Important Information

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH SOFTWARE LICENSE AGREEMENT OR CLICKWRAP END USER LICENSE AGREEMENT, THE LICENSE(S) LOCATED IN THE "LICENSE" FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

This document contains confidential information that is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of TIBCO Software Inc.

TIBCO, The Power of Now, TIBCO ActiveMatrix, TIBCO ActiveMatrix BusinessWorks, TIBCO Administrator, TIBCO ActiveSpaces, TIBCO Designer, TIBCO Enterprise Message Service, TIBCO Hawk, TIBCO Runtime Agent, TIBCO Rendezvous, are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and/or other countries.

EJB, Java EE, J2EE, and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only.

THIS SOFTWARE MAY BE AVAILABLE ON MULTIPLE OPERATING SYSTEMS. HOWEVER, NOT ALL OPERATING SYSTEM PLATFORMS FOR A SPECIFIC SOFTWARE VERSION ARE RELEASED AT THE SAME TIME. SEE THE README.TXT FILE FOR THE AVAILABILITY OF THIS SOFTWARE VERSION ON A SPECIFIC OPERATING SYSTEM PLATFORM.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "READ ME" FILES.

Copyright © 2004-2011 TIBCO Software Inc. ALL RIGHTS RESERVED.

TIBCO Software Inc. Confidential Information
Contents

Tables .................................................................................................................. vii

Preface ................................................................................................................ x

Changes from the Previous Release of this Guide ................................................. x

Related Documentation ........................................................................................ xi

TIBCO BusinessEvents and Add-On Product Documentation .............................. xi

Accessing TIBCO BusinessEvents Functions Reference Documentation ........... xiv

Other TIBCO Product Documentation ............................................................... xiv

Typographical Conventions ................................................................................. xv

Connecting with TIBCO Resources .................................................................... xviii

How to Join TIBCOmmunity .............................................................................. xviii

How to Access All TIBCO Documentation ......................................................... xviii

How to Contact TIBCO Support .......................................................................... xviii

Chapter 1 Query Features Overview ................................................................. 1

Query Features Overview ................................................................................... 2

Queries are Executed in Query Agents ............................................................... 2

Querying the Cache ............................................................................................. 2

Querying the Event Stream ................................................................................ 2

Using a Co-deployed Inference Agent ............................................................... 3

Two Types of Queries—Snapshot and Continuous ............................................ 3

Structure of a Query Select Statement .............................................................. 4

Summary of Functions Used to Create and Execute Queries ............................ 4

Two Example Uses for Cache Queries ............................................................... 6

Triggering a Query from a Rule (in an Inference Agent) ................................ 6

Using a Query as a Pre-filter ............................................................................. 6

Chapter 2 Query Language Components ....................................................... 7

Select Clause ....................................................................................................... 8

Delete Clause ..................................................................................................... 9

From Clause ....................................................................................................... 10

Where Clause .................................................................................................... 11

Group by Clause ............................................................................................... 12

Using a Dummy Group Expression for Aggregation ....................................... 12

Optional having Clause ................................................................................... 13
Identifier-Dependent Expression ................................................................. 71
Operators for Unary Expressions .................................................................. 72
Operators for Binary Expressions ................................................................ 73
Operators for Other Expressions ................................................................. 76
Wildcards, Datatypes, Literals, Identifiers, and Keywords. .......................... 77
  Wildcard Characters ................................................................................. 77
  Datatypes .................................................................................................. 77
  Literals ..................................................................................................... 77
  Types and Literals ................................................................................... 77
  Identifiers ................................................................................................ 78
  Keywords and Other Reserved Words. ....................................................... 79
  Escaping the Keywords ............................................................................ 79

Index ............................................................................................................ 81
## Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>General Typographical Conventions</td>
<td>xv</td>
</tr>
<tr>
<td>2</td>
<td>Syntax Typographical Conventions</td>
<td>xvi</td>
</tr>
<tr>
<td>3</td>
<td>Miscellaneous Terms Used in Query Syntax Diagrams</td>
<td>62</td>
</tr>
<tr>
<td>4</td>
<td>Operators for Unary Expressions in Queries</td>
<td>72</td>
</tr>
<tr>
<td>5</td>
<td>Operators for Binary Expressions in Queries</td>
<td>73</td>
</tr>
<tr>
<td>6</td>
<td>Operators for Other Expressions in Queries</td>
<td>76</td>
</tr>
<tr>
<td>7</td>
<td>Query Language Types and Literals</td>
<td>77</td>
</tr>
</tbody>
</table>
Preface

TIBCO BusinessEvents® Event Stream Processing software provides query and event pattern matching features that help you to identify and create valuable information in TIBCO BusinessEvents® event streams and data caches.

TIBCO BusinessEvents Event Stream Processing Add-on Query Developer’s Guide presents the syntax of the query language and explains how it is used in TIBCO BusinessEvents.

TIBCO BusinessEvents
The Power to Predict™

Topics

• Changes from the Previous Release of this Guide on page x
• Related Documentation, page xi
• Typographical Conventions, page xv
• Connecting with TIBCO Resources, page xviii
Changes from the Previous Release of this Guide

This section itemizes the major changes from the previous release of this guide.

**Dummy Groups**

Content explaining use of dummy groups in a group by clause has been expanded and clarified. See Using a Dummy Group Expression for Aggregation on page 12.

**Getting the Count of Records in Certain Result Sets**

A section is added to explain the use of this function: `Query.ResultSet.getRowCountIfPossible()`

See Getting the Count of Records in Certain Result Sets on page 27.

**Configuration of Standalone ESP Projects**

Configuration of standalone ESP projects has changed. The property `be.network.mode.standalone` is no longer required. See Configuring a Standalone ESP Project on page 60.
Related Documentation

This section lists documentation resources you may find useful.

