

TIBCO BusinessEvents™

Language Reference

*Software Release 3.0.1
November 2008*

The Power to Predict™

 **TIBCO®**
The Power of Now®

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 LICENSE.PDF) 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.

TIB, TIBCO, TIBCO Software, TIBCO Adapter, Predictive Business, Information Bus, The Power of Now, The Power to Predict, TIBCO BusinessEvents, TIBCO ActiveMatrix BusinessWorks, TIBCO Rendezvous, TIBCO Enterprise Message Service, TIBCO PortalBuilder, TIBCO Administrator, TIBCO Runtime Agent, TIBCO General Interface, and TIBCO Hawk are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and/or other countries.

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

Excerpts from Oracle Coherence documentation are included with permission from Oracle and/or its affiliates. Copyright © 2000, 2006 Oracle and/or its affiliates. All rights reserved.

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 © 2004-2008 TIBCO Software Inc. ALL RIGHTS RESERVED.

TIBCO Software Inc. Confidential Information

Contents

Preface	vii
Enterprise Suite and Inference Edition Features	viii
Related Documentation	ix
TIBCO BusinessEvents Documentation	ix
Other TIBCO Product Documentation	x
Typographical Conventions	xi
How to Contact TIBCO Support	xiv
 Chapter 1 Rule Language Grammar	 1
Rule Language Basics	2
Whitespace	2
Comments	2
Separators	3
Identifiers (Names)	3
Local Variables	4
Keywords and other Reserved Words	5
Literals	6
Escape Sequences	7
Operators	7
Rule Components	8
Attributes	9
Accessing Concept and Event Properties	11
Concept Property Atom	11
Concept Property Array	12
Event Property	13
Exception Handling	14
Syntax	14
Examples	15
Flow Control	17
if/else	17
for	17
while	18
 Chapter 2 Working With Rule Language Datatypes	 19
Concept Properties to XML Datatype Conversions	20

Compatibility of Operators with Types	21
Correcting Inconsistencies of Type	23
String Operands	23
Arithmetic Expressions	23
Assignment Conversion	24
Function Argument Conversion	24
Chapter 3 Rule Language Syntax	25
Rule Language Syntax	26
Chapter 4 Creating Custom Functions.	29
Overview of Creating Custom Functions.	30
Restrictions	30
Task Summary	31
Structure of a Function Catalog	32
Elements	32
Example Function Catalog	34
Java Archive Resource	36
Chapter 5 Query Features Overview.	37
Query Features Overview	38
Queries are Executed in Query Agents	38
Queries Retrieve Information from Cache	38
Two Main Types of Queries	38
Structure of a Query Select Statement	39
Summary of Functions Used to Create and Execute Queries	39
For More Information	40
Two Common Ways to Use Queries	41
Triggering a Query from a Rule (in an Inference Agent)	41
Using a Query as a Pre-filter	41
Chapter 6 Query Language Components.	43
Select Clause	44
From Clause	45
Where Clause	46
Group by Clause	47
Order by Clause	48
Limit Clause	49
Stream Clause	50
Stream Policy	51

Chapter 7 Working With the Query Language	53
Querying the Cache and Using Query Results	54
Query Function Catalog	54
Using Functions Within Queries	54
Using Bind Variables	55
Limitation in Use of Arrays	55
Lifecycle of a Query—Use of Query Functions	56
Create the Query Definition	56
Open a Query Statement	57
Set Bind Variables (if Used)	57
Execute an Instance of the Query Statement and Obtain Results	58
Closing a Statement and Deleting a Query Definition	60
Using Data from a Result Set	61
Using Data from a Callback Rule Function	62
The Callback Rule Function Required Signature	62
Simple Snapshot Query Example	64
Simple Continuous Query Example	65
Example Showing Batching of Return Values	66
Using Bind Variables in Query Text	69
 Chapter 8 Working With Continuous Queries	 71
Overview of Continuous Queries	72
Executing a Continuous Query	72
Ending a Continuous Query	72
Understanding Query Windows	72
Working With Implicit Windows	74
Implicit Window Examples	74
Working With Sliding, Tumbling, and Time Windows	76
Use Sliding, Tumbling, or Time Windows for Events and not Concepts	76
Explicit Window Example	77
Sliding Window Examples	79
Tumbling Window Examples	81
Time Window Examples	82
Optimizing the Design	83
Reuse Existing Queries and Statements Whenever Possible	83
Improve Performance by Pre-fetching Objects	83
Use Filtering for Efficient Joins	83
Effect of the Cache on Continuous Queries	84
Effect of Time on Queries	84

Chapter 9 Query Language Reference 87

Miscellaneous Terms Used in Query Syntax Diagrams 88

 Reading Query Language Syntax Diagrams 88

Query Syntax 89

 Select Clause 89

 From Clause 89

 Where Clause 89

 Group by Clause 89

 Order By Clause 90

 Limit 90

 Stream Clause 90

 Stream Policy 90

Expression Syntax 91

 Expression 91

 Boolean Expression 91

 DateTime Expression 93

 Entity Expression 93

 Number Expression 93

 String Expression 96

 Identifier-Dependent Expression 97

Operators for Unary Expressions 98

Operators for Binary Expressions 99

Operators for Other Expressions 101

Wildcards, Datatypes, and Literals 102

 Wildcard Characters 102

 Datatypes 102

 Literals 102

 Types and Literals 102

 Identifier 103

Reserved Words 104

 Escaping the Keywords 104

Index 105

Preface

TIBCO BusinessEvents™ allows you to abstract and correlate meaningful business information from the data flowing through your information systems and take appropriate action using business rules. By detecting complex patterns within the real-time flow of simple events, BusinessEvents™ can help you to detect and understand unusual activity, recognize trends, problems, and opportunities. BusinessEvents delivers this business critical information in real time to your critical enterprise systems or custom dashboards. With BusinessEvents you can predict the needs of your customers, make faster decisions, and take faster action.

BusinessEvents
The Power to Predict™

Topics

- *[Enterprise Suite and Inference Edition Features, page viii](#)*
- *[Related Documentation, page ix](#)*
- *[Typographical Conventions, page xi](#)*
- *[How to Contact TIBCO Support, page xiv](#)*

Enterprise Suite and Inference Edition Features

BusinessEvents is available in the Inference Edition and in the Enterprise Suite. The components available in each option are listed below.

Inference Edition and Enterprise Suite

Inference Edition provides inferencing features and comprises the following components (also included in Enterprise Suite):

- **Server**—The BusinessEvents runtime engine.
- **Workbench**—A TIBCO Designer[™] palette of BusinessEvents resources.
- **TIBCO BusinessWorks 5.x Plug-in**—A TIBCO Designer palette of activities that enables communication between BusinessEvents and BusinessWorks[™]. (When you select this option, BusinessEvents Workbench and Server are also automatically selected.)
- **Documentation**—TIBCO BusinessEvents documentation. The doc folder contains an HTML and a PDF folder. If you do not install documentation, this folder is not included in the installation.

Enterprise Suite Only

All of the above components plus the following:

- **Decision Manager application**—A business user rule-building application.



The Decision Manager application is available only on Windows.

- **Rules Management Server**—A rules server for the Decision Manager application.
- **Query**—A language and set of functions for querying cache data.
- **Database Concepts**—A utility for creating concepts from database metadata, with functions for updating the associated database tables or views.
- **State Modeler**—A component that enables you to model the life cycle of concept instances.

Related Documentation

This section lists documentation resources you may find useful.

TIBCO BusinessEvents Documentation

- *TIBCO BusinessEvents Installation*: Read this manual for instructions on site preparation and installation.
- *TIBCO BusinessEvents Getting Started*: After the product is installed, use this manual to learn the basics of BusinessEvents. This guide provides step-by-step instructions to implement an example project and also explains the main ideas so you gain understanding as well as practical knowledge.
- *TIBCO BusinessEvents User's Guide*: Read this manual for instructions on using TIBCO BusinessEvents to create, manage, and monitor complex event processing projects.
- *TIBCO BusinessEvents Decision Manager*: This manual explains how to use decision tables to create rules using a spreadsheet-like interface, as well as how to administer the Rules Management Server.
- *TIBCO BusinessEvents Language Reference*: This manual provides reference and usage information for the BusinessEvents rule language and the BusinessEvents query language.
- *TIBCO BusinessEvents Cache Configuration Guide*: This online reference is available from the HTML documentation interface. It provides configuration details for cache-based object management. Cache-based object management is explained in *TIBCO BusinessEvents User's Guide*.
- *TIBCO BusinessEvents Java API Reference*: This online reference is available from the HTML documentation interface. It provides the Javadoc-based documentation for the BusinessEvents API.
- *TIBCO BusinessEvents Functions Reference*: This online reference is available from the HTML documentation interface. It provides a listing of all functions provided with BusinessEvents, showing the same details as the tooltips available in the TIBCO Designer rule editor interface.
- *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.

Other TIBCO Product Documentation

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

- TIBCO BusinessWorks™
- TIBCO Rendezvous®
- TIBCO Enterprise Message Service™
- TIBCO Designer™
- TIBCO Hawk™
- TIBCO Runtime Agent™

Typographical Conventions

The following typographical conventions are used in this manual.

Table 1 General Typographical Conventions

Convention	Use
<i>TIBCO_HOME</i> <i>BE_HOME</i>	<p>Many TIBCO products must be installed within the same home directory. This directory is referenced in documentation as <i>TIBCO_HOME</i>. The value of <i>TIBCO_HOME</i> depends on the operating system. For example, on Windows systems, the default value is C:\tibco.</p> <p>Other TIBCO products are installed into an installation environment. Incompatible products and multiple instances of the same product are installed into different installation environments. The directory into which such products are installed is referenced in documentation as <i>ENV_HOME</i>. The value of <i>ENV_HOME</i> depends on the operating system. For example, on Windows systems the default value is C:\tibco.</p> <p>TIBCO BusinessEvents installs into a version-specific directory within <i>TIBCO_HOME</i>. This directory is referenced in documentation as <i>BE_HOME</i>. The value of <i>BE_HOME</i> depends on the operating system. For example on Windows systems, the default value is C:\tibco\be\3.0.</p>
code font	<p>Code font identifies commands, code examples, filenames, pathnames, and output displayed in a command window. For example:</p> <p>Use MyCommand to start the foo process.</p>
bold code font	<p>Bold code font is used in the following ways:</p> <ul style="list-style-type: none"> • 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]

Table 1 General Typographical Conventions (Cont'd)




Convention	Use
<i>italic font</i>	Italic font is used in the following ways: <ul style="list-style-type: none">• To indicate a document title. For example: See <i>TIBCO BusinessWorks Concepts</i>.• To introduce new terms For example: A portal page may contain several <i>portlets</i>. Portlets are mini-applications that run in a portal.• To indicate a variable in a command or code syntax that you must replace. For example: <code>MyCommand <i>pathname</i></code>
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

Convention	Use
[]	An optional item in a command or code syntax. For example: <code>MyCommand [optional_parameter] required_parameter</code>
	A logical 'OR' that separates multiple items of which only one may be chosen. For example, you can select only one of the following parameters: <code>MyCommand param1 param2 param3</code>

Table 2 Syntax Typographical Conventions

Convention	Use
{ }	<p>A logical group of items in a command. Other syntax notations may appear within each logical group.</p> <p>For example, the following command requires two parameters, which can be either the pair param1 and param2, or the pair param3 and param4.</p> <pre>MyCommand {param1 param2} {param3 param4}</pre> <p>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:</p> <pre>MyCommand {param1 param2} {param3 param4}</pre> <p>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.</p> <pre>MyCommand param1 [param2] {param3 param4}</pre>

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 **Rule Language Grammar**

This chapter describes the grammar for TIBCO BusinessEvents rules.

