TIBCO BusinessEvents® Enterprise Edition

Event Stream Processing Query Developer Guide

Version 6.4.0 | December 2025

Contents

Query Features Overview

The query language enables you to make queries using an SQL-like language. Queries are executed in query agents. You can query cache content (requires Cache OM), and you can also query events (event stream processing).

Note: Configuration of query agents is explained in TIBCO BusinessEvents Developer Guide.

Query Agents

Queries can only be executed by specialized agents called *query agents*. One engine (node) can have multiple query agents, or a mixture of inference agents and query agents.

Query agents have channels and destinations. They can execute rule functions, but not rules. Query agents have no Rete network for inferencing.

Querying the Cache

When a query agent is deployed as part of a TIBCO BusinessEvents application that uses cache object management, you can query data in the cache.

Query features provide view-only access into the cache. You cannot use query language to do any updates to data in the cache.

It’s important to understand basic cache configuration and the part query agents play in a cache cluster. See chapters on Cache OM in TIBCO BusinessEvents Developer Guide.

Tip: You can load objects into the cache so you can then query them. To load objects into the cache, use the `DataGrid.CacheLoad*()` functions. For details on these functions, see their tooltips, and also see TIBCO BusinessEvents Developer Guide.

Querying the Event Stream

Query agents can listen to an event stream. The event stream can consist of messages sent out on a Rendezvous subject, or a JMS topic or queue, or other source that a TIBCO BusinessEvents destination can listen to. Events can also be generated internally and piped straight to a query.

Event stream processing in the query agent is highly performant and can handle very large numbers of incoming messages. The query agent runs continuous (or snapshot) queries against the events.

Continuous queries against the event stream make comparisons across event streams, as if they were tables. Thus, event stream processing can be termed *channel-centric computing*. This approach is ideal when you need to operate on sets of events (such as for aggregations). This is traditionally associated with financial data feeds, although it might also be used in detecting patterns in streams for smart grid meter feeds, website monitoring feeds, and so on.

Distilling Data

The query agent can assert events, such that another query (or a locally deployed inference agent) can listen to them. These internally generated events enable you to build several tiers of queries, each aggregating and abstracting the data into ever more interesting information. The distilled data can be sent out through a channel to a TIBCO BusinessEvents application or external application as needed.

See Event Stream Processing (ESP) Queries for details.

Note: You cannot use hot deployment for query-related resources.

Two Types of Queries—Snapshot and Continuous

Two types of queries are available, snapshot queries and continuous queries.

Snapshot Queries

Snapshot queries return data from the cache as it exists at a moment in time. A snapshot query returns a single, finite collection of entities that exist in the cache.

See The Query Language Usage and in particular, see Simple Snapshot Query Example for better understanding.

Continuous Queries

Continuous queries collect data as objects are added, deleted, or modified in the cache. That is, continuous queries work on data streaming through the query. Continuous queries continue to gather and return data when notified of changes, until you stop the query. Continuous queries use windows (explicit or implicit) to process data (snapshot queries do not). Snapshot queries are not used for event stream processing.

See Continuous Queries for more details.

Summary of Functions Used to Create and Execute Queries

All queries are created and executed using a set of query functions. The query functions are called from rule functions in the query agent.

Three functions are mandatory, and additional functions are available for different purposes.

Create the Query

First a `Query.create()` function creates the *query definition* which contains the query text and a name for the definition.

Create the Query Statement

Then the `Query.Statement.open()` function is used to create a *query statement*, which is a named instance of the query definition.

Execute an Instance of the Query Statement and Obtain Results

Choose one of these ways to execute a query instance:

- For snapshot queries, you can use either the `Query.Statement.execute()` function or a `Query.Statement.executeWithCallback()` function.
- For continuous queries you must use the `Query.Statement.executeWithCallback()` function or `Query.Statement.executeWithBatchCallback()` function, with the `IsContinuous` parameter set to true.

  These functions are generally placed in an event preprocessor rule function.

Use Results

To use results returned by a query, you can create events to send information between query and inference agents. You could also send results out to some other system. The use to which results are put depends on the business need.

See The Query Language Usage for more details.

Query From a Rule (in an Inference Agent)

Queries can only run in a query agent. Rules can only run in an inference agent. In order for a rule to trigger a query to execute, the rule must send an event to the query agent. In order for the query results to be used in a rule, the query agent must send them in an event to an inference agent.

A rule in the inference agent sends an event to destination D1, including any necessary query parameters.

The query agent listens for messages on destination D1.

When event E1 arrives, an event preprocessor executes a query statement.

A query function collects results into event, E2 and sends it to destination D2.

The inference agent listens on destination D2.

When event E2 arrives, a rule in the inference agent collects the results from the event and processes them as needed.

Query as a Pre-filter

Query agents can act as pre-filters and routers. Suppose you want to check for the existence of a concept in the cache, using properties of an event. If the concept does not exist, you want to create it.

You can achieve this result as follows:

The query agent listens for messages on a destination D1.

On receiving a message (event A) at D1, the query agent executes the query statement to determine if the corresponding concept exists in cache.

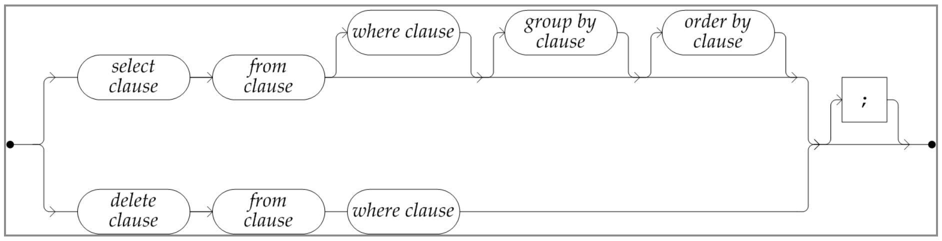
- If the query finds an existing concept, nothing happens.
- If the query does not find an existing concept the agent sends event A to destination D2.

  The inference agent listens for events (messages) on destination D2.

  On receiving an event at D2, a rule in the inference agent creates the concept.

Query Language Components

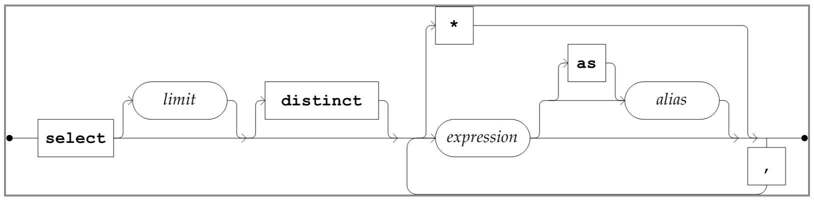
The text of a query uses a structure similar to the structure of a SELECT statement in SQL, and it has parallels with the structure of a TIBCO BusinessEvents rule, too. The query text is provided as an argument to the `Query.create()` function.



The syntax diagrams shows the structure of a query and of each clause in a query. Read them from left to right. Items above or below the main line are optional. Items that can repeat are shown by lines that loop back from the end to the beginning of the repeating section, along with the separator character.

Select Clause

In the select clause, you specify columns that will appear in the query results.



In the example, a `select` clause projects two columns, `address` and `name`, properties of the concept `/customer`. The alias for the customer concept is the letter `c`:

```
select c.name, c.address from /customer c
```

You can also give each projection an alias, for example:

```
select c.name as name
```

You can check for NULL values using equality `[=]` operator:

```
Select * from /concepts/test as cp where cp.value = null;
```

The use of the optional "`as`" makes the code more readable.

In the `select` clause you can use the following:

- Literal values
- Catalog functions and rule functions
- Entities that are declared in the `from` clause, unless you are using a `group by` clause (see Group by Clause)

You can use an optional `limit` clause to specify the maximum number of rows to return, and you can use an `offset` to ignore the first *n* rows.

You can use an optional `distinct` clause to prevent the query from returning duplicate rows.

Examples of Select Clauses

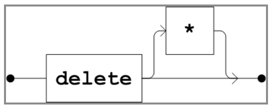
These examples show only the `select` clause. A complete query requires a `select` and a `from` clause. (# is the escape character. See Keywords and Other Reserved Words.)

```
select A.*
select {limit: first 10} A.name
select /#DateTime/now() as C
select /RuleFunctions/GetState() as D
select /#String/concat(B.customerId,”ABC”) as E
select B.*, A.custId id, B@extId as extId
```

Delete Clause

The delete clause is used only in a delete query. Delete queries are used in a specific situation only.

See The Delete Query for more details.



In the delete clause, you cannot specify columns. The concept specified in the from clause is deleted.

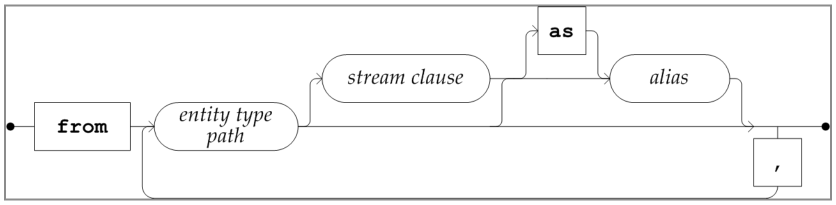
Examples of Delete Clause

These examples show only the `delete` clause. A complete query requires a `delete` and a `from` clause. The `from` clause can specify only one concept type.

```
delete *
delete
```

From Clause

Just as a rule declaration specifies the scope of the rule, the `from` clause specifies the scope of the query. The items in the `from` clause must exist in the project ontology.



Using Strings (Instead of Variables) in From Clauses

Bind variables cannot be used in the `from` clause—you cannot use `select * from $someConcept`. However, to achieve a similar result you can use a new string to construct the query as shown in the following examples:

```
String conceptName1 = "/Concepts/Concept1";
Query.create("newQuery1", "select * from " + conceptName1);
String conceptName2 = "/Concepts/Concept2";
Query.create("newQuery2", "select * from " + conceptName2);
```

Continuous Queries

The `from` clause in a continuous query can specify *window policies*. See Overview of Continuous Queries for more information.

Examples

The `select` and `from` clauses are required for all queries.

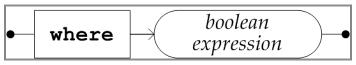
```
select * from /Concepts/Address as A
select * from /Concepts/Customer B
select * from /EntityA as A
select * from /EntityB  B
select * from /EntityX, /EntityY, /EntityZ
```

Where Clause

The optional `where` clause is analogous to a rule’s conditions. The expression in the `where` clause can be simple or complex.

In the `where` clause you can use following:

- Literal values
- Catalog functions and rule functions
- Entities that are declared in the `from` clause



Examples

Following checks for NULL values using equality `[=]` operator:

```
Select * from /concepts/test as cp where cp.value = null;
```

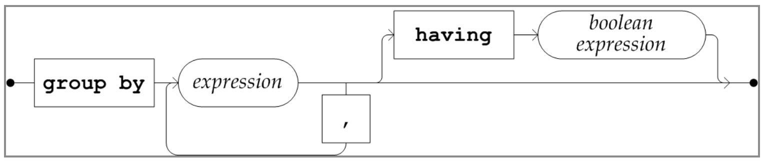
Pound or hash (#) is the escape character. See Keywords and Other Reserved Words.

```
where A.customerId = B.customerId
where A.id = B@extid                    // Entity attributes
and ( B@parent.name = 'ABCD' or C.name = "EFGH" )
and  A.tokens[5]  = 50                  // array property
and  ( A.containedConceptE.price > 100
or B.startTime > /#DateTime/addMinute(/#DateTime/now(),5) )
and B.value between 2 and 5
```

Tip: The pound sign (#) is used to escape reserved (key) words. See Keywords and Other Reserved Words for a complete listing.

Group by Clause

The optional `group by` clause allows you to group entities that share one or more criteria into a single row. Each group is represented by one row.



This allows you to use any of the standard group functions that are applicable, such as those used to calculate minimum, maximum, count, sum, average.

Aggregation functions operate on all entities (and their attributes and properties) that make up a given group. For example, you could find out how many customers are in each zip code as follows:

```
select c.zipcode from customer c group by c.zipcode;
```

Note that, although the `group by` clause reduces the result set to a list—in this example to a list of zip codes—additional information from the query is internally available to the aggregation functions.

Group By Usage

The select clause can use only the group by criteria and aggregation functions.

For example, the following example is valid:

```
select s.deptName, count(*)
  from /Student s
  group by s.deptName
```

However, the following example is *invalid*:

```
INVALID  select s.deptName, s.deptNo, count(*)
           from /Student s
           group by s.deptName
```

In the second example, `s.deptNo` does not appear in the `group by` clause and therefore it cannot be used in the `select` clause.

Using a Dummy Group Expression for Aggregation

Suppose you want to get a count of all entities in the `from` clause. In this case you must use a `group by` clause that creates a *dummy group*. In this case, all the rows are in the same group. As an example:

```
select count(*)
  from /Student s
  group by 1
```

The group by clause restricts the columns that can be used in the select clause. So, as an example, this usage is *invalid*:

```
INVALID  select s.deptName, count(*)
           from /Student s
           group by 1
```

Dummy groups are created when you specify a constant in the group by clause. For example, you can specify a dummy group in any of the following ways:

```
group by ""
group by 1
group by 2
group by "hello"
```

Any constant can be used.