TIBCO BusinessEvents and Add-On Product Documentation

The following diagram shows the main documents in the TIBCO BusinessEvents documentation set, and the documentation sets for the optional add-on products.

Each set also contains an installation guide, release notes, and a readme file.

TIBCO BusinessEvents Documentation

TIBCO BusinessEvents Studio, the design-time UI, is supported on Windows and Linux. The documentation set for TIBCO BusinessEvents is as follows.

- **TIBCO BusinessEvents Installation**: Read this manual for instructions on site preparation, installation, upgrading from an earlier release, and project migration.

- **TIBCO BusinessEvents Getting Started**: After the product is installed, use this manual to learn the basics of TIBCO BusinessEvents: project design, cache
OM, and backing store. This guide explains the main ideas so you gain understanding as well as practical knowledge.

- **TIBCO BusinessEvents Architect’s Guide**: If you are architecting an application using TIBCO BusinessEvents, read this guide for overview and detailed technical information to guide your work.

- **TIBCO BusinessEvents Developer’s Guide**: Use this guide when you implement a project design in TIBCO BusinessEvents Studio. It covers topics such as project-level tasks, resource-level tasks, debugging, and integration with TIBCO ActiveMatrix BusinessWorks.

- **TIBCO BusinessEvents Administration**: This book explains how to configure, deploy, monitor, and manage a TIBCO BusinessEvents application and the data it generates. It explains how to configure the CDD file for different object management options, set up a backing store, and how to work with TIBCO BusinessEvents Monitoring and Management component, and with TIBCO Administrator.

- Online References:
  - **TIBCO BusinessEvents Java API Reference**: This online reference is available from the HTML documentation interface. It provides the Javadoc-based documentation for the TIBCO BusinessEvents API.
  - **TIBCO BusinessEvents Functions Reference**: This reference is available from the HTML documentation interface. It provides a listing of all functions provided with TIBCO BusinessEvents, showing the same details as the tooltips available in TIBCO BusinessEvents Studio.

- **TIBCO BusinessEvents Release Notes**: Read the release notes for a list of new and changed features. This document also contains lists of known issues and closed issues for this release.

### TIBCO BusinessEvents Event Stream Processing

This TIBCO BusinessEvents add-on is available separately, and includes the TIBCO BusinessEvents Query Language features and the Pattern Matcher Service.

- **TIBCO BusinessEvents Event Stream Processing Installation**: Read this brief manual for installation instructions. A compatible version of TIBCO BusinessEvents must be installed before you install any add-on.

- **TIBCO BusinessEvents Event Stream Processing Add-on Query Developer’s Guide**: This manual explains how to use the object query language to query various aspects of the running system. For details on configuring and deploying query agents, see **TIBCO BusinessEvents Administration**.
TIBCO BusinessEvents Event Stream Processing Pattern Matcher Developer’s Guide: This manual explains how to use the pattern matcher language and engine to correlate event patterns in a running system.

TIBCO BusinessEvents Event Stream Processing Release Notes: Read the release notes for a list of new and changed features. This document also contains lists of known issues and closed issues for this release.

TIBCO BusinessEvents Decision Manager

This TIBCO BusinessEvents add-on is available separately. It incorporates the Decision Manager decision modeling business user interface (supported on Windows and Linux), and the Rules Management Server (supported on all platforms supported by TIBCO BusinessEvents).

TIBCO BusinessEvents Decision Manager Installation: Read this brief manual for installation instructions. A compatible version of TIBCO BusinessEvents must be installed before you install any add-on.

TIBCO BusinessEvents Decision Manager User’s Guide: This manual explains how business users can use decision tables and other decision artifacts to create business rules. It also covers configuration and administration of Rules Management Server, which is used for authentication, authorization, and approval processes.

TIBCO BusinessEvents Decision Manager Release Notes: Read the release notes for a list of new and changed features. This document also contains lists of known issues and closed issues for this release.

TIBCO BusinessEvents Data Modeling

This TIBCO BusinessEvents add-on is available separately. It contains state models and database concept features.

TIBCO BusinessEvents Data Modeling Installation: Read this brief manual for installation instructions. A compatible version of TIBCO BusinessEvents must be installed before you install any add-on.

TIBCO BusinessEvents Data Modeling Developer’s Guide: This manual explains data modeling add-on features for TIBCO BusinessEvents. The database concepts feature enables you to model TIBCO BusinessEvents concepts on Database tables. The state modeler feature enables you to create state machines.

TIBCO BusinessEvents Data Modeling Release Notes: Read the release notes for a list of new and changed features. This document also contains lists of known issues and closed issues for this release.
TIBCO BusinessEvents Views

This TIBCO BusinessEvents add-on is available separately. It includes graphical dashboard components for run-time event monitoring.

- **TIBCO BusinessEvents Views Installation**: Read this manual for instructions on site preparation and installation. A compatible version of TIBCO BusinessEvents must be installed before you install any add-on.

- **TIBCO BusinessEvents Views Developer’s Guide**: This guide explains how to use TIBCO BusinessEvents Views to create meaningful metrics that are presented to business users in real-time for proactive decision making.


- **TIBCO BusinessEvents Views Release Notes**: Read the release notes for a list of new and changed features. This document also contains lists of known issues and closed issues for this release.

Accessing TIBCO BusinessEvents Functions Reference Documentation

Reference documentation for functions, including those used in add-ons, is available in the HTML documentation interface for the TIBCO BusinessEvents documentation set, and as tooltips in TIBCO BusinessEvents Studio. To use the HTML-based functions reference from the file system do the following:


2. In the left panel, browse to Online References and in the right panel choose TIBCO BusinessEvents Functions Reference. The reference opens in a new tab.