Topics

- *[Rule Language Basics, page 2](#)*
- *[Attributes, page 9](#)*
- *[Accessing Concept and Event Properties, page 11](#)*
- *[Exception Handling, page 14](#)*
- *[Flow Control, page 17](#)*

Rule Language Basics

Whitespace

Whitespace is used to separate tokens (identifiers, keywords, literals, separators, and operators) just as it is used in any written language to separate words. Whitespace is also used to format code.

These are whitespace characters, excluding line terminators:

- the ASCII SP character, also known as "space"
- the ASCII HT character, also known as "horizontal tab"
- the ASCII FF character, also known as "form feed"

Line terminators include these characters:

- the ASCII LF character, also known as "newline"
- the ASCII CR character, also known as "return"
- the ASCII CR character followed by the ASCII LF character

Line terminators are not significant in condition or action rules.

Comments

Comment rules as shown:

/ text */* BusinessEvents ignores the text from *"/"* to *"*/"*.

// text BusinessEvents ignores the text from *"/"* to the end of the line.

Separators

The following tokens are used for separators:

- ; Statement separator for conditions and actions.
- (Expression Grouping begin, or function argument list begin.
-) Expression Grouping end or function argument list end.
- , Argument list and statement expression list separator.

Identifiers (Names)

An identifier (or *name*, to use the user interface label) is an unlimited-length sequence of letters and digits, the first of which must be a letter. Letters include uppercase and lowercase ASCII Latin letters A-Z, a-z, and the underscore (_).



Do not use the dollar sign (\$).

Letters and digits may be drawn from the entire Unicode character set, which supports most writing scripts in use in the world today, including the large sets for Chinese, Japanese, and Korean. This allows programmers to use identifiers in their programs that are written in their native languages.

Digits include the ASCII digits 0-9.

Two identifiers are the same only if they have the same Unicode character for each letter or digit. Note that some letters look the same even though they are different Unicode characters. For example, a representation of the letter A using \u0041 is not the same as a representation of the letter A using \u0391.

Example identifiers:

- new_order
- E72526
- creditCheck

Identifiers may not be the same as any literal, keyword, or other reserved word. See [Keywords and other Reserved Words on page 5](#) and [Literals on page 6](#).

Local Variables

You can use local variables of the following types in action rules and rule functions:

- Primitives
- Concepts
- Event definitions

You can also use primitive arrays, which are fixed length. Here are examples of array declaration, initialization, and array creation expressions:

- Array declaration and initialization:

```
int i;                // int
int[] ii = {1,2,i};  // array of int
```

- Array creation with initialization expression:

```
ii = int[] {1,2,3};
```

- Array creation without initialization expression:

```
int[] arr = int[5] {};  
arr = int[5]{};
```

- Getting the length of the array:

```
int arrLength = arr@length;
```

Keywords and other Reserved Words

Keywords are words that have special meaning to BusinessEvents. Keywords are not available for use as identifiers. Other words are used internally by BusinessEvents. These words are also not available for use as identifiers. *Do not* use the words listed in this section as identifiers.

The following words are for BusinessEvents internal use. Do not use them as identifiers, resource names, or folder names.

attribute	moveto	requeue	then
declare	priority	rule	when

The following reserved words are either implemented as keywords in BusinessEvents or may be implemented in the future. Do not use them as identifiers, resource names, or folder names.

abstract	double	interface	switch
boolean	else	long	synchronized
break	Event	native	this
byte	extends	new	throw
case	final	package	throws
catch	finally	private	TimeEvent
char	float	protected	transient
class	for	public	try
const	goto	return	void
Concept	if	short	volatile
ContainedConcept	implements	SimpleEvent	while
continue	import	static	
default	instanceof	strictfp	
do	int	super	

Literals

The BusinessEvents rule Language supports these literals:

- **Integer** — One or more digits without a decimal. May be positive or negative.
Examples: `4` `45` `-321` `787878`
- **Long** — An integer literal suffixed with the letter L. The suffixed L can be either upper or lower case, but keep in mind that the lower case L (l) can be difficult to distinguish from the number one (1).
Examples: `01` `0777L` `0x100000000L` `2147483648L` `0xC0B0L`
- **Double** — A number that has a decimal. D suffix is optional unless there is no decimal point or exponent.
Examples: `1D` `1e1` `2.` `.3` `0.0D` `3.14` `1e-9d` `1e137`
- **String** — Zero or more characters enclosed in double quotes ("). Opening and closing double quotes cannot be interrupted by a line terminator (CR or LF). Use the plus sign (+) to concatenate string segments.
Examples: `" " " " "P0QSTN3" "The quick brown fox had quite a feast!"`
`"The quick brown fox " +`
`"had quite a feast!"`
- **Boolean** — One of these two values: `true` `false`
- **Null** — This value: `null`



Do not use Unicode escapes for newlines or carriage returns as character literals. They will be transformed into actual line terminators. Use `'\n'` or `'\r'` instead.

The characters CR and LF are line terminators, not input characters.

Escape Sequences

You can represent single and double quotes, the backslash character, and some nongraphic characters in literals using these escape sequences:

Table 3 *Escape Sequences*

Character	Escape Sequence
backspace	\b
horizontal tab	\t
linefeed	\n
form feed	\f
carriage return	\r
double quote	\"
single quote	\'
backslash	\\

Operators

The language defines the following operators:

<code>+</code> , <code>-</code> (unary)	unary plus, unary minus
<code>*</code> , <code>\</code> , <code>%</code>	multiplication, division, remainder
<code>+</code> , <code>-</code>	addition, subtraction
<code>></code> , <code><</code> , <code>>=</code> , <code><=</code>	greater than, less than, greater than or equal to, less than or equal to
<code>instanceof</code>	Tests whether an object is an instance of specified type. Restricted to use with concepts and events.

Example:

```
boolean b = customer instanceof USCustomer;
```

<code>==</code> , <code>!=</code>	equality, inequality
<code>&&</code> , <code> </code> , <code>!</code>	boolean AND, OR, NOT
<code>=</code>	assignment
<code>.</code>	property access
<code>@</code>	attribute access

Rule Components

A TIBCO BusinessEvents rule has three components:

- **Declaration** — Use the declaration to declare which concepts and events the rule will depend on, and the names by which instances of these entities can be referred to in the conditions and actions. Aliases must be valid identifiers. Declaring multiple terms of the same type allows the rule to consider multiple instances of the corresponding Entity.
- **Conditions** — Each statement in the condition must evaluate to a boolean value. All of these statements must be true for the rule's action to be executed. Assignments and calls to certain functions are disallowed in the condition.
- **Actions** — List of statements that will be executed, when the rule is fired, for each combination of terms that matches all the conditions.

Attributes

You can use the attributes described in [Table 4](#) in rules to return information about an entity instance. These attributes are also documented in *TIBCO BusinessEvents User's Guide* for convenience.

Table 4 Attributes

Entity	Attributes	Type	Returns
Event	@id	long	The event's unique internal ID.
	@extId	string	The event's unique external ID.
	@ttl	long	The time to live of the event as specified in the configuration. (This is not the time-to-live remaining.)
	@payload	string	The payload as a string value.
Repeating TimeEvent	@id	long	The time event's unique internal ID.
	@closure	string	null.
	@interval	long	The number of units between creation of successive time events.
	@scheduledTime	dateTime	The time scheduled for asserting into the WorkingMemory.
	@ttl	long	0.
Rule-Based TimeEvent	@id	long	The time event's unique internal ID.
	@closure	string	A string that was specified when the event was scheduled.
	@interval	long	0.
	@scheduledTime	dateTime	The time scheduled for asserting into the WorkingMemory.
	@ttl	long	The time to live of the event as specified when scheduling the event. (This is not the time-to-live remaining.)

Table 4 Attributes (Cont'd)

Entity	Attributes	Type	Returns
Advisory Event	id	long	The advisory event's unique internal ID
	extId	String	Null
	category	String	Broad category of advisory, for example, an exception.
	type	String	Type of advisory within the category.
	message	String	Message for the user.
Concept	@id	long	The concept instance's unique internal ID.
	@extId	string	The concept instance's unique external ID.
ContainedConcept	@id	long	The contained concept instance's unique internal ID.
	@extId	string	The contained concept instance's unique external ID.
	@parent	concept	The parent concept instance. (This is treated as a concept reference in the language.)
PropertyAtom	@isSet	boolean	True if the property value has been set. Otherwise, false.
PropertyArray	@length	int	The number of PropertyAtom entries in the array.



The internal ID is automatically generated by BusinessEvents. You cannot assign it.

Accessing Concept and Event Properties

This section describes how to access concept properties and event properties using the BusinessEvents language.

Concept Property Atom

This is the syntax for accessing a concept property atom:

instanceName.propertyName

where *instanceName* is the identifier of the concept instance, and *propertyName* is the name of the concept property that you want to access.

For example to get the current value of the cost propertyAtom:

```
int x = instanceA.cost;
```

For example, to set a value with the current system time stamp:

```
instanceA.cost = value;
```



If the history size is 0, BusinessEvents does not record a time stamp.

Get and Set PropertyAtom Value With User-Specified Time

You can get and set PropertyAtom values as follows:

- You can specify a time and get the PropertyAtom value stored in the history at that time using one of the standard functions:

```
type Instance.PropertyAtom.get(type(PropertyAtom propertyName, \
                                long time)
```

where *type* is the type of the PropertyAtom and *propertyName* is the name of the PropertyAtom, and *time* is the time from which you want to retrieve the value.

- You can set a value in the PropertyAtom History using one of the standard functions:

```
Instance.PropertyAtom.set(type(PropertyAtom propertyName, \
                                type value, long time)
```

where *type* is the type of the PropertyAtom and the type of the new value, *propertyName* is the name of the PropertyAtom, *value* is the value to store in the ring buffer, and *time* is the time stamp for the new entry.

BusinessEvents manages these requests as follows:

- **If the ring buffer has vacancies**, BusinessEvents inserts the new entry into the correct place based on its time stamp, shifts the older values out one place, and returns True.
- **If the ring buffer is full, and the new value has a more recent time stamp than the oldest value**, BusinessEvents inserts the new value into the correct place, shifts older values if necessary, drops the oldest value, and returns True.
- **If the ring buffer is full, and the new value has a time stamp that is older than the oldest value in the ring buffer**, BusinessEvents does not insert the new value into the ring buffer, and it returns False.

Concept Property Array

This is the syntax for accessing a concept property array:

instanceName.propertyName

where *instanceName* is the identifier of the concept instance, and *propertyName* is the name of the concept property that you want to access.

Accessing a Value in the Property Array

To access a value in a property array, identify the position in the array of the value as shown:

instanceName.propertyName[indexPosition]

For example:

```
String x = instanceA.lineItem[0];
```

This gets the current value of the first property atom in the array, *lineItem*, and assigns it to the local variable, *x*.



Array index difference In the BusinessEvents language, array indexes start from zero (0). However, in XSLT and XPath languages, they start from one (1). It's important to remember this difference when using the rule language in the rule editor, and when working in the XSLT mapper and the XPath builder.

