webMethods Continuous Query Development

Continuous Query Development Help

Version 9.5 SP1

November 2013

This document applies to webMethods Continuous Query Development Version 9.5 SP1.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: WWEP-CEP-CQDDEV-95SP1-20130929

Table of Contents

Preface

1 Working with the Continuous Query Development Perspective

This section provides basic information about working with the Continuous Query Development perspective in Software AG Designer. You use the Continuous Query Development perspective to build and test continuous query applications.

Starting the Continuous Query Development Perspective in Windows

To start the Continuous Query Development perspective, proceed as follows:

To start the Continuous Query Development perspective

1 Start the Software AG Designer.

2 From the Eclipse menu bar, select **Window > Open Perspective > Other > Continuous Query Development**.

The Project Explorer View

The project explorer view displays the hierarchical contents of the currently open project or projects. When you create a project for continuous query processing, the project is displayed in the project explorer view. When you expand the continuous query project, the available folders typically include:

Folder	Contents	File extension of files in this folder
Event Sequences	Files containing event sequences.	*.events* Example: *AuctionStream.events*
Event Types	Files containing XSD schemas that describe event types.	*.xsd* Example: *OpenAuction.xsd*
Input Streams	Files describing the input streams that will be used for query processing.	*.instream* Example: *Auction.instream*
Queries	Files containing query statements or query models.	*.ceq* Example: *AuctionQuery.ceq*

Note: The editors provided in the Continuous Query Development perspective require you to use the file extensions shown in the table above. For example, the query editor requires the file extension of a query file name to be "ceq", otherwise the file cannot be opened for editing with this editor. Also, if you change the file extension of such a file that is already open in an editor, the editor will close automatically.

Important: Do not add Java code to a continuous query project, unless the code is associated with a user-defined extension. Doing so can result in class-loading errors when the project is deployed. For more information about user-defined extensions, see *Developing User-*

Defined Functions, *Developing User-Defined Aggregates* and *Developing User-Defined Operators*.

The "Servers" and "Event Servers Admin" Views

The **Servers** view in Software AG Designer, in conjunction with the **Event Servers Admin** view, allows you to inspect all assets (except event types) that are registered with the Event Server.

The **Servers** view additionally allows you to activate event recording for selected input streams and queries. Event recording results in the creation of an event log file. In a default installation, the event log file is located in the folder *SAGInstallDir*/EventServer/eventlogs.

Context Menu Commands

In general, each entry in a view has associated commands that can be activated from the context menu of the entry. The context menu can be activated by right-clicking on the entry.

The functions available in the context menu are self-explanatory, but note the following additional information.

- **Rename**
  When you use the rename feature to rename a query or input stream, this not only renames the file but also where necessary adapts the contents of the file. This is because the contents of files of these types include a reference to the file name. Additionally, when you use the rename feature to rename a query model, both the .ceq and the .cqm file for the model are renamed.
- **Delete**
  When you use the delete feature to delete a query model, Software AG Designer deletes both the .ceq and the .cqm file from the project.
- **Copy**
  When you use the copy feature to copy a query model, Software AG Designer copies both the .ceq and the .cqm file for the model.

For the reasons stated above, always use the Rename, Delete and Copy commands in Software AG Designer to rename, delete or copy items in a continuous query project, not the file-system commands provided by the host operating system.

2 Overview of Building a Continuous Query Application

This section describes the high level steps you take to create a continuous query application.

Building a Continuous Query Application

The following describes the high-level steps involved in creating a continuous query application.

Note: You can build and test a continuous query application entirely within you local Software AG Designer environment. An Event Server is not required for initial development and testing. You do not need access to an Event Server until you are ready to deploy your application and test it in a "live" environment.

Step	Description
1	Create a new, empty continuous query project in Software AG Designer. For information about creating a new project, see ***Working with Projects***.
2	Create event types that define the structure of the events that your application will consume and publish. If the event types you need already exist in an Event Type Store or a CentraSite registry, you can reuse the existing type definitions instead of creating new ones. For information about adding event types to a project, see ***Working with Event Types***.
3	Define the input streams that your application will consume. Associate an event type with each input stream that you define. For information about defining input streams, see ***Working with Input Streams***.
4	Create your query by specifying it using the Event Query Language (EQL) or building it graphically using the Continuous Query Modeler. During this step, you define the query that the application will apply to the events it receives. For information about creating queries using EQL, see ***Querying an Event Stream***. For information about creating queries using the Continuous Query Modeler, see ***Building a Query Using the Continuous Query Modeler***. **Note:** Although you can create queries using EQL or the Continuous Query Modeler, you cannot convert a modeled query to EQL or vice versa. If you are unsure which tool to use to create a query, see ***Should You Use the Continuous Query Modeler or EQL?***
5	*Optional*. If your query requires the use of user-defined extensions, add that functionality to your project. For information about adding user-defined extensions to a query application, see ***Developing User-Defined Functions***, ***Developing User-Defined Aggregates*** and ***Developing User-Defined Operators***.
6	*Optional*. If you want to process input or output streams in which the events can have variable structure, adapt your event type definitions and query logic accordingly. For information, see ***Using Variable XML Schema Components***.
7	*Optional*. If you want to combine an event stream with data from an external database, define a data source for each table that you want to reference. For information about creating data sources, see ***Combining Events with Data from a Database***.
8	Test your application in Software AG Designer. For information about testing your application in the design-time environment, see ***Testing a Project in Software AG Designer***.
9	Deploy your application to an Event Server. For information about deploying your application, see ***Deploying a Continuous Query Application to an Event Server***.

Step	Description
10	Test your application on the Event Server. For information about testing your application in the run-time environment, see ***Testing, Debugging, and Troubleshooting a Deployed Project***.