Optional having Clause

The optional `having` clause allows you to apply conditions after entities are grouped. For example this query returns the number of customers in each zip code, except for those zip codes where there are three or fewer customers:

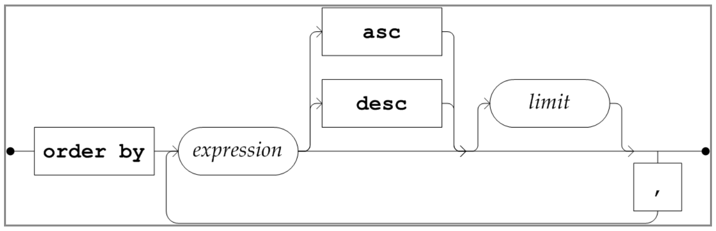
```
select c.zipcode, count(*) as count_zipcode
from /customer c
group by c.zipcode
having count_zipcode > 3;
```

Note that the `having` clause accepts aliases declared in the `select` clause.

You can also use aggregation functions in the `having` clause in order to apply conditions on the whole group.

Order by Clause

The optional `order by` clause enables you to sort the results in ascending or descending order.



In a continuous query, each set of ordered results in a window constitutes one *batch* of results. For an example, see Example Showing Batching of Return Values (Continuous Queries).

See also Limit Clause.

Examples

Pound or hash (#) is the escape character. See Keywords and Other Reserved Words.

In the following example, each row in the result shows the ID of a customer who has placed three or more orders each of which contained 5 or more lines.

```
order by A.State, C, D, E
order by A@extId, B.name {limit : first 10}
select o.customerId as cid
from /Concepts/#Order o
where o.lines@length >= 5
group by o.customerId
having count(*) >= 3
order by cid desc;
```

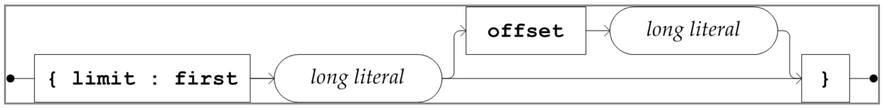
Limit Clause

You can use an optional `limit` clause in a `select` or an `order by` clause.

When used in a `select` clause, it limits the maximum number of rows to return. The limit clause is applied last after the all the clauses are executed on the result set.

You can also use an optional `offset` to ignore the first *n* rows.

When used in an ordered by clause, the limit applies to each of the items in the ordered list (after the ordering is executed). See Working With Implicit Windows.



Example Showing Use in Select Clause

```
select {limit: first 10 offset 20} c.name from /Customer c
```

Without the limit clause, this query would return all customers. With the limit, it returns 10 customers, with an offset of 20. That is, it returns customers 20-30.

Example Showing Use in Order By Clause

The following query keeps count of the number of students per department. Every time a student enrolls or leaves, the count changes and the query produces the entire list sorted on the count, sorted in descending order, and limited to the first two.

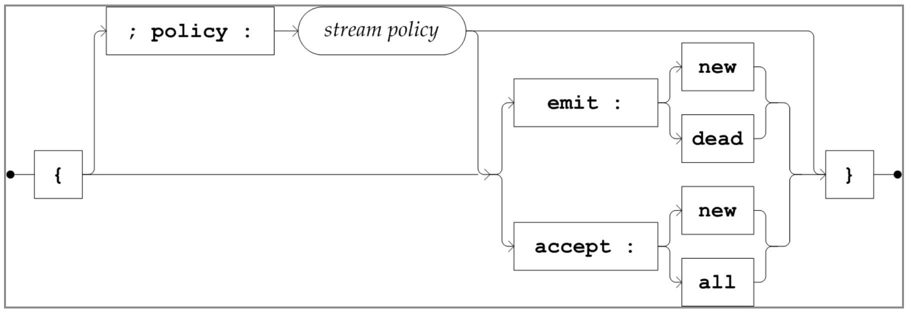
```
select s.deptName, count(*)
  from /Student s
  group by s.deptName
  order by count(*) desc {limit: first 2};
```

The `limit` clause specifies that only the first two of the ordered lists of departments are returned by the query: the list of departments with the largest number of students, and the list of departments with the second largest number of students.

Stream Clause

The stream clause is used for continuous queries only. It is used within a `from` clause.

See Stream Policy for details on how a window is defined.



Use of Accept:New and Accept:All

Events and concepts (entities) can be deleted by rules. By default (`accept:all`), if a continuous query has already seen an entity before, then it will expect a delete or modify notification from the cache cluster. Therefore the query must keep track of such things.

However, if you specify the `accept:new` clause, then the continuous query does not have to track such things, and the memory footprint of the query is reduced.

The `accept:new` clause is required for event stream processing (ESP) queries. See Event Stream Processing (ESP) Queries.

Use of Emit: New and Emit: Dead

The `emit` keyword determines whether the query is evaluated when an entity enters the window (`emit : new`) or when an entity leaves the window (`emit : dead`).

The default value is `emit:new`.

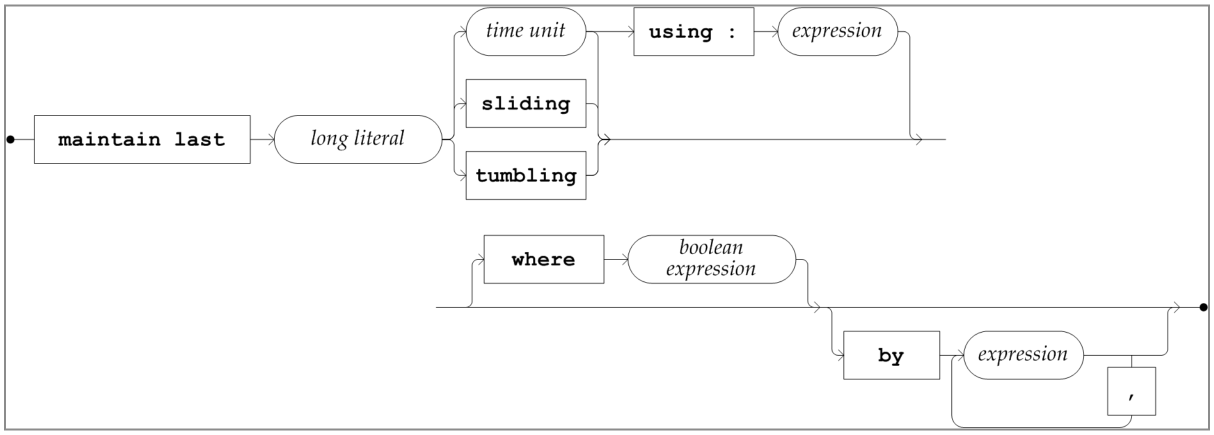
Note: Do not use emit clauses with aggregations.

For examples showing usage, see the following:

- Using Emit New to Create a Counter
- Delaying Output with an Emit Dead Clause

Stream Policy

The stream policy (also known as a window policy) is used for continuous queries only. It determines what kind of window is used: a time window, sliding window, or tumbling window.



See Working With Sliding Tumbling and Time Windows and examples following: Sliding Window Examples (Cache Queries), Tumbling Window Examples (Cache Queries), and Time Window Examples (Cache Queries).

Note that continuous queries that use an implicit window do not have a stream policy. See Working With Implicit Windows.

The value of *`long literal`* specifies the size of the window. When used for a time window, the value refers to a time unit specified by *`time unit`*. The time unit can be specified in milliseconds, seconds, minutes, hours or days. For example: `maintain last 5 minutes` defines a time window of five minutes.

For sliding and tumbling windows, the number refers to a number of entities.

Using Clause

When the query specifies time units, you can specify a start time by including a `using` clause. The expression could refer to a timestamp property in the entity, for example. If the `using` clause is absent, the start time is the moment the entity enters the window.

Where Clause

The optional `where` clause is used as a pre-filter (a filter on results that enter the window). It eliminates entities that are not useful for the query, optimizing performance.

By Clause

Maintaining a single window (like a sliding window) over all the events in the window may not be what you need for a query. The (optional) `by` clause allows you to do aggregations within the window. In this regard, the `by` clause is similar to the `group by` clause.

For example, instead of a single window of size 50 that contains all the entities, you can maintain a window of size 50 for each combination of values for the fields in the `by` section:

```
select car.id, car.color from "CarEvent" {policy: maintain last 50
sliding  where type = "Sedan" by country, state, city} car;
```

See Explicit Window Example (Cache Query) for a detailed discussion of an example that uses a stream policy. See Time Window Examples (Cache Queries) for more examples.

The Query Language Usage

The query language can be used to create and execute queries, and use results returned by a query.

Queries Construction and Query Results Usage

To implement queries, you put query text (SQL-like statements) as arguments to an appropriate function from the CEP Query function catalog and place the query functions in one or more rule functions. You can also use bind variables in many clauses to create prepared statements.

When you deploy an agent to query cache data, you can query concepts and simple events in the cache. You cannot query scorecards or time events because they do not exist in the cache. You cannot query the objects in the Rete network itself, or those in the backing store, just those in the cache.

When you deploy an agent to query an incoming event stream, you can query events.

Note: You can use arrays within expressions in a query, but returning arrays in the results of the query is not supported in this release.

Query Function Catalog

A catalog of functions called CEP Query is provided for use in writing and managing queries.

The following categories and functions are provided in the catalog:

- `Query` category: `create()`, `delete()`, `exists()`
- `Callback` category: `delete()`, `exists()`
- `ResultSet` category: `close()`, `get()`, `isBatchEnd()`, `isOpen()`, `next()`

- `Statement` category: `clearSnapshotRequired()`, `clearVars()`, `close()`, `execute()`, `executeWithCallback()`, `executeWithBatchCallback()`, `getSnapshotRequired()`, `getVar()`, `isOpen()`, `open()`, `setSnapshotRequired()`, `setVar()`

Each category also has a `Metadata` subcategory, which contains functions such as `findColumn()`, `getColumnCount()`, `getColumnName()`, `getColumnType()`, `getQueryName()`, and `getStatementName()`.

Tooltips associated with all these functions show the function signatures and other helpful text. The tooltips are available in TIBCO BusinessEvents Studio and you can also refer *TIBCO BusinessEvents Functions Reference* for more information about the catalog functions.

For general information on using the functions provided with TIBCO BusinessEvents, see TIBCO BusinessEvents Developer Guide.

Functions within Queries

Many of the available catalog functions as well as custom functions can be used in a query agent. You can also use rule functions from the same project.

Functions that Can Be Used in a Query Agent

Functions that can be used in a query agent are marked with a blue *q*. (They may have more decorations if they are usable in other areas such as Decision Manager).

Functions that Cannot be Used in a Query Agent

The following functions cannot be used in a query agent:

- Rule functions with a Validity attribute that is set to anything other than "Action, Condition, Query."
- Ontology functions.
- All catalog functions that assert, modify or delete objects in the cache or in working memory. Queries cannot change the cache.

Bind Variables Usage

You can place bind variables in the query text argument of the query definition. The values of the variables can be set when a query statement is opened, enabling a single query definition to be reused.

See Bind Variables in Query Text for details.

Lifecycle of a Query—Use of Query Functions

Lifecycle of a query involves creation, execution, and gathering of results. Functions can be used to create and execute queries, and to gather query results.

Also see Result Set Data Usage (Snapshot Queries) and Callback Rule Function Data Usage for details on how to get and use query results.

Query Definition Creation

A *query definition* is a Java runtime object (similar to a factory class).

Creating a query definition is a separate step from opening and executing a query statement. Creating a query definition is the most expensive step in the process of making the query available for execution. Therefore it is often best done at engine startup. The syntax of the function is:

```
Query.create(String QueryDefinitionName, String QueryText, boolean
isNativeQueryRawResults);
```

The `QueryDefinitionName` is used in other functions to identify the query definition. The query text contains the select statement.

For example,

```
Query.create("report","select zipcode, total_sales, agent_name from
/Concepts/Sales where total_sales > $min");
```

Where `$min` is a bind variable whose value is provided at runtime.

If a query statement based on this definition is executed and returns a result set, the result set columns would be, `zipcode`, `total_sales`, and `agent_name`, with rows of entity values that match the condition specified at the time the query was executed.

Query Statement Opening

A *query statement* is an object that represents one instance of the query. You can create multiple statements that can run in parallel.

Use the open() function to open a query statement.

The syntax of the function is:

```
Query.Statement.open(String QueryDefinitionName, String StatementName);
```

The *`QueryDefinitionName`* references the query definition that contains the query text. The statement name defined here is used in other functions to identify this query statement.

For example,

```
Query.Statement.open("report", S_Id);
```

Where `S_Id` is a string variable that contains the statement name. Names can be constructed in various ways, as shown in Simple Snapshot Query Example.

Bind Variables Value Setting (if Used)

For the named query statement, set values for bind variables (if any are used in the query definition) before executing them. This sequence is required.

The functions need not be executed right after each other, however. For example, the `Query.Statement.open()` function could be in a startup rule function and the `Query.Statement.setvar()` function could be in a rule function called on assertion of an event, followed by the `Query.Statement.execute()` function.

Note: Open a named query statement for each set of variable values that are used at execution time. For example, if you set the variable values two different ways, you would provide two open query statements, each with its own name, to keep the configured queries and their returned information separate from each other

The syntax of the function is:

```
Query.Statement.setVar(String StatementName, String BindVariableName, Object
Value);
```

For example,

```
Query.Statement.setVar(S_Id, "min", evt.min_total_sales);
```

See Bind Variables in Query Text for more details.

Query Statement Execution

To execute a query and specify how a query returns values, you can use either of the execute functions.