Adding a Value to a Property Array

You can append a value to the end of a property array — you cannot add a value to any other position in an array. This is the syntax:

instanceName.propertyName[indexPosition] = value

To use the syntax shown above you must know the index position of the end of the array. You can append a value to the end of an array without knowing the index position of the end of the array using the `@length` attribute as shown:

```
instanceName.propertyName[instanceName.propertyName@length] = value
```

Event Property

This is the syntax for accessing an event property:

```
eventName.propertyName
```

For example:

```
String x = eventA.customer;
```

where *eventName* is the identifier of the concept instance and *propertyName* is the name of the event property that you want to access.

Exception Handling

The BusinessEvents rule Language includes an Exception type that has `try/catch/finally` commands to handle exceptions. The `try/catch/finally` commands behave like their same-name Java counterparts.



Advisory Events You can also use the special `AdvisoryEvent` event type to be notified of exceptions that originate in user code but that are not caught with the `catch` command. To use the `AdvisoryEvent`, click the plus sign used to add a resource to the declaration. `AdvisoryEvent` is always available in the list of resources.

This section describes the `try/catch/finally` commands.

Syntax

These combinations are allowed:

- `try/catch`
- `try/finally`
- `try/catch/finally`

```
try    try {
        try_statements
    }

catch catch (Exception identifier) {
        catch_statements
    }

finally finally {
        finally_statements
    }
```



When using the `catch` command, assignment of the Exception type is mandatory, and you are limited to one `catch` block.

Examples

This section provides some examples to demonstrate use of exception handling.

try/finally Example

```
String localStatus = "default status";
try {
    //readStatus might throw an exception
    localStatus = readStatus();
} finally {
    //If readStatus throws an exception,
    //ScoreCard.status will be set to "default status"
    //but the exception won't be caught here.
    //Otherwise ScoreCard.status will be set to the
    //return value of readStatus()
    Scorecard.status = localStatus;
}
```

try/catch/finally Example

```
String localStatus = "default status";
try {
    //readStatus might throw an exception
    localStatus = readStatus();
} catch(Exception exp) {
    System.debugOut("readStatus() threw an exception with message"
        + exp.getMessage());
} finally {
    //If readStatus throws an exception,
    //ScoreCard.status will be set to "default status"
    //Otherwise ScoreCard.status will be set to the
    //return value of readStatus()
    Scorecard.status = localStatus;
}
```

try/catch Example

```
String localStatus = "default status";
try {
    //readStatus might throw an exception
    localStatus = readStatus();
} catch(Exception exp) {
    System.debugOut("readStatus() threw an exception with message "
        + exp.getMessage());
}

//If readStatus throws an exception,
//ScoreCard.status will be set to "default status"
//Otherwise ScoreCard.status will be set to the
//return value of readStatus()
Scorecard.status = localStatus;
```

Flow Control

The BusinessEvents rule Language includes commands to perform conditional branching and iteration loops. This section describes these commands.

if/else

The `if/else` command allows you to perform different tasks based on conditions.

Syntax:

```
if(condition){  
  code_block;  
}  
else{  
  code_block;  
}
```

for

The `for` command allows you to create a loop, executing a code block until the condition you specify is false.

Syntax:

```
for(initialization; exit condition; incrementor){  
  code_block;  
  [break;]  
  [continue;]  
}
```

`break` allows you to break out of the loop.

`continue` allows you to stop executing the code block but continue the loop.

For example:

```
int i;  
for(i=0; i<10; i=i+1){  
  System.debugOut("Hello World!");  
}
```

This example prints "Hello World!" to debugOut ten times.

while

The `while` command allows you to perform one or more tasks repeatedly until a given condition becomes false.

Syntax:

```
while(condition){  
    code_block;  
    [break;]  
    [continue;]  
}
```

`break` allows you to break out of the loop.

`continue` allows you to stop executing the code block but continue the loop.

Chapter 2 **Working With Rule Language Datatypes**

This chapter provides datatype conversion tables, information about operators and types, and information about how TIBCO BusinessEvents handles inconstancy problems with datatypes.

Topics

- [*Concept Properties to XML Datatype Conversions, page 20*](#)
- [*Compatibility of Operators with Types, page 21*](#)
- [*Correcting Inconsistencies of Type, page 23*](#)

Concept Properties to XML Datatype Conversions

Table 5 Concept Properties to XML Datatype Conversions

Property Type	Int	Long	Float	Double	Boolean	String	DateTime	ComplexType	@ref
Int	L	L	L	L		L			
Long	N	L	N	N		L			
Double	N	N	N	L		L			
String	L	L	L	L	L	L	L		
Boolean					L	L			
Datetime						L	L		
ContainedConcept								D	
ConceptReference									ID

- N - Numeric conversion loss of information possible
- L - Shallow copy — Copies only the current value; not the history.
- D - Deep copy — Copies the entire structure of the contained concept (current value of property only).
- ID - Copies the ID of the referred concept.



Datatype conversion tables for events are located in the TIBCO Rendezvous and TIBCO Enterprise Message Service documentation.

Compatibility of Operators with Types

Table 6, Operator Matrix, defines the compatibility of operators with types.

Table 6 Operator Matrix

		Right Side of Operator							
		str	int	lon	dou	boo	ent	obj	dat
Left Side of Operator	str	=, +, eq, cmp, inst	+	+	+	+	+	=, +, eq, cmp, inst	+
	int	+	=, math, eq, cmp	=, math, eq, cmp	=, math, eq, cmp			=, math, eq, cmp	
	lon	+	=, math, eq, cmp	=, math, eq, cmp	=, math, eq, cmp			=, math, eq, cmp	
	dou	+	=, math, eq, cmp	=, math, eq, cmp	=, math, eq, cmp			=, math, eq, cmp	
	boo	+				=, eq		=, eq	
	ent	+					=, eq, inst	=, eq, inst	
	obj	=, +, eq, cmp, inst	=, math, eq, cmp	=, math, eq, cmp	=, math, eq, cmp	=, eq	=, eq, inst	=, eq, inst	=, eq, inst
	dat	+,						=, eq, inst	=, eq, cmp, inst

Abbreviation	Meaning and Notes
boo	Boolean.
cmp	Comparison operators: <, >, <=, >=

Abbreviation	Meaning and Notes
dat	Date/Time
dou	Double
ent	Entity. Type includes Concept, Event and ScoreCard. Both operands must either be of the same type or have a subtype-supertype relationship
eq	Equality operators: ==, !=
inst	instanceof
int	Integer
lon	Long
math	Numerical operators: unary +, unary -, =, -, *, /, %
obj	Object
str	String

Correcting Inconsistencies of Type

BusinessEvents attempts to correct inconsistencies of type whenever possible by converting expressions to the appropriate type. BusinessEvents converts expression types in the following cases:

- An expression uses the plus sign (+) with a string operand.
- An arithmetic expression includes numbers of differing types.
- The value of an expression is assigned to a variable of a different type.
- The arguments to a function are of the wrong type.

There are some inconsistencies of type that BusinessEvents cannot correct. For example, all expressions within conditions must be of type boolean. If an expression within a condition evaluates to anything other than boolean, it would be illogical for BusinessEvents to convert the expression to boolean. In cases like this, BusinessEvents returns an error at compile time.

String Operands

When an expression uses the plus sign (+) with a string operand, BusinessEvents treats the expression as a request for concatenation rather than addition. It converts the second operand to a string and concatenates the two strings.

For example:

```
"area code: " + 650 becomes
"area code: 650"
```

Arithmetic Expressions

The following information applies to these operators:

```
* / % + - < <= > = == !=
```

When an expression uses one of the above arithmetic operators with two numbers of different numeric types, BusinessEvents promotes one of the two operands to the numeric type of the other. It makes these promotions as follows:

- If either operand is a double, BusinessEvents promotes the other to a double.
- Otherwise, if either operand is a long, it promotes the other to a long.

Assignment Conversion

If the value of an expression is assigned to a variable, BusinessEvents converts the expression's type to that of the variable. This might include, for example, converting a double to an int, or converting a generic model type to a more specific model type.

Function Argument Conversion

Conversions of function arguments are handled in the same way as assignment conversions.

Chapter 3 **Rule Language Syntax**

This chapter lists the syntax of the rule language.

Topics

- [*Rule Language Syntax, page 26*](#)

Rule Language Syntax

```

<Conditions>                := { <Predicate> }*
<Actions>                   := { <StatementOrLocalDecl> }*
<Predicate>                 := <Expression> ";"
<StatementOrLocalDecl>     := <Statement> | <LocalVariableDeclaration> ";"
<Statement>                 := <LineStatement> | <BlockStatement>
<LineStatement>            := ";" | <StatementExpression> ";" | "break" ";" |
    "continue" ";" | "return" { <Expression> }? ";"
<BlockStatement>           := <Block> | <If> | <While> | <For> | <TryCatchFinally>
<StatementExpression>      := <PrimaryExpression> "=" <Expression> | <Name>
<Arguments>
<Name>                     := <Identifier> { .<Identifier> }*
<Arguments>                 := "(" Expression() { "," Expression() }* ")"
<LocalVariableDeclaration> := <Type> <VariableNameAndInit> { ","
    <VariableNameAndInit> }*
<VariableNameAndInit>      := <Identifier> { "=" <Expression> |
    <DeclarationArrayLiteral> }?
<Type>                     := <TypeName> { "[" ] }?
<TypeName>                 := <Name> | "boolean" | "int" | "long" | "double"
<DeclarationArrayLiteral> := "{" <Expression> { "," <Expression> }* "}"
<Block>                    := "{" { <StatementOrLocalDecl> }* "}"
<If>                       := "if" "(" <Expression> ")" <Statement>
<While>                    := "while" "(" <Expression> ")" <Statement>
<For>                      := "for" "(" { <LocalVariableDeclaration> |
    <StatementExpressionList> }? ";" { <Expression> }?
    ";" { <StatementExpressionList> }? ")" <Statement>
    <StatementExpressionList> := <StatementExpression> {
    "," <StatementExpression> }*
<TryCatchFinally>          := "try" <Block> { "catch" "(" "Exception" <Identifier>
    ")" <Block> { "finally" <Block> }? | "finally"
    := <ConditionalExpression> { '||'
    <ConditionalExpression> }*
<ConditionalExpression>    := <EquivalenceExpression> { '&&'
    <EquivalenceExpression> }*
<EquivalenceExpression>    := <LogicalExpression> { <EquivalenceOps>
    <LogicalExpression> }*
<LogicalExpression>        := <AdditiveExpression> { <LogicalOps>
    <AdditiveExpression> | "instanceof" <Type> } }*
<AdditiveExpression>       := <MultiplicativeExpression> { <AdditiveOps>
    <MultiplicativeExpression> }*
<MultiplicativeExpression> := <PrimaryExpression> { <MultiplicativeOps>
    <PrimaryExpression> }*
<UnaryExpression>          := <PrimaryExpression> { <UnaryOps>
    <PrimaryExpression> }*
<PrimaryExpression>        := <PrimaryPrefix> { <PrimarySuffix> }*
<PrimaryPrefix>             := <Literal> | "(" <Expression> ")" | <Name> |
    <ArrayLiteral> | <ArrayAllocator>
<ArrayLiteral>              := <Type> "{" <Expression> { "," <Expression> }* "}"
<ArrayAllocator>            := <Type> "[" <Expression> "]" "{" }"
<PrimarySuffix>             := <ArrayIndex> | <Arguments> | <PropertyOrAttrAccess>
<PropertyOrAttrAccess>      := [ "." , "@" ] <Identifier>
<ArrayIndex>                := "[" <Expression> "]"
<EquivalenceOps>            := [ "==" , "!=" ]