3 Working with Projects

A *project* encompasses the resources that are associated with a continuous query application.

Creating a New Project

Use the following procedure to create a new continuous query project.

To create a new project

1 In the Continuous Query Development, choose **File > New > Continuous Query Project**.

2 In the field **Project Name**, specify a name for the new project. The project name must begin with a letter and include only the following characters:

- Letters A-Z or a-z
- Digits 0-9
- The following special characters: _ $

3 Click **Finish**.

Deleting a Project

Use the following procedure to delete a continuous query project.

To delete a project

1 In the Project Explorer, right-click the project name.

2 From the context menu, select **Delete**.

Renaming a Project

Use the following procedure to rename a continuous query project.

To rename a project

1 1. In the Project Explorer, select the project that you want to rename..

2 2. Choose **File** > **Rename**.

3 Specify a new name for the project. The project name must begin with a letter and include only the following characters:

- Letters A-Z or a-z
- Digits 0-9
- The following special characters: _ $

4 Click **OK**.

5 If this project is referenced by other projects in the workspace, update the **Project** > **Properties** > **Project References** settings in those projects so that they refer to this project by its new name.

Tip: You can also perform the rename operation from the context menu for the project. (Note that if the project contains Java resources, such as user-defined extensions, the **Rename** command will reside under the **Refactor** menu entry.)

4 Working with Event Types

Introduction

Event types are XML schemas that define the structure of an event that components can publish or consume. In a continuous query application, an event type describes the event instances that a given input stream or output stream will carry.

Creating and Editing Event Types

You create event types using the Event Type Editor. If you want to reuse an event type has already been defined, you can add it to your project from the Event Type Store.

Continuous query applications cannot consume XML documents directly. They consume *records* (also called tuples), which are flat structures that resemble the rows in a database table. When the Event Server receives an event from the Event Bus, it immediately transforms the event from an XML document to a flat record. When a continuous query application produces an output event, the Event Server transforms the event from a record to an XML document. When you define event types for a continuous query application, you must take care only to define types that can be appropriately rendered in record format.

The following lists some specific points to keep in mind when defining event types for a continuous query application:

- The Event Type Editor allows you to define the cardinality of fields in the schema. In event instances in an input stream, each field defined in the schema must be present once and only once in each event instance. Therefore, the cardinality in the schema must always be restricted to specifying exactly one instance of each field.
- The Event Type Editor allows you to create hierarchical structures by using composite fields with subfields. Event instances in input streams do not have hierarchical structures, but you can nevertheless use composite fields in the event type, to any hierarchical depth you require. The fields of the event type that are relevant for the structure of event instances are the simple fields that are contained in the composite fields.

  For example, you could define a composite field called "address" in the event type, containing the simple fields "street", "number", "city" and "zipcode". The relevant fields for the input stream would be the fields "street", "number", "city" and "zipcode".

  In queries that process the input stream, you would refer to these fields as "address$street", "address$number", "address$city" and "address$zipcode". For an illustration of this principle, see the Device Monitoring example in *Getting Started with Complex Event Processing*.
- The Event Query Language (EQL) does not directly support the use of EQL keywords or Java type names as field names. If you wish to use an EQL keyword or a Java type name as a field name in an EQL query, you must enclose the field name in quotation marks in the query. See

the description of the ObjectName statement in the *EQL Reference Guide* document for more information.

For information about creating and editing event types, see the separate documentation for the Event Type Editor.

Working with the Event Type Store

When you define an input stream, the Input Stream wizard includes a field where you can specify the name of the event type to be used for the input stream. Similarly, when you use the Input Stream Editor, there is a field where you can provide the name of the event type. In both cases, there is a **Browse** button that allows you to browse all defined lookup locations for event types.

One of the lookup locations is the location of the Event Type Store. The Event Type Store provides a central location where predefined event types are stored. When you install webMethods Business Events, an entry is made in the list of Eclipse preferences, defining the location of the predefined event types. You can see or modify the setting of this preference under **Preferences > Software AG > Events**. For more information about the lookup path, see the separate documentation for the Event Type Editor.

You can make the predefined event types visible in the Software AG Designer as follows:

To make the predefined event types visible in the Software AG Designer

1 In the Software AG Designer, select **File > Import**.

2 Select **General > File System** to indicate that you wish to import from the file system.

3 Specify a location in the file system where the required predefined event types are stored. By default, this is: *C:\SoftwareAG\common\PredefinedEventTypes*.

4 Specify the project folder where the selected predefined event types will be imported to.

5 Click **Finish**. The predefined event types will then be copied to the specified folder.

Publishing and Retrieving Event Types from CentraSite

If your organization is using CentraSite for SOA governance, you can add event types to, and retrieve event types from the CentraSite registry. For example, if your organization publishes its event types to CentraSite, you can search the CentraSite registry for the event type you need and drop it into your continuous query project.

Continuous query applications use event types that you might wish to publish in the CentraSite Registry Repository. The ability to publish to CentraSite is available also in the Event Type Editor.

For a description of the available functionality for publishing to CentraSite, see the documentation for the Event Type Editor.

5 Working with Input Streams

Introduction

An input stream identifies a stream of events from a specified topic on the Event Bus. An input stream has an associated event type, which describes the structure of the events that the input stream carries. An input stream can carry only one type of event.

A continuous query application can consume one or more input streams.

Note: You can only use input streams to send data into a continuous query. You cannot publish the data from the input stream as events.

Chronon Streams

A *chronon* is defined as a non-decomposable time period at the finest time granularity supported by the product, namely one millisecond. In a chronon stream, each event lasts the minimum amount of time possible for an event, namely one millisecond; therefore the end timestamp of an event in a chronon stream is the start timestamp plus one millisecond.