The following are the available execute functions:

- `Query.Statement.execute()` provides results using a result set. This function is used for snapshot queries only.
- `Query.Statement.executeWithCallback()` provides results using a callback rule function, which is called once for each matching result. This function can be used with snapshot or continuous queries.
- `Query.Statement.executeWithBatchCallback()` provides results using a callback rule function, which is called once at the end of each batch of results. This function can be used only with continuous queries.

Obtain Results Using a Result Set

The `Query.Statement.execute()` function returns values in a *result set*.

The result set is a tabular form (with rows and columns) on which you can perform operations to return data. It is used for snapshot queries only. Execution is synchronous.

```
Query.Statement.execute(String StatementName, String resultsetName);
```

For example:

```
Query.Statement.execute(S_Id, evt@extId);
```

In the example, `S_Id` is a string variable providing the name that was given in the `Query.Statement.open()` function. The example shows use of the external ID of event `evt` (`evt@extId`) as the result set name, as a way to ensure that each result set has a unique name.

See Result Set Data Usage (Snapshot Queries) for more information.

Close the Result Set after Collection

After you have collected the data you need, close the result set. You can close the result set directly, or close it indirectly by closing a higher-level item such as the statement or the query definition. To close the result set use the following function:

```
Query.ResultSet.close(String ResultsetName);
```

For example:

```
Query.ResultSet.close("rset");
```

Obtain Results Using a Callback Rule Function

You can pass query results to a callback rule function.

Two functions are available for this purpose:

- `Query.Statement.executeWithCallback()` calls the rule function once for each row of results, as well as at the end of a batch (if ordering is used) and at the end of the execution. Results are sent to the callback rule function as individual rows of data. (See Example Showing Batching of Return Values (Continuous Queries) for an example.)
- `Query.Statement.executeWithBatchCallback()` calls the rule function at the end of a batch and at the end of the execution. The results are sent to the callback rule function as an array of rows of data, at batch end. It is generally used for queries that contain an `order by` clause, which results in useful batches of data. It is useful, for example, when you want to send an outbound message containing all the results of a batch.

Only `Query.Statement.executeWithCallback()` can be used for snapshot queries. When used with snapshot queries, the query looks at the current state of the cache and calls the rule function once for each matching row, in quick succession. Batching is not used with snapshot queries.

Both functions are used for continuous queries. You set the `IsContinuous` argument to `true` so that the query runs as a continuous query. When used in continuous queries, the query listens for changes to the cache, or listens to events if the query is listening to events, and calls the rule function as matches occur over the lifetime of the query.

Note: Use `Query.Statement.executeWithCallback()` only when batches of results will be small.

The format of the `Query.Statement.executeWithCallback()` function is shown in following sample. The format of the `Query.Statement.executeWithBatch Callback()` function is the same (but the way it sends results to the callback function is a little different).

```
Query.Statement.executeWithCallback(
String   statementName,
String   listenerName,
String   callbackUri,
boolean  isContinuous,
Object   closure)
```

The *`listenerName`* parameter keeps results from different executions separate from each other.

The *`callbackUri`* parameter value provides the project path to the callback rule function.

The `isContinuous` parameter defines if the query is a snapshot or continuous query.

The `closure` parameter is stored during the execution of the query, and provided as a parameter to the callback function every time that function is called.

For example:

```
String execID = evt@extId;
Query.Statement.executeWithCallback(
MyStmt, MyexecID, "/MyRuleFunction", false, evt);
```

See Callback Rule Function Data Usage for details.

Statement Closing and Query Definition Deletion

You can close or delete at different levels. You can delete a query definition to make room for new query definitions. You can also delete (close) the statement that is running, without deleting the query definition itself.

Use the following functions as needed for your situation:

```
Query.Statement.close(String StatementName);
Query.delete(String QueryDefinitionName);
```

When you delete a query or a statement, all their subordinate artifacts are deleted as well, including result sets.

You can also close just the result set. See Close the Result Set after Collection.

Result Set Data Usage (Snapshot Queries)

Use the `Query.Statement.execute()` function to returns values in a *result set*.

See Obtain Results Using a Result Set for details about obtaining results.

Move the Cursor to the Next Row

The result set maintains a cursor (that is, a reference) on the current row, initially positioned just before the first row so that you can perform operations on the table. The only way to do operations on the table is through the cursor.

You can move the cursor to the next row, using the following function:

```
boolean Query.ResultSet.next(String ResultsetName)
```

The function returns false when the cursor moves after the last row (or when there is no row).

To get the value of a column in the row referenced by the cursor, pass the index of that column to the following function:

```
Object Query.ResultSet.get(String ResultsetName, int ZeroBasedColumnIndex)
```

The following example shows how you can get the value of column 1 in each row of the result set and simply display it on the console:

```
while(Query.ResultSet.next("rset")) {
    System.debugOut(Query.ResultSet.get("rset",1));
}
```

Where `"rset"` is the name of the result set.

Count of Records in Certain Result Sets

You can use the `getRowCountIfPossible()` function to get the count of records in a result set when using Query Functions.

```
Query.ResultSet.getRowCountIfPossible()
```

This function can be used only with snapshot queries that use joins and aggregations (order by and group by clauses). Only in such cases is the result set known. In other cases the query begins filtering and feeding results to the result set without knowing when the query will end.

Callback Rule Function Data Usage

The data provided to the callback rule function depends on which callback function you are using.

Execute with Callback Function

When you use the `Query.Statement.executeWithCallback()` function, the agent calls the specified callback rule function once for each row of data generated. The row of data is provided as an array of columns.

The callback rule function is called in the following circumstances:

- Once for each row of data generated by the query.
- At the end of a batch of rows (continuous queries only). A blank row with the `isBatchEnd` flag is sent.
- Once, when there are no more results, indicating the end of the results (snapshot queries) or that the statement was closed or the query deleted (continuous queries). See Statement Closing and Query Definition Deletion.

You can provide a closure object when executing the statement. The closure object is provided to each rule function call. It can contain anything useful in the execution context. For example, you can use an object array to accumulate each row of results returned in each callback rule function call.

Execute with Batch Callback Function

When you use the `Query.Statement.executeWithBatchCallback()` function, the agent calls the specified callback rule function once at the end of each batch of results. The data is provided as an array of all the rows in that batch.

The callback rule function is called in the following circumstances:

- At the end of a batch of rows generated by the query.
- Once, when there are no more results, indicating that the statement was closed or the query deleted. See Statement Closing and Query Definition Deletion.

As with the `Query.Statement.executeWithCallback()` function, you can provide a closure object when executing the statement.

The Callback Rule Function Required Signature

The callback function must have a signature with the parameter types provided in a specific order.

The following table lists the parameter types in the order of their usage:

Parameter	Notes
`String` *`id`*	Identifies the current execution. Uses the value of *`listenerName`*, which was provided when calling the `Query.Statement.executeWithCallback()` function. The ID enables you to identify rows of data belonging to different executions of the same query (or different queries).
`boolean` *`isBatchEnd`*	Used in the case of continuous queries only, and is useful only when the query text contains an `order by` clause (see Order by Clause). Only true at the end of a batch of rows of data generated by the query. In the case of continuous queries where no sorting is used, each row of data is a batch. See Example Showing Batching of Return Values (Continuous Queries).
`boolean` *`hasEnded`*	When true, signals the end of the execution.

`Object` *`row`*
 	
OR
`Object` *`rows`*
	For `Query.Statement.executeWithCallback()`: An array of columns representing one row of data generated by the query. Each column corresponds to an item in the projection (see Select Clause).
	
	For `Query.Statement.executeWithBatchCallback()`: an array of rows comprising one batch of results.

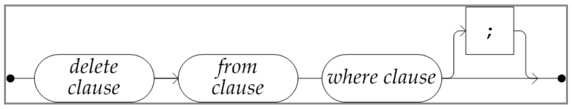
`Object` *`closure`*	Closure object provided when executing the `Query.Statement.executeWithCallback()` function, or `null`. The object provided depends on your needs. For example, it could be a

Parameter	Notes
	simple string, or it could be an array of objects used to add a row of data from each callback rule function.

The Delete Query

The delete query is typically used to delete the temporary concepts, which are no longer required.

Temporary concepts can be created in the query agent to hold rows of data returned by a query. Such data can then be transformed into an XML string and sent out of the system through a channel, or used to perform computations. When the temporary concepts are no longer needed, use the delete query to delete them.



You can create one concept (to act as a container), with an array of contained concepts to hold each row of results. In a rule function you can use the `Concept.serializeUsingDefaults()` function to create an XML string with all the results nested within the container. After you send the results out of the system, you can then use a delete query to remove the temporary concepts, which are no longer needed.

See Delete Clause for reference details.

Delete Query Limitations

Because of its limited context of use, this query has various limitations, listed next.

- The delete query does not use locking. Use the delete query to delete concepts created in the query agent only. The delete action does not go through an RTC. Do not delete concepts that are used in inference agents; doing so may cause issues such as data integrity issues and rule processing issues. Do not attempt to delete concepts that could be accessed at the same time in any other agent as results could be unpredictable.
- The delete query does not delete contained or referenced concepts. You must delete each contained concept individually.

- The delete query does not delete child concepts (inherited concepts). For example deleting Customer does not delete RedCustomer and BlueCustomer.
- Each statement deletes instances of one specified concept type, which is specified in the from clause. You cannot use more than one concept type in the from clause (that is, joins are not supported).
- Only use the delete query with concepts that use cache only mode. The delete query deletes concepts from the cache only. (Not from Rete network or backing store).
- Use the delete query only with write behind database write strategy (not cache-aside).
- The delete clause can be used only in snapshot queries.

Simple Snapshot Query Example

The snapshot query example code could be placed in a preprocessor rule function that receives an event called `requestEvent`. The example code includes all steps from creating to closing the query.

The example is simplified for clarity. In a real-world use case, the creation step could be performed in a startup rule function, and the output could be sent in an event to an inference agent or other destination.

```
Query.create("report853", "select agent_name, total_sales, zipcode from
/Concepts/Sales");
String id = requestEvent@extId;
String stmt = "S" + id;
String rset = "R" + id;
Query.Statement.open("report853", stmt);
Query.Statement.execute(stmt, rset);
while(Query.ResultSet.next(rset)) {
    String agent = Query.ResultSet.get(rset, 0);
    double sales = Query.ResultSet.get(rset, 1);
    String zip = Query.ResultSet.get(rset, 2);
    System.debugOut(rset + ": Agent " + agent
        + " sold $" + sales
        + " in " + zip
        + ".");
}
System.debugOut(rset + ": ========");
Query.ResultSet.close(rset);
Query.Statement.close(stmt);
Query.Close("report853");
```

The last three lines are provided for completeness. However, if the `Query.Close()` function is used, you would not need to include the `Query.ResultSet.close()` or `Query.Statement.close()` functions. See Statement Closing and Query Definition Deletion for details about these hierarchical relationships.

Sample Output

```
R123: Agent Mary Smith sold $15063.28 in 94304.
R123: Agent Robert Jones sold $14983.05 in 94304.
R123: ========
```

Simple Continuous Query Example

The continuous query example shows how a callback rule function is used to gather results generated by the query.

An example callback rule function is as follows:

```
MyRF(ID, isBatchEnd, hasEnded, row, closure)
if (hasEnded) {
    System.debugOut(ID + ": ========");
} else if (isBatchEnd) {
    System.debugOut(ID + ": --------");
} else {
    Object[] columns = row;
    String agent = columns[0];
    double sales = columns[1];
    String zip = columns[2];
    System.debugOut(id
        + ": Agent " + agent
        + " sold $" + sales
        + " in " + zip
        + ". " + closure);
}
```

Create the Query

```
Query.create("report853", "select agent_name, total_sales, zipcode from
/Concepts/Sales");
```

Open and Execute the Query Statement

```
String id = requestEvent@extId;
String stmt = "S" + id;
String clbk = "C" + id;
Query.Statement.open("report853", stmt);
Query.Statement.executeWithCallback(
stmt, clbk, "/MyRF", true, "@@@@");
```

Where `requestEvent` is an event, and `"/MyRF"` is the path to the rule function shown at the beginning of the section. The `true` parameter indicates that this is a continuous query.

Sample Output

In the following sample output , each row of data (generated when a relevant change occurs in the cache) is one batch, because the query does not involve ordering or aggregation. The last line in the sample indicates that the query has ended. For example, someone closed the statement (not shown in the code sample).

```
C123: Agent Mary Smith sold $15063.28 in 94304. @@@@
C123: --------
Time passes…
C123: Agent Robert Ng sold $14983.05 in 94304. @@@@
C123: --------
Time passes…
C123: Agent Jose Ortiz sold $16244.78 in 94304. @@@@
C123: --------
C123: ========
```

Function Calls

The following example shows the parameter values for each function call.

As a reminder: the first Boolean indicates whether the batch has ended or not; the second Boolean indicates whether the execution has ended or not.

- *Mary Smith makes a sale.*

  ```
  MyRF(clbk, false, false, ["Mary Smith", 15063.28, 94304], "@@@@")
  MyRF(clbk, true, false, null, "@@@@")
  ```

- *Time passes… Robert Ng makes a sale.*

  ```
  MyRF(clbk, false, false, ["Robert Ng", 14983.05, 94304], "@@@@")
  MyRF(clbk, true, false, null, "@@@@")
  ```

- *Time passes… Jose Ortiz makes a sale.*

  ```
  MyRF(clbk, false, false, ["Jose Ortiz", 16244.78, 94304], "@@@@")
  MyRF(clbk, true, false, null, "@@@@")
  MyRF(clbk, true, true, null, "@@@@")
  ```

Example Showing Batching of Return Values (Continuous Queries)

The example is the same as the simple continuous query example, with the addition of an `order by` clause in the query text, to show batching behavior. Only the output and function calls differ.