```



```

<LogicalOps>           := [ "<" , "<=" , "=>" , ">" ]
<AditiveOps>           := [ "+" , "-" ]
<MultiplicativeOps>    := [ "*" , "/" , "%" ]
<UnaryOps>              := [ "!" , "-" , "+" ]

<Literal>               := <INTEGRAL_LITERAL> | <DOUBLE_LITERAL> |
                           <STRING_LITERAL> | <BOOLEAN_LITERAL> | "null"
<INTEGRAL_LITERAL>      := <DECIMAL_INT_LITERAL> | <HEX_INT_LITERAL> |
                           <OCTAL_INT_LITERAL>
                           | <DECIMAL_LONG_LITERAL> | <HEX_LONG_LITERAL> |
                           <OCTAL_LONG_LITERAL>
<DOUBLE_LITERAL>        := { ["0"-"9"] }+ "." { ["0"-"9"] }* { <EXPONENT> }? {
                           ["d" , "D"] }? | "." { ["0"-"9"] }+ { <EXPONENT> }? {
                           ["d" , "D"] }? | { ["0"-"9"] }+ { <EXPONENT> } {
                           ["d" , "D"] }? | { ["0"-"9"] }+ { <EXPONENT> }? ["d" ,
                           "D"]
<STRING_LITERAL>        := "'" <STRING_CONTENTS> "'"
<BOOLEAN_LITERAL>       := "true" | "false"
<DECIMAL_INT_LITERAL>    := ["1"-"9"] { ["0" - "9"] }*
<HEX_INT_LITERAL>       := "0" ["x" , "X"] { ["0"-"9" , "a"-"f" , "A"-"F"] }+
<OCTAL_INT_LITERAL>     := "0" { ["0"-"7"] }*
<DECIMAL_LONG_LITERAL>  := <DECIMAL_INT_LITERAL> ["l" , "L"]
<HEX_LONG_LITERAL>      := <HEX_INT_LITERAL> ["l" , "L"]
<OCTAL_LONG_LITERAL>    := <OCTAL_INT_LITERAL> ["l" , "L"]
<EXPONENT>              := ["e" , "E"] { ["+" , "-"] }? { ["0"-"9"] }+
<STRING_CONTENTS>       := { ~["'" , '\'" , "\n" , "\r"] | { '\'"
                           { ["n" , "t" , "b" , "r" , "f" , '\'" , '"'"' , '"'] |
                           ["0"-"7"] } { ["0"-"7"] }? | ["0"-"3"] ["0"-"7"]
                           ["0"-"7"]
                           }
                           }
                           }*

<ReserevedWord>         := [ "true", "false", "null", "if", "else", "while",
                           "for", "continue", "break", "return", "int", "long",
                           "double", "boolean", "instanceof", "rule",
                           "attribute", "declare", "when", "then", "this",
                           "moveto", "requeue", "priority", "package", "char",
                           "float", "abstract", "default", "private", "do",
                           "implements", "protected", "throw", "import",
                           "public", "throws", "byte", "transient", "case",
                           "extends", "short", "try", "catch", "final",
                           "interface", "static", "void", "finally",
                           "strictfp", "volatile", "class", "native", "super",
                           "const", "new", "switch", "goto", "synchronized" ]

// Identifiers additionally are restricted not to be any of the above reserved words

<Identifier>            := [ <ID_START> { <ID_PART> }* ]
<ID_START>              := except '$', any character for which
                           java.lang.Character.isJavaIdentifierStart()
                           returns true
<ID_PART>               := except '$', any character for which
                           java.lang.Character.isJavaIdentifierPart()
                           returns true

```

```
<SINGLE_LN_COMMENT>      :=  "//" { ~[ "\n", "\r", "\r\n" ] }*  
                           { [ "\n", "\r", "\r\n" ] }  
<MULTI_LN_COMMENT>      :=  "/*" { ~[ "*/" ] }* "*/"
```

Chapter 4 **Creating Custom Functions**

This chapter describes how to use custom functions with TIBCO BusinessEvents. It does not provide any information about programming the functions.

Topics

- [*Overview of Creating Custom Functions, page 30*](#)
- [*Structure of a Function Catalog, page 32*](#)
- [*Java Archive Resource, page 36*](#)

Overview of Creating Custom Functions

This chapter does not contain detailed instructions for creating custom functions or programming in Java. It is assumed that you are comfortable programming in Java, and have at a minimum already implemented a class and a static function.

Restrictions

Note the following restrictions that pertain to using custom functions.

Static and Non-Static Functions

Custom functions must be written in Java and have public static modifiers.

As a workaround, encapsulate a non-static function in a static function and compile the encapsulating class to get the `.class` file.

Return Types

BusinessEvents custom functions support the following return types:

- Java types supported are: `Object`, `String`, `Calendar` (which displays in BusinessEvents as `DateTime`), `Integer`, `Long`, `Double`, `Boolean`, `int`, `long`, `double`, and `boolean` (but not `byte`, `short`, `float`, or `char`).
- Entity model object, for example, concept or event. Also any well defined entities in the model of that type.
- Arrays are not supported return types, with this exception: One-dimensional Java arrays of any supported type are supported.

Name Overloading

The `functions.catalog` file makes functions available for use in TIBCO Designer. The structure of the file requires each function within a class to have a unique name. Because of this structure, you cannot refer to an overloaded function in `functions.catalog`.

For example, the standard Java library has several `String.valueOf()` functions overloaded for each primitive type (`String.valueOf(int i)`, `String.valueOf(double d)` and so on). However, the BusinessEvents standard function catalog has a separate function name for each data type: `valueOfBoolean()`, `valueOfDouble()`, `valueOfInt()`, and `valueOfLong()`.

See [Structure of a Function Catalog on page 32](#), for more about the `functions.catalog` file.

Task Summary

TIBCO BusinessEvents allows you to write your own custom functions in Java and add them to the function registry, making them available from within the function registry along with the prepackaged functions in the rule editor.

The steps below summarize the tasks required to integrate your own custom functions with TIBCO BusinessEvents:

1. Write your custom static function in Java and compile it.
2. Create a file called `functions.catalog`, an XML file that makes it possible to access your custom functions from the functions registry within the rule editor. The XML can also include information for a tool tip for each function.
3. Create a `.jar` file that includes your `.class` file and `functions.catalog`.
4. Add the location of the `.jar` file to the class path for the BusinessEvents Server and Workbench. Also add locations for any dependent classes.

The following pages provide more information about each of these tasks.

Structure of a Function Catalog

A function catalog is an XML file that conforms to `function_catalog.xsd` — a schema. This allows BusinessEvents to integrate your custom functions with the function registry in the rule editor. The function catalog must be in the XML format described in [Table 7](#) to map properly to the schema.



- Name the function catalog `functions.catalog`.
- Place `functions.catalog` in the root folder of the required Java archive resource (`.jar`) file.

Elements

[Table 7](#) lists and describes the elements used in the function catalog. Each element’s horizontal position within the [Element Name](#) column indicates the correct nesting position within the XML file.

Table 7 Function Catalog Elements

Element Name	Sub-Elements	Description
<code><catalog name="name "></code>		The root element. Attribute: <code>name="name"</code> where <i>name</i> is a name you provide for this functions catalog. Example: <code><catalog name="custom"></code>
<code><category></code>		This is a sub-element of <code><catalog></code> . <code><category></code> is a nesting container for a set of related functions within this functions catalog.
	<code><name></code>	A name you provide for this category.
<code><function></code>		A container for the information about a single function.
	<code><name></code>	The name of the function.
	<code><class></code>	The java class that implements the function.
	<code><desc></code>	Optional. A description of the function.
	<code><args></code>	A comma separated list of descriptive names for the function’s arguments. BusinessEvents takes the argument type from the function itself.

Table 7 Function Catalog Elements (Cont'd)

Element Name	Sub-Elements	Description
	<isActionOnly>	Valid values: true, false Optional. If this function has side effects, for example, if it can modify values, you can only use it in action rules. Set this parameter to true to alert BusinessEvents that this function has side effects.
<tooltip>		This is a sub-element of <function>. This element contains the elements of a tool tip. A tool tip provides information about a function when the user floats the cursor over the name of the function in the functions catalog in the rule editor.
	<synopsis>	A brief description of the function for display within the tool tip. Example: <synopsis>Takes value of "amount" and returns item name.
	<args>	A container for descriptive information about the function's arguments. The information you provide will be displayed in the function's tool tip.
	<paramdesc>	A sub-element of <args>. Provides descriptive information for one argument. Use this tag once for each argument. Attributes: <ul style="list-style-type: none"> name='arg_name' type='arg_type' Example: <paramdesc name='amount' type='int'>arg1</paramdesc>
	<returns>	Describes the return value. Attribute: type='return_type' Example: <returns type='string'>item name</returns>

Example Function Catalog

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<catalog name="Custom">
  <category>
    <name>Categories</name>
    <category>
      <name>SimpleOnes</name>
      <function>
        <name>firstSample</name>
        <method>firstSample</method>
        <class>com.tibco.be.functions.custom.CustomJavaHelper</class>
        <args></args>
        <tooltip>
          <synopsis>You could write java here</synopsis>
        </tooltip>
      </function>
    </category>
  </category>
  <category>
    <name>TimeBasedPropertyValues</name>
    <function>
      <name>snapshotOverTime</name>
      <method>snapshotOverTime</method>
      <class>com.tibco.be.functions.custom.CustomJavaHelper</class>
      <args>entity, startTime, endTime, namespace</args>
      <isActionOnly>true</isActionOnly>
      <tooltip>
        <synopsis>This is the synopsis</synopsis>
        <args>
          <paramdesc name="propertyDouble"
            type="PropertyAtomDouble">Property to serialize.</paramdesc>
          <paramdesc name="startTime" type="long">Start time for
            serialising the history values.</paramdesc>
          <paramdesc name="endTime" type="long">End time for
            serialising the history values.</paramdesc>
        </args>
        <returns type="String">An XML String Serialization
          of the PropertyAtomDouble passed.</returns>
      </tooltip>
    </function>
  </category>
  <category>
    <name>Serialize</name>
    <function>
      <name>serializeConcept</name>
      <method>serializeConcept</method>
      <class>com.tibco.be.functions.custom.CustomJavaHelper</class>
      <args>concept, changedOnly, nameSpace, root</args>
      <isActionOnly>true</isActionOnly>
      <tooltip>
        <synopsis>Serializes the Concept passed to an XML String
          which is returned.</synopsis>
        <args>
          <paramdesc name='concept' type='Concept'>The Concept to
            serialize.</paramdesc>
          <paramdesc name='changedOnly' type='boolean'>true - data
            modified since last conflict resolution cycle
            serialized</paramdesc>
          <paramdesc name='nameSpace' type='String'>A path describing
            the nameSpace to save in.</paramdesc>
          <paramdesc name='root' type='String'>The name of
            the serialized data.</paramdesc>
        </args>
        <returns type='String'>An XML String Serialization of
          the Concept passed.</returns>
      </tooltip>
    </function>
  </category>
</catalog>
```



