defined in the `DEFINE` clause. If you specify a correlation name that is not defined in the `DEFINE` clause, it is considered to be true for every input.
- `W+`: corresponds to the first decreasing arm of the double bottom pattern.

  It is defined by `W.price < prev(W.price)`. This definition implies that the current price is less than the previous one.
- `X+`: corresponds to the first increasing arm of the double bottom pattern.
- `Y+`: corresponds to the second decreasing arm of the double bottom pattern.
- `Z+`: corresponds to the second increasing arm of the double bottom pattern.

Example 15-3 Pattern Detection

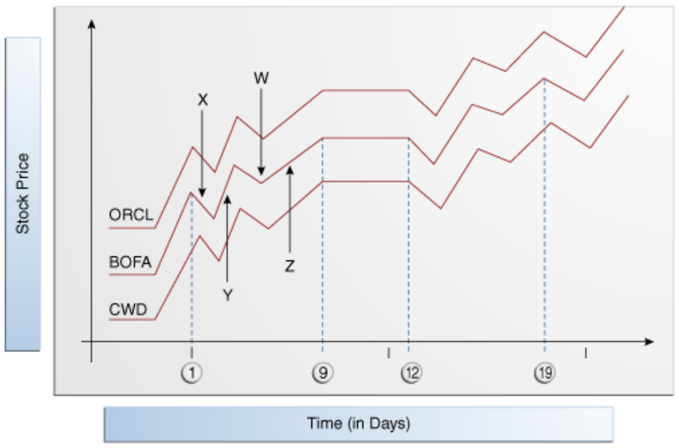
```
<query id="q"><![CDATA[
    SELECT
        T.firstW,
        T.lastZ
    FROM
        S2
    MATCH_RECOGNIZE (
        MEASURES
```

```
                A.stockTick as firstW,
                last(Z) as lastZ
            PATTERN(A W+ X+ Y+ Z+)
            DEFINE
                W as W.price < prev(W.price),
                X as X.price > prev(X.price),
                Y as Y.price < prev(Y.price),
                Z as Z.price > prev(Z.price)
        ) as T
        WHERE
            S2.symbol = "oracle"
]]></query>
```

15.11.2 Pattern Detection With PARTITION BY

Consider the stock fluctuations that Figure 15-17 shows. This data can be represented as a stream of stock ticks (index number or time) and stock price. In this case, the stream contains data for more than one stock ticker symbol. Figure 15-17 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 15-17   Pattern Detection With Partition By: Stock Fluctuations



Example 15-4 shows a query q on stream S2 of stock price events with schema symbol, stockTick, and price. This query detects double bottom patterns on the incoming stock trades using the PATTERN clause (A W+ X+ Y+ Z+). The correlation names in this clause are:

- A: corresponds to the start point of the double bottom pattern.
- W+: corresponds to the first decreasing arm of the double bottom pattern as defined by W.price < prev(W.price), which implies that the current price is less than the previous one.
- X+: corresponds to the first increasing arm of the double bottom pattern.
- Y+: corresponds to the second decreasing arm of the double bottom pattern.
- Z+: corresponds to the second increasing arm of the double bottom pattern.

The query partitions the input stream by stock ticker symbol using the PARTITION BY clause and applies this PATTERN clause to each logical stream.

Example 15-4   Pattern Detection With PARTITION BY

```
<query id="q"><![CDATA[
    SELECT
        T.firstW,
        T.lastZ
    FROM
        S2
    MATCH_RECOGNIZE (
        PARTITION BY
            A.symbol
        MEASURES
            A.stockTick as firstW,
            last(Z) as lastZ
        PATTERN(A W+ X+ Y+ Z+)
        DEFINE
            W as W.price < prev(W.price),
            X as X.price > prev(X.price),
            Y as Y.price < prev(Y.price),
            Z as Z.price > prev(Z.price)
    ) as T
]]></query>
```

15.11.3 Pattern Detection With Aggregates

Consider the query `q1` and the data stream `S`. Stream `S` has schema `(c1 integer)`. The query returns the stream.

```
<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>
```

```
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

Timestamp   Tuple
8000        338
12000       52
```

15.11.4 Pattern Detection With the WITHIN Clause

Consider the queries and the data stream `S`. Stream `S` has schema `(c1 integer, c2 integer)`. Table 15-3 compares the output of these queries.

```
<query id="queryWithin"><![CDATA[
    SELECT T.Ac2, T.Bc2, T.Cc2
    FROM S
    MATCH_RECOGNIZE(
        MEASURES A.c2 as Ac2, B.c2 as Bc2, C.c2 as Cc2
        PATTERN (A (B+ | C)) within 3000 milliseconds
        DEFINE
            A as A.c1=10 or A.c1=25,
            B as B.c1=20 or B.c1=15 or B.c1=25,
            C as C.c1=15
    ) as T
]]></query>

<query id="queryWithinInclusive"><![CDATA[
    SELECT T.Ac2, T.Bc2, T.Cc2
    FROM S
    MATCH_RECOGNIZE(
        MEASURES A.c2 as Ac2, B.c2 as Bc2, C.c2 as Cc2
        PATTERN (A (B+ | C)) within inclusive 3000 milliseconds
        DEFINE
            A as A.c1=10 or A.c1=25,
            B as B.c1=20 or B.c1=15 or B.c1=25,
            C as C.c1=15
    ) as T
]]></query>

Timestamp   Tuple
  1000       10,100
h 2000
  3000       15,200
  3000       20,300
  4000       25,400
  5000       20,500
  6000       20,600
  7000       35,700
  8000       10,800
  9000       15,900
h 11000
 11000       20,1000
 11000       50,1100
```

Table 15-3 WITHIN and WITHIN INCLUSIVE Query Output

Query queryWithin	Query queryWithinInclusive

`Timestamp   Tuple Kind  Tuple`
 	
`3000:       +            100,300,`
`6000:       +            400,600,`
`9000:       +            800,900,`
	`Timestamp   Tuple Kind  Tuple`
	
	` 4000:       +            100,400,`
	`11000:       +            800,1000,`

As Table 15-3 shows for the `queryWithin` query, the candidate match starts with the event at `TimeStamp=1000` and since the `WITHIN` clause duration is 3 seconds, the query will output the match only if it completes before the event at `TimeStamp=4000`. When the query receives the event at `TimeStampe=4000`, the longest match up to that point (since we are not using `ALL MATCHES`) is output. Note that although the event at `TimeStamp=4000` matches `B`, it is not included in the match. The next match starts with the event at `TimeStamp=4000` since that event also matches `A` and the previous match ends at `TimeStamp=3000`.