The special feature of chronon streams is that temporal windows may be applied to them by using WINDOW clauses in queries. Using a WINDOWS clause on a stream that is not a chronon stream can lead to extreme processing overload and is therefore not allowed.

Time-Ordering of Events in an Input Stream

The Event Server expects events in an input stream to arrive in the chronological order of the starting timestamps stored in the events.

Depending on the way in which event producers publish events to the input stream, it can happen that the starting timestamps are not in chronological order. This can happen, for example, if the stream is fed by multiple event producers whose clocks are not synchronized, or if an event producer does not guarantee that events will be produced in chronological order. If the Event Server receives an event X that has a starting timestamp that is before the starting timestamp of the previously received event Y, the event X will be rejected and an error will be reported in the error log.

You can modify the behavior of the Event Server to permit fluctuations in the chronological order for a given input stream. The dialog **Input Processing** of the input stream definition allows you to re-order out-of-sequence events in the input stream before they are processed.

Important: Re-ordering events increases the processing latency accordingly, since it activates a buffer as a sorting area for events before the events are released to the application. Therefore this feature might not be suitable for use in time-critical real time monitoring applications.

Creating an Input Stream Definition

To create a new input stream definition, proceed as follows:

To create a new input stream definition

1 In the Continuous Query Development, choose **File > New > Input Stream**.

This opens the wizard for creating a new input stream.

2 In the field **Project**, specify the name of the project in which you want to store the input stream definition.

3 In the field **Name**, specify a name for the new input stream. The name of an input stream must begin with a letter and include only the following characters:

- Letters A-Z or a-z
- Digits 0-9
- The following special characters: _ $

Note: The Event Query Language (EQL) does not directly support the use of EQL keywords or Java type names as stream names. If you want to use an EQL keyword or a Java type name as a stream name in an EQL query, you must enclose the stream name in quotation marks in the query. See the description of the ObjectName statement in the *EQL Reference Guide* document for more information.

4 In the field **Event Type**, specify the name of the Event Type upon which events in the new input stream will be based.

You can click **Browse** to see a list of the available Event Types. When you click **Browse**, a dialog is displayed that shows a flat list of the Event Types in all of the defined locations in the lookup path. In the dialog, you can use wildcards at the beginning or at the end of the Event Type name to filter the selection of Event Types. If, for example, you specify "obj*" as the name of the Event Type, the display with show all Event Types in the lookup path that begin with the letters "obj".

If two or more Event Types in the lookup path have the same name, the paths of each of the Event Types are shown in the dialog.

The lookup path contains all of the following Event Types:

- Custom Event Types within the same Eclipse project
- Custom Event Types in the projects to which you explicitly declare a reference (**Project > Properties > Project References**)
- Predefined Event Types in the Event Type Store.

For more information about the lookup path, see the separate documentation for the Event Type Editor.

For more information about the Event Type Store, see *Working with the Event Type Store*.

For more information about creating Event Types, see *Creating and Editing an Event Type*.

5 Click **Finish**.

The input stream definition is created and stored as a new file in the current project.

At this point, the input stream definition exists, but you still need to supply further data to complete the definition. To complete the definition, perform the editing steps described below.

To edit an input stream definition, proceed as follows:

To edit an input stream definition

1 Open the editor by doing one of the following:

- Create a new input stream definition using the wizard as described above. When you close the wizard, the editor opens automatically.

  Or:

- Select the existing input stream definition file from the project tree and open it using any of the standard Eclipse methods.

2 Select the tab **General**.

The **Event Type** field contains the name of the Event Type for the input stream. If you want to use a different Event Type, you can click **Browse** to obtain a list of defined Event Types.

For information about creating Event Types, see *Creating and Editing an Event Type*.

3 Specify whether you want to specify an upper limit for the duration of an event in the input stream.

- To specify an upper limit, select **Interval**, and specify an amount of time by entering an integer value and selecting a unit of time from the list.

For example, to specify 5 milliseconds as the maximum allowed duration of an event, enter "5" as the integer value and select "Milliseconds" from the list.

The smallest possible duration you can specify is 1 millisecond, and the term *chronon* is used to refer to this. An input stream that consists entirely of events of the smallest possible duration is called a *chronon stream*. Events in such a stream are valid at exactly one time instant (for example, sensor measurements). The query language provides the WINDOWS operator that you can use when querying a chronon stream. For more information on querying chronon streams, see ***Notion of Time in Continuous Queries***.

When you specify an upper limit for the duration of an event, Event Server logs and discards any event that exceeds this limit.

- If you do not want to specify an upper limit, select **Unlimited**.

Note: This setting is relevant in a high availability environment for synchronizing master and slave. The larger the upper limit is, the longer the time required for synchronization.

4 To check whether the Event Type you selected contains the expected fields, click the **Event Schema** tab. This tab displays a list of fields defined in the Event Type. The list contains the following information for each field:

- Name of the field.
- Data type of the field.
- Flag to indicate whether the field is mandatory.

The displayed names are the names that you need to use when creating queries for the input stream. Usually the name relates 1:1 to the name of the corresponding element in the XSD document, but could be different, for example if complex (nested) types are used. For an example of nested types, see *Complex schema structures* in ***Conversion from Schema Syntax to Event Sequence Syntax***.

5 Specify how events are fed into the input stream in the server environment using the **Input Processing** tab.

For the binding, you can use either a JMS connection or the Business Events client API.

- To use a JMS connection for the binding, select **JMS connection** and fill in the fields described below. Event Server uses the JMS binding properties to receive events from the Event Bus. During the deployment process, Event Server connects the input stream to the Event Bus using settings you specify on this tab.
  - In the **JMS connection alias** field, specify the name of the JMS connection alias to use for the JMS binding.
  - In the **Channel** field, specify the name of the JMS topic to use for the JMS binding.