Create the Query

```
Query.create("report853", "select agent_name, total_sales, zipcode from
/Concepts/Sales order by agent_name");
```

Sample Output

- *Mary Smith makes a sale.*

  ```
  C123: Agent Mary Smith sold $15063.28 in 94304. @@@@
  C123: --------
  ```

- *Time passes… Robert Ng makes a sale.*

```
C123: Agent Mary Smith sold $15063.28 in 94304. @@@@
C123: Agent Robert Ng sold $14983.05 in 94304. @@@@
C123: --------
```

- *Time passes… Jose Ortiz makes a sale.*

```
C123: Agent Jose Ortiz sold $16244.78 in 94304. @@@@
C123: Agent Mary Smith sold $15063.28 in 94304. @@@@
C123: Agent Robert Ng sold $14983.05 in 94304. @@@@
C123: --------
C123: ========
```

Function Calls

The following example shows the parameter values for each function call.

As a reminder: the first Boolean indicates whether the batch has ended or not; the second Boolean indicates whether the execution has ended or not.

```
Mary Smith makes a sale.
MyRF(clbk, false, false, ["Mary Smith", 15063.28, 94304], "@@@@")
MyRF(clbk, true, false, null, "@@@@")
Time passes… Robert Ng makes a sale.
MyRF(clbk, false, false, ["Mary Smith", 15063.28, 94304], "@@@@")
MyRF(clbk, false, false, ["Robert Ng", 14983.05, 94304], "@@@@")
MyRF(clbk, true, false, null, "@@@@")
Time passes… Jose Ortiz makes a sale.
MyRF(clbk, false, false, ["Jose Ortiz", 16244.78, 94304], "@@@@")
MyRF(clbk, false, false, ["Mary Smith", 15063.28, 94304], "@@@@")
MyRF(clbk, false, false, ["Robert Ng", 14983.05, 94304], "@@@@")
MyRF(clbk, true, false, null, "@@@@")
MyRF(clbk, true, true, null, "@@@@")
```

Note: The `Query.Statement.executeWithBatch Callback()` function works in a similar way, except that the callback rule function is called once for each batch, and the results are sent as an array of rows.

Bind Variables in Query Text

Query definitions can use literal values for entity attributes in query text, or they can use *bind variables* whose values are provided at runtime.

In the `Query.create()` function, use a dollar sign ($) to indicate a bind variable in the query text. (See `$min` in the following example.)

The values for all bind variables must be supplied to a statement when it executes. Set the value of a bind variable, using the `Query.Statement.setVar()` function, from the CEP Query Functions catalog, as shown next.

```
Query.Statement.setVar(String StatementName, String BindVariableName, Object
value);
```

When you use the `Query.Statement.setVar()` function, functions must be called in the following order:

- `Query.Statement.open()`
- `Query.Statement.setVar()`
- `Query.Statement.execute()` OR `Query.Statement.executeWithCallback()` OR `Query.Statement.executeWithBatchCallback()`

All functions must reference the same query statement name.

Note: Bind variables cannot be used with the `like` operator.

Bind variables cannot be used with the `from` clause. However see Using Strings (Instead of Variables) in From Clauses for an alternative.

The following example shows how a bind variable in a query definition is set as the value of an event property by the `Query.Statement.setVar()` function. The value could be defined as a literal value as desired, or in some other way, depending on context and need.

For example:

```
Query.create("report927", "select agent_name, total_sales, zipcode from
/Concepts/Sales where total_sales >= $min");
Query.Statement.open("report927", S_Id);
Query.Statement.setVar(S_Id, "min", evt.min_total_sales);
Query.Statement.execute(S_Id, "rset");
```

Where `evt.min_total_sales` is an event property of a numeric type.

Clearing Bind Variables

You can use `Query.Statement.clearVars()` to clear all bind variable values associated with the named statement.

Datatype Assignment to a Bind Variable

In queries, the type of a bind variable is enforced by its surrounding expression. In the query, use the specific expressions to assign the desired type to the bind variable.

Datatype	Expression
`int long`	(+ 0)
`double`	(+ .0)
`String`	(||)
`Boolean`	(or false)
`DateTime`	**Note** : `DateTime` is not supported so use the following instead. Pass a `long` instead of a `DateTime` and use: `/#Datetime/parseLong()` Pound or hash (#) is the escape character. See Keywords and Other Reserved Words.

Collocated Inference Agents and Dynamic Query Agent Sessions

Depending on the need, it can be useful to deploy an inference agent in the same processing unit (node) as the query agent. Another way to integrate query and inference functionality is to dynamically start a collocated query agent session from an inference agent.

Query agent and inference agent functionality is complementary. You can work with these two agent types in different ways.

Collocated Query and Inference Agents

The inference agent can process and enrich the event data, create concepts, modify concepts, and so on. The query agent can send events to the inference agent using a local channel.

Inference agent rules can process the data and send an event to the query agent (where another query is listening for that event), or send the event out of the node.

Tip: Modifying concepts retrieved from a query agent

The inference agent can modify concepts retrieved from a query agent using the following functions. Use the appropriate function for the type of cluster.

As with all actions that modify concepts, ensure that correct locking is used before executing the query.

The rule `ExecuteSelectInQueryAgent`, in the following example project demonstrates this technique:

```
BE_HOME/examples/event_stream_
processing/CollocatedInferenceAndQuery
```

The collocated inference agent can use Cache OM or In Memory OM. Performance of In Memory OM systems is very high. However, the processing potential of Cache OM is greater because the inference agent has access to all the cache data as well as the data in memory. Choose the option that fits your needs.

Dynamic Query Agent Sessions

You can dynamically start a collocated query agent session from an inference agent and make queries, using startup and preprocessor rule functions.

executeInDynamicQuerySession

The following function is used in the inference agent's preprocessor. It enables you to execute a query in the dynamic query agent session (without the need for collocated query agent),, and use the results:

```
Query.Util.executeInDynamicQuerySession(String sqlString, Object
mapOfParameters, boolean reuse, long timeout, boolean isNativeQueryRawResults);
```

The is an example showing how you might use this rule function:

```
void rulefunction Inference.RuleFunctions.MyPreProcessor {
       attribute {
   validity = ACTION;
}
scope {
   Events.AccountOperations request;
}
body {
   String queryString = "select acc" +
      " from /Concepts/Account as acc" +
      " where acc.Status = \"Normal\"";
   Object resultList = Query.Util.executeInDynamicQuerySession
(queryString, null, true);
   int size = Query.Util.sizeOfList(resultList);
   System.debugOut("Result list has " + size + " items");

   for(int i = 0;  i < size; i = i + 1){
      Object row = Query.Util.removeFromList(resultList, 0);
      Concepts.Account acc = row;
      System.debugOut("  Result row: " + acc);
      }
   }
}
```

executeInQuerySession

The following rule functions executes the SQL string synchronously in the collocated query and returns the results:

```
Query.Util.executeInQuerySession(String querySessionName, String sqlString,
Object mapOfParameters, boolean reuse, long timeout, boolean isNativeQueryRawResults)
```

invokeFunctionInQuerySession

The following rule function invokes a rule function in another query session/agent and needs a collocated query agent. The name of the query session/agent, which is deployed in the same processing unit, is provided as a parameter:

```
Query.Util.invokeFunctionInQuerySession(String querySessionName, String
queryRuleFunctionUri, Object[] parameters)
```

Design Optimization

You can implement few basic strategies for optimizing the design when working with queries.

Reuse Existing Queries and Statements Whenever Possible

Creating a new query is an "expensive" operation. If possible, create the queries ahead of time (in a startup function), then keep reusing those existing query definitions in new statements.

(See Lifecycle of a Query—Use of Query Functions for more details)

For example, you could create a query in a startup function. That query may use bind variables, for more flexibility (see Bind Variables in Query Text. Then, in a preprocessor rule function, you could create a new statement using that query, set values in the statement for all the bind variables of the query using the data in the event, and execute the statement. As a result, the query would be customized and executed for each event reaching the preprocessor.

Depending on your situation, it might be possible to create a single statement, and keep reusing that same statement, executing it multiple times.

Note: The function that creates a new query requires that you provide a globally unique name. You can later refer to that query using its name. The function that opens a new statement requires you to provide an existing query name, and a new globally unique statement name. You can later refer to that statement using its name.

Improve Performance by Pre-fetching Objects (Cache Queries)

When a query executes, objects are fetched from the cache as needed for query processing. Objects are placed in the local query cache for use by the query. You can improve performance by prefetching the objects from the backing store. See TIBCO BusinessEvents Developer Guide for more information.

Optimize WHERE Clause Expressions

In the where clause, ensure that the most selective operators appear first.

For example, suppose you have a query like this:

```
select * from /Customer c where c.location = "CA" and c.age > 95
```

If the number of customers in the dataset whose age is greater than 95 is very small compared to the number of people living in California, then `age > 95` is a more selective operator than `location = "CA"`.

Rewrite the query as follows:

```
select * from /Customer c where c.age > 95 and c.location = "CA"
```

The more selective operator now appears first, so the query is more efficient.

Use Indexing for More Efficient Cache Queries

You can index concept and event properties to make searches faster. Use of indexing avoids the need to deserialize the entire object before running the filter—indexing is of greatest value with large objects that have many properties.

You can index more than one of an entity type’s properties. When indexing is used, memory use will also increase.

The efficiency of a filter is increased when you index the properties that are used in the most selective operators (see Optimize WHERE Clause Expressions for details).

The cache provider, however, may or may not use the index, depending on how complex the filter is. Complex `where` clauses containing function calls and joins will not be optimized. Only simple filters, such as `age > 60`, or `name in (“a”, “b”, “c”)`, are re-written to use indexes.

For example, indexing the `age` property for the Customer concept would make the following search more efficient:

```
select * from /Customer c where c.age > 95
```

However, indexing would not work for a more complex expression such as the following:

```
select * from /Customer c where /MyFunctions/roundup(c.age) > 95
```

You can create the indexes using index catalog function or domain object override.

To Enable Query Optimization

Only query agents enabled for query optimization use this feature. In the project CDD file, add the following property to the query agent properties:

```
be.agent.query.enable.filter.optimizer=true
```

Only agents with this property set to true will attempt to use indexing that you define.

Creating an Index Using a Domain Object Override Setting

You can create an unordered index in the project's Cluster Deployment Descriptor (CDD) using a domain object setting.

Procedure

1. Open the project CDD in TIBCO BusinessEvents Studio and go to **Cluster tab > Object Management > Domain Objects > Overrides**.
2. Edit or create an override entry as needed for the desired entity or entities
3. In the override entry's **Properties Metadata** section, select the **Present in Index** checkbox for the property you want to index.
4. Save the CDD.

Use Filtering for Efficient Joins (Cache Queries)

When performing a join between two or more entities in a query, put the most selective operators before the actual join expression. This makes the join more efficient.

See Optimize WHERE Clause Expressions for more information.

Joins that test for equality (equivalent joins), that is, joins between two entities that use the equals operator (=), perform better than joins that test for inequality (non-equivalent joins), that is, joins using operators such as greater than, less than, and so on. Comparison operators supported for filtering are as follows:

```
>, >=, <, <=, ==, !=, In, Between, And, Or, Not, Like
```

Example

In the following example, the two entities `Trade` and `StockTick` are joined by matching their respective `securityId` and `symbol`. But the query also places the restriction that only TIBX trades and stock ticks are of interest, and only if the trade's settlement status is `FULLY_SETTLED`. These filters appear before the actual join expression, which is more efficient than if they were placed after the join (`t.securityId = tick.symbol`).

```
select tick.symbol,
    sum(tick.price) * 1000 / count(*),
    avg(tick.volume),
    count(*),
    t.counterpartyId
from /Trade t, /StockTick {policy: maintain last 1 sliding where symbol
= "TIBX"} tick
where t.securityId = "TIBX"
    and t.settlestatus = "FULLY_SETTLED"
    and t.securityId = tick.symbol
group by tick.symbol, t.counterpartyId
having count(*) > 2;
```

Effect of the Cache on Continuous Queries

Cache queries are run against the object cache, not against the contents of working memory. Ensure that the objects you want to query are in the cache when the query is run, and are not, for example, removed from the cache before the query executes.

For example, while a continuous query is running, multiple batches of results may be received. At the time it is first received, a batch of continuous query results contains items that are in the cache. If you wait for another batch, some (or all) of the objects in the old results may have been evicted from the cache.

Effect of Time on Cache Queries

While running continuous queries, errors can occur if entity creation and deletion happen in rapid succession.

Example

Consider a continuous query that is monitoring entities of type `/OrderEvents`. Suppose that `OrderEvents` entities are created, asserted, and consumed, at a fast rate. When an `OrderEvent` entity is asserted, it is also added to the cache. When it is consumed, an `OrderEvent` entity is deleted from the cache. The continuous query receives the creation notification and the deletion notification one after the other.

If there is a long enough delay between the creation and deletion actions and the moment a query agent attempts to process the related notifications, the agent will try to retrieve

`OrderEvent` entities that have already been removed from the cache, resulting in runtime errors.