As Table 15-3 shows for the `queryWithinInclusive` query, the candidate match starts with the event at `TimeStamp=1000`. When the query receives the event at `TimeStamp=4000`, that event is included in the match because the query uses `WITHIN INCLUSIVE` and the event matches `B`. Note that although the event at `TimeStamp=5000` matches `B`, the pattern is not grown further since it exceeds the duration (3 seconds) measured from the start of the match (`TimeStamp=1000`). Since this match ends at `TimeStamp=4000` and we are not using `ALL MATCHES`, the next match does not start at `TimeStamp=4000`, even though it matches `A`.

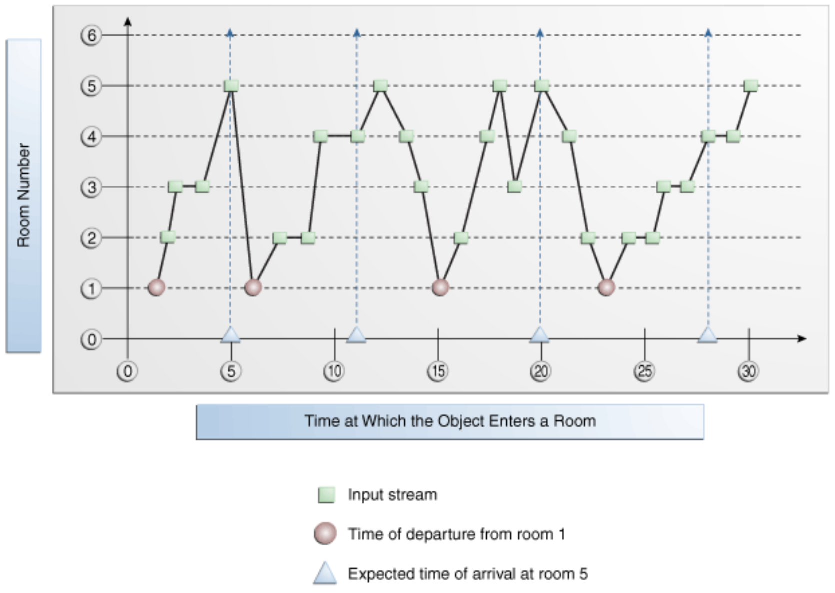
For more information, see:

- WITHIN Clause
- ALL MATCHES Clause.

15.11.5 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 15-18 shows the object's performance. This data can be represented as a stream of time and room number. Note that when the object started from room 1 at time 1, it reached room 5 at time 5, as expected. However, when the object started from room 1 at time 6, it failed to reach room 5 at time 11; it reached room 5 at time 12. When the object started from room 1 at time 15, it was in room 5 at time 20, as expected. However, when the object started from room 1 at time 23, it failed to reach room 5 at time 28; it reached room 5 at time 30. The successes at times 5 and 20 are considered events: the arrival of the object in room 5 at the appropriate time. The failures at time 11 and 28 are considered non-events: the expected arrival event did not occur. Using Oracle CQL, you can query for such non-events.

Figure 15-18 Fixed Duration Non-Event Detection



The following example 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.

```
<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 DURATION Clause.

16 Oracle CQL Statements

Data definition language (DDL) statements in Oracle Continuous Query Language (Oracle CQL) is described.

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 Event Processing that no further input will have a timestamp lesser than the heartbeat value.

For more information, see:

- Lexical Conventions
- Syntactic Shortcuts and Defaults
- Documentation Conventions
- Basic Elements of Oracle CQL
- Common Oracle CQL DDL Clauses
- Oracle CQL Queries, Views, and Joins.

16.2.1 Query

Purpose

Use the query statement to define a Oracle CQL query that you reference by *`identifier`* in subsequent Oracle CQL statements.

Prerequisites

If your query references a stream or view, then the stream or view must already exist.

If the query already exists, Oracle Event Processing server throws an exception.

For more information, see:

- View
- Oracle CQL Queries, Views, and Joins

Syntax

You express a query in a `<query></query>` element.

The `query` element has one attribute:

- `id`: Specify the `identifier` as the `query` element `id` attribute.