```
        </tooltip>  
        </function>  
    </category>  
</catalog>
```

Java Archive Resource

To use your custom function with BusinessEvents, place the following resource files in a Java archive resource (.jar):

- The static function .class file. If the function itself is non-static, this is the encapsulating static function.
- The XML file described in [Structure of a Function Catalog on page 32](#).

Make the .jar file and any dependent class files available to BusinessEvents in one of these ways:

- Add the location of the JAR file to the `tibco.env.STD_EXT_CP` variable in the configuration file: `BE_HOME\bin\be-engine.tra`.
- Locate the file in `BE_HOME\lib`.

`BE_HOME` is the directory in which *BusinessEvents* is installed.

Chapter 5 **Query Features Overview**

This chapter provides a short overview of query features.

Topics

- [*Query Features Overview, page 38*](#)
- [*Two Common Ways to Use Queries, page 41*](#)

Query Features Overview



Query features are available only in the TIBCO BusinessEvents Enterprise Suite.

Query features are available in conjunction with cache-based object management.

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, but no Rete network (working memory).

For instructions on how to configure a query agent, see Chapter 23, Configuring Query Agents (Cache OM) in *TIBCO BusinessEvents User's Guide*.

Queries Retrieve Information from Cache

The query language enables you to query concepts and simple events in the cache, using an SQL-like language.

You can't query scorecards or time events (they don't exist in the cache). You can't query the objects in the Rete network itself, or those in the backing store.

Query Feature is Read-Only

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

Two Main Types of Queries

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.

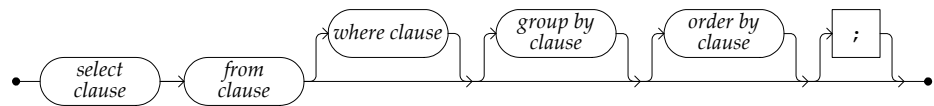
Continuous Queries

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 to process data (snapshot queries do not).

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 BusinessEvents rule, too:



Each clause in a statement is examined in detail in [Chapter 6, Query Language Components, on page 43](#). For a quick reference to the query syntax, see [Chapter 9, Query Language Reference, on page 87](#).

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.

The terminology shown in *italics* is used throughout documentation.

- **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** Two ways to execute a query instance are provided:
 - 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, 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 [Lifecycle of a Query—Use of Query Functions on page 56](#) for more details.

For More Information

The query features are documented in detail in the following chapters:

- [Chapter 6, Query Language Components, on page 43](#) Provides a detailed reference to each clause of a query statement.
- [Chapter 7, Working With the Query Language, on page 53](#) Explains how to use the query functions and how to work with results.
- [Working With Continuous Queries on page 71](#) Explains how continuous queries use different kinds of windows

[Chapter 9, Query Language Reference, on page 87](#) Provides a reference to the query language components, expressions, operators, wildcards, datatypes, literals, and reserved words,

Two Common Ways to Use Queries

The following sections show two ways queries can be used.

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 another 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

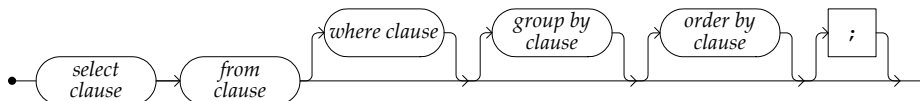
Query agents can act as pre-filters and routers. Suppose you want to check for the existence of a concept in the cache, using properties of an event. If the concept does not exist, you want to create it. You can achieve this result as follows:

- | | |
|-----------------|---|
| 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 messages on destination D2.
4. On receiving a message (event) at D2, a rule in the inference agent creates the concept. |

Chapter 6

Query Language Components

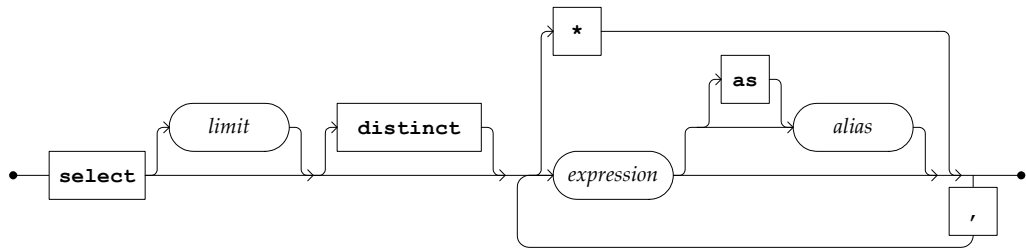
This chapter expands each of the components of the `select` statement:



Topics

- [Select Clause, page 44](#)
- [From Clause, page 45](#)
- [Where Clause, page 46](#)
- [Group by Clause, page 47](#)
- [Order by Clause, page 48](#)
- [Limit Clause, page 49](#)
- [Stream Clause, page 50](#)
- [Stream Policy, page 51](#)

Select Clause



In the select clause, you specify the columns that will appear in the query results. In the example below, a select clause projects two columns, address and name, which are properties of the concept /customer. The alias for the customer concept is the letter c:

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

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

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

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 47](#))

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.

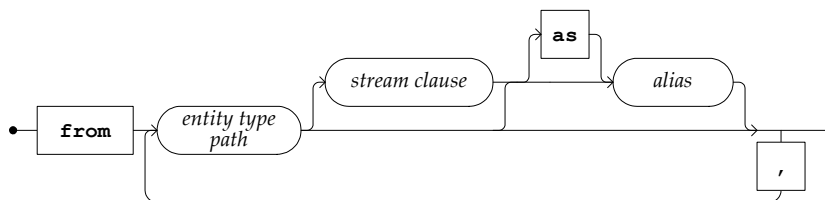
You can use an optional distinct clause to prevent the query from returning duplicate rows.

Examples of Select Clauses

These examples show only the select clause. A complete query requires a select and a from clause.

```
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
```

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.

Continuous Queries

The `from` clause in a continuous query can specify *window policies*. See [Overview of Continuous Queries on page 72](#) and sections following, for more information.

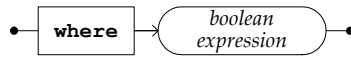
Examples

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

```

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

Where Clause



The optional where clause is analogous to a rule's conditions. The expression in the where clause can be simple or complex. In the where clause you can use following:

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

Examples

```
where A.customerId = B.customerId
```

```

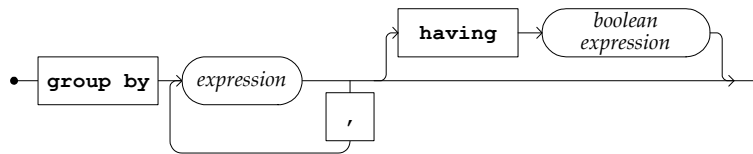
where A.id = B@extid // Entity attributes
and ( B@parent.name = 'ABCD' or C.name = "EFGH" )
and A.tokens[5] = 50 // array property
and ( A.containedConceptE.price > 100
or B.startTime > /#DateTime/addMinute(/#DateTime/now(),5) )
and B.value between 2 and 5

```



The pound sign (#) is used to escape reserved (key) words. See [Reserved Words on page 104](#) 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.



When you use a `group by` clause, the `select` clause cannot use the entities specified in the `from` clause What is available to the `select` clause has been, in effect, reduced using the `group by` clause. When you use a `group by` clause, the `select` clause can use only the `group by` criteria, and aggregation functions.

For example:

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

The above query would return a list of zip codes.

Although the `group by` clause in this example reduces the result set to a list of zip codes, additional information from the query is internally available to the aggregation functions. You can use any of the standard group functions that are applicable, such as those used to calculate count, sum, average, maximum, and minimum.

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:

```
select count(*) from customer c group by c.zipcode;
```

Optional having Clause

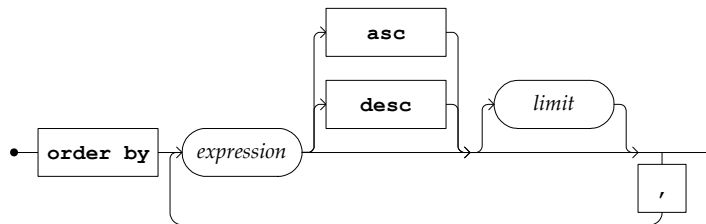
The optional `having` clause allows you to apply conditions after entities are grouped. For example this query returns the number of customers in each zip code, except for those zip codes where there are three or fewer customers:

```
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 on page 66](#).

See also [Limit Clause on page 49](#).

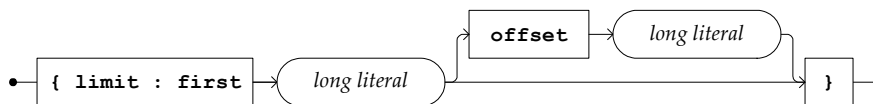
Examples

```
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 74](#).

Example Showing Use in Select Clause

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

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

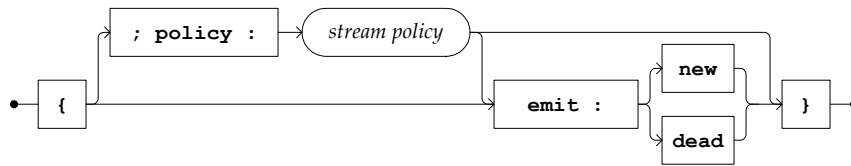
Example Showing Use in Order By Clause

```
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 Emit: New and Emit: Dead

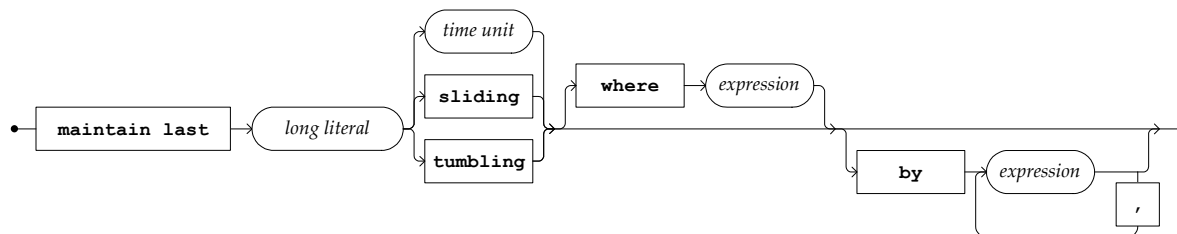
The `emit` key word determines whether the query is evaluated when an entity enters the window (`emit : new`) or when an entity leaves the window (`emit : dead`).

The default value is `emit : new`.

See [Time Window Examples on page 82](#) for some example uses.

See [Stream Policy on page 51](#) 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 76](#).

Note that continuous queries that use an implicit window do not have a stream policy. See [Working With Implicit Windows on page 74](#).

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.

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.

By Clause

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

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

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

See [Explicit Window Example on page 77](#) for a detailed discussion of an example that uses a stream policy. See [Time Window Examples on page 82](#) for more examples.

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

- *Querying the Cache and Using Query Results, page 54*
- *Lifecycle of a Query—Use of Query Functions, page 56*
- *Using Data from a Result Set, page 61*
- *Using Data from a Callback Rule Function, page 62*
- *Simple Snapshot Query Example, page 64*
- *Simple Continuous Query Example, page 65*
- *Using Bind Variables in Query Text, page 69*

Querying the Cache and Using Query Results

To use the query language features, you put query text (SQL-like statements that retrieve information from the cache) 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 to create prepared statements.

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()`, `getStatementName()`
- ResultSet category: `close()`, `get()`, `isBatchEnd()`, `isOpen()`, `next()`
- Statement category: `clearSnapshotRequired()`, `clearVars()`, `close()`, `execute()`, `executeWithCallback()`, `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 Designer and in the online reference, [TIBCO BusinessEvents Functions Reference](#).

For general information on using the functions provided with BusinessEvents, see Understanding and Working With Functions in *TIBCO BusinessEvents User'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't be used in
queries

The following functions cannot be used within queries:

- Rule functions with a Validity field that is set to anything other than "Action, Condition, Query." (The Validity field is in the rule function Configuration tab.)

- Ontology functions.
- All catalog functions that assert, modify or delete objects in the cache or in working memory. Queries cannot change the cache.

For information on custom functions see [Chapter 4, Creating Custom Functions, on page 29](#).

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 69](#) 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 on page 61](#) and [Using Data from a Callback Rule Function on page 62](#) for details on how to get and use query results.

Create the Query Definition

Creating a query definition is a distinct and 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:

```
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:

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

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

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

Open a Query Statement

Format:

```
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:

```
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 64](#).