This situation may occur when, for example, a very quick succession of notifications is sent, or the network traffic suffers delays, and so on.

Query Agent Local Cache

The query agent retains the most recently processed entities in a local cache to avoid frequent network lookups. But in the earlier example, the `OrderEvent` is deleted from the cache even before the create-notification is processed by the query, so the `Orderevent` cannot be copied into the query agent's private cache.

Keep such situations in mind as you design your queries.

Continuous Queries

A continuous query returns results throughout its lifetime, as changes occur. When nothing changes, the query waits.

Overview of Continuous Queries

Continuous queries, when used in a query agent deployed in a TIBCO BusinessEvents® cluster, listen to and process the data stream of notifications sent from the cache. Notifications are sent when entities are added to, modified, or deleted from the cache.

Unlike snapshot queries, continuous queries do not examine the cached entities themselves. Entities that were created before a query start are not visible to it—unless they are modified while the query is running.

When used in a query agent deployed stand-alone to perform event stream processing, continuous queries listen to and process the data stream for a specified event.

Enabling Continuous Query

Only query agents enabled for continuous query use a continuous query. In the project CDD file, add the following property to the query agent properties:

```
be.agent.query.continuous.allow=true
```

By default this property is set to false. Only agents with this property set to `true` attempts to use continuous query.

Tip: A continuous query must be run using the `Query.Statement.executeWithCallback()` function. Snapshot queries can also use this function. However, when you set the argument `IsContinuous` to true, the query runs as a continuous query. See Overview of Continuous Queries for more details.

Optimizing Continuous Query

To optimize continuous queries containing aggregate functions, such as sum(), add the property to the project CDD file to the query agent properties:

```
be.engine.query.optimize.aggregate=true
```

The default value is `false`.

Setting this property to `true` maintains a single value in aggregate functions instead of accumulating all the previous values and thus optimizes memory consumption.

Note: The property is applicable only to BQL continuous queries.

Running a Continuous Query

For continuous queries, use the `Query.Statement.executeWithCallback()` function (and a variant of this function called `Query.Statement.executeWithBatchCallback()`) with the `IsContinuous` argument set to `true`.

See Callback Rule Function Data Usage and See Two Types of Queries—Snapshot and Continuous for more information.

Ending a Continuous Query

A continuous query only ends when one of the following occurs:

- You explicitly stop it.
- Its query statement is closed.
- Its query definition is deleted.
- The query agent engine stops.

Query Windows

Continuous queries use windows. A *window* is a boundary for analyzing data streams. It is a container in which events and concepts are held and processed by the query. The events or entities enter and leave the window as determined by the window type and how it is configured.

One query can contain multiple windows, and the contents of these windows can be analyzed and compared.

Windows can be divided into two basic types, explicit and implicit.

Explicit windows (sliding, tumbling, and time windows) define the window boundary, that is, a condition that limits the lifecycle of the entities in the window.

With implicit windows, the lifetime of the entities themselves control the lifecycle of the entities in the implicit window. Implicit windows process changes, additions, and deletions affecting the specified entities until the query ends.

TIBCO BusinessEvents supports the following two types of Continuous Queries:

- Business Query Language (BQL) Continuous Query
- Ignite Native Continuous Query

Business Query Language (BQL) Continuous Query

A Business Query Language (BQL) continuous query operates persistently over live event streams. It maintains its state instead of returning a one-time result. It emits incremental updates as events arrive, change, or expire within the stream.

You use continuous queries for real-time analytics, event correlation, and alerting within BusinessEvents environments. This capability allows for immediate responses to evolving data conditions. You use continuous queries for real-time analytics, event correlation, and alerting within BusinessEvents environments. This capability allows for immediate responses to evolving data conditions.

Type of Windows

See Working With Sliding Tumbling and Time Windows for content that applies to all these types of explicit windows.

Implicit Window

Has no `policy` clause. Instead a group by clause in the select statement of a continuous query determines that the query is using an implicit window. See Working With Implicit Windows.

Sliding Window

The `policy` clause specifies a queue size, into which entities flow. When the queue is full and a new entity arrives, the oldest entity in the queue is removed from the window (FIFO). See Sliding Window Examples (Cache Queries).

Tumbling Window

The `policy` clause specifies a queue size as a certain number of entities, and empties each time the maximum size is exceeded. Emptying the window completes one cycle. The lifetime of an entity in the window, therefore, is one cycle. See Tumbling Window Examples (Cache Queries)

Time Window

The `policy` clause specifies a time period during which entities remain in the window. See Time Window Examples (Cache Queries).

Working With Implicit Windows

Implicit windows are created when the continuous query does not have an explicit policy (window) clause.

The lifecycle of an entity within an implicit window is affected by the life cycle of that entity in the cache:

- When an entity in the scope of the query is added to the cache or is updated in the cache, it is automatically added to the window.
- When an entity is deleted from the cache, it automatically exits the window.

Deletion of entities may cause an update of the query output, depending on the query text.

Example 1

```
select count(*) from /EventA evt group by 1;
```

This example uses a dummy group, required because aggregate functions, `count(*)` in this case, require a `group by` clause to work on all the rows. See Using a Dummy Group Expression for Aggregation for more details.

Suppose that for the first 10 minutes after the query statement is executed, 100 events are created in quick succession. Every time the query receives a new event notification, the count goes up progressively until it stabilizes at 100.

Suppose that thirty minutes later, 50 of those 100 events are consumed by a rule or expire because of their time to live (TTL) settings. The events are deleted from the cache. The query receives deletion notifications and the query output, `count(*)`, changes until it drops and stabilizes at 50.

Example 2

The following query joins Department and Student entities using the department name. It then keeps a count and an average of age of students per department.

```
select d.name, count(*), avg(s.age)
  from /Department d, /Student s
  where d.name = s.deptName
  group by d.name;
```

The following query keeps count of the number of students per department. Every time a student enrolls or leaves, the count changes and the query produces the entire list sorted on the count.

```
select s.deptName, count(*)
  from /Student s
  group by s.deptName
  order by count(*);
```

Working With Sliding Tumbling and Time Windows

Sliding, tumbling, and time windows are *explicit* windows. In an explicit window, the lifecycle of an entity in a window is determined either by a specified duration of the entity in the window, or by setting a maximum number of entities that can be in the window at any time.

The *stream policy* (also known as a *window policy*) determines what kind of lifecycle and what kind of window to use: a time window, sliding window, or tumbling window.

You can filter entities entering the query using a `where` in the stream policy. You can also do aggregations within the window using a `by` clause. See Stream Policy.

Use Explicit Windows for Events and not Concepts

Concepts are mutable. Events are immutable after they are asserted. The mutability of concepts makes them generally unsuitable for cache queries that use sliding, tumbling, or time windows, as explained next.

Entities enter a sliding, tumbling, or time window when they are added to the cache and they remain in the window according to criteria such as duration in the window or number of items in the window. This characteristic enables you to gather statistical information such as how many transactions were processed in an hour.

Deleting an entity from the cache does not cause it to be removed from such a window. (This behavior is different from the behavior of implicit windows.)

When a concept is modified, internal actions delete the old value from the cache and add the new one. Sliding, tumbling, and time windows ignore the deletion, but recognize the addition. Therefore the old and the new value both appear in the window, leading to unexpected results.

Events are immutable (after assertion), so this issue does not arise in the case of events.

Note: If you know that in your environment concepts will not be modified, then you can safely use concepts in sliding, tumbling, and time windows.

Explicit Window Example (Cache Query)

In SQL, the order in which the clauses are presented in a query string is not the order in which they are processed.

For example, following is a fairly complex example formatted to make each clause clear:

```
select
    tick.symbol, trade.counterpartyId, avg(tick.volume), count(*),
from
    /Trade trade,
    /StockTick
      {policy: maintain last 5 sliding
         where symbol = "TIBX" or symbol = "GOOG"
         by symbol}
    tick
where
    trade.settlestatus = "FULLY_SETTLED"
    and
    trade.securityId = tick.symbol
group by
    tick.symbol,
    trade.counterpartyId
having
    count(*) > 2;
```

In fact, the clauses are processed in the following order, as shown in How a Query String is Processed:

`from` (including stream clause)

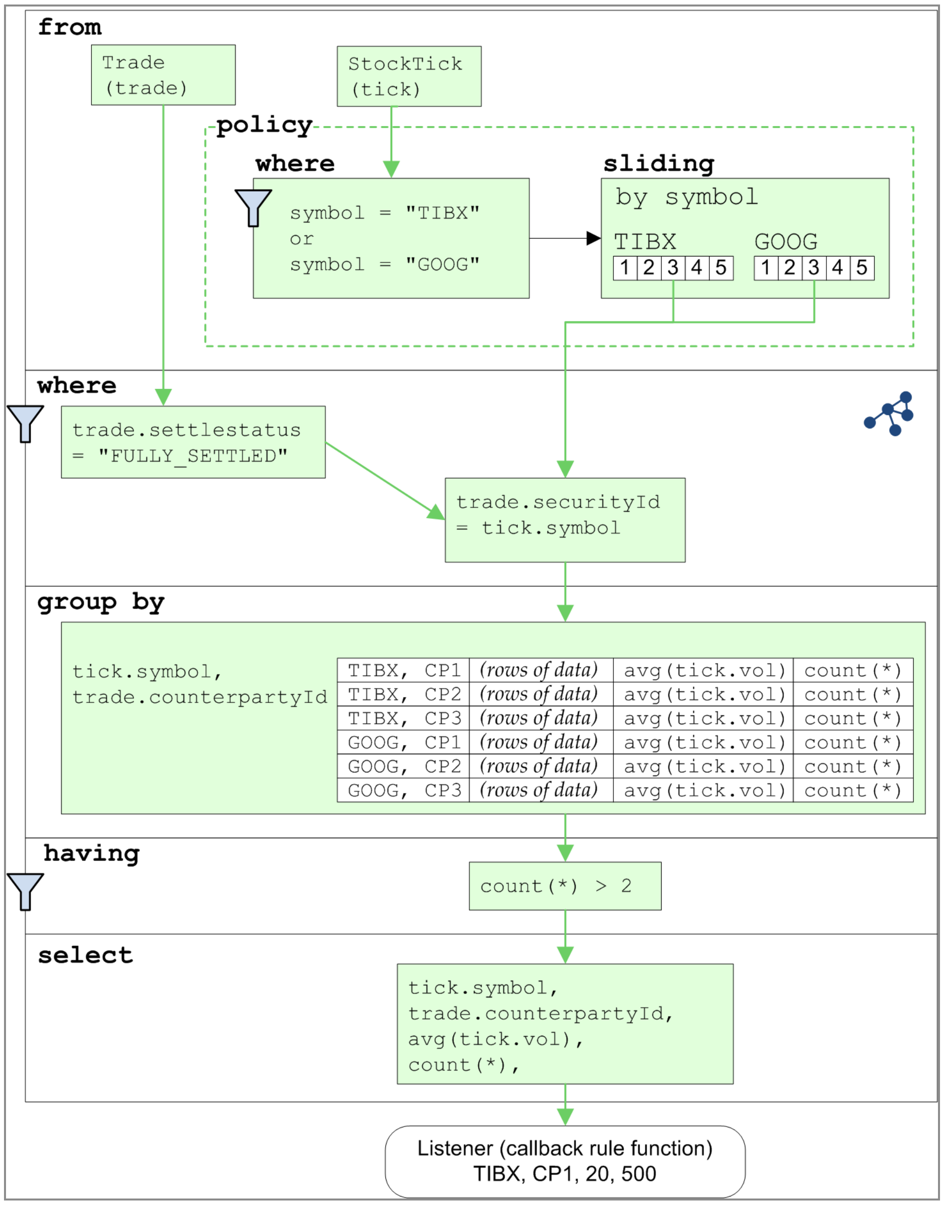
```
where
```

`group by` (including `having`)

```
select
```

Of special interest is how the where clause in the stream policy operates with the main where clause; and how the stream policy can create multiple windows.

Figure 1: How a Query String is Processed


from
Trade (trade)
StockTick (tick)
policy
where
symbol = "TIBX" or symbol = "GOOG"
sliding
by symbol
TIBX
GOOG
1 2 3 4 5
1 2 3 4 5
where
trade.settlestatus = "FULLY_SETTLED"
trade.securityId = tick.symbol
group by
tick.symbol, trade.counterpartyId
TIBX, CP1 (rows of data) avg(tick.vol) count(*)
TIBX, CP2 (rows of data) avg(tick.vol) count(*)
TIBX, CP3 (rows of data) avg(tick.vol) count(*)
GOOG, CP1 (rows of data) avg(tick.vol) count(*)
GOOG, CP2 (rows of data) avg(tick.vol) count(*)
GOOG, CP3 (rows of data) avg(tick.vol) count(*)
having
count(*) > 2
select
tick.symbol, trade.counterpartyId, avg(tick.vol), count(*),
Listener (callback rule function)
TIBX, CP1, 20, 500

Sliding Window Examples (Cache Queries)

A sliding window policy maintains a queue of a specified size, into which entities flow. When the queue is full and a new entity arrives, the oldest entity in the queue is removed from the window (FIFO).

The following query has a sliding window over Car events. It retains the last 5 car events that have passed through the query. Every time a new event arrives, the query produces output that matches the latest event that arrived.