```
<query id="q0"><![CDATA[
    select * from OrderStream where orderAmount > 10000.0
]]></query>
```

Figure 16-1 ***query*::=**

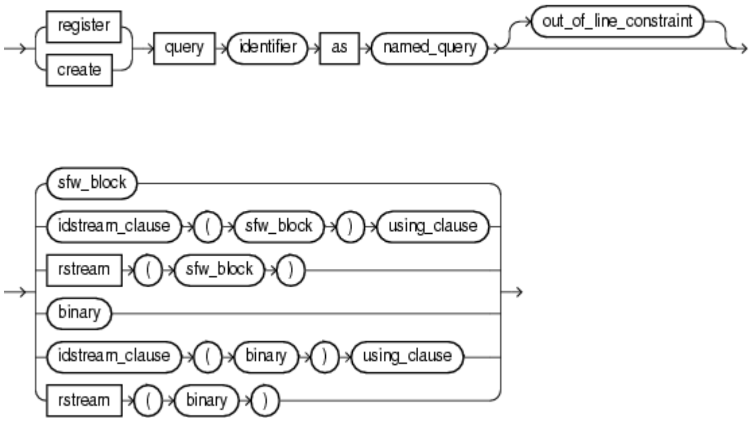


Figure 16-2 ***sfw_block*::=**

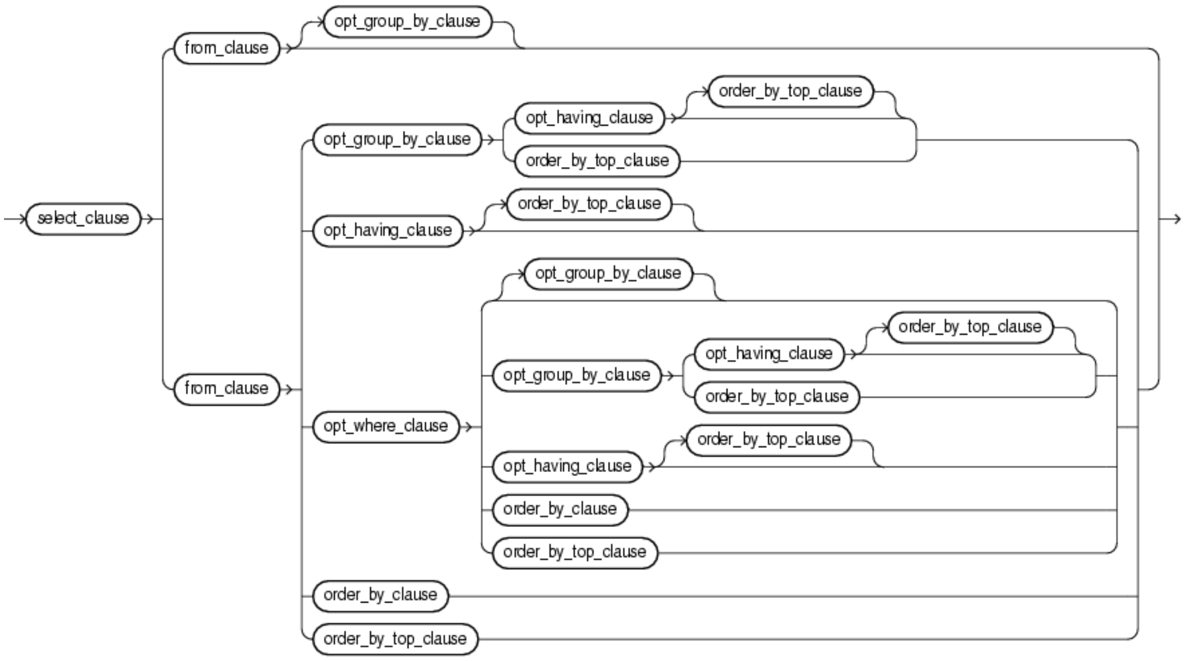


Figure 16-3 ***select_clause*::=**

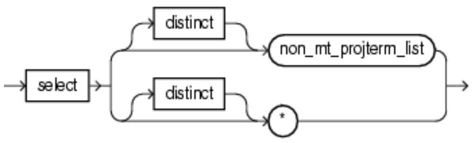


Figure 16-4 ***non_mt_projterm_list*::=**

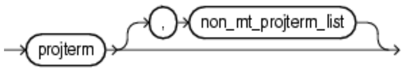
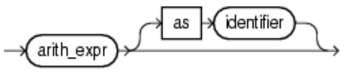


Figure 16-5 ***projterm*::=**



(*arith_expr*)

Figure 16-6 ***from_clause*::=**

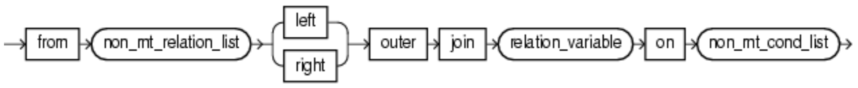


Figure 16-7 ***non_mt_relation_list*::=**

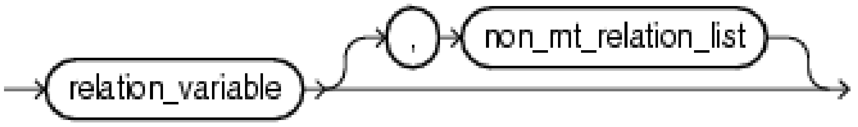
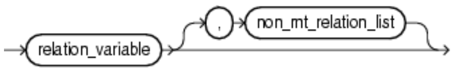


Figure 16-8 ***relation_variable*::=**



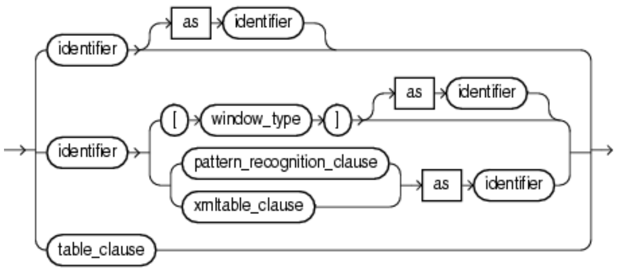


Figure 16-9 *window_type*::=

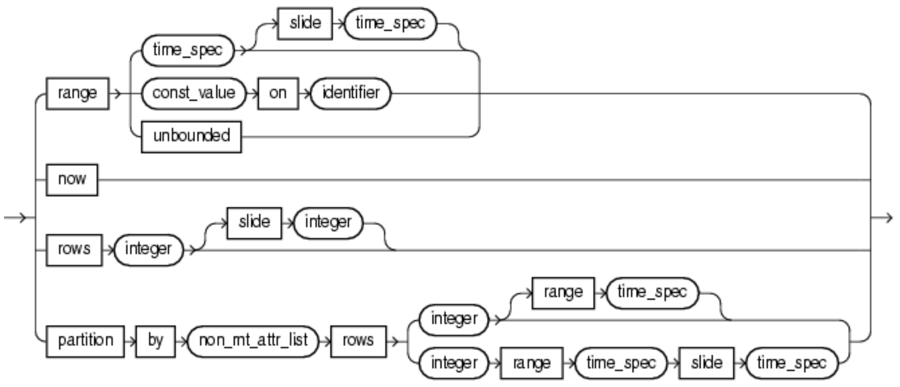


Figure 16-10 *table_clause*::=

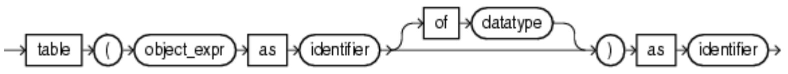


Figure 16-11 *opt_where_clause*::=

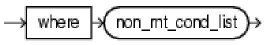


Figure 16-12 *opt_group_by_clause*::=

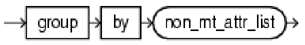


Figure 16-13 *order_by_clause*::=

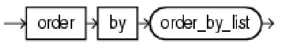


Figure 16-14 *order_by_top_clause*::=

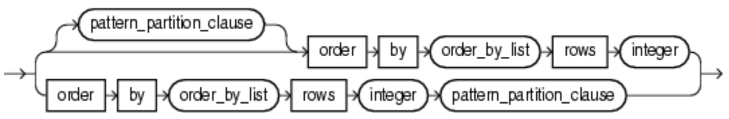


Figure 16-15 *order_by_list*::=

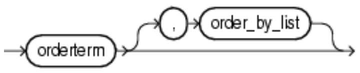


Figure 16-16 *orderterm*::=

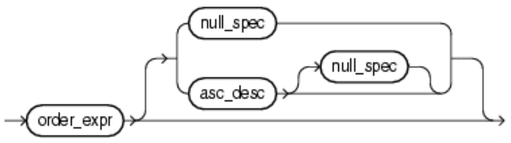


Figure 16-17 *null_spec*::=

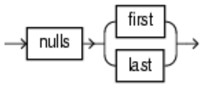


Figure 16-18 *asc_desc*::=

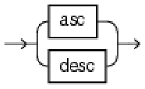


Figure 16-19 *opt_having_clause*::=

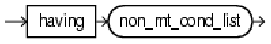


Figure 16-20 *binary*::=

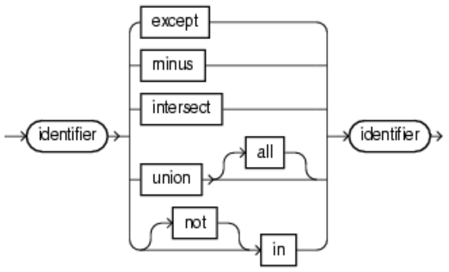


Figure 16-21 *idstream_clause*::=

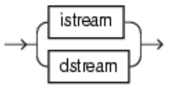


Figure 16-22 *using_clause*::=

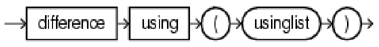


Figure 16-23 *usinglist*::=

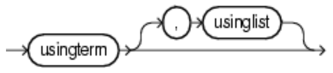


Figure 16-24 *usingterm*::=

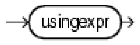


Figure 16-25 *usingexpr*::=

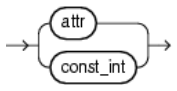


Figure 16-26 *xmltable_clause*::=

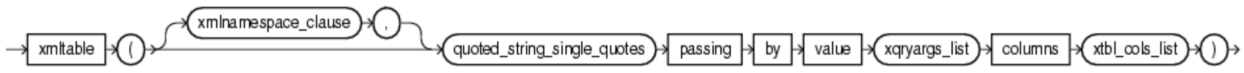


Figure 16-27 *xmlnamespace_clause*::=

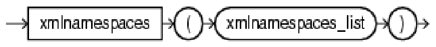


Figure 16-28 *xmlnamespaces_list*::=

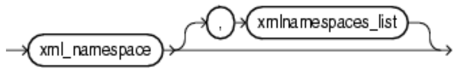


Figure 16-29 *xml_namespace*::=

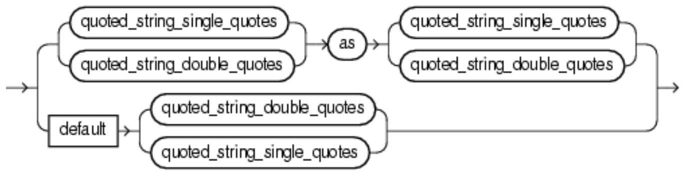

quoted_string_single_quotes
quoted_string_double_quotes
as
quoted_string_single_quotes
quoted_string_double_quotes
default
quoted_string_double_quotes
quoted_string_single_quotes

Figure 16-30 *xtbl_cols_list*::=

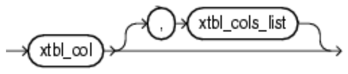
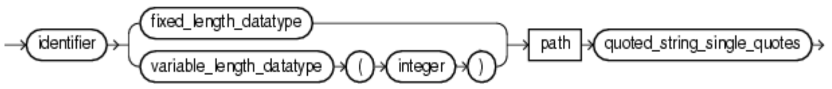


Figure 16-31 *xtbl_col*::=



16.2.1.1 Query Semantics

named_query

Specify the Oracle CQL query statement itself.

For syntax, see Query.

query

You can create an Oracle CQL query from any of the following clauses:

- *`sfw_block`*: a select, from, and other optional clauses.
- *`binary`*: an optional clause, often a set operation.
- *`xstream_clause`*: apply an optional relation-to-stream operator to your *`sfw_block`* or *`binary`* clause to control how the query returns its results.

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.
- *`from_clause`*: the stream or view from which your query selects.
- *`opt_where_clause`*: optional conditions your query applies to its selection
- *`opt_group_by_clause`*: optional grouping conditions your query applies to its result

  `order_by_clause`: optional ordering conditions your query applies to its results
- *`order_by_top_clause`*: optional ordering conditions your query applies to the top-`n` elements in its results
- *`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`

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.

Optionally, specify `distinct` if you want Event Processing to return only one copy of each set of duplicate tuples selected. Duplicate tuples are those with matching values for each expression in the select list.

non_mt_projterm_list

Specify the projection term or comma separated list of projection terms in the select clause of the Oracle CQL query statement.

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` 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.

from_clause

Specify the from clause of the Oracle CQL query statement by specifying the individual streams or views from which your query selects.

To perform an outer join, use the `LEFT` or `RIGHT OUTER JOIN ... ON` syntax. To perform an inner join, use the `WHERE` clause.

non_mt_relation_list

Specify the stream or view or comma separated list of streams or views in the from clause of the Oracle CQL query statement.

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. 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` . 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`. Optionally, use the `AS` keyword to specify an alias for the stream or view instead of using its name as is.

To access, as a relation, the multiple rows returned by a data cartridge function in the `FROM` clause of an Oracle CQL query, use a `table_clause`.

For more information, see:

- View.

window_type

Specify a built-in stream-to-relation operator.

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

table_clause

Use the data cartridge `TABLE` clause to access the multiple rows returned by a data cartridge function in the `FROM` clause of an Oracle CQL query.

The `TABLE` clause converts the set of returned rows into an Oracle CQL relation. Because this is an external relation, you must join the `TABLE` function clause with a stream. Oracle Event Processing invokes the data cartridge method only on the arrival of elements on the joined stream.

Use the optional `OF` keyword to specify the type contained by the returned array type or `Collection` type.

Use the `AS` keyword to specify an alias for the *`object_expr`* and for the returned relation.

Note the following:

- The data cartridge method must return an array type or `Collection` type.
- You must join the `TABLE` function clause with a stream.

time_spec

Specify the time over which a range or partitioned range sliding window should slide.

Default: if units are not specified, Oracle Event Processing assumes `[second|seconds]`.

For more information, see Range-Based Stream-to-Relation Window Operators and Partitioned Stream-to-Relation Window Operators.

opt_where_clause

Specify the (optional) where clause of the Oracle CQL query statement.

Because Oracle CQL applies the `WHERE` clause before `GROUP BY` or `HAVING`, if you specify an aggregate function in the `SELECT` clause, you must test the aggregate function result in a `HAVING` clause, not the `WHERE` clause.

In Oracle CQL (as in SQL), the `FROM` clause is evaluated before the `WHERE` clause. Consider the following Oracle CQL query:

```
SELECT ... FROM S MATCH_RECOGNIZE ( .... ) as T WHERE ...
```

In this query, the `S MATCH_RECOGNIZE ( .... ) as T` is like a subquery in the `FROM` clause and is evaluated first, before the `WHERE` clause. Consequently, you rarely use both a `MATCH_RECOGNIZE` clause and a `WHERE` clause in the same Oracle CQL query. Instead, you typically use views to apply the required `WHERE` clause to a stream and then select from the views in a query that applies the `MATCH_RECOGNIZE` clause.

For more information, see:

- Built-In Aggregate Functions and the Where, Group By, and Having Clauses
- Colt Aggregate Functions and the Where, Group By, and Having Clauses

- MATCH_RECOGNIZE and the WHERE Clause.

opt_group_by_clause

Specify the (optional) `GROUP BY` clause of the Oracle CQL query statement. Use the `GROUP BY` clause if you want Oracle Event Processing to group the selected stream elements based on the value of *`expr`*(s) and return a single (aggregate) summary result for each group.

Expressions in the `GROUP BY` clause can contain any stream elements or views in the `FROM` clause, regardless of whether the stream elements appear in the select list.

The `GROUP BY` clause groups stream elements but does not guarantee the order of the result set. To order the groupings, use the `ORDER BY` clause.

Because Oracle CQL applies the `WHERE` clause before `GROUP BY` or `HAVING`, if you specify an aggregate function in the `SELECT` clause, you must test the aggregate function result in a `HAVING` clause, not the `WHERE` clause.

For more information, see:

- Built-In Aggregate Functions and the Where, Group By, and Having Clauses
- Colt Aggregate Functions and the Where, Group By, and Having Clauses.

order_by_clause

Specify the `ORDER BY` clause of the Oracle CQL query statement as a comma-delimited list of one or more order terms. Use the `ORDER BY` clause to specify the order in which stream elements on the left-hand side of the rule are to be evaluated. The *`expr`* must resolve to a dimension or measure column. This clause returns a stream.

Both ORDER BY and ORDER BY ROWS support specifying the direction of sort as ascending or descending by using the ASC or DESC keywords. They also support specifying whether null items should be listed first or last when sorting by using NULLS FIRST or NULLS LAST.

For more information, see:

- Sorting Query Results

order_by_top_clause

Specify the `ORDER BY` clause of the Oracle CQL query statement as a comma-delimited list of one or more order terms followed by a `ROWS` keyword and integer number (`n`) of elements. Use this form of the `ORDER BY` clause to select the top-`n` elements over a stream or relation. This clause returns a relation.

Consider the following example queries:

- At any point of time, the output of the following example query will be a relation having top 10 stock symbols throughout the stream.