3. Click the navigation links to browse to the functions as desired.

Other TIBCO Product Documentation

You may find it useful to refer to the documentation for the following TIBCO products:

- TIBCO ActiveSpaces®
- TIBCO Hawk®
- TIBCO Rendezvous®
- TIBCO Enterprise Message Service™
- TIBCO ActiveMatrix BusinessWorks™
Typographical Conventions

The following typographical conventions are used in this manual.

Table 1  General Typographical Conventions

<table>
<thead>
<tr>
<th>Convention</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIBCO_HOME</td>
<td>Many TIBCO products must be installed within the same home directory. This directory is referenced in documentation as TIBCO_HOME. The value of TIBCO_HOME depends on the operating system. For example, on Windows systems, the default value is C:\tibco.</td>
</tr>
<tr>
<td>ENV_HOME</td>
<td>Other TIBCO products are installed into an installation environment. Incompatible products and multiple instances of the same product are installed into different installation environments. An environment home directory is referenced in documentation as ENV_HOME. The value of ENV_HOME depends on the operating system. For example, on Windows systems the default value is C:\tibco.</td>
</tr>
<tr>
<td>BE_HOME</td>
<td>TIBCO BusinessEvents installs into a directory within an ENV_HOME. This directory is referenced in documentation as BE_HOME. The value of BE_HOME depends on the operating system. For example on Windows systems, the default value is C:\tibco\be\5.0.</td>
</tr>
</tbody>
</table>

Code font identifies commands, code examples, filenames, pathnames, and output displayed in a command window. For example:

Use MyCommand to start the foo process.

Bold code font is used in the following ways:

- In procedures, to indicate what a user types. For example: Type admin.
- In large code samples, to indicate the parts of the sample that are of particular interest.
- In command syntax, to indicate the default parameter for a command. For example, if no parameter is specified, MyCommand is enabled: MyCommand [enable | disable]
Typographical Conventions

Table 1  General Typographical Conventions (Cont’d)

<table>
<thead>
<tr>
<th>Convention</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>italic font</td>
<td>Italic font is used in the following ways:</td>
</tr>
<tr>
<td></td>
<td>• To indicate a document title. For example: See TIBCO BusinessWorks Concepts.</td>
</tr>
<tr>
<td></td>
<td>• To introduce new terms For example: A portal page may contain several portlets. Portlets are mini-applications that run in a portal.</td>
</tr>
<tr>
<td></td>
<td>• To indicate a variable in a command or code syntax that you must replace. For example: MyCommand pathname</td>
</tr>
</tbody>
</table>

Key combinations

Key name separated by a plus sign indicate keys pressed simultaneously. For example: Ctrl+C.

Key names separated by a comma and space indicate keys pressed one after the other. For example: Esc, Ctrl+Q.

The note icon indicates information that is of special interest or importance, for example, an additional action required only in certain circumstances.

The tip icon indicates an idea that could be useful, for example, a way to apply the information provided in the current section to achieve a specific result.

The warning icon indicates the potential for a damaging situation, for example, data loss or corruption if certain steps are taken or not taken.

Table 2  Syntax Typographical Conventions

<table>
<thead>
<tr>
<th>Convention</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>An optional item in a command or code syntax. For example:</td>
</tr>
<tr>
<td></td>
<td>MyCommand [optional_parameter] required_parameter</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MyCommand param1</td>
</tr>
</tbody>
</table>
A logical group of items in a command. Other syntax notations may appear within each logical group.

For example, the following command requires two parameters, which can be either the pair `param1` and `param2`, or the pair `param3` and `param4`:

```
MyCommand {param1 param2} | {param3 param4}
```

In the next example, the command requires two parameters. The first parameter can be either `param1` or `param2` and the second can be either `param3` or `param4`:

```
MyCommand {param1 | param2} {param3 | param4}
```

In the next example, the command can accept either two or three parameters. The first parameter must be `param1`. You can optionally include `param2` as the second parameter. And the last parameter is either `param3` or `param4`:

```
MyCommand param1 [param2] {param3 | param4}
```
Connecting with TIBCO Resources

This section provides links to helpful TIBCO resources.

How to Join TIBCOmmunity

TIBCOmmunity is an online destination for TIBCO customers, partners, and resident experts, a place to share and access the collective experience of the TIBCO community. TIBCOmmunity offers forums, blogs, and access to a variety of resources. To register, go to http://www.tibcommunity.com.

How to Access All TIBCO Documentation

After you join TIBCOmmunity, you can access the documentation for all supported product versions here:

http://docs.tibco.com/TibcoDoc

How to Contact TIBCO Support

For comments or problems with this manual or the software it addresses, please contact TIBCO Support as follows.

- For an overview of TIBCO Support, and information about getting started with TIBCO Support, visit this site:
  
  http://www.tibco.com/services/support

- If you already have a valid maintenance or support contract, visit this site:
  
  https://support.tibco.com

Entry to this site requires a user name and password. If you do not have a user name, you can request one.
Chapter 1  Query Features Overview

This chapter provides a short overview of query features.

Topics

• Query Features Overview, page 2
• Two Example Uses for Cache Queries, page 6
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).

Configuration of query agents is explained in *TIBCO BusinessEvents Administration*.

Queries are Executed in 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.

For instructions on how to configure a cache query agent, see *TIBCO BusinessEvents Administration*.

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 can’t 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 Administration*.

You can load objects into the cache so you can then query them. To load objects into the cache, use the `Coherence.C_CacheLoad*()` functions. For details on these functions, see their tooltips, and also see *TIBCO BusinessEvents Developer’s 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, web site 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 Chapter 5, Event Stream Processing (ESP) Queries, on page 55 for details.

### Using a Co-deployed Inference Agent

Depending on the need, it can be useful to deploy an inference agent in the same node as the query agent. The inference agent can process and enrich the event data, create 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.

The co-deployed 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.