If you have specified the Event Type of the input stream in the **General** tab, the Input Stream editor automatically assigns default JMS binding values. You can modify the name by editing the **Channel** field. If you later decide you want to revert to the automatically generated name, click **Assign default values**.

The default channel name has the format `Event::Namespace::EventTypeName`. If the Event Type of the input stream is located in the Event Type Store, the namespace is the path within the Event Type Store where the Event Type is located. Backslash characters in the path are replaced by the channel name delimiter (by default, two colons "::").

For example, if the Event Type is *AuctionType* that is located in the *MyEventTypes/BusinessTypes* folder in the Event Type Store, the Input Stream editor automatically assigns channel name `Event::MyEventTypes::BusinessTypes::AuctionType`.

Note: If you are using webMethods Broker as the Event Bus, be aware that it has certain conventions for topic names. For example, the total number of characters in a topic name cannot exceed 256, and some characters (such as "/" and "-") are not permitted in topic names. If you want to ensure that the topic name you enter in the **Channel** field complies with the naming rules, activate the **Enable use of webMethods Broker rules** option in the Eclipse preferences under **Software AG > Continuous Event Development**. When this option is activated, topic names that do not comply with the rules are marked as errors when you save your input, and an error message appears in the **Problems** view.

The JMS connection options are for JMS communication with the Event Bus. Event Server uses them to make a subscription to the Event Bus to receive events. An input stream defined in a continuous query application subscribes to the topic specified in the **Channel** field. When an event is published to the topic, the continuous query application receives the event as a subscriber of that topic.

When you select **JMS connection**, the values for the JMS binding are required when you deploy the application to the target server. See *Deploying a Continuous Query Application to an Event Server*. For local testing of your application, you do not need to supply a value for these fields.

- To use the Business Events client API for feeding events into the input stream, select **Use Business Events client API for input connection**.

  With this option, ensure you have a Business Events client that is connected to the input stream and delivers events based on that connection. Otherwise the input stream will not provide events to subsequent queries.

  When using this option, there is no need for JMS topics or triggers for the input stream. As a result, this option can save performance and bandwidth.

6 Specify if you wish to allow re-ordering of events in the input stream, in cases where the required chronological ordering of the events might not be guaranteed. The section **Time-Ordering of Events in an Input Stream** gives examples of how this situation can occur, and also points out that re-ordering events might be unsuitable for use in certain applications.

The re-ordering mechanism uses a buffer to collect and chronologically sort events before they are passed to the continuous query. In the field **Interval**, specify the maximum time difference allowed in the buffer between the newest starting timestamp and the oldest starting timestamp.

Note: The interval value refers to the comparison of the starting timestamps stored in the events, not to the elapsed real time when the Event Server is running.

When a new event arrives, the following actions are possible:

- If the starting timestamp of the new event is newer than the newest starting timestamp in the buffer, the new event is added to the buffer as the newest event.

  Then, any existing events in the buffer with a starting timestamp older than the starting timestamp of this event by at least the amount given in the field **Interval** are released from the buffer and passed to the continuous query in the correct chronological order.
- If the new event has a starting timestamp that is older than the newest timestamp already in the buffer, but the starting timestamp is within the stated Interval, the new event will be added to the buffer at the correct sorting position.
- If the new event has a starting timestamp that is older than the newest timestamp already in the buffer, and the starting timestamp is also outside the stated Interval, the new event will NOT be added to the buffer. In this case, the event is considered to be too old to be included in the sorting process, so instead it is passed directly to the continuous query.

The default value of **Interval** is 0, indicating that no sorting of events takes place.

Example:

Assume that the interval is set to 60 seconds, and 5 events arrive in the input stream in the following order, with the given starting timestamps:

```
EVENT, IS, ...T10:00:01, ...
EVENT, IS, ...T10:01:01, ...
EVENT, IS, ...T10:00:03, ...
EVENT, IS, ...T10:01:05, ...
EVENT, IS, ...T10:00:04, ...
```

The steps involved in the re-ordering of this example are as follows:

1. The events with starting timestamps T10:00:01 and T10:01:01 arrive, and are in the required chronological order. The timestamps are 60 seconds apart, which is within the stated interval, therefore they are both added to the sorting buffer.
2. The event with starting timestamp T10:00:03 arrives. This is out of chronological order compared to the timestamp of the newest event (T10:01:01), but is within the 60 second interval that we have set. Therefore this event is moved to the correct position in the sorting buffer (between the events with the timestamps T10:00:01 and T10:01:01).
3. The event with starting timestamp T10:01:05 arrives. This timestamp is newer than the newest event in the sorting buffer, so the event is added to the sorting buffer as the newest event.

   Taking the interval of 60 seconds into account, the oldest starting timestamp allowed in the sorting buffer is now T10:01:05 - 00:01:00 = T10:00:05. This means that the elements with starting timestamps T10:00:01 and T10:00:03 are removed from the sorting buffer and passed to the continuous query.
4. The event with starting timestamp T10:00:04 arrives. This timestamp is 61 seconds before the timestamp of the newest event (T10:01:05), so it is not in chronological order and is also outside the 60 second interval that we have set. Therefore this event will not be considered for sorting, and is passed directly to the continuous query.

   Since the latest event that was passed to the continuous query was the event with starting timestamp T10:00:03 in the step above, the event with starting timestamp T10:00:04 will (in this example) still be in correct chronological order, since the events with newer timestamps T10:01:01 and T10:01:05 are still in the sorting buffer. Had the timestamp of the event been T10:00:02 instead of T10:00:04, it would have been rejected by the continuous query as being out of chronological sequence compared to the event with starting timestamp T10:00:03.
5. At some later time, more new events will arrive in the sort buffer and will finally displace the events with timestamps T10:01:01 and T10:01:05 from the buffer, at which time these two events will be passed to the continuous query.