  ```
  select stock_symbols from StockQuotes order by stock_price rows 10
  ```

- At any point of time, the output of the following example query will be a relation having top 10 stock symbols from last 1 hour of data.

  ```
  select stock_symbols from StockQuotes[range 1 hour] order by stock_price rows 10
  ```

For more information, see:

- Sorting Query Results.

order_by_list

Specify a comma-delimited list of one ore more order terms in an (optional) `ORDER BY` clause.

orderterm

A stream element 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.

order_expr

order_expr can be an `attr` or `constant_int`. The `attr` can be any stream element or pseudo column.

null_spec

Specify whether or not nulls are ordered first (`NULLS FIRST`) or last (`NULLS LAST`) for a given order term.

asc_desc

Specify whether an order term is ordered in ascending (`ASC`) or descending (`DESC`) order.

opt_having_clause

Use the `HAVING` clause to restrict the groups of returned stream elements to those groups for which the specified *`condition`* is `TRUE`. If you omit this clause, then Oracle Event Processing returns summary results for all groups.

Specify `GROUP BY` and `HAVING` after the *`opt_where_clause`*. If you specify both `GROUP BY` and `HAVING`, then they can appear in either order.

Because Oracle CQL applies the `WHERE` clause before `GROUP BY` or `HAVING`, if you specify an aggregate function in the `SELECT` clause, you must test the aggregate function result in a `HAVING` clause, not the `WHERE` clause.

For more information, see:

- Built-In Aggregate Functions and the Where, Group By, and Having Clauses
- Colt Aggregate Functions and the Where, Group By, and Having Clauses.

binary

Use the *`binary`* clause to perform operations on the tuples that two streams or views return. Most of these perform set operations, receiving two relations as operands. However, the UNION ALL operator can instead receive two streams, which are by nature unbounded.

idstream_clause

Use an *`idstream_clause`* to specify an `IStream` or `DStream` relation-to-stream operator that applies to the query.

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

using_clause

Use a `DIFFERENCE USING` clause to succinctly detect differences in the `IStream` or `DStream` of a query.

For more information, see Detecting Differences in Query Results.

usinglist

Use a `usinglist` clause to specify the columns to use to detect differences in the `IStream` or `DStream` of a query. You may specify columns by:

- attribute name: use this option when you are selecting by attribute name.
- alias: use this option when you want to include the results of an expression where an alias is specified.
- position: use this option when you want to include the results of an expression where no alias is specified.

  Specify position as a constant, positive integer starting at 1, reading from left to right.

  The following example specifies the result of expression `funct(c2, c3)` by its position (3) in the `DIFFERENCE USING` clause `usinglist`.