`select car from /CarEvent {policy: maintain last 5 sliding} car;`

The following query is similar to the previous one except for the `emit` clause. The query maintains a sliding window over the last 5 events. However, instead of echoing the event that just arrived, it emits the oldest event in the window that got displaced when the new event arrived. The query starts producing output only after the window has filled up and reached its full size.

```
select car from /CarEvent {policy: maintain last 5 sliding; emit: dead}
car;
```

The following query maintains a count of the number of events in the sliding window. Every time an event arrives or drops out of the window (or both), the query produces output. Note that when the query starts and events start arriving, the count progresses towards the maximum window size (25). Once it reaches 25, the number stops changing, because the window will always have a count of 25 from then on.

```
select count(*) from /CarEvent {policy: maintain last 25 sliding} car
group by 1;
```

The following query performs a rolling average and a count over a sliding window of size 30. The window has a pre-filter clause that only consumes `StockTick` events whose symbols match "ABCD" or "WXYZ." All other symbol types are dropped and prevented from entering the window. Also, the `by` clause indicates that a sliding window must be maintained per symbol. The `group by` clause matches the `by` clause because both of them specify grouping on `symbol`. As result, the query correctly maintains a rolling average and count over the last 30 events, per symbol.

```
select stock.symbol, avg(stock.price), count(*)
  from /StockTick {policy: maintain last 30 sliding
    where symbol = "ABCD" or symbol = "WXYZ"
    by symbol} stock
  group by stock.symbol;
```

The `by` and `group by` clauses in the following query are used differently here from the way they are used in the prior example. This query maintains a sliding window of size 30 based on price. However, the `group by` clause is on the symbol. So, the windowing based on price is of little use here.

```
select stock.symbol, avg(stock.price), count(*)
  from /StockTick {policy: maintain last 30 sliding
    where symbol = "ABCD" or symbol = "WXYZ"
    by price} stock
  group by stock.symbol;
```

Tumbling Window Examples (Cache Queries)

A tumbling window a specified queue size, specified as a certain number of entities, and empties each time the maximum size is exceeded. Emptying the window completes one cycle. The lifetime of an entity in the window, therefore, is one cycle.

The following query maintains a count over a tumbling window of events. Every time events arrive, the query picks up a maximum of 500 events, passes them through the query processing stages, in this case a counter, and produces the count as the result. Because this is a tumbling window, all those 500 or less events expire immediately and so the query runs once again and flushes all the events from the window. Now, the count drops to 0 and the query produces "0" as the count.

Tip: The following example uses a constant (in this case `group by ""`) to create a dummy group. The next example uses a different constant. See Using a Dummy Group Expression for Aggregation for more details.

```
select count(*) from /BurgerSoldEvent {policy: maintain last 500
tumbling} burger group by "";
```

The following query is not very useful because it forgets how many events have been processed every time the window "tumbles." One way to solve this problem is to store all the events in a very large window, forever—but this is impractical. Another way is shown next.

```
select count(*) from /BurgerSoldEvent {policy: maintain last 500
tumbling; emit: new} burger group by 1;
```

Using Emit New to Create a Counter

You can define a tumbling window which retains events for just one cycle and then keep a counter that remains pinned even if the window appears to disappear after it empties itself.

To create such a counter, use the `emit: new` clause. This clause indicates to the query that it should only record events entering the window and not those exiting it. So, in this case the count keeps increasing as new events arrive and it never decreases.

Time Window Examples (Cache Queries)

Time windows use a stream policy that specifies how long an entity remains in the window.

For information, see Stream Policy.

The expiry time is calculated from a start time. You can use an event or concept's timestamp property to define the start time. Otherwise, the time the event or concept entered the window is used as the default start time.

Note: Events whose expiry time is exceeded when they arrive in the window

A query that uses a time window processes events that have already expired when they enter the window. The expired events appear in the window for one cycle and then leave the window in the next cycle.

The following query holds `PizzaOrderEvents` for 45 minutes after the `OrderTime` in a time window.

```
select coldpizza from /PizzaOrderEvent {policy: maintain last 45 minutes
using coldpizza.OrderTime; emit: dead} coldpizza;
```

When the using clause is omitted, the window uses the default timestamp that is associated with the event when it enters the query.

Delaying Output with an Emit Dead Clause

Without an `emit: dead` clause, the query would produce the event as its output as soon as it arrives. But because of the `emit: dead` clause, it is delayed for the amount of time specified in the window.

The following query maintains the count on a 2-minute time window over network ping events. Whenever the number of pings in the last two minutes goes above 120, it produces output that can be treated as an attack.

```
select count(*) from /NetworkPing {policy: maintain last 2 minutes}
dosattack group by 1 having count(*) > 120;
```

Ignite Native Continuous Query

Ignite Native Continuous Query provides a low-latency, window-aware streaming events directly from the Ignite data grid. This section describes the window semantics and EMIT behavior model for Native Continuous Queries in the Query module. It also provides examples and troubleshooting information.

To enable the Native Continuous Query execution, see the following table.

Property	Type	Description
`be.ignite.cluster.continuous.query.enabled`	Boolean	Set this property to `true` to enable the Native Continuous Query execution. The default value is `true`.

Best Practices

- Always specify an ordered time column for windowed queries.
- Prefer explicit column selection over wildcards.
- Use `DISTINCT` sparingly in high-throughput environments.
- Choose Tumbling Windows for batch processing workflows.
- Close the statements when no longer needed to release memory.

Window Types and Semantics

Native continuous queries support the following four window types:

- Sliding Window
- Tumbling Window
- Time Window
- Implicit Window

For an Explicit Query, the following are window semantics:

```
WINDOW <Type>(<duration|count>, <slideDuration|slideCount>) WHERE <Window
expression> [ON <column>] [EMIT: NEW | EMIT: DEAD]
```

- `WINDOW <Type>(<duration|count>, <slideDuration|slideCount>):`

  This defines the time or count-based window and the sliding mechanism (duration or count).

  - `<Type>`: The type of window (example, TUMBLING, SLIDING, or TIME).
  - `<duration|count>`: The duration or count for the window.
  - `<slideDuration|slideCount>`: The sliding duration or count for the window. It is applicable for Window Type=SLIDING (*Optional*).

- `WHERE <Window expression>` (*Optional*): This is an optional condition for filtering events inside the window based on the window expression.
- `ON <column>` (*Optional*): This specifies the column on which the windowing operation is applied.
- `[EMIT: NEW | EMIT: DEAD]` (*Optional*):
  `EMIT: NEW`: Emits new events as they arrive or after they fall into the window.
  `EMIT: DEAD`: Emits events that have expired or are no longer in the window.

Note:

- Query having date type properties supports ISO-8601 UTC-based standard formats. For example,
    - `2025-12-12T10:15:30Z`
    - `2025-12-12T10:15:30+02:00`
    - `2025-12-12T10:15:30-05:00`
- The window time unit supports the following values:
    - `s` for second
    - `m` for minute
    - `h` for hour
- Implicit window does not support DateTime values in `epoch`.

Sliding Window

A sliding window continuously advances, retaining the last `N` events or `N` time units (or at a configured slide interval).

Syntax

```
WINDOW SLIDING(<duration|count>,<slideDuration|slideCount>) ON <column>
[EMIT:NEW|EMIT:DEAD]
```

Characteristics

- Supports overlapping windows.
- Slide interval defaults to window size, if omitted.
- Time-based or count-based depending on units.
- Window size and window slide should be of the same unit type.

Use Cases

- Rolling averages (moving KPIs)
- Frequent refresh of anomaly detection metrics

EMIT Behavior

- **EMIT:NEW** publishes entries entering the window.
- **EMIT:DEAD** publishes entries leaving the window.

Example

```
native-query: SELECT deviceId, AVG(temp) FROM SensorEvent WINDOW SLIDING
(15m,2m) ON eventTime EMIT:NEW GROUP BY deviceId;
```

Tumbling Window

A tumbling window is a non-overlapping, fixed-size window, where each event belongs to exactly one window.

Syntax

```
WINDOW TUMBLING(<duration|count>) ON <column> [EMIT:NEW|EMIT:DEAD]
```

Use Cases

- Periodic summarization
- Batch Processing
- Roll-up Statistics

Example

```
native-query: SELECT COUNT(*) FROM TradeEvent WINDOW TUMBLING(1m) ON
tradeTime EMIT:DEAD;
```

Time Window

A time window is a time-range window whose definition is based exclusively on its duration.

Syntax

```
WINDOW TIME(<duration>) ON <column> [EMIT:NEW|EMIT:DEAD]
```

Use Cases

- Monitoring the most recent time horizon.
- Threshold or alert detection.

Example

```
native-query: SELECT symbol, SUM(volume) FROM TradeEvent WINDOW TIME
(30m) ON tradeTime EMIT:NEW GROUP BY symbol;
```

Implicit Window

An implicit window applies no explicit window definition. The stream passes through without accumulation. The native query must be provided in the standard TIBCO BusinessEvents (BE) native SQL query format, similar to non-continuous queries.

Syntax

```
<Standard native query>
```

Use Cases

- Simple filtering.
- Fast projection of incoming rows.

Example

```
native-query: SELECT id, status FROM AlertEvent
```

Extended Properties

Configuration Property	Type	Description
`be.ignite.query.dedupe.emitOnChange`	Boolean	This setting controls whether deduplication is enabled for implicit continuous queries, which affects when tuple signatures are emitted. When set to `true`, Deduplication is enabled. Only new or changed tuple signatures are emitted. When set to `false`, Deduplication is disabled. All tuples are emitted, regardless of duplication. The default value is `true`.
`be.ignite.query.dedupe.maxEntries`	Integer	This property sets the maximum number of tuple signatures held in the Least Recently Used (LRU) cache. This limit detects and prevents the re-emission of

Configuration Property	Type	Description
		duplicate tuples and also prevents the unbounded memory consumption. Default value is `20000`. **Larger Value**: It tracks more history, leading to higher deduplication accuracy. **Smaller Value**: It remembers fewer signatures. Duplicates may be re-emitted once their older signatures are evicted from the cache.

EMIT Options

EMIT options control whether entering or exiting tuples are published. You have to define two queries to model entry behavior and exit behavior separately.

EMIT Behavior

- **EMIT:NEW** publishes rows that are entering the window or being updated.
- **EMIT:DEAD** publishes rows that are leaving or expiring from the window.

Common Errors

For an Ignite Native Continuous Query, the following are the common errors:

- Already started: The `execute` method is called more than once on the same statement.
- Invalid policy: Native Continuous Queries does not support a snapshot-only policy.

- Missing rule function: Invalid callback URI.
- Unknown alias: Snapshot configuration error.
- Out-of-memory risk: Large batch window with large datasets.

Strict Column Validation

The system property for strict column validation dictates whether the parser enforces strict validation of column references in Continuous Native Queries. Column validation is performed during metadata analysis, after query parsing but prior to registration or execution.

Property	Type	Description
`be.native.query.strictColumnValidation`	Boolean	Set this property to `true` to enforce strict column validation of column references in a native continuous query. The default value is `false`.

When strict column validation is enabled:

- All column references appearing in the query must match the known schema properties for the target cache.
- Strict mode reports a fatal error if the set of schema property names is empty (for example, when schema resolution fails).
- The parser raises a `QueryException` for unknown column references.
- The system columns, including normalized forms, such as `_id`, `_extid`, `_typeid` are excluded from unknown-column checks.

Limitations of an Ignite Native Continuous Query

The Native Continuous Query engine has a defined set of functional constraints to ensure deterministic behavior, predictable window processing semantics, and consistent integration with the underlying EventStream.

All implicit and explicit Native Continuous Queries have the following limitations:

- **Subqueries Not Supported**

  Subqueries are not supported in the current Native Continuous Query implementation.

  Unsupported patterns include, scalar subqueries, inline subqueries in `FROM`, nested `SELECT` statements, `IN (SELECT ...)` constructs, and correlated subqueries.

- **Join Support Limited to Self-Joins**

  Continuous queries support only self-joins on the same stream or table.

- **Explicit Queries Applicable Only to Event Streams**

  Explicit continuous queries using: `WINDOW SLIDING(...)`, `WINDOW TUMBLING(...)`, `WINDOW TIME(...)` are suitable only with run over event types and not with concept types. No error is encountered but results might be inconsistent.

- **Additional Windowing and Query Restrictions**

  All Continuous Query Window definitions have the following limitations:

  - Only one window clause is allowed per query.
  - Mixing window types (for example, sliding + tumbling) is not supported.
  - The window column must be a valid, monotonic field. Calculated fields are not allowed.

Troubleshooting

In this section, problems encountered, causes, and their resolution are listed in the following table.

Problem	Reason	Resolution
No Output	Wrong EMIT type.	Switch to `EMIT:NEW` or correct window.
High memory use	Large batch accumulation.	Use smaller windows or non-batch.
Snapshot ignored	Snapshot flag set too late.	Set the snapshot before `execute`.
Expensive computation	Large window + tiny slide.	Increase the slide or reduce the window size.

Event Stream Processing (ESP) Queries

You can configure a query agent to process events arriving through a channel, using continuous queries.

Event Stream Processing Queries Overview

Event stream processing (ESP) queries respond directly to events from the channel, as they happen. Instances of events specified in a query statement are piped directly to the query. ESP uses continuous queries only.

ESP queries are very performant because the data does not go through the inference engine and then cache and then finally to the query, as with cache queries. Instead the query engine listens to events directly, reducing latency.

It is more efficient to process very large numbers of events in a query agent than in an inference agent. Using ESP queries you can reduce and enrich the data before sending it to an inference agent. For example, using a sliding 10-minute window, a query could process all the router status messages that arrive in that time period and its callback rule function is executed for each event (that enter and leaves the time window) and can send out summary information for that event.