Therefore the final order of events that arrive at the continuous query will be:

```
EVENT, IS, ...T10:00:01, ...
EVENT, IS, ...T10:00:03, ...
EVENT, IS, ...T10:00:04, ...
EVENT, IS, ...T10:01:01, ...
EVENT, IS, ...T10:01:05, ...
```

7 You can see the XML serialization of the input stream object by selecting the **Source** tab.

Software AG Designer uses the **Source** display for highlighting errors identified at a later stage by the builder.

8 Save your changes.

Renaming an Input Stream

Use the following procedure to rename an input stream.

To rename an input stream

1 In the Project Explorer, select the input stream that you want to rename..

2 Choose **File** > **Rename**.

3 Specify a new name for the input stream. The name of an input stream must begin with a letter and include only the following characters:

- Letters A-Z or a-z
- Digits 0-9
- The following special characters: _ $

Important: Do not modify or delete the file extension. An input stream must have the extension ".stream".

4 Click **OK**.

5 If this input stream is referenced by queries in the project, update those queries so that they refer to the stream by its new name. (Queries that refer to this stream by its old name are listed in the **Problems** view.)

Tip: You can also perform the rename operation from the context menu for the input stream.

6 Querying an Event Stream

Querying an Event Stream Using EQL

This chapter describes how to define a continuous query using Event Query Language (EQL). EQL enables you to query an event stream using a language that is virtually identical to SQL. Using EQL, you can create queries that utilize the full range of continuous-query capabilities provided by the Event Server.

Note: You can alternatively use the Continuous Query Modeler to build a query using graphical elements. The Continuous Query Modeler enables you to create a query without having to know EQL. If you are not familiar with SQL, it might be easier for you to use the Continuous Query Modeler than EQL. For more information about using the Continuous Query Modeler, see ***Building a Query Using the Continuous Query Modeler***.

Main Features of the Event Query Language

In a Complex Event Processing (CEP) system, there are one or more input streams in which events are being continuously transported. These potentially unbounded streams can be analyzed in real-time and the results presented in the form of graphical dashboards or other output formats. A powerful feature of CEP is its ability continuously to detect complex events in incoming simple events. A *simple event* is a raw data item that includes temporal information, often in the form of a timestamp. A *complex event* is a specific combination of simple events; it typically represents a meaningful pattern, trend or relationship. For example, a complex event could be the fraudulent usage of a credit card. You specify the criteria for the detection of a complex event using query statements that are expressed in the Event Query Language.

webMethods Business Events follows a declarative approach to the Event Query Language, which is based on the ANSI standard SQL language (ANSI X3.135-1992, Entry SQL), for the following reasons:

- Its standardized language syntax and semantics are well understood and widely used. This ensures high developer productivity and acceptance.
- SQL provides a high level of abstraction, due to the principle of independence of logical and physical data. This means that the underlying data, the data representation and data management are cleanly separated from the applications that use the data. In turn, data independence enables the system to perform sophisticated query optimizations.

Some aspects of the Complex Event Processing scenario differ from the database scenario for which the ANSI SQL standard was developed; therefore the Event Query Language differs in some respects from ANSI SQL. The main changes can be summarized as follows:

- In a conventional data system, data are stored and queries are transient. By contrast, in a CEP system queries are stored and data are transient.

- A CEP system executes continuous queries; in other words, once a query has been subscribed to the system, it is evaluated continuously over the input streams. As new data arrive, new query results are produced and streamed directly to the data sinks that are connected.
- Continuous queries incorporate a notion of time, so the Event Query Language has been extended to allow queries along the time axis.
- The Event Query Language can be used only for READ operations on the input streams; there are no constructs in the Event Query Language for writing events to an input stream or to external data storage such as a file or database. The Event Query Language can be used to filter and combine input streams to create transient output streams that exist only for the duration of the query.
- Several query features that are used when dealing with large volumes of data in a database are not meaningful for querying input streams, so such features have been omitted. An example is the standard SQL capability of ordering query output using criteria such as alphabetical values in table columns, which is meaningless in the context of continuous input and output streams.

The specification of a continuous query resembles the formulation of a query in native SQL using the common clauses such as `SELECT`, `FROM`, `WHERE`, and `GROUP BY`. Whilst orienting itself as closely as possible on SQL, the Event Query Language supports most of the standard query functionality, for example, control over duplicates, nested subqueries, aggregates, quantifiers, and disjunctions. In addition to standard SQL features, the Event Query Language also provides powerful extensions for event analysis, such as **pattern matching** functionality.

Notion of Time in Continuous Queries

The basic goal of CEP is to derive complex events from the simple events that are streaming in. In this context, a crucial feature is the ability to combine content-based and temporal information given by the events. For example, an indicator of fraudulent credit card usage is an unusually high number of transactions within a short time frame, i.e., not only the number of transactions and the amounts are relevant but also the transaction times. In order to incorporate temporal information in query processing, webMethods Business Events exploits the *time interval approach*.

The Time Interval Approach

The time interval approach refers to the format of input and output streams of a query. Each event in a stream is defined as a pair $(e,[t_1,t_2))$, in which e is the data part and $[t_1,t_2)$ is the validity interval. This interval describes the validity of the event, i.e., the event is valid in the time between start timestamp t_1 (inclusive) and end timestamp t_2 (exclusive). (Note: Square brackets $[a,b]$ denote a closed, or inclusive, interval, which includes its endpoints; parentheses (a,b) denote an open, or exclusive, interval, which excludes its endpoints. Square brackets and parentheses can be mixed in an interval specification, for example $[t,t+10)$, which denotes the interval from t inclusive to t+10 exclusive.) Concerning the ordering of the stream, the events are always in non-descending chronological order with respect to their start timestamps.