```
<query id="q1">
    ISTREAM (
        SELECT c1, log(c4) as logval, funct(c2, c3) FROM S [RANGE 1 NANOSECONDS]
    )    DIFFERENCE USING (c1, logval, 3)
</query>
```

For more information, see Detecting Differences in Query Results.

xmltable_clause

Use an *`xmltable_clause`* to process `xmltype` stream elements using XPath and XQuery. You can specify a comma separated list of one or more XML table columns, with or without an XML namespace.

pattern_recognition_clause

Use a *`pattern_recognition_clause`* to perform advanced comparisons optimized for data streams.

For more information and examples, see Pattern Recognition With MATCH_RECOGNIZE.

16.2.1.2 Query Examples

For more examples, see Oracle CQL Queries, Views, and Joins.

Simple Query Example

The following example shows how to register a simple query `q0` that selects all (`*`) tuples from stream `OrderStream` where stream element `orderAmount` is greater than 10000.

```
<query id="q0"><![CDATA[
    select * from OrderStream where orderAmount > 10000.0
]]></query>
```

HAVING Example

Consider the query `q4` and the data stream `S2`. Stream `S2` has schema `(c1 integer, c2 integer)`. The query returns the relation.

```
<query id="q4"><![CDATA[
    select
        c1,
        sum(c1)
    from
        S2[range 10]
    group by
        c1
    having
        c1 > 0 and sum(c1) > 1
]]></query>
```

```
Timestamp   Tuple
1000        ,2
2000        ,4
3000        1,4
5000        1,
6000        1,6
7000        ,9
8000        ,
```

```
Timestamp   Tuple Kind  Tuple
5000:       +           1,2
6000:       -           1,2
6000:       +           1,3
```

BINARY Example: UNION and UNION ALL

The UNION and UNION ALL operators both take two operands and combine their elements. The result of the UNION ALL operator includes all of the elements from the two operands, including duplicates. The result of the UNION operator omits duplicates.

The UNION operator accepts only two relations and produces a relation as its output. This operator cannot accept streams because in order to remove duplicates, the Oracle CQL engine must keep track of all of the elements contained in both operands. This is not possible with streams, which are by nature unbounded.

The UNION ALL operator accepts either two streams (and producing a stream) or two relations (producing a relation) as its operands. Using one stream and one relation as operands is invalid for both operators.

Given the relations `R1` and `R2`, respectively, the `UNION` query `q1` returns the relation and the `UNION ALL` query `q2` returns the relation.

```
<query id="q1"><![CDATA[
    R1 UNION R2
]]></query>
<query id="q2"><![CDATA[
    R1 UNION ALL R2
]]></query>
```

```
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

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

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
```

Note:

UNION is one of the binary operator which removes duplicates in its output. The above is a trivial case where the attributes are of CQL native types. In case of an input relation with an OBJECT type attribute, user must add a correct implementation of hashCode() and equals() method in their Object definition class so that CQL Engine can compute the equality of member object attributes in two comparing tuples.

The following output is from using UNION ALL with two relations as operands.

```
Timestamp   Tuple Kind  Tuple
     1002:  +           15,0.14
     2002:  -           15,0.14
   200000:  +           20,0.2
   200000:  +           20,0.2
    20100:  -           20,0.2
   201000:  -           20,0.2
   400000:  +           30,0.3
   400000:  +           30,0.3
   401000:  -           30,0.3
   401000:  -           30,0.3
100000000:  +           40,4.04
100000000:  +           40,4.04
 10001000:  -           40,4.04
100001000:  -           40,4.04

Timestamp     Tuple
   100000:    20,0.1
   150000:    15,0.1
   200000:    5,0.2
   400000:    30,0.3
100000000:    8,4.04

Timestamp     Tuple
     1001:    10,0.1
     1002:    15,0.14
   200000:    20,0.2
   400000:    30,0.3
100000000:    40,4.04
```

The following output is from using UNION ALL with the two preceding streams as operands. Note that all of the elements are inserted events.

```
Timestamp   Tuple Kind  Tuple
     1001:  +           10,0.1
     1002:  +           15,0.14
   100000:  +           20,0.1
   150000:  +           15,0.14
   200000:  +           20,0.2
   200000:  +           5,0.2
   400000:  +           30,0.3
   400000:  +           30,0.3
100000000:  +           40,4.04
100000000:  +           8,4.04
```

BINARY Example: INTERSECT

The INTERSECT operator returns a relation (with duplicates removed) with only those elements that appear in both of its operand relations.

Given the relations `R1` and `R2`, respectively, the `INTERSECT` query `q1` returns the relation.

```
<query id="q1"><![CDATA[
    R1 INTERSECT R2
]]></query>

Timestamp   Tuple Kind  Tuple
1000:        +          10,30
1000:        +          10,40
2000:        +          11,20
3000:        -          10,30
3000:        -          10,40

Timestamp   Tuple Kind  Tuple
1000:       +           10,40
2000:       +           10,30
2000:       -           10,40
3000:       -           10,30

Timestamp   Tuple Kind  Tuple
1000:       +           10,40
2000:       +           10,30
2000:       -           10,40
3000:       -           10,30
```

BINARY Example: EXCEPT and MINUS

As in database programming, the `EXCEPT` and `MINUS` operators are very similar. They both take relations as their two operands. Both result in a relation that is essentially elements of the first operand relation that are not also in the second operand relation.

An important difference between `EXCEPT` and `MINUS` is in how they handle duplicate occurrences between the first and second relation operands, as follows:

- The EXCEPT operator results in a relation made up of elements from the first relation, removing elements that were also found in the second relation up to the number of duplicate elements found in the second relation. In other words, if an element occurs m times in the first relation and n times in the second, the number of that element in the result will be n subtracted from m, or 0 if there were fewer in m than n.
- The MINUS operator results in a relation made up of elements in the first relation minus those elements that were also found in the second relation, regardless of how many of those elements were found in each. The MINUS operator also

removes duplicate elements found in the first relation, so that each duplicate item is unique in the result.

The following examples illustrate the MINUS operator. Given the relations `R1` and `R2`, respectively, the `MINUS` query `q1` returns the relation.