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 distinct from each other

Format

```
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 69](#) 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.

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 on page 61](#) 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 the statement or the query definition. To close the result set use the following function:


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

For example:

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

To Obtain Results Using a Callback Rule Function

You can use `Query.Statement.executeWithCallback()`, for snapshot or continuous queries. If you set the `IsContinuous` argument to `true`, the query runs as a continuous query.

The behavior depends on whether the query executes as a snapshot or as a continuous query:

- 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.
- When used in continuous queries, the query listens for changes to the cache, and calls the rule function as matches occur over the lifetime of the query.

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

Format

```
Query.Statement.executeWithCallback(
String  StatementName,
String  ExecutionName,
String  CallbackUri,
boolean IsContinuous,
Object  Closure)
```

The *ExecutionName* parameter keeps results from different executions distinct 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

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

See [Using Data from a Callback Rule Function on page 62](#) for more 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:

```
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 58](#).

Using Data from a Result Set

See [To Obtain Results Using a Result Set on page 58](#) for details about obtaining results.

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

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

The 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:

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

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

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

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

Using Data from a Callback Rule 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).
- 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 60](#).

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 Callback Rule Function Required Signature

The callback function must have a signature with the following parameter types, provided in the order specified:

Parameter	Notes
<code>String id</code>	Identifies the current execution. Uses the value of <i>ExecutionName</i> , 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).

Parameter	Notes
boolean <i>isBatchEnd</i>	<p>Used in the case of continuous queries only, and is useful only when the query text contains an <code>order by</code> clause (see Order by Clause on page 48).</p> <p>Only true at the end of a batch of rows of data generated by the query.</p> <p>In the case of continuous queries where no sorting is used, each row of data is a batch.</p> <p>See Example Showing Batching of Return Values on page 66.</p>
boolean <i>hasEnded</i>	When true, signals the end of the execution.
Object <i>row</i>	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 44).
Object <i>closure</i>	<p>Closure object provided when executing the executing the <code>Query.Statement.executeWithCallback()</code> function, or null.</p> <p>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.</p>

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.

```
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 60](#) 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:

```
MyRF(ID, isBatchEnd, hasEnded, row, closure)

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

Create the Query

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

Open and Execute the Query Statement

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

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

Sample Output

In the 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

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.

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, "@@@")
```

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.

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

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

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

All three functions must reference the same query statement name.

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.

Example

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

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

Clearing Bind Variables

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

This chapter focuses on the features of continuous queries.

Topics

- *Overview of Continuous Queries, page 72*
- *Working With Implicit Windows, page 74*
- *Working With Sliding, Tumbling, and Time Windows, page 76*
- *Explicit Window Example, page 77*
- *Sliding Window Examples, page 79*
- *Tumbling Window Examples, page 81*
- *Time Window Examples, page 82*
- *Optimizing the Design, page 83*

Overview of Continuous Queries

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.

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 72](#) for more details.

Executing a Continuous Query

You use the `Query.Statement.executeWithCallback()` function with the `IsContinuous` argument set to `true` to execute a continuous query statement. See [Using Data from a Callback Rule Function on page 62](#), and See [Two Main Types of Queries on page 38](#) 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 to contain the entities that are added or changed. 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. 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.



Ensure that the lifetime of an entity in an explicit window is shorter than the lifetime of the entity itself. Otherwise errors occur.

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 76](#) for content that applies to all these types of explicit windows.

Implicit Window A group by clause in the select statement determines that the query is a continuous query using an implicit window. See [Working With Implicit Windows on page 74](#).

Sliding Window Has a specified 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 on page 79](#)

Tumbling Window Has 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. See [Tumbling Window Examples on page 81](#)

Time Window Specifies a time period during which entities remain in the window. See [Time Window Examples on page 82](#).

Working With Implicit Windows

Implicit windows are created when the query does not have an explicit policy (window) clause, and the query is executed as a continuous query (see [Executing a Continuous Query on page 72](#)).

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;
```

The example shows a `group by 1` clause. This is a dummy group, required because aggregate functions, `count(*)` in this case, require a `group by` clause.

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.

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

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

The lifecycle of an entity in a window can be 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 51](#).

Use Sliding, Tumbling, or Time Windows for Events and not Concepts

Concepts are mutable and events are immutable after they are asserted. The mutability of concepts makes them unsuitable for 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

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 distinct:

```
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

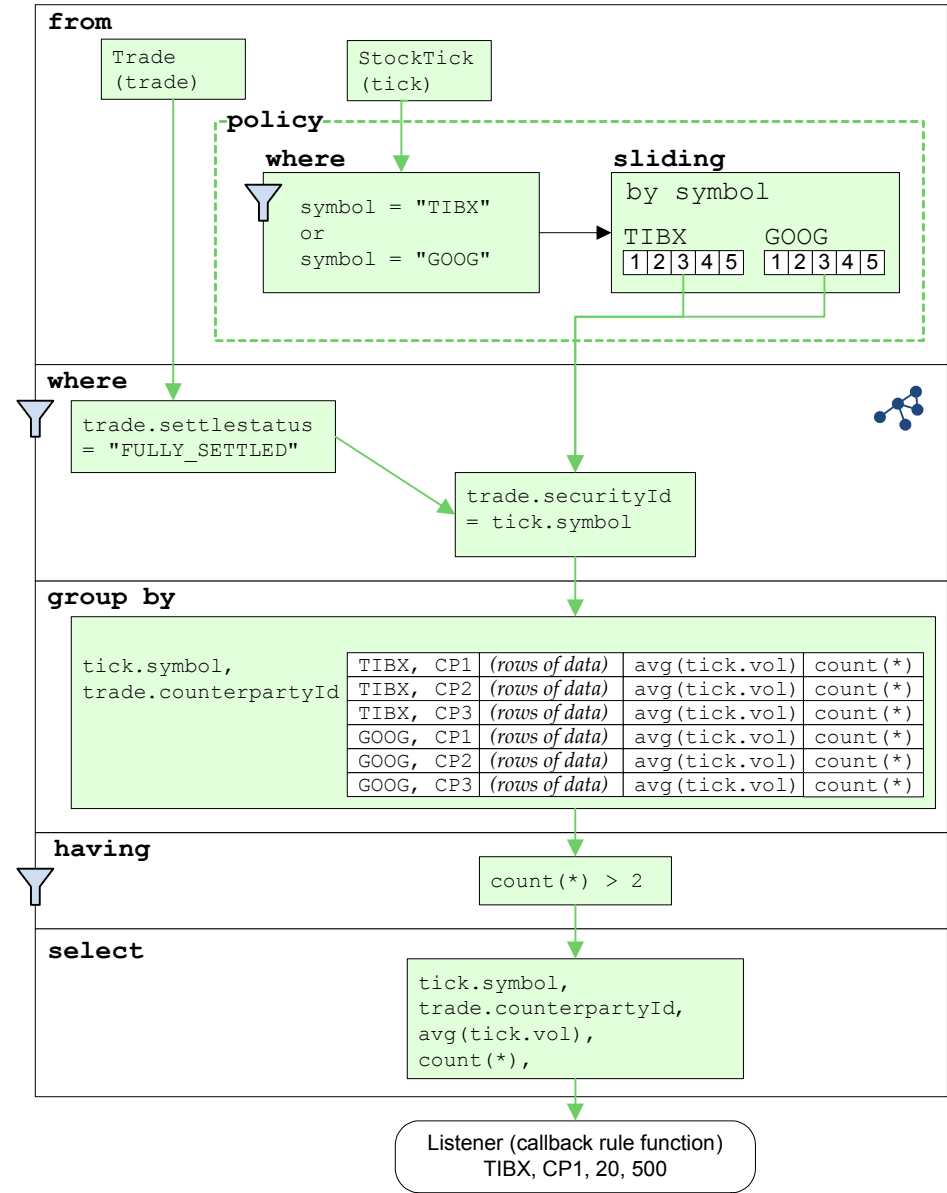
group by
    tick.symbol,
    trade.counterpartyId

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 78](#): from (including stream clause), where, group by (including having), select.

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

Figure 1 How a Query String is Processed



Sliding Window Examples

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.

```
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

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 latest 500} burger group by 1;
```

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
clause

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

Time Window Examples

Time windows use a stream policy that specifies how long an entity remains in the window. See [Stream Policy on page 51](#).

The expiry time is calculated from a start time. You can use the 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.

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

The above query holds `PizzaOrderEvents` for 45 minutes in a time window. 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.
----------------------	--

```
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 56](#) 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 69](#)). 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

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 pre-fetching the objects. See the section [Configuring Query Agents—Engine Properties for Performance in TIBCO BusinessEvents User's Guide](#) for details.

Use Filtering for Efficient Joins

When performing a join between two or more entities in a query, the most selective operators (filter on entity attributes) must appear before the actual join expression. 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 (>, <, >=, <=).

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.securityId = tick.symbol`).

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

Effect of the Cache on Continuous Queries

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 Queries

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

Example	<p>Consider a continuous query that is monitoring entities of type <code>/OrderEvents</code>. Suppose that <code>OrderEvents</code> entities are created, asserted, and consumed, at a fast rate. When an <code>OrderEvent</code> entity is asserted, it is also added to the cache. When it is consumed, an <code>OrderEvent</code> entity is deleted from the cache. The continuous query receives the creation notification and the deletion notification one after the other.</p> <p>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 <code>OrderEvent</code> entities that have already been removed from the cache, resulting in runtime errors.</p> <p>This situation may occur when, for example, a very quick succession of notifications is sent, or the network traffic suffers delays, and so on.</p>
Query Agent Local Cache	<p>The query agent retains the most recently processed entities in a local cache to avoid frequent network lookups. But in the example above, the <code>OrderEvent</code> is deleted from the cache even before the create-notification is processed by the query, so the <code>OrderEvent</code> can't be copied into the query agent's private cache.</p> <p>Keep such situations in mind as you design your queries.</p>

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 88*
- *Query Syntax, page 89*
- *Expression Syntax, page 91*
- *Operators for Unary Expressions, page 98*
- *Operators for Binary Expressions, page 99*
- *Operators for Other Expressions, page 101*
- *Wildcards, Datatypes, and Literals, page 102*
- *Reserved Words, page 104*

Miscellaneous Terms Used in Query Syntax Diagrams

The following table defines terms used in syntax diagrams shown in [Chapter 9, Query Language Reference, on page 87](#), and that don't fall into categories documented in other tables, such as operators,

Table 8 Miscellaneous Terms Used in Query Syntax Diagrams

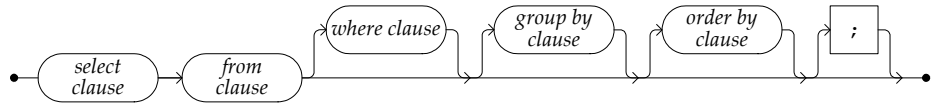
alias	Each alias must be globally unique in the whole query (this includes aliases defined in the projection—that is, aliases used in the select clause and in the from clause.
entity	Use the fully qualified ontology name of an entity, with its project path. From <code>/concepts/customer data</code> Remember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