The validity intervals are crucial for the computation of query results. The events in the query output stream that are valid for a given time instant are defined by the result that a database system would compute for the same query if given the events that are valid in the continuous input streams at this instant. In other words, this criterion, which is known as *snapshot reducibility*, ensures that the results of the query are exactly the same as they would be if the events had been stored in a database and then the query had been evaluated per time instant.

See also the section ***Theoretical Foundation of Query Semantics***.

In general, milliseconds are the finest time granularity in query processing and a *chronon* refers to a millisecond tick on the time axis.

In the time interval approach, we distinguish the chronon stream as a special case of the more general event stream. A chronon stream has the special property that a query relating to a chronon stream can include WINDOW clause, and it is defined as follows:

Chronon stream

Each event in the stream has a validity interval equal to the minimum interval duration, which is one millisecond. Thus, the end timestamp t_2 must be equal to the start timestamp $t_1 + 1$.

webMethods Business Events also allows events to be specified by just the data part and the start timestamp, t_1; the end timestamp is implicit in this case and is set automatically to $t_1 + 1$. This is common in real-world scenarios, as many applications only stamp events with a single timestamp.

Window Specification for Continuous Queries

In general, an event from a chronon stream should contribute to the query result for time longer than one millisecond, so you can extend its validity. For this purpose, the Event Query Language of webMethods Business Events supports a number of temporal windows that can be applied to a chronon stream.

When a query with a temporal window is specified, the validity of each incoming chronon event is extended to the window range. For example, given the query

```
SELECT MIN(X) FROM A WINDOW(RANGE 5);
```

each incoming event `(in,[t,t+1))` is given the validity interval `[t,t+5)`. Thus, the query only considers events that are in the current time window of length five milliseconds. As new events arrive, the window continuously slides forward.

The operators in the query – in the case of the example, the aggregation operator – process the events with time intervals.

The output stream that is yielded by the query also consists of events with time intervals. If the example yielded the result `(out,[5,8))`, this would indicate that at the times 5, 6, and 7 the value "`out`" was the minimum value in the corresponding last 5 milliseconds; in other words, "out" was

the minimum value in the intervals `[1,5]`, `[2,6]`, `[3,7]`. The interval notation is a concise and convenient way of expressing this. Note that the windowing concept allows the specification of queries using different window sizes over the same chronon stream.

To model the sliding window concept in continuous queries, we have enhanced the FROM clause by adding a WINDOW specification. To define a window over a stream, specify the window after the stream reference in the FROM clause. This enables us to let windows slide directly over streams, e.g., to express windowed stream joins. Rather than defining an entirely new syntax for windows from scratch, we have adopted the WINDOW specification of SQL:2003.

Windows in SQL:2003 are restricted to built-in OLAP (online analytical processing) functions and thus are only permitted in the SELECT clause. In contrast, we use windows in the FROM clause in order to specify the streams to which they should be applied. As we are solely considering preceding windows, we do not use the keyword PRECEDING from SQL:2003.

The grammar and all details of the window specification are shown in the description of the WINDOW clause in the *EQL Language Reference*.

Derived Streams and Nested Queries

The result of a continuous query is also a stream, which we denote as a *derived stream*. Such a derived stream can also serve as input for other queries. A derived stream can be compared to the views that are familiar from traditional database systems. A derived stream is a subquery that bundles and makes available frequently-used query functionality. The stream is derived only once, but it can be used in many queries; therefore, using derived streams can save system resources. Additionally, derived streams make the language more powerful, as complex queries can be expressed more easily and more intuitively. The Hot Items Query illustrates the usage of derived streams. A derived stream can be accessed by its unique name, under which the corresponding query was registered at the system.

Besides the ability to use derived streams as subqueries, the Event Query Language also supports nested queries, i.e. unnamed subqueries specified within a query. Note that as we only permit windows over **chronon streams**, windows can only be applied to derived streams that are chronon streams; this is appropriate for the majority of continuous queries. More details can be found in the section **Support for Nested Windows**.

Event Query Language Keywords

The following list shows all the keywords, and the clauses in which they are used. The keywords are reserved in the grammar, i.e., they cannot be used directly as identifiers in Event Query Language statements.

Note: If you wish to use an EQL keyword as an identifier in an EQL query, you must enclose the identifier in quotation marks in the query. See the description of the ObjectName statement in the *EQL Reference Guide* document for more information.

ABS

FunctionCall

ALL

SelectWithoutOrder
SetClause
AggregateFunctionCall
UserDefinedAggregateFunctionCall
SetCondition

AND

SqlAndExpression
SqlBetweenClause
SetCondition

ARRAY

CastableDataTypeDeclaration
ArrayValueConstructor

AS

Alias
DefineItem
FunctionCall
WithClause

AVG

AggregateFunctionCall

BETWEEN

SqlBetweenClause
SetCondition

BIG_DECIMAL

CastableDataTypeDeclaration

BLOB

CastableDataTypeDeclaration

BOOLEAN

CastableDataTypeDeclaration

BY

WindowClause
ShiftClause
MatchingClause
GroupByClause

BYTE

BasicDataTypeDeclaration
CastableDataTypeDeclaration

CASE

FunctionCall

CAST

FunctionCall

CEIL

FunctionCall

CEILING

FunctionCall

CHARACTER

BasicDataTypeDeclaration
CastableDataTypeDeclaration

CHARACTER_LENGTH

FunctionCall

CHAR_LENGTH

FunctionCall

CLOB

CastableDataTypeDeclaration

COUNT

AggregateFunctionCall
SetCondition

CROSS

JoinedTable

DATE

CastableDataTypeDeclaration

DAY

NonConstantTimeUnit

DAYS

NonConstantTimeUnit

DEFAULT

MeasureItem

DEFINE

DefineClause

DISTINCT

SelectWithoutOrder
SetClause
AggregateFunctionCall
UserDefinedAggregateFunctionCall