A query agent can perform both ESP queries and cache queries, when deployed in a TIBCO BusinessEvents application that uses cache OM. You can also configure standalone nodes that perform only ESP queries and do not use any cache functionality. An inference agent using In Memory OM could also be deployed in the same node.

Example ESP Query Strings

ESP uses continuous queries only, and the query string must include `accept: new` in the stream policy clause.

See Stream Clause. For example:

```
select count(*) as theCount from /InferenceOntology/DirectToQueryEvent
{policy: maintain last 10 seconds; accept: new} as dtq\n group by 1;
select sum(currentCount) as theSum from /QueryOntology/Level2QueryEvent
{policy: maintain last 25 seconds; accept: new} as l2q\n group by 1;
```

An example project demonstrating ESP queries is provided in the directory BE_HOME`/Examples/event_stream_processing/QueryESP`.

Event Assertion in a Query Agent

In a query agent, a channel executes an automatic `Event.assertEvent(e)` when its destination receives a message and converts it to the destination’s default event type.

However, query agents do not have a Rete network for rule inferencing, so the event is not asserted in the same way that it is asserted in an inference agent. Also, events asserted in a query agent are not persisted in the cache. Asserted events cannot be modified or explicitly deleted.

You can assert events in a callback rule function and they are asserted locally, within the query agent.

Note: There is no need to associate a locally asserted event with a destination. You only have to associate the event with a destination if you want to send the event out of the agent.

Events Asserted Locally Feed Second-Level Queries

Asserting events locally in the ESP query agent enables the output of one query to used by another query for processing and condensation.

The process can be repeated as many times as required, each query asserting an event that another query listens to. The end result is generally a smaller set of events with condensed, high-value information which can be sent to a TIBCO BusinessEvents application or other external application.

The following methods of asserting events locally are available within an ESP query agent:

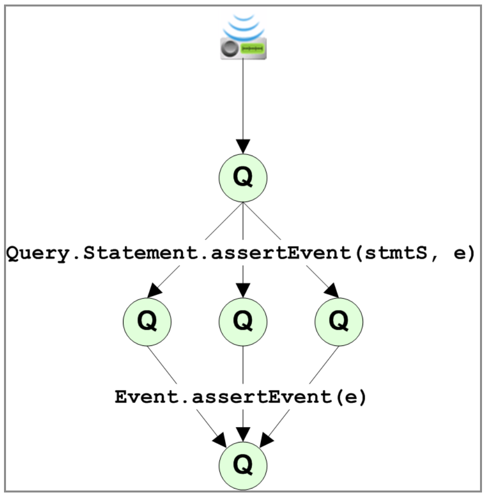
- The callback rule function executes an `Event.assertEvent(e)` after creating an event using data from the query.
- The callback rule function executes a `Query.Statement.assertEvent (statementName, e)` after creating an event using data from the query. This function pipes the named event to registered instances of the named query statement.

Some ESP Query Use Cases

The use cases can be elaborated and can be applied as per your needs.

Map and Reduce

ESP queries can be used to implement a kind of "map and reduce" data processing pattern.



A message arrives at a destination that transforms it into an event of a certain type.

A query configured to listen for events of that type (as specified in the query's `from` clause) then executes:

In the callback rule function for the following query, you would perform some kind of mapping operation. You would create event instances and then assert them using the following function:

```
Query.Statement.assertEvent("myStatement", myEvent);
```

The function sends the named event directly to all query instances registered under the given statement name. You can use the function one or more times in the callback rule function, according to your needs.

The callback rule function for each of the second-level queries could perform some kind of "reduce" operation, and then create and assert an event locally using the function `Event.assertEvent(e)`. The event is piped to any query that is listening for it.

The callback rule function for the final level of query would create and send out an event containing the reduced, higher-value information, for example to a TIBCO BusinessEvents application.

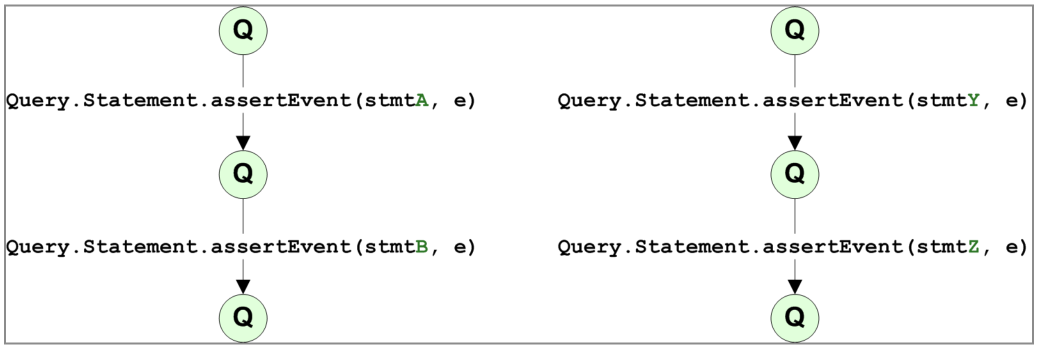
ETL (Extract Transform Load) Pattern

Another pattern you can implement is ETL.

Using the following function you can implement dedicated queries that are strung together like beads on a thread, each listening for the output of the one before:

```
Query.Statement.assertEvent(statementName, e)
```

As a result you can process multiple streams of events in parallel.


Q
Query.Statement.assertEvent(stmtA, e)
Q
Query.Statement.assertEvent(stmtB, e)
Q
Q
Query.Statement.assertEvent(stmtY, e)
Q
Query.Statement.assertEvent(stmtZ, e)
Q

Standalone ESP Project Configuration

No special configuration is required if you want to use ESP queries in a query agent deployed in a cache cluster.

Agent configuration is documented in the TIBCO BusinessEvents Developer Guide.

However, if you are running a standalone node for ESP queries only (and not cache queries), configure it as follows, in the project CDD file:

- Use Cache object management. (Cache OM is required only for the query agent to function.)
- Configure events and concepts to use Memory Only mode. (In the Cluster tab, expand Domain Objects > Default and set Mode to Memory Only.)

If you deploy an inference agent in the same node, configure it to use In Memory OM.

Note that the query agent's local cache is used only if the agent is getting objects from a cache cluster. It is not used for events arriving from a channel.

Tip: Performance Tuning—Garbage Collection Settings for Sun JDK

The following are some tips for tuning the JVM.

Suggested settings are as follows: Replace *n* with the number of CPUs in a multi-CPU machine.

```
-Xms2g -Xmx2g -XX:+AggressiveOpts -XX:+UseParallelOldGC -
XX:+UseParallelGC -XX:ParallelGCThreads=n
```

It is recommended that you use a 2GB heap or larger for high volume applications.

Query Language Reference

The syntax diagrams show the structure of a query and of each clause in a query. Operators and other items are also used in the syntax diagrams to add more clarity to the clauses.

Miscellaneous Terms Used in Syntax Diagrams

Some miscellaneous terms are used in the syntax diagrams that do not fall into categories documented in other tables.

Miscellaneous Terms Used in Query Syntax Diagrams

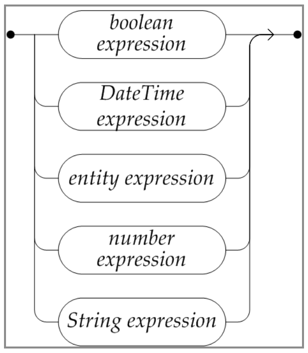
Terms	Descriptions
alias	Each alias must be globally unique in the whole query (this includes aliases defined in the projection—that is, aliases used in the select clause and in the from clause.
entity	Use the fully qualified ontology name of an entity, with its project path. `From /concepts/customer data` Remember that names are case sensitive
time unit	Allowable time units are as follows: `milliseconds`, `seconds`, `minutes`, `hours`, `days`

Syntax Diagrams

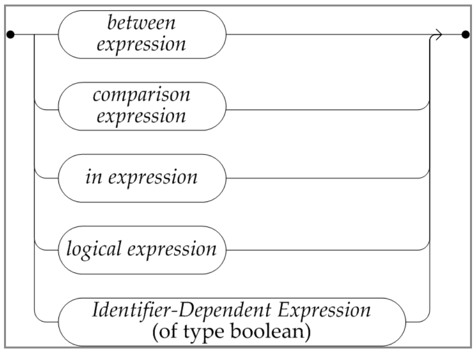
The syntax diagrams show the structure of a query and of each clause in a query.

Read them from left to right. Items above or below the main line are optional. Items that can repeat are shown by lines that loop back from the end to the beginning of the repeating section, along with the separator character.