### 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 Chapter 3, Working With the Query Language, on page 19 and in particular, see Simple Snapshot Query Example, page 32 for
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 Chapter 4, Continuous Queries, on page 39 for more details.

Structure of a Query Select Statement

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:

```
select clause  from clause  where clause  group by clause  order by clause
```

Each clause in a statement is examined in detail in Chapter 2, Query Language Components, on page 7. For a quick reference to the query syntax, see Chapter 6, Query Language Reference, on page 61.

The query text is provided as an argument to the `Query.create()` function. Use of functions to create and execute queries is explained next.

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, explained later in this guide.

- **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 Chapter 3, Working With the Query Language, on page 19 for more details.
Two Example Uses for Cache Queries

The following sections show two ways queries can be used. (For information on event stream processing queries, see Chapter 5, Event Stream Processing (ESP) Queries, on page 55.)

Triggering a 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.

Inference Agent 1. A rule in the inference agent sends an event to destination D1, including any necessary query parameters.

Query Agent 2. The query agent listens for messages on destination D1.

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

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

Inference Agent 5. The inference agent listens on destination D2.

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

Using a 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:

Query Agent 1. The query agent listens for messages on a destination D1.

2. 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.

Inference Agent 3. The inference agent listens for events (messages) on destination D2.

4. On receiving an event at D2, a rule in the inference agent creates the concept.
Chapter 2  Query Language Components

This chapter expands each of the components of the `select` and `delete` statements:

```
select clause
from clause
where clause
group by clause
order by clause
;
```

```
delete clause
from clause
where clause
```

Topics

- Select Clause, page 8
- Delete Clause, page 9
- From Clause, page 10
- Where Clause, page 11
- Group by Clause, page 12
- Order by Clause, page 14
- Limit Clause, page 15
- Stream Clause, page 16
- Stream Policy, page 17
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:

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

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

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

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 on page 12)

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 on page 79.)

```sql
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
```
The delete clause is used only in a delete query. Delete queries are used in a specific situation only. See Using the Delete Query on page 30 for more details.

In the delete clause, you can’t 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.

```plaintext
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 can’t 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:

```java
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 on page 40 and sections following, for more information.

Examples

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

```sql
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**

Pound or hash (#) is the escape character. See *Keywords and Other Reserved Words on page 79*.

```plaintext
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
```

The pound sign (#) is used to escape reserved (key) words. See *Keywords and Other Reserved Words on page 79* 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:

```sql
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.

When you use a group by clause, the group by criteria control what criteria you can use in the select clause. What is available to the select clause has been, in effect, reduced using the group by clause—the select clause can use only the group by criteria, and aggregation functions. For example, this is valid:

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

But the next example is invalid:

```sql
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:

```sql
select count(*)
```
As explained in the note above, 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 these 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) on page 34.

See also Limit Clause on page 15.

Examples

Pound or hash (#) is the escape character. See Keywords and Other Reserved Words on page 79.

```
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;
```

In the above 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.
Limit Clause

You can use an optional limit clause in a select or an order by clause. When used in a select clause, the limit the maximum number of rows to return. 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 Implicit Window Examples on page 42.

Example Showing Use in Select Clause

```sql
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

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

The above 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.

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.

**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 Chapter 5, Event Stream Processing (ESP) Queries, on page 55.

**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`.

See Time Window Examples (Cache Queries) on page 50 for some example uses.

See Stream Policy on page 17 for more details on how the window is defined.
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 on page 44 and examples following: Sliding Window Examples (Cache Queries) on page 47, Tumbling Window Examples (Cache Queries) on page 49, and Time Window Examples (Cache Queries) on page 50.

Note that continuous queries that use an implicit window do not have a stream policy. See Working With Implicit Windows on page 42.

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:

```sql
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) on page 45 for a detailed discussion of an example that uses a stream policy. See Time Window Examples (Cache Queries) on page 50 for more examples.
Chapter 3  Working With the Query Language

This chapter explains how to use the query language to create and execute queries and use results returned by a query. It covers snapshot and continuous queries.

Topics

- Constructing Queries and Using Query Results, page 20
- Lifecycle of a Query—Use of Query Functions, page 22
- Using Data from a Result Set (Snapshot Queries), page 27
- Using Data from a Callback Rule Function, page 28
- Using the Delete Query, page 30
- Simple Snapshot Query Example, page 32
- Simple Continuous Query Example, page 33
- Using Bind Variables in Query Text, page 37
Constructing Queries and Using Query Results

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 can’t query scorecards or time events because they don’t exist in the cache. You can’t 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.

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 in the online reference, *TIBCO BusinessEvents Functions Reference*.

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

Using Functions Within Queries

Many of the available catalog functions as well as custom functions can be used in the text of a query. You can also use rule functions from the same project.
Functions that Can Be Used in Queries
Functions that can be used in queries are marked with a blue $q$. (They may have more decorations if they are usable in other areas such as Decision Manager). You can call such functions in a query string.

Functions that Can’t be Used in Queries
The following functions cannot be used within queries:

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

Using Bind Variables
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 Using Bind Variables in Query Text on page 37 for details.

Limitation in Use of Arrays
You can use arrays within expressions in a query, but returning arrays in the results of the query is not supported in this release.
Lifecycle of a Query—Use of Query Functions

This section explains how to use functions to create and execute queries, and to gather query results. In summary:

1. **Create the Query Definition**: A *query definition* is a Java runtime object (similar to a factory class).
2. **Open a Query Statement**: A *query statement* is an object that represents one instance of the query. You can create multiple statements that can run in parallel.
3. **Set Bind Variables (if Used)**: For the named query statement, set values for bind variables (if any are used in the query definition).
4. **Execute an Instance of the Query Statement and Obtain Results**
5. **See Closing a Statement and Deleting a Query Definition**, for details on how queries and query statements that are no longer used are removed.

Also see Using Data from a Result Set (Snapshot Queries) on page 27 and Using Data from a Callback Rule Function on page 28 for details on how to get and use query results.

Create the Query Definition

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.

**Format:**

```java
Query.create(String QueryDefinitionName, String QueryText);
```

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

**Example:**

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

Where $\text{min}$ 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.

**Open a Query Statement**

Format:

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

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

Example:

```java
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 on page 32.