```
<query id="q1BBAQuery"><![CDATA[
    R1 MINUS R2
]]></query>

Timestamp   Tuple Kind  Tuple
1500:       +           10,40
1800:       +           10,30
2000:       +           10,40
2000:       +           10,40
2100:       -           10,40
3000:       -           10,30

Timestamp   Tuple Kind  Tuple
1000:       +           11,20
2000:       +           10,40
3000:       -           10,30

Timestamp   Tuple Kind  Tuple
1500:       +           10,40
1800:       +           10,30
2000:       -           10,40
```

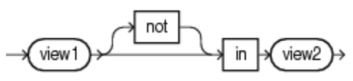
The following examples illustrate the EXCEPT operator. Given the relations `R1` and `R2`, respectively, the `EXCEPT` query `q1` returns the relation.

```
<query id="exceptQuery"><![CDATA[
    R1 EXCEPT R2
]]></query>

Timestamp   Tuple Kind  Tuple
1500:       +           10,40
1800:       +           10,30
2000:       +           10,40
2000:       +           10,40
2100:       -           10,40
3000:       -           10,30

Timestamp   Tuple Kind  Tuple
1000:       +           11,20
2000:       +           10,40
3000:       -           10,40

Timestamp   Tuple Kind  Tuple
1500:       +           10,40
1800:       +           10,30
2000:       +           10,40
2000:       -           10,40
```

BINARY Example: IN and NOT IN

In this usage, the query will be a binary query.

***in_condition_set*::=**



Note:

You cannot combine this usage with *in_condition_membership* as Using IN and NOT IN as a Membership Condition describes.

Consider the views V3 and V4 and the query Q1 and the data streams S3 (with schema (c1 integer, c2 integer)) and S4 (with schema (c1 integer, c2 integer)). In this condition test, the numbers and data types of attributes in left relation should be same as number and types of attributes of the right relation. The following example shows the relation that the query returns.

```
<view id="V3" schema="c1 c2"><![CDATA[
    select * from S3[range 2]
]]></query>
<view id="V4" schema="c1 d1"><![CDATA[
    select * from S4[range 1]
]]></query>
<query id="Q1"><![CDATA[
     v3 not in v4
]]></query>

Timestamp   Tuple
1000        10, 30
1000        10, 40
2000        11, 20
3000        12, 40
3000        12, 30
3000        15, 50
h 2000000

Timestamp   Tuple
1000        10, 40
2000        10, 30
2000        12, 40
h 2000000

Timestamp   Tuple Kind  Tuple
1000:       +           10,30
1000:       +           10,40
1000:       -           10,30
1000:       -           10,40
2000:       +           11,20
2000:       +           10,30
2000:       +           10,40
2000:       -           10,30
2000:       -           10,40
3000:       +           15,50
3000:       +           12,40
3000:       +           12,30
4000:       -           11,20
5000:       -           12,40
5000:       -           12,30
5000:       -           15,50
```

Select and Distinct Examples

Consider the query q1. Given the data stream S, the query returns the relation.

```
<query id="q1"><![CDATA[
    SELECT DISTINCT c1 FROM S WHERE c1 > 10
]]></query>
```

```
Timestamp   Tuple
 1000       23
 2000       14
 3000       13
 5000       22
 6000       11
 7000       10
 8000        9
10000        8
11000        7
12000       13
13000       14

Timestamp   Tuple
1000        23
2000        14
3000        13
5000        22
6000        11
```

XMLTABLE Query Example

Consider the query `q1` and the data stream `S`. Stream `S` has schema `(c1 xmltype)`. The query returns the relation.

For a more complete description of XMLTABLE, see 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>
```

```
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>"
```

```
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` and the data stream `S1`. Stream `S1` has schema `(c1 xmltype)`. The query returns the relation.

For a more complete description of XMLTABLE, see XMLTABLE 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>

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

Timestamp   Tuple Kind  Tuple
3000:       +     John,1
3000:       +     Jack,2
3000:       +     Jill,3
```

Data Cartridge TABLE Query Example: Iterator

Consider a data cartridge (`MyCartridge`) with method `getIterator`.

```
...
    public static Iterator<Integer> getIterator() {
        ArrayList<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        return list.iterator();
    }
...
```

Consider the query `q1`. Given the data stream `S0`, the query returns the relation.

```
<query id="q1"><![CDATA[

    select S1.c1, S1.c2, S2.c1
    from
        S0[now] as S1,
        table (com.acme.MyCartridge.getIterator() as c1) of integer as S2

]]></query>


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

Timestamp   Tuple Kind  Tuple
1:          +           1,abc,1
1:          +           1,abc,2
1:          -           1,abc,1
1:          -           1,abc,2
2:          +           2,ab,1
2:          +           2,ab,2
2:          -           2,ab,1
2:          -           2,ab,2
3:          +           3,abc,1
3:          +           3,abc,2
3:          -           3,abc,1
3:          -           3,abc,2
4:          +           4,a,1
4:          +           4,a,2
4:          -           4,a,1
4:          -           4,a,2
```

Data Cartridge TABLE Query Example: Array

Consider a data cartridge (`MyCartridge`) with method `getArray`.

```
...
    public static Integer[] getArray(int c1) {
        ArrayList<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        return list.toArray(new Integer[2]);;
    }
...
```

Consider the query `q1`. Given the data stream `S0`, the query returns the relation.

```
<query id="q1"><![CDATA[

    select S1.c1, S1.c2, S2.c1
    from
        S0[now] as S1,
        table (com.acme.MyCartridge.getArrayS1.c1) as c1) of integer as S2

]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
1:          +           1,abc,1
1:          +           1,abc,2
1:          -           1,abc,1
1:          -           1,abc,2
2:          +           2,ab,2
2:          +           2,ab,4
2:          -           2,ab,2
2:          -           2,ab,4
3:          +           3,abc,3
3:          +           3,abc,6
3:          -           3,abc,3
3:          -           3,abc,6
4:          +           4,a,4
4:          +           4,a,8
4:          -           4,a,4
4:          -           4,a,8
```