DO

DefineItem

DOUBLE

BasicDataTypeDeclaration
CastableDataTypeDeclaration

DURATION

MatchingClause

ELSE

SimpleCaseCall
BooleanCaseCall

END

SimpleCaseCall
BooleanCaseCall

EXCEPT

SetClause

EXP

FunctionCall

FALSE

Constant

FLOAT

BasicDataTypeDeclaration
CastableDataTypeDeclaration

FLOOR

FunctionCall

FROM

FromClause

GROUP

GroupByClause

HAVING

HavingClause

HOUR

ConstantTimeUnit

HOURS

ConstantTimeUnit

IN

SqlInClause
SingleFieldSetCondition

MultipleFieldSetCondition

INNER

JoinedTable

INTEGER

BasicDataTypeDeclaration
CastableDataTypeDeclaration

INTERSECT

SetClause

IS

IsNullClause

JOIN

JoinedTable

LIKE

SqlLikeClause

LN

FunctionCall

LONG

BasicDataTypeDeclaration
CastableDataTypeDeclaration

MATCHING

MatchingClause

MAX

WindowClause
AggregateFunctionCall
SetCondition

MEASURES

MeasureClause

MILLISECOND

ConstantTimeUnit

MILLISECONDS

ConstantTimeUnit

MIN

AggregateFunctionCall
SetCondition

MINUS

SetClause

MINUTE

ConstantTimeUnit

MINUTES

ConstantTimeUnit

MOD

FunctionCall

MONTH

NonConstantTimeUnit

MONTHS

NonConstantTimeUnit

NATURAL

JoinedTable

NOT

SqlUnaryLogicalExpression
SqlInClause
SqlBetweenClause
SqlLikeClause
IsNullClause
FieldDefinition

NOW

WindowClause

NULL

IsNullClause
Constant

FieldDefinition

ON

JoinedTable

OR

SqlExpression

PARTITION

WindowClause
MatchingClause

PATTERN

MatchingClause

POWER

FunctionCall

RANGE

WindowClause

RECURSIVE

WithClause

REF

CastableDataTypeDeclaration

RELATIVE

WindowClause

ROUND

FunctionCall

ROWS

WindowClause

SECOND

ConstantTimeUnit

SECONDS

ConstantTimeUnit

SELECT

SelectWithoutOrder

SHIFT

ShiftClause

SHORT

BasicDataTypeDeclaration
CastableDataTypeDeclaration

SLIDE

WindowClause

SQRT

FunctionCall

STDDEV

AggregateFunctionCall

STDDEV_POP

AggregateFunctionCall

STDDEV_SAMP

AggregateFunctionCall

STRING

BasicDataTypeDeclaration
CastableDataTypeDeclaration

STRUCT

CastableDataTypeDeclaration

SUM

AggregateFunctionCall

THEN

SimpleCaseCall
BooleanCaseCall

TIME

CastableDataTypeDeclaration

TIMESTAMP

CastableDataTypeDeclaration

TO

WindowClause

TRUE

Constant

TRUNC

FunctionCall

TRUNCATE

FunctionCall

UDA_

S_UDA_IDENTIFIER

UDF_

S_UDF_IDENTIFIER

UDO_

S_UDO_IDENTIFIER

UNBOUNDED

WindowClause

UNION

SetClause

UNKNOWN

CastableDataTypeDeclaration

USING

JoinedTable

VARIANCE

AggregateFunctionCall

VAR_POP

AggregateFunctionCall

VAR_SAMP

AggregateFunctionCall

WEEK

NonConstantTimeUnit

WEEKS

NonConstantTimeUnit

WHEN

SimpleCaseCall
BooleanCaseCall

WHERE

WhereClause

WINDOW

WindowClause

WITH

WithClause

WITHIN

MatchingClause

YEAR

NonConstantTimeUnit

YEARS

NonConstantTimeUnit

Additional Notes on Event Query Language Statements

This section lists additional notes to consider while writing Event Query Language statements.

- Efficient Evaluation of Equi-Joins and Natural Joins
- Naming of Subqueries
- Naming of Input Stream after Window Declaration
- Support for Nested Windows

Efficient Evaluation of Equi-Joins and Natural Joins

Given two streams, both an equi-join and a natural join check for the equality of attributes. Both joins can be efficiently implemented by a hash join, which outperforms a standard nested-loops join significantly. The software currently does not include a query optimizer, which would automatically choose the better implementation. Therefore, the hash-join implementation for an equi-join or a natural join must be explicitly specified in the statement by coding the USING clause, which is part of the JoinedTable clause (see also the *Short Auctions Query*).

Naming of Subqueries

Subqueries must be explicitly named in an Event Query Language statement in order to be correctly translated. In the current version of the software, this refers to all subqueries used in the FROM clause.

Naming of Input Stream after Window Declaration

To name an input stream, code the optional ALIAS clause ("AS" and the name) after the WINDOW(…) clause. For an example, see the *New Users Query*.

Support for Nested Windows

The software does not currently support nested windows; in other words, a WINDOW Clause cannot be applied to a subquery.

Comments in Event Query Language Statements

For the sake of readability, it can be helpful to comment Event Query Language statements. The language allows single-line comments and also comments that can extend over more than one line: the syntax is defined under LINE_COMMENT and MULTI_LINE_COMMENT.

```
-- this is a single line comment
SELECT Bid.*
/* this is a
multiline comment */
FROM Bid;
```

Language Extensions

Besides the window constructs, the Event Query Language includes additional SQL extensions with powerful event analysis capabilities.