**Set Bind Variables (if Used)**

If you used bind variables in the query definition, then you set the values after opening the query statement, and before executing it. 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.

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.

Format

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

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

See Using Bind Variables in Query Text on page 37 for more details.

### Execute an Instance of the Query Statement and Obtain Results

To execute a query and specify how a query returns values, you use one of the following 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.

### To 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.

**Format**

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

**Example**

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

In the above 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 Using Data from a Result Set (Snapshot Queries) on page 27 for more information.
Closing a Result Set

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:

```java
Query.ResultSet.close(String resultsetName);
```

For example:

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

To 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) on page 34 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.

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

The format of the `Query.Statement.executeWithCallback()` function is shown below. The format of the `Query.Statement.executeWithBatchCallback()` 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.

Example

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

See Using Data from a Callback Rule Function on page 28 for details.

**Closing a Statement and Deleting a Query Definition**

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:

```java
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 Closing a Result Set on page 25.
Using Data from a Result Set (Snapshot Queries)

See To Obtain Results Using a Result Set on page 24 for details about obtaining results.

Moving 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:

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

The above 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:

```java
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:

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

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

Getting the Count of Records in Certain Result Sets

You can use the following function to get the count of records in a result set when using Query Functions:

```java
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 resultset without knowing when the query will end.
Using Data from a Callback Rule Function

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

The 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 Closing a Statement and Deleting a Query Definition on page 26.

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.

The 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 Closing a Statement and Deleting a Query Definition on page 26.

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 following parameter types, provided in the order specified:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>String</strong> id</td>
<td>Identifies the current execution. Uses the value of <code>listenerName</code>, which was provided when calling the <code>Query.Statement.executeWithCallback()</code> function. The ID enables you to identify rows of data belonging to different executions of the same query (or different queries).</td>
</tr>
<tr>
<td><strong>boolean</strong> isBatchEnd</td>
<td>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 on page 14). 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) on page 34.</td>
</tr>
<tr>
<td><strong>boolean</strong> hasEnded</td>
<td>When true, signals the end of the execution.</td>
</tr>
<tr>
<td><strong>Object</strong> row</td>
<td>For <code>Query.Statement.executeWithCallback()</code>: 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 on page 8). For <code>Query.Statement.executeWithBatchCallback()</code>: an array of rows comprising one batch of results.</td>
</tr>
<tr>
<td>OR</td>
<td><strong>Object</strong> rows</td>
</tr>
<tr>
<td><strong>Object</strong> closure</td>
<td>Closure object provided when executing the <code>Query.Statement.executeWithCallback()</code> function, or <code>null</code>. The object provided depends on your needs. For example, it could be a simple string, or it could be an array of objects used to add a row of data from each callback rule function.</td>
</tr>
</tbody>
</table>
Using the Delete Query

The delete query is typically used for the following specific use case. 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 on page 9 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 can’t 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 following example code could be placed in a preprocessor rule function that receives an event called `requestEvent`. It 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.

```java
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 Closing a Statement and Deleting a Query Definition on page 26 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 example provided in this section shows how a callback rule function is used to gather results generated by the query. The callback rule function is shown next:

```java
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

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

Open and Execute the Query Statement

```java
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 sample output below, 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 below 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

This section 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)

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

TIBCO BusinessEvents Event Stream Processing Query Developer's Guide
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

This section 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, "@@@")

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.
Using 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 example below.)

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.

```java
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()

All functions must reference the same query statement name.

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 on page 10 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.

```java
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.

Assigning a Datatype to a Bind Variable

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

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>(+ 0)</td>
</tr>
<tr>
<td>long</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>(+ .0)</td>
</tr>
<tr>
<td>String</td>
<td>(</td>
</tr>
<tr>
<td>Boolean</td>
<td>( or false)</td>
</tr>
<tr>
<td>DateTime</td>
<td><strong>Note:</strong> <code>DateTime</code> is not supported so use the following instead. Pass a <code>long</code> instead of a <code>DateTime</code> and use: <code>/#datetime/parseLong()</code></td>
</tr>
</tbody>
</table>

Pound or hash (#) is the escape character. See Keywords and Other Reserved Words on page 79.
Chapter 4  

Continuous Queries

This chapter focuses on the features of continuous queries.

Topics

- Overview of Continuous Queries, page 40
- Working With Implicit Windows, page 42
- Working With Sliding, Tumbling, and Time Windows, page 44
- Explicit Window Example (Cache Query), page 45
- Sliding Window Examples (Cache Queries), page 47
- Tumbling Window Examples (Cache Queries), page 49
- Time Window Examples (Cache Queries), page 50
- Optimizing the Design, page 51
Overview of Continuous Queries

When used in a query agent deployed in a TIBCO BusinessEvents cluster, continuous queries 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 starts 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.

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

What makes a query run as a continuous query? A continuous query must be executed 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 on page 40 for more details.

Executing 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 Using Data from a Callback Rule Function on page 28. and See Two Types of Queries—Snapshot and Continuous on page 3 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.
Understanding 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.

Types of Windows

See Working With Sliding, Tumbling, and Time Windows on page 44 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 on page 42.

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) on page 47.

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) on page 49.