Data Cartridge TABLE Query Example: Collection

Consider a data cartridge (`MyCartridge`) with method `getCollection`.

```
...
    public HashMap<Integer,String> developers;
    developers = new HashMap<Integer,String>();
    developers.put(2, "Mohit");
    developers.put(4, "Unmesh");
    developers.put(3, "Sandeep");
    developers.put(1, "Swagat");

    public HashMap<Integer,String> qaengineers;
    qaengineers = new HashMap<Integer,String>();
    qaengineers.put(4, "Terry");
    qaengineers.put(5, "Tony");
    qaengineers.put(3, "Junger");
    qaengineers.put(1, "Arthur");
...
    public Collection<String> getEmployees(int exp_yrs) {
        LinkedList<String> employees = new LinkedList<String>();
        employees.add(developers.get(exp_yrs));
        employees.add(qaengineers.get(exp_yrs));
        return employees;
  }
...
```

Consider the query `q1`. Given the data stream `S0`, the query returns the relation.

```
<query id="q1"><![CDATA[

    RStream(
        select S1.c1, S2.c1
        from
            S0[now] as S1,
            table(S1.c2.getEmployees(S1.c1) as c1) of char as S2
    )

]]></query>
```

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

```
Timestamp   Tuple Kind  Tuple
1:          +           1,Swagat
1:          +           1,Arthur
2:          +           2,Mohit
3:          +           3,Sandeep
3:          +           3,Junger
4:          +           4,Unmesh
4:          +           4,Terry
```

ORDER BY Query Example

Use the ORDER BY clause with stream input to sort events that have duplicate timestamps. ORDER BY is only valid when the input is a stream and only sorts among events of the same timestamp. Its output is a stream with the sorted events.

Consider the query `q1`. Given the data stream `S0`, the query returns the relation. The query sorts events of duplicate timestamps in ascending order by tuple values.

```
<query id="q1"><![CDATA[
    SELECT *
    FROM S0
    ORDER BY c1,c2 ASC
]]></query>
```

```
Timestamp   Tuple
1000        7, 15
2000        7, 14
2000        5, 23
2000        5, 15
2000        5, 15
```

```
2000        5, 25
3000        3, 12
3000        2, 13
4000        4, 17
5000        1, 9
h 1000000000

Timestamp   Tuple Kind  Tuple
1000:       +           7,15
2000:       +           5,15
2000:       +           5,15
2000:       +           5,23
2000:       +           5,25
3000:       +           2,13
3000:       +           3,19
4000:       +           4,17
5000:       +           1,9
```

ORDER BY ROWS Query Example

Use the ORDER BY clause with the ROWS keyword to use ordering criteria to determine whether an event received by the query should be included in output. ORDER BY ROWS accepts either stream or relation input and outputs a relation.

The ORDER BY ROWS clause maintains a set of events whose maximum size is the number specified by the ROWS keyword. As new events are received, they are evaluated, based on thr order criteria and the ROWS limit, to determine whether they will be added to the output.

Note that the output of ORDER BY ROWS is not arranged based on the ordering criteria, as is the output of the ORDER BY clause. Instead, ORDER BY ROWS uses the ordering criteria and specified number of rows to determine whether to admit events into the output as they are received.

Consider the query `q1`. Given the data stream `S0`, the query returns the relation.

```
<query id="q1"><![CDATA[
    SELECT c1 ,c2
    FROM S0
    ORDER BY c1,c2 ROWS 5
]]></query>

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

Timestamp   Tuple Kind  Tuple
1000:       +           7,15
2000:       +           7,14
2000:       +           5,23
2000:       +           5,15
2000:       +           5,15
2000:       -           7,15
2000:       +           5,25
3000:       -           7,14
3000:       +           2,13
3000:       -           5,25
```

```
3000:       +           3,19
4000:       -           5,23
4000:       +           4,17
5000:       -           5,15
5000:       +           1,9
```

In the following example, the query uses the PARTITION keyword to specify the tuple property within which to sort events and constrain output size. Here, the PARTITION keyword specifies that events in the input should be evaluated based on their symbol value.

In other words, when determining whether to include an event in the output, the query looks at the existing set of events in output that have the same symbol. The ROWS limit is two, meaning that the query will maintain a set of sorted events that has no more than two events in it. For example, if there are already two events with the ORCL symbol, adding another ORCL event to the output will require deleting the oldest element in output having the ORCL symbol.

Also, the query is ordering events by the value property, so that is also considered when a new event is being considered for output. Here, the DESC keyword specifies that event be ordered in descending order. A new event that does not come after events already in the output set will not be included in output.

```
<query id="q1"><![CDATA[
    SELECT symbol, value
    FROM S0
    ORDER BY value DESC ROWS 2
    PARTITION BY symbol
]]></query>

Timestamp   Tuple
1000        ORCL, 500
1100        MSFT, 400
1200        INFY, 200
1300        ORCL, 503
1400        ORCL, 509
1500        ORCL, 502
1600        MSFT, 405
1700        INFY, 212
1800        INFY, 209
1900        ORCL, 512
2000        ORCL, 499
2100        MSFT, 404
2200        MSFT, 403
2300        INFY, 215
2400        MSFT, 415
2500        ORCL, 499
2600        INFY, 211

Timestamp    Tuple Kind    Tuple
1000         +             ORCL,500
1100         +             MSFT,400
1200         +             INFY,200
1300         +             ORCL,503
1400         -             ORCL,500
1400         +             ORCL,509
1600         +             MSFT,405
1700         +             INFY,212
1800         -             INFY,200
1800         +             INFY,209
1900         -             ORCL,503
```

```
1900        +            ORCL,512
2100        -            MSFT,400
2100        +            MSFT,404
2300        -            INFY,209
2300        +            INFY,215
2400        -            MSFT,404
2400        +            MSFT,415
```

16.2.2 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
- Oracle CQL Queries, Views, and Joins.

Syntax

You express the a view in a `<view></view>` element as the example below shows.

The `view` element has two attributes:

- `id`: Specify the `identifier` as the `view` element `id` attribute.

  The `id` value must conform with the specification given by Figure 7-6.
- `schema`: Optionally, specify the schema of the view as a space delimited list of attribute names.

  Oracle Event Processing server infers the types.

```
<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.

Examples

The following examples illustrate the various semantics that this statement supports. For more examples, see Oracle CQL Queries, Views, and Joins.

The following example shows how to register view `v2`.

Example 16-1   Registering a View Example

```
<view id="v2" schema="cusip bid ask"><![CDATA[
    IStream(select * from S1[range 10 slide 10])
]]></view>
```