Reading Query Language Syntax Diagrams

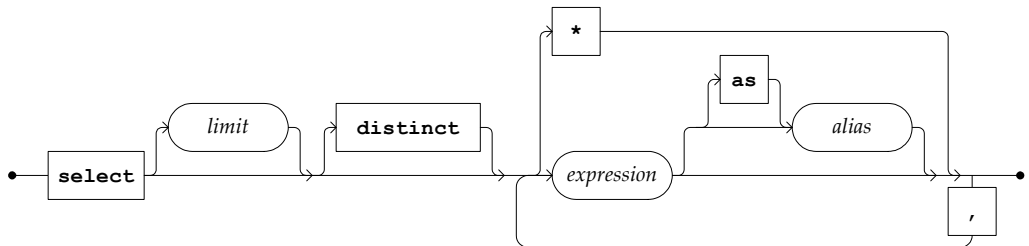
The syntax diagrams in this section show the structure of a query and of each clause in a query. Read them from left to right. Items above or below the main line are optional. Items that can repeat are shown by lines that loop back from the end to the beginning of the repeating section, along with the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

Query Syntax

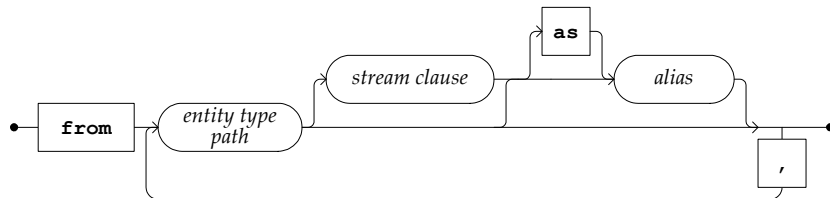
The top level syntax for a query is as follows:



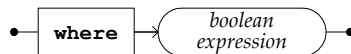
Select Clause



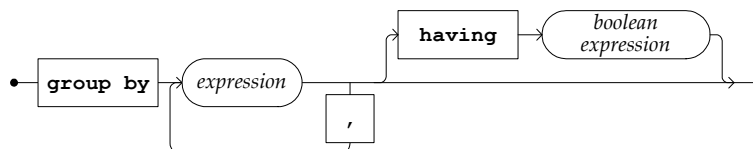
From Clause



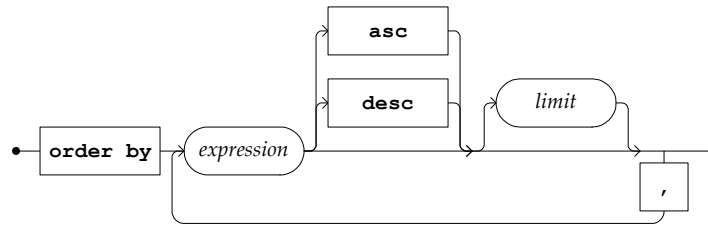
Where Clause



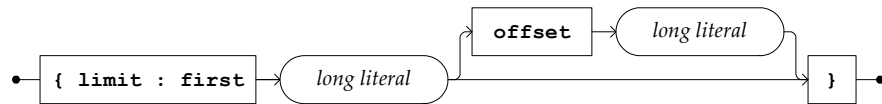
Group by Clause



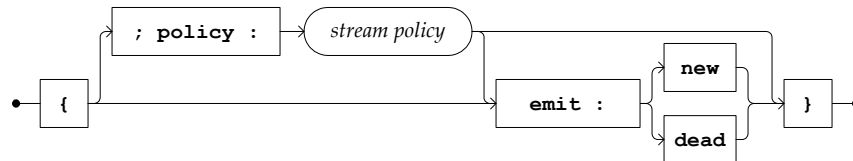
Order By Clause



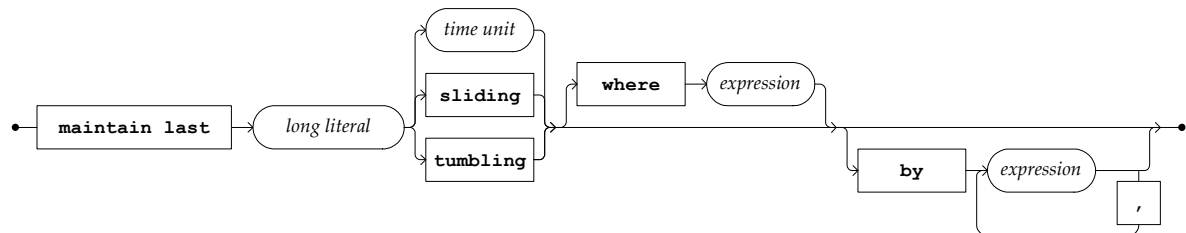
Limit



Stream Clause

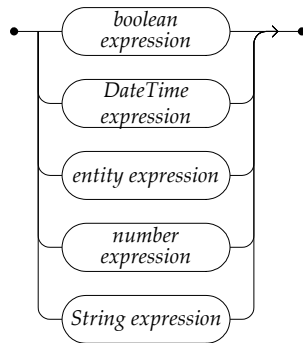


Stream Policy

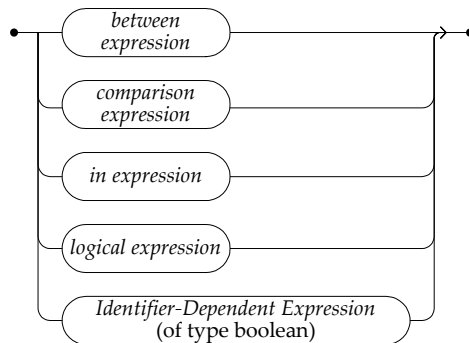


Expression Syntax

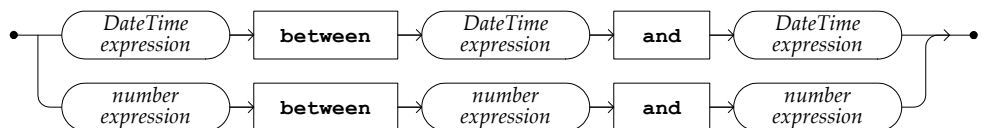
Expression



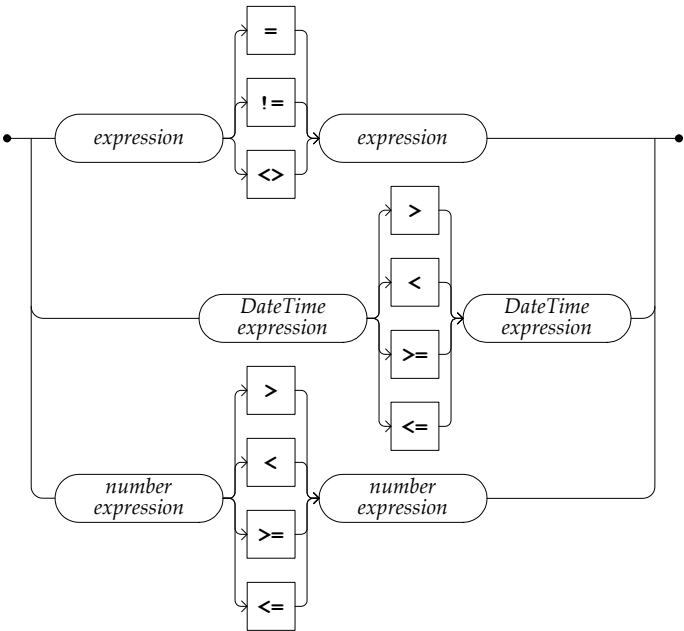
Boolean Expression



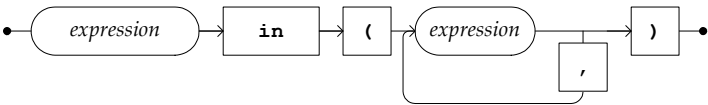
Between Expression



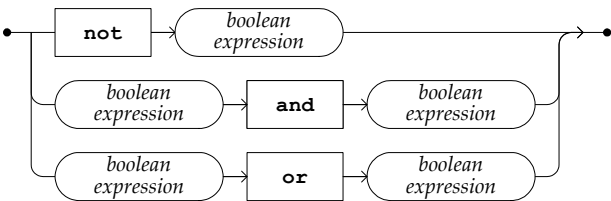
Comparison Expression



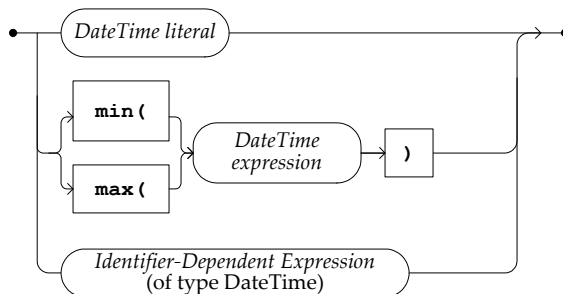
In Expression



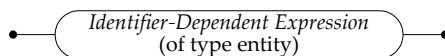
Logical Expression



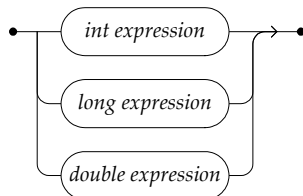
DateTime Expression



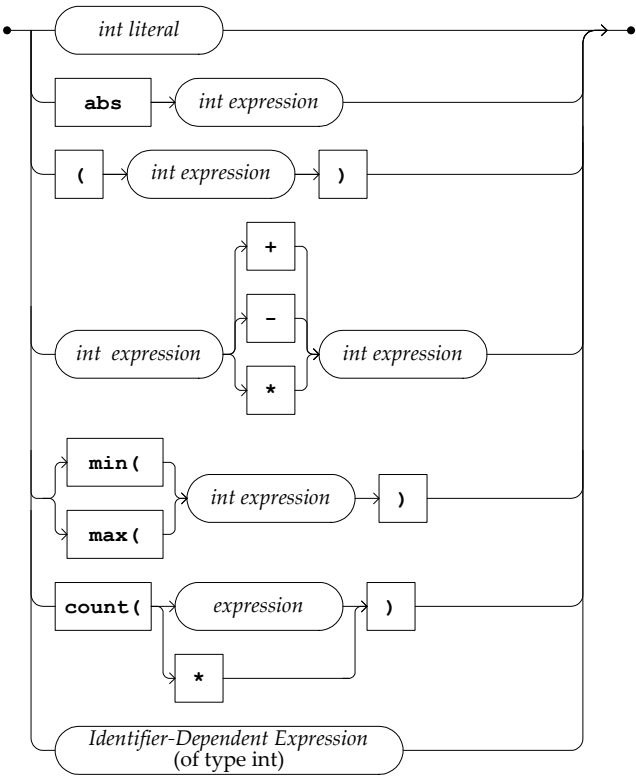
Entity Expression



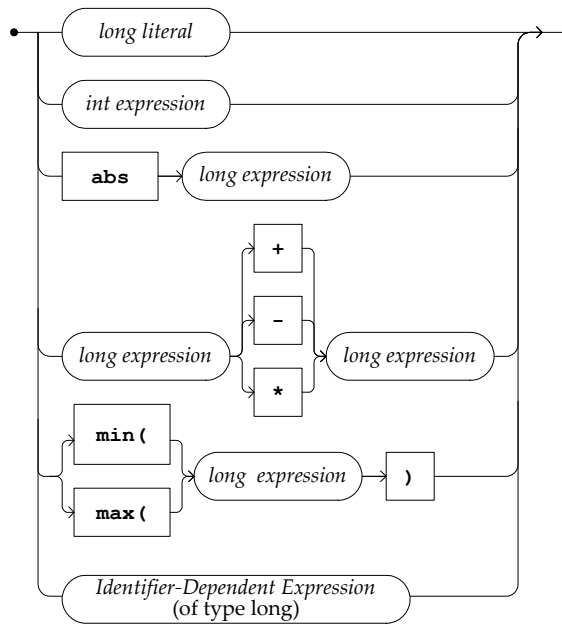
Number Expression



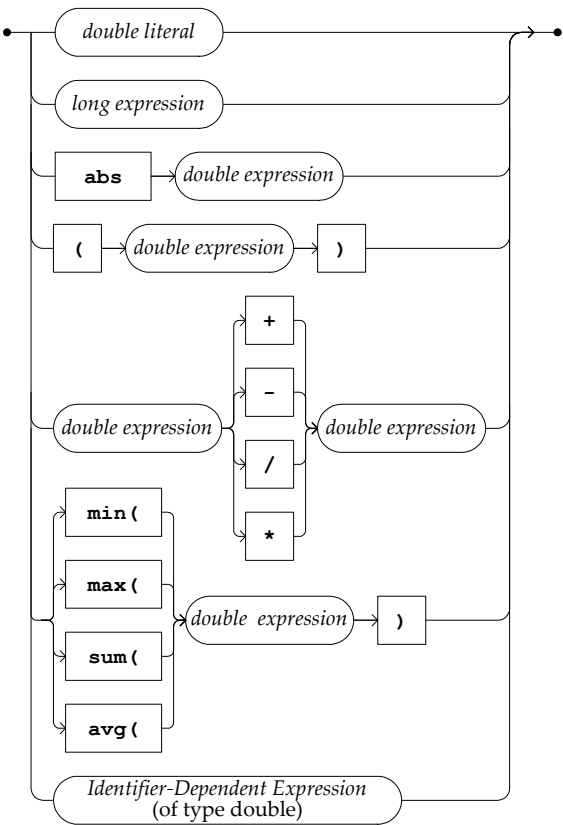
Int Expression



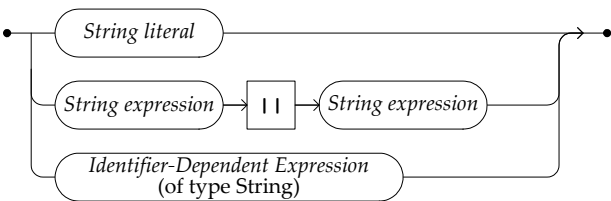
Long Expression



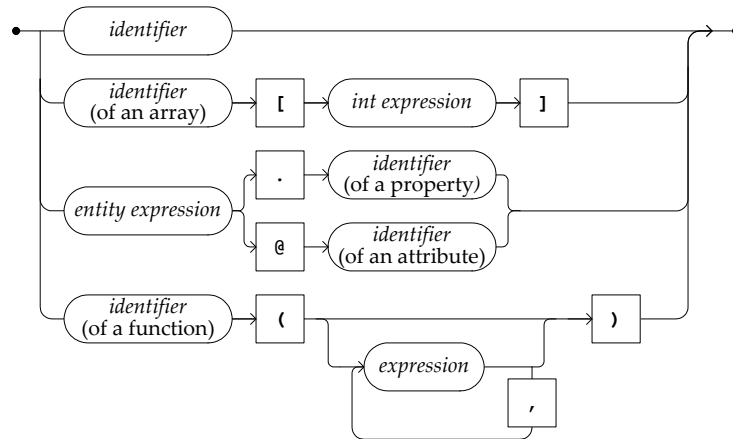
Double Expression



String Expression



Identifier-Dependent Expression



Operators for Unary Expressions

Table 9 Operators for Unary Expressions in Queries

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

Operators for Binary Expressions

Table 10 Operators for Binary Expressions in Queries (Sheet 1 of 2)

Operator	Description and Examples	Datatypes	Result type
Relational Expression Operators			
=	equality x = y	x and y can be any type	Boolean
!= <>	inequality x != y x <> y	x and y can be any type.	Boolean
> < >= <=	Greater than Less than Greater than or equal to Less than or equal to x > y (and so on) Generically known as comparison operators	x and y must both be number types, or both be Datetime types.	Boolean
Logical Operators			
and or	Logical (Boolean) and, or. x and y x or y	x and y must be Boolean	Boolean
Mathematical Operators			
Also used in the projection (select clause)			
*	Multiplication x * y	x and y must both be numbers.	Either the type of x or y, whichever has the larger capacity.
\	Division x \ y	x and y must both be numbers.	double

Table 10 Operators for Binary Expressions in Queries (Sheet 2 of 2)

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

Operators for Other Expressions

Table 11 Operators for Other Expressions in Queries

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

Wildcards, Datatypes, and Literals

Wildcard Characters

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

Datatypes

All types supported by BusinessEvents.

Literals

Literal values can be of any of the data types described below, plus the following:

- hex
- octal
- char

Types and Literals

Table 12 Query Language Types and Literals

Type	Syntax of Literals	Example
int	A signed integer expressed using only digits and an optional sign prefix. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive).	1234567
long	A signed integer expressed using only digits 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).	digits 1234567
double	A double-precision 64-bit IEEE 754 floating point.	12345.67 1.234e+56

Table 12 Query Language Types and Literals

Type	Syntax of Literals	Example
String	String literals are surrounded by double quotes. To escape double quote and backslash characters, prefix them with a backslash.	"hello" "She says: \"Hello.\"" "c:\\temp\\myfile"
boolean	The boolean data type has only two possible values: true and false. Use for simple flags that track true and false conditions.	true false
DateTime	yyyy-MM-dd 'T' HH:mm:ss.SSSZ where yyyy: four digit year MM: two digit month dd: two digit day of month HH: two digit hour of day in 24 hour format mm: two digit minutes in hour ss: two digit seconds in minute SSS: three digit milliseconds in second 'T': the letter T z: timezone expressed as defined in RFC 822.	2008-04-23T13:30:25. 123-0700
Entity type	"entity-project-path" Entity project path begins with a forward slash and folders are separated with a forward slash.	"/a/b/MyConcept"
Entity	No literal is used for entity instances.	(Not applicable)

Identifier

First character must be alphabetical (upper or lower case) or the underscore character. Other characters can be alphabetical or numeric or the underscore character.

Reserved Words

The following key words are used by the object query language.

abs	and	as	asc
avg	between	by	concept
count	days	dead	desc
distinct	emit	entity	event
false	first	from	group
having	hours	in	last
latest	like	maintain	max
milliseconds	min	minutes	mod
new	not	null	object
or	order	policy	seconds
select	sum	true	using
where			

Escaping the Keywords

You cannot use key words as identifiers, resource names, or folder names without prefixing 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 [104](#)
 @closure [9](#)
 @extId [9](#)
 @id [9](#)
 @interval [9, 9](#)
 @isSet [10](#)
 @length [10](#)
 @parent [10](#)
 @payload [9](#)
 @scheduledTime [9, 9](#)
 @ttl [9, 9, 9](#)

A

actions [8](#)
 AdvisoryEvent event type [14](#)
 array indexes, start from zero or one [12](#)
 arrays
 accessing and appending values [12](#)
 indexes start from zero or one [12](#)
 primitive [4](#)
 attributes [9](#)

B

batching of return values [66](#)
 BE_HOME [xi](#)
 between expression [91](#)
 binary expressions, operators for [99](#)
 bind variables [55](#)
 in query text [69](#)
 boolean expression [91](#)

C

callback rule functions [59](#)
 required signature [62](#)
 using data from [62](#)
 closing a statement [60](#)
 comparison expression [92](#)
 concept properties, accessing [11](#)
 concepts, when not to use in explicit windows [76](#)
 conditions [8](#)
 continuous queries
 executing [72](#)
 overview [72](#)
 simple example [65](#)
 create the query definition [56](#)
 custom functions
 name overloading not supported [30](#)
 return types supported [30](#)
 static modifiers [30](#)
 tool tips [33](#)
 customer support [xiv](#)

D

datatypes [102](#)
 datetime expression [93](#)
 declaration [8](#)
 deleting a query definition [60](#)
 double expression [96](#)

E

effect of the cache on continuous queries [84](#)
 effect of time on queries [84](#)

emit

 dead [50, 82](#)

 new [50, 81](#)

ending a continuous query [72](#)

entity expression [93](#)

escape sequences [7](#)

escaping the keywords [104](#)

event properties, accessing [13](#)

execute an instance of the query statement and obtain results [58](#)

executing a continuous query [72](#)

explicit windows, example [77](#)

expression [91](#)

 syntax [91](#)

F

filtering for efficient joins [83](#)

from clause [45](#)

functions

 that can't be used in queries [54](#)

 used to create and execute queries [39](#)

 within queries [54](#)

G

group by clause [47](#)

I

identifier [103](#)

identifier-dependent expression [97](#)

implicit windows [74](#)

 examples [74](#)

improve performance by pre-fetching objects [83](#)

in expression [92](#)

inference agents [41](#)

int expression [94](#)

L

lifecycle of a query [56](#)

limit [49, 90](#)

limitation in use of arrays [55](#)

literals [102, 102](#)

local variables [4](#)

logical expression [92](#)

long expression [95](#)

N

number expression [93](#)

O

open a query statement [57](#)

operators

 for binary expressions [99](#)

 for other expressions [101](#)

 for unary expressions [98](#)

optimizing the design [83](#)

order by clause [48, 90](#)

P

pre-filter [41](#)

primitive arrays [4](#)

property arrays, index from zero or one [12](#)

property values, accessing [11, 11](#)

Q

queries are executed in query agents [38](#)

queries retrieve information from cache [38](#)

query agents [41](#)

 local cache [85](#)

- query as a pre-filter [41](#)
- query features overview [38](#)
- query function catalog [54](#)
- query statement, open [57](#)
- query string, how processed [78](#)
- query syntax [89](#)
 - terms used in diagrams [88](#)
- query windows [72](#)
- querying the cache and using query results [54](#)

R

- reserved words [104](#)
- result set [58](#)
 - using data from [61](#)
- rules
 - triggering queries from [41](#)

S

- select clause [44, 89](#)
- select statement, structure of [39](#)
- set bind variables (if used) [57](#)
- sliding windows [76](#)
 - examples [79](#)
- snapshot queries, example [64](#)
- stream clause [50, 90](#)
- stream policy [51, 90](#)
- string expression [96](#)
- structure of a query select statement [39](#)
- support, contacting [xiv](#)
- syntax diagrams [88](#)

T

- technical support [xiv](#)
- TIBCO_HOME [xi](#)
- time windows [76](#)
 - examples [82](#)

- tool tips [33](#)
- triggering a query from a rule [41](#)
- tumbling windows [76](#)
 - examples [81](#)
- types and literals [102](#)
- types of windows [73](#)

U

- unary expressions, operators for [98](#)
- use of query functions [56](#)

V

- variables, local [4](#)

W

- where clause [46, 89](#)
- wildcard characters [102](#)
- wildcards, datatypes, and literals [102](#)
- windows, types of [73](#)