Time Window The policy clause specifies a time period during which entities remain in the window. See Time Window Examples (Cache Queries) on page 50.
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

```
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 on page 12 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.

Implicit Window Examples

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

The above query joins Department and Student entities using the department name. It then keeps a count and an average of age of students per department.
The above 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.
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 on page 17.

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.

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)

If you are familiar with SQL, you know that the order in which the clauses are presented in a query string is not the order in which they are processed. For example, here is a fairly complex example formatted to make each clause clear:

```sql
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 Figure 1, How a Query String is Processed, on page 46:

1. `from` (including stream clause)
2. `where`
3. `group by` (including `having`)
4. `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
   1 2 3 4 5
GOOG
   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).

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

The above 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; emit: dead} car;
```

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

The above 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 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 query above 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 a result, the query correctly maintains a rolling average and count over the last 30 events, per symbol.

```sql
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;
```

The `by` and `group by` clauses in the above 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.
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.

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

The above 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.

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

The query above 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 to 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.

```
emit: new
```

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.

**Dummy groups** The above 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 on page 12 for more details.
Time Window Examples (Cache Queries)

Time windows use a stream policy that specifies how long an entity remains in the window. See Stream Policy on page 17.

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.

**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.

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

The above query holds *PizzaOrderEvents* for 45 minutes after the *OrderTime* in a time window.

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

*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.

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

The above 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.
Optimizing the Design

It is important to be aware of the following points 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 on page 22 for more details)

For example, you could create a query in a startup function. That query may use bind variables, for more flexibility (see Using Bind Variables in Query Text on page 37. 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.

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. See TIBCO BusinessEvents Administration 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 $\text{age} > 95$ is a more selective operator than $\text{location} = \text{"CA"}$.

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

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 on page 51 for more on this). This section explains how to use indexing if Oracle Coherence is the cache provider.

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.

For example, suppose you have a query like this:

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

You would use `Coherence.C_index()` to index the `age` property for the Customer cache. The format and an example follow:

```
C_Index(String cacheName, Object property, boolean isOrdered)
```

```
Coherence.C_Index(Coherence.C_CacheName("/Customer"), "age", true)
```

The cache manager then builds an index called `customer` using the `age` property.

---

#### How to Use Indexes

- Add indexes using startup rule functions in any inference engine in the cluster. You can index more than one of an entity type’s properties.

- Indexing is used for simple `WHERE` clause expressions like this:

  ```
  where c.age > 95.
  ```

  It doesn’t work for expressions like this:

  ```
  where /MyFunctions/roundup(c.age) > 95
  ```

- When indexing is used, memory use will also increase.

---

### 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 (see Optimize WHERE Clause Expressions on page 51). This makes the join more efficient.
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:

\[\geq, \leq, <, >, =, \neq, \text{In}, \text{Between}, \text{And}, \text{Or}, \text{Not}, \text{Like}\]

**Example**

In the example below, 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 (\(t\text{.securityId} = t\text{.symbol}\)).

```sql
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 example above, the OrderEvent is deleted from the cache even before the create-notification is processed by the query, so the OrderEvent can’t be copied into the query agent’s private cache.

Keep such situations in mind as you design your queries.
Chapter 5  

Event Stream Processing (ESP) Queries

Chapter 4, Continuous Queries, on page 39 focused on cache queries, though general points also apply to event stream processing (ESP) queries.

This chapter explains how to configure a query agent to process events arriving through a channel, using continuous queries.

Topics

- Event Stream Processing Overview, page 56
- Some ESP Query Use Cases, page 58
- Configuring a Standalone ESP Project, page 60
Event Stream Processing 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 could send out summary information in an 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 \texttt{accept: new} in the stream policy clause (see Stream Clause, page 16). For example:

\begin{verbatim}
select count(*) as theCount from /InferenceOntology/DirectToQueryEvent {policy: maintain last 10 seconds; accept: new} as dtq
  group by 1;
\end{verbatim}

\begin{verbatim}
select sum(currentCount) as theSum from /QueryOntology/Level2QueryEvent {policy: maintain last 25 seconds; accept: new} as l2q
  group by 1;
\end{verbatim}

An example project demonstrating ESP queries is provided in the directory \texttt{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.

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 be 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 `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

This section assumes familiarity with continuous queries. You can elaborate and apply the kinds or cases described below to your needs.

Map and Reduce

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

1. A message arrives at a destination that transforms it into an event of a certain type.
2. A query configured to listen for events of that type (as specified in the query’s from clause) then executes:
3. In the callback rule function for the above query, you would perform some kind of mapping operation. You would create event instances and then assert them using the following function:
   
   ```java
   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.

4. 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.

5. 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:

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

As a result you can process multiple streams of events in parallel.
Configuring a Standalone ESP Project

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 Administration guide.

However, if you are running a standalone node for ESP queries only, 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.

**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.
Chapter 6  Query Language Reference

The syntax diagrams in this section show the structure of a query and of each clause in a query. Operators and other items used in the syntax diagrams (except standard SQL terms) are also defined in this chapter.

Topics

- Miscellaneous Terms Used in Query Syntax Diagrams, page 62
- Query Syntax, page 63
- Expression Syntax, page 65
- Operators for Unary Expressions, page 72
- Operators for Binary Expressions, page 73
- Operators for Other Expressions, page 76
- Wildcards, Datatypes, Literals, Identifiers, and Keywords, page 77
Miscellaneous Terms Used in Query Syntax Diagrams

The following table defines terms used in syntax diagrams shown in Chapter 6, Query Language Reference, on page 61, and that don’t fall into categories documented in other tables, such as operators.

Table 3 Miscellaneous Terms Used in Query Syntax Diagrams

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias</td>
<td>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.)</td>
</tr>
<tr>
<td>entity</td>
<td>Use the fully qualified ontology name of an entity, with its project path. From /concepts/customer data. Remember that names are case sensitive.</td>
</tr>
<tr>
<td>time unit</td>
<td>Allowable time units are as follows: milliseconds, seconds, minutes, hours, days.</td>
</tr>
</tbody>
</table>

Reading Query Language Syntax Diagrams

The syntax diagrams in this section 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.
Query Syntax

The top level syntax for a query is as follows:

```
select clause  from clause  where clause  group by clause  order by clause ;
```

**Select Clause**

```
select  limit  distinct  as  expression  as  *,
```

**From Clause**

```
from  entity type path  as  alias,
```

**Where Clause**

```
where  boolean expression
```

**Group by Clause**

```
group by  expression  having  boolean expression
```
Order By Clause

Limit

Stream Clause

Stream Policy
Expression Syntax

Expression

```
boolean expression

DateTime expression

entity expression

number expression

String expression
```

Boolean Expression

```
between expression

comparison expression

in expression

logical expression

Identifier-Dependent Expression
(of type boolean)
```

Between Expression

```
DateTime expression between DateTime expression and DateTime expression

number expression between number expression and number expression
```
Comparison Expression

In Expression

Logical Expression
**DateTime Expression**

- **DateTime literal**
- **min** (DateTime expression)
- **max** (DateTime expression)

**Entity Expression**

- **Identifier-Dependent Expression** (of type DateTime)

**Number Expression**

- **int expression**
- **long expression**
- **double expression**
Int Expression
Long Expression

- long literal
- int expression
- abs (long expression)
- long expression + long expression
- long expression - long expression
- long expression * long expression
- min(long expression)
- max(long expression)

Identifier-Dependent Expression (of type long)
Double Expression

String Expression
Identifier-Dependent Expression

- **identifier**
  - (of an array) 
  - int expression

- **entity expression**
  - . identifier (of a property)

- **identifier** (of a function)
  - ( expression )
Operators for Unary Expressions

Table 4  Operators for Unary Expressions in Queries

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description and Examples</th>
<th>Datatypes</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>not</td>
<td>Negation</td>
<td>x must be a Boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td>not x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>abs</td>
<td>absolute value</td>
<td>x must be a number</td>
<td>The type of the operand</td>
</tr>
<tr>
<td></td>
<td>abs x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>unary plus</td>
<td>x must be a number</td>
<td>The type of the operand</td>
</tr>
<tr>
<td></td>
<td>+ x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>unary minus</td>
<td>x must be a number</td>
<td>The type of the operand</td>
</tr>
<tr>
<td></td>
<td>- x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>()</td>
<td>Group (that is, parentheses)</td>
<td>Any</td>
<td>The type of the operand</td>
</tr>
<tr>
<td></td>
<td>(a+b)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Operators for Binary Expressions

### Table 5  Operators for Binary Expressions in Queries  (Sheet 1 of 3)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description and Examples</th>
<th>Datatypes</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Relational Expression Operators</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>equality</td>
<td>x and y can be any type</td>
<td>Boolean</td>
</tr>
<tr>
<td>x = y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td>inequality</td>
<td>x and y can be any type.</td>
<td>Boolean</td>
</tr>
<tr>
<td>x &lt;&gt; y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>x and y must both be number types, or both be Datetime types.</td>
<td>Boolean</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x &gt; y (and so on)</td>
<td>Generically known as comparison operators</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Logical Operators</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>Logical (Boolean) and, or.</td>
<td>x and y must be Boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>or</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x and y</td>
<td></td>
<td>Boolean</td>
<td></td>
</tr>
<tr>
<td>x or y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Mathematical Operators</strong></td>
<td>Also used in the projection (select clause)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>x and y must both be numbers.</td>
<td>Either the type of x or y, whichever has the larger capacity.</td>
</tr>
<tr>
<td>x * y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\</td>
<td>Division</td>
<td>x and y must both be numbers.</td>
<td>double</td>
</tr>
<tr>
<td>x \ y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 5  Operators for Binary Expressions in Queries  (Sheet 2 of 3)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description and Examples</th>
<th>Datatypes</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mod</code></td>
<td>Remainder</td>
<td>x and y must both be numbers.</td>
<td>Either the type of x or y, whichever has the larger capacity.</td>
</tr>
<tr>
<td></td>
<td><code>x mod y</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>+</code></td>
<td>Addition</td>
<td>x and y must both be numbers.</td>
<td>Either the type of x or y, whichever has the larger capacity.</td>
</tr>
<tr>
<td></td>
<td><code>x + y</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>-</code></td>
<td>Subtraction</td>
<td>x and y must both be numbers.</td>
<td>Either the type of x or y, whichever has the larger capacity.</td>
</tr>
<tr>
<td></td>
<td><code>x - y</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Postfix Operators

| `[`      | Array dereferencing, to access an array element. | x must be an array and y must be an int. | Type of the array element. |
|          | `x[y]`                                           |                                           |                           |
| `.`      | For object graph traversal, to access a property | x must be an entity and y must be a property. | Type of y. |
|          | `x.y`                                            |                                           |                           |
| `@`      | For object graph traversal, to access an attribute | x must be an entity and y must be a attribute. | Type of y. |
|          | `customer@extId`                                  |                                           |                           |

### String Operator

| `||`     | String concatenation | x and y must be String | String |
|          | `x || y`             |                       |        |

---

TIBCO BusinessEvents Event Stream Processing Query Developer's Guide
The like operator matches all strings that match the regular expression provided in double quotes. The regular expression syntax is the same as Java RegEx.

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description and Examples</th>
<th>Datatypes</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>like</td>
<td>The like operator matches all strings that match the regular expression provided in double quotes. The regular expression syntax is the same as Java RegEx.</td>
<td>String</td>
<td>Boolean</td>
</tr>
</tbody>
</table>