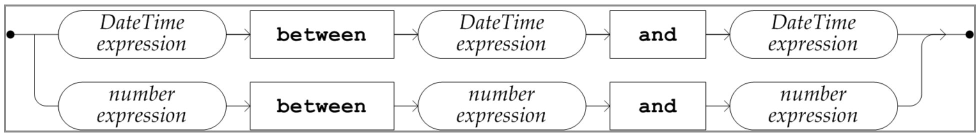
Expression



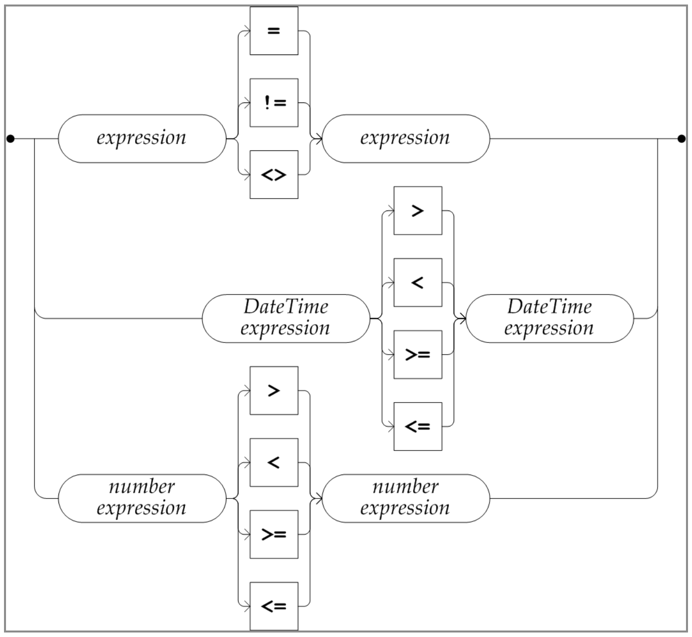
Boolean Expression



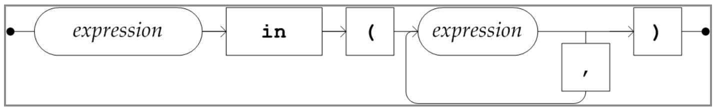
Between Expression



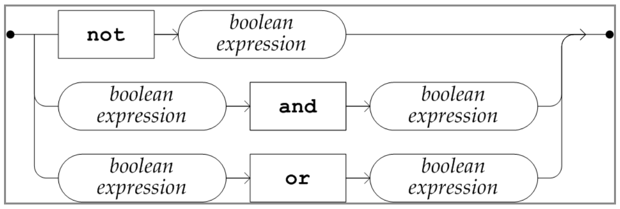
Comparison Expression



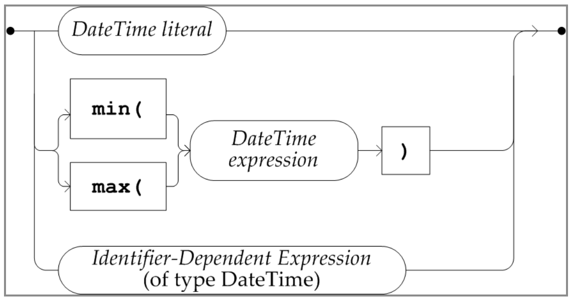
In Expression



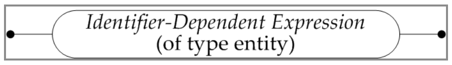
Logical Expression



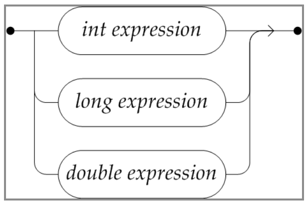
DateTime Expression



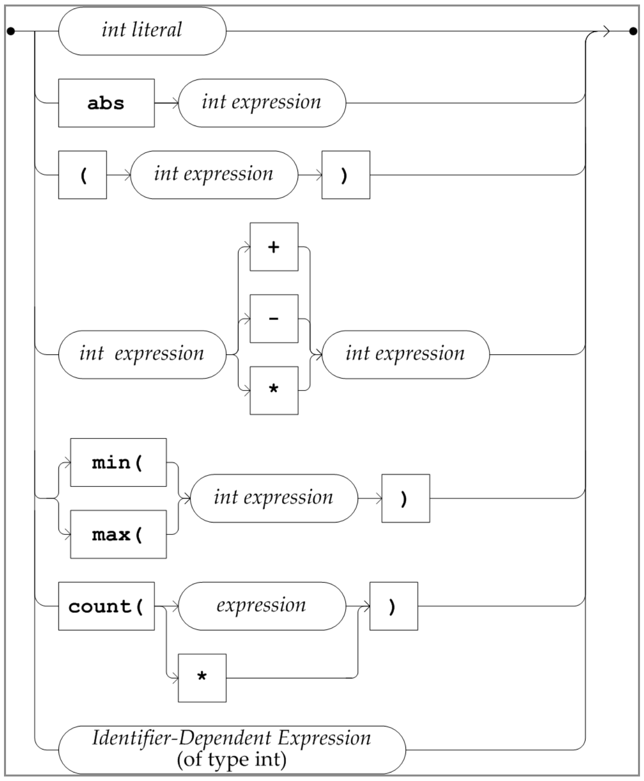
Entity Expression



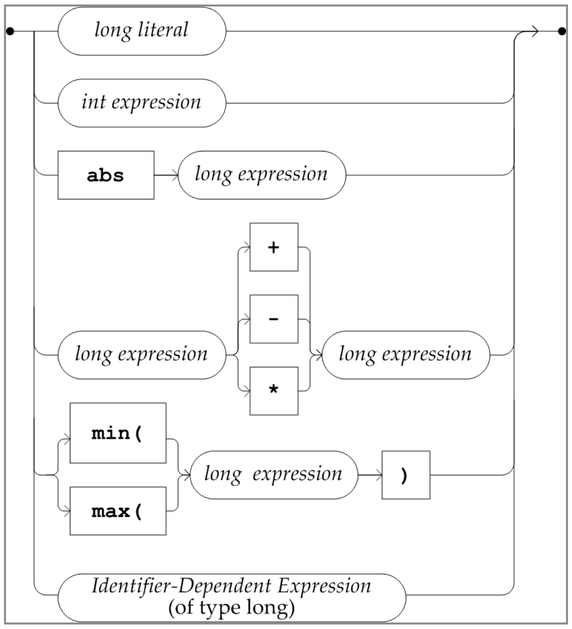
Number Expression



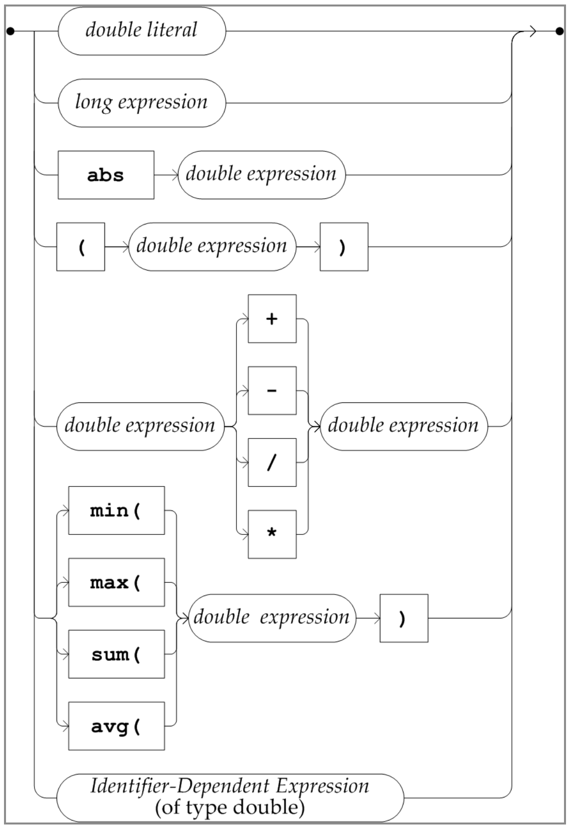
Int Expression



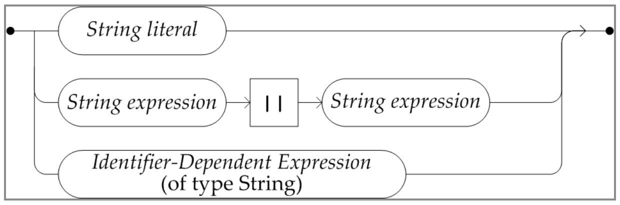
Long Expression



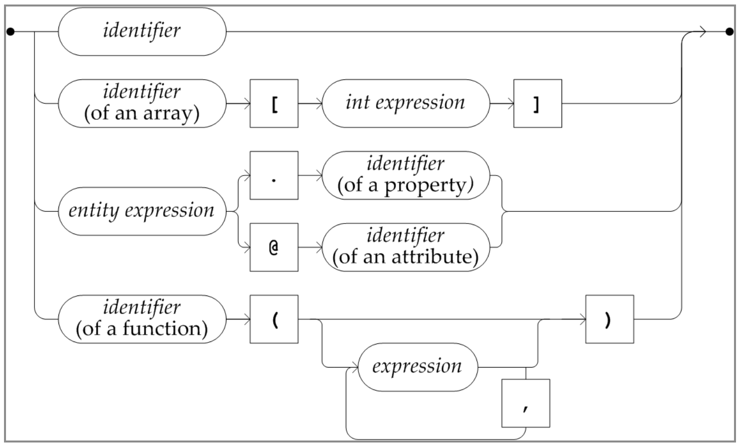
Double Expression



String Expression



Identifier-Dependent Expression



Operators for Unary Expressions

Few operators can be used to work on single operand.

Operators for Unary Expressions in Queries

Operator	Description and Examples	Datatypes	Result type
`not`	Negation `not x`	x must be a Boolean	Boolean
`abs`	absolute value `abs x`	x must be a number	The type of the operand
`+`	unary plus `+ x`	x must be a number	The type of the operand
`-`	unary minus `-x`	x must be a number	The type of the operand
`()`	Group (that is, parentheses) `(a+b)`	Any	The type of the operand

Operators for Binary Expressions

Various logical, mathematical and string operators are provided to created queries using binary expressions.

Operators for Binary Expressions in Queries (Sheet of)

Operator	**Description and Examples**	**Datatypes**	**Result type**
Relational Expression Operators			
`=`	equality `x = y`	x and y can be any type	Boolean

`!=`
`<>`
	inequality
	`x != y`
	`x <> y`
		x and y can be any type.	Boolean

`>`
`<`
`>=`
`<=`
	Greater than
	Less than
	Greater than or equal to
	Less than or equal to
	`x > y` (and so on)
	Generically known as comparison operators
		x and y must both be number types, or both be Datetime types.	Boolean

Logical Operators			

`and` `or`	Logical (Boolean) and, or.
	`x and y`
	`x or y`
		x and y must be Boolean	Boolean

Mathematical Operators Also used in the projection (`select` clause)			

Operator	Description and Examples	Datatypes	Result type
`*`	Multiplication `x * y`	x and y must both be numbers.	Either the type of x or y, whichever has the larger capacity.
`\`	Division `x \ y`	x and y must both be numbers.	double
`mod`	Remainder `x mod y`	x and y must both be numbers.	Either the type of x or y, whichever has the larger capacity.
`+`	Addition `x + y`	x and y must both be numbers.	Either the type of x or y, whichever has the larger capacity.
`-`	Subtraction `x - y`	x and y must both be numbers.	Either the type of x or y, whichever has the larger capacity.
Postfix Operators			
`[]`	Array dereferencing, to access an array element. `x[y]`	x must be an array and y must be an int.	Type of the array element.
`.`	For object graph traversal, to access a property `x.y`	x must be an entity and y must be a property.	Type of y.
`@`	For object graph traversal, to access an attribute `customer@extId`	x must be an entity and y must be a attribute.	Type of y.
String Operator			
`||`	String concatenation	x and y must be	String

Operator	Description and Examples	Datatypes	Result type
	`x || y`	String	

`like`	The like operator matches all strings that match the regular expression provided in double quotes.
	
	With the TIBCO cache provider, you can use `"*.*"` syntax, for example:
	
	`select symbol from /ConceptModel/StockTick where symbol like ".*T.*"`
	
	Results: TIBX, MSFT,
	
	`where symbol like ".*T"`
	
	Results: MSFT
	
	`where symbol like "TIBX"`
	
	Results: TIBX
	
	`where symbol like "TIB."`
	
	Results: TIBX
	
	`where symbol like ".*"`
	
	Results: JNJ, VMW, TIBX, HPQ, MSFT, HPQ
		String	Boolean

Operators for Other Expressions

Event Stream Processing also provide few operators for queries other than unary and binary expressions.

Operators for Other Expressions in Queries

Operator	Description and Examples	Datatypes	Result type
`between and`	Between operator for range expressions. Range is inclusive. `x between y and z`	x and y must all be number types, or all be Datetime types.	Boolean
`in()`	Inclusion operator. Checks if an expression is in a group of items. `x in (y1, y2, ..., yn)`	Any	Boolean
`$`	Bind variable prefix. `$name`	*name* has no type. It is just a label.	The type of `$name` is determined by its surrounding expression. For example, in the expression: `($minimum + 14.58)` `$minimum` is a bind variable of type double.

Wildcards Datatypes Literals Identifiers and Keywords

Event Stream Processing supports various wildcards, datatypes and keyword which can help in creating the queries.

Wildcard Characters

- The asterisk (*) is a wildcard character, meaning "all"
- The single quote (’) is a single character wildcard

Datatypes

All types supported by TIBCO BusinessEvents.

Literals

Literal values can be of any of the following data types as well as those mentioned in the following table:

- `hex`
- `octal`
- `char`

Note: Octal Values

To specify an octal number, begin the number with a zero (0), for example, 01223 is treated as an octal number.

Do not start decimal numbers with a leading zero. To specify a decimal zero use zero and a decimal point (0.). Do not use 0.0.

Types and Literals

Query Language Types and Literals

Type	Syntax of Literals	Example
int	A signed integer expressed using only digits and an optional sign prefix. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive).	1234567
long	A signed integer expressed using only digits	digits

Type	Syntax of Literals	Example
	and an optional sign prefix. It has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive).	1234567
double	A double-precision 64-bit IEEE 754 floating point.	12345.67 1.234e+56

String	String literals are surrounded by double quotes.
	
	To escape double quote and backslash characters, prefix them with a backslash.
		`"hello"`
		`"She says: \"Hello.\""`
		`"c:\\temp\\myfile"`

boolean	The boolean data type has only two possible values: true and false. Use for simple flags that track true and false conditions.	`true` `false`
DateTime	`yyyy-MM-dd'T'HH:mm:ss.SSSZ` where `yyyy`: four digit year `MM`: two digit month `dd`: two digit day of month `HH`: two digit hour of day in 24-hour format `mm`: two digit minutes in hour `ss`: two digit seconds in minute `SSS`: three digit milliseconds in second `’T’`: the letter T `Z`: timezone expressed as defined in RFC 822.	`2008-04-23T13:30:25.123-0700`

Type	Syntax of Literals	Example
Entity type	"*entity-project-path*" Entity project path begins with a forward slash and folders are separated with a forward slash.	`"/a/b/MyConcept"`
Entity	No literal is used for entity instances.	(Not applicable)

Identifiers

The first character of an identifier must be alphabetical (upper or lower case) or the underscore character. Other characters can be alphabetical or numeric or the underscore character.

Keywords and Other Reserved Words

The complete list of keywords and reserved words used by TIBCO BusinessEvents and its add-on products is provided in the section Keywords and Other Reserved Words in TIBCO BusinessEvents Developer Guide.

In TIBCO BusinessEvents Event Stream Processing, the restriction is not case sensitive. For example, `last`, `Last` and `LAST` are all reserved.

Escaping the Keywords

If you want to use keywords as identifiers, resource names, or folder names in your query string, prefix them with the # escape character.

Examples:

```
select id from /PO/#Order o
select /#DateTime/format(birthDate, "yyyy-MM-dd") from /Person
select e.sender as #from from /Email e
```

Where the following are the types items that use keywords:

`#Order` is a concept name

`#DateTime` is a function catalog category whose name happens to be a keyword (only category names that are keywords need to be escaped)

`#from` is an alias

TIBCO Documentation and Support Services

For information about this product, you can read the documentation, contact TIBCO Support, and join TIBCO Community.

How to Access TIBCO Documentation

Documentation for TIBCO products is available on the Product Documentation website, mainly in HTML and PDF formats.

The Product Documentation website is updated frequently and is more current than any other documentation included with the product.

Product-Specific Documentation

The documentation for this product is available on the TIBCO BusinessEvents® Enterprise Edition Documentation page.

Other TIBCO Product Documentation

When working with TIBCO BusinessEvents Enterprise Edition, you may find it useful to read the documentation of the following TIBCO products:

- TIBCO ActiveSpaces®: It is used as the store provider for the TIBCO BusinessEvents Enterprise Edition project.
- TIBCO FTL®: It is used as the cluster provider for the TIBCO BusinessEvents Enterprise Edition project.

How to Access Related Third-Party Documentation

When working with TIBCO BusinessEvents® Enterprise Edition, you may find it useful to read the documentation of the following third-party products:

- Apache Ignite
- Apache Kafka

- Confluent Kafka Schema Registry
- TIBCO Messaging - Schema Repository for Apache Kafka
- Apache Pulsar
- GridGain
- Apache Cassandra
- Grafana
- InfluxDB
- OpenTelemetry
- OTel Collector
- Apache Maven

How to Contact Support for TIBCO Products

You can contact the Support team in the following ways:

- To access the Support Knowledge Base and getting personalized content about products you are interested in, visit our product Support website.
- To create a Support case, you must have a valid maintenance or support contract with a Cloud Software Group entity. You also need a username and password to log in to the product Support website. If you do not have a username, you can request one by clicking **Register** on the website.

How to Join TIBCO Community

TIBCO Community is the official channel for TIBCO customers, partners, and employee subject matter experts to share and access their collective experience. TIBCO Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from TIBCO products. For a free registration, go to TIBCO Community.

Legal and Third-Party Notices