### Operators for Other Expressions

#### Table 6  Operators for Other Expressions in Queries

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description and Examples</th>
<th>Datatypes</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>between and</strong></td>
<td>Between operator for range expressions. Range is inclusive. x between y and z</td>
<td>x and y must all be number types, or all be Datetime types.</td>
<td>Boolean</td>
</tr>
<tr>
<td><strong>in()</strong></td>
<td>Inclusion operator. Checks if an expression is in a group of items. x in (y1, y2, ..., yN)</td>
<td>Any</td>
<td>Boolean</td>
</tr>
<tr>
<td><strong>$</strong></td>
<td>Bind variable prefix. $name has no type. It is just a label.</td>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>
Wildcards, Datatypes, Literals, Identifiers, and Keywords

Wildcard Characters

- The asterisk (*) is a wild card 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 data types described below, plus the following:

- hex
- octal
- char

Types and Literals

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax of Literals</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>A signed integer expressed using only digits and an optimal sign prefix. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive).</td>
<td>1234567</td>
</tr>
<tr>
<td>long</td>
<td>A signed integer expressed using only digits 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).</td>
<td>digits 1234567</td>
</tr>
<tr>
<td>double</td>
<td>A double-precision 64-bit IEEE 754 floating point.</td>
<td>12345.67 1.234e+56</td>
</tr>
</tbody>
</table>
Table 7  Query Language Types and Literals

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax of Literals</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>String literals are surrounded by double quotes.</td>
<td>&quot;hello&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;She says:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;Hello.&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;c:\temp\myfile&quot;</td>
</tr>
<tr>
<td></td>
<td>To escape double quote and backslash characters,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>prefix them with a backslash.</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>The boolean data type has only two possible values:</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>true and false. Use for simple flags that track true</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>and false conditions.</td>
<td></td>
</tr>
<tr>
<td>DateTime</td>
<td>yyyy-MM-dd'T'HH:mm:ss.SSSZ</td>
<td>2008-04-23T13:30:25.123-0700</td>
</tr>
<tr>
<td></td>
<td>where</td>
<td></td>
</tr>
<tr>
<td></td>
<td>yyyy: four digit year</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MM: two digit month</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dd: two digit day of month</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HH: two digit hour of day in 24 hour format</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mm: two digit minutes in hour</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ss: two digit seconds in minute</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sss: three digit milliseconds in second</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'T': the letter T</td>
<td></td>
</tr>
<tr>
<td></td>
<td>z: timezone expressed as defined in RFC 822.</td>
<td></td>
</tr>
<tr>
<td>Entity type</td>
<td>&quot;entity-project-path&quot;</td>
<td>=&quot;/a/b/MyConcept&quot;</td>
</tr>
<tr>
<td></td>
<td>Entity project path begins with a forward slash and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>folders are separated with a forward slash.</td>
<td></td>
</tr>
<tr>
<td>Entity</td>
<td>No literal is used for entity instances.</td>
<td>(Not applicable)</td>
</tr>
</tbody>
</table>

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 Keywords and Other Reserved Words in TIBCO BusinessEvents Developer’s 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

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

Examples:

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

Symbols

# (pound sign) used to escape keywords 79

B

batching of return values 34
BE_HOME xv
between expression 65
binary expressions, operators for 73
bind variables 21
  in query text 37
boolean expression 65

C

callback rule functions 25
  required signature 29
  using data from 28
changes from the previous release of TIBCO BusinessEvents Event Stream Processing Query Developer’s Guide x
closing a statement 26
comparison expression 66
concepts, when not to use in explicit windows 44
continuous queries
  executing 40
  overview 40
  simple example 33
create the query definition 22
customer support xviii

d

datatypes 77
datetime expression 67
deleting a query definition 26
double expression 70

effect of the cache on continuous queries 53
effect of time on queries 53
emit
  dead 16, 50
  new 16, 49
ending a continuous query 40
entity expression 67
ENV_HOME xv
escaping the keywords 79
execute an instance of the query statement and obtain results 24
executing a continuous query 40
explicit windows, example 45
expression 65
  syntax 65

F

filtering for efficient joins 52
from clause 10
  and variables 10
functions
  that can’t be used in queries 21
  used to create and execute queries 4
  within queries 20
functions documentation, accessing xiv
G

garbage collection tips for ESP query agents 60

group by clause 12

I

identifier 78

identifier-dependent expression 71

implicit windows 42

examples 42

improve performance by pre-fetching objects 51

in expression 66

inference agents 6

int expression 68

L

lifecycle of a query 22

like operator syntax 75

limit 15, 64

limitation in use of arrays 21

literals 77, 77

logical expression 66

long expression 69

N

number expression 67

P

pre-filter 6

Q

queries are executed in query agents 2

queries retrieve information from cache 2

query agents 6

local cache 54

query as a pre-filter 6

query features overview 2

query function catalog 20

query statement, open 23

query string, how processed 46

query syntax 63

terms used in diagrams 62

query windows 41

Query.ResultSet.getRowCountIfPossible() 27

Query.Statement.executeWithBatch Callback() 36

Query.Statement.executeWithBatchCallback() 25, 28, 37, 40

Query.Statement.executeWithCallback 33, 37

Query.Statement.executeWithCallback() 25, 28, 40

querying the cache and using query results 20

R

reserved words 79

result set 24

using data from 27

rules

triggering queries from 6
S

select clause 8, 63
select statement, structure of 4
set bind variables (if used) 23
sliding windows 44
  examples 47
snapshot queries, example 32
stream clause 16, 64
stream policy 17, 64
string expression 70
structure of a query select statement 4
support, contacting xviii
syntax diagrams 62

T

technical support xviii
TIBCO_HOME xv
time windows 44
  examples 50
triggering a query from a rule 6
tumbling windows 44
  examples 49
types and literals 77
types of windows 41

U

unary expressions, operators for 72
use of query functions 22

W

where clause 11, 63
wildcard characters 77
wildcards, datatypes, and literals 77
windows, types of 41