- Recursion
- Pattern Matching
- Temporal Shift
- Set Membership
- User-Defined Functions
- User-Defined Aggregates
- User-Defined Operators

Recursion

Recursion is another language extension that adds great analytical power. Using recursive queries, you can uncover and leverage nested relationships that are hidden in the events.

In general, a continuous query can use another query as a subquery. In particular, a recursive query is a query that uses itself as the subquery.

A recursive query is specified by the WITH clause followed by SelectWithoutOrder, which has three parts:

1. A recursive data stream. The recursive data stream is assigned a name, and its relational metadata are specified with the FieldDefinitionList clause.
2. A recursive subquery that defines the recursive data stream. The subquery can refer to all defined data streams, including the recursive data stream specified in the first part of the WITH clause. The subquery specifies the computation of the recursive query. This implies that the recursive data stream is both the input and the output of the recursive query; therefore the relational metadata of the recursive subquery must correspond to the FieldDefinitionList. The following constructs are currently not allowed in the subquery: UNION without ALL, MINUS, INTERSECT, WINDOW clause, DISTINCT.
3. A query that uses the recursive subquery. This query is registered to the system and can be accessed from other queries; by contrast, the recursive subquery can only be accessed within this query.

Click the link to see the syntax of the WITH clause. SelectStatement represents a general SQL query, with or without a recursive subquery.

The following example illustrates the use of recursion. It recursively determines the transitive closure for a stream of directed edges.

```
WITH RECURSIVE Closure (startPoint STRING NOT NULL, endPoint STRING NOT NULL) AS
(
    SELECT *
    FROM EdgeStream
  UNION ALL
    SELECT a.startPoint, b.endPoint
    FROM EdgeStream a JOIN Closure b ON a.endPoint = b.startPoint
)
SELECT * FROM Closure;
```

When setting up recursive queries, please keep the following notes in mind:

Notes:

1. There is no mechanism for loop detection, therefore the query can yield an unlimited number of results for a single time instant. In the example, this would occur if the graph contained cycles.
2. Aggregation is only allowed in combination with grouping, and it requires independent groups, i.e., a group generated by a recursive result must not overlap with a previous group.
3. The SHIFT BY clause is not supported in recursive queries.
4. Recursion is a very powerful mechanism and it is often the only practical way to generate the desired result. However, if a non-recursive alternative is feasible it should be used, because its performance will most likely be better than the performance of the equivalent recursive version.

Pattern Matching

A common requirement in continuous query applications is the detection of patterns in event streams. The expressive power of patterns in terms of SQL queries is limited; additional language constructs for sophisticated pattern matching have therefore been provided.

A pattern is a sequence of events with event attributes that match certain conditions. Following the CEP paradigm, these conditions also include temporal relationships, for example the occurrence of a given pattern within the last 10 minutes. Analogously to the SQL approach, the semantics underlying the pattern matcher are well-defined and ensure deterministic, reproducible results.

A pattern matching query takes a stream as its input and delivers a chronon stream as its output. It is important to note that valid matches are only defined for event sequences with strictly monotonically increasing start timestamps. If events satisfy the pattern conditions but have equal start timestamps, they do not qualify for a match.

If events have equal timestamps, the query engine has to consider the possible combinations with subsequent events due to snapshot reducibility considerations; therefore, a pattern matching query that refers to a stream that may contain many events with equal timestamps can be computationally very expensive and can generate output events at a very high rate.

For optimum usability, the pattern matching constructs are seamlessly integrated into the Event Query Language. For that purpose, the FROM clause has been extended by the addition of the MATCHING clause.

All clauses in the MATCHING clause except the PATTERN clause are optional. The clauses of the pattern matcher are described in detail below:

- **PARTITION BY Clause**
  With this parameter the input stream can be grouped with respect to the specified list of columns (similar to GROUP BY in SQL). Then the pattern matching runs separately for each group.
- **MEASURES Clause**
  The MEASURES clause defines the output schema of the pattern matcher as a set of output variables, i.e. the schema of valid matches. Each output variable can be initialized with a default value. Besides these variables, the output schema also contains two timestamps that describe the temporal range of the pattern. The first timestamp corresponds to the start timestamp of the first event of the sequence constituting a pattern, and the second timestamp corresponds to the end timestamp of the last event. The timestamps can be accessed with their attribute names `START_TIMESTAMP` and `END_TIMESTAMP`.
- **PATTERN Clause**
  The pattern to be matched is defined as a regular expression over the pattern variables defined in the DEFINE clause. The regular expression itself must be enclosed in single quotation marks. All regular expression metacharacters except [^] can be used.
- **DURATION Clause**
  The DURATION clause can be used to set the time after which a match becomes valid. More precisely, a match is returned if a sequence matches a pattern and no other event arrives until the duration expires, i.e., during the time span start timestamp given by the first event of the sequence plus duration. This concept is important for the detection of non-events. It should only be used for that class of patterns. If no time unit is set explicitly in the DURATION clause, milliseconds are used by default.
- **WITHIN Clause**
  The WITHIN clause defines a time span as an additional condition. Given a sequence that satisfies the conditions of the regular expression, the time span between the start timestamp of its first event and the end timestamp of its last event must be less than the specified duration. If no time unit is set explicitly, milliseconds are used by default.
- **DEFINE Clause**
  In the DEFINE clause, pattern variables are used to define the conditions on events of the input stream. The condition of a pattern variable is defined as a predicate, i.e. a logical expression, in the AS clause. In the predicate the attributes of the input stream and the output variables of the MEASURES clause can be accessed. The pattern variables in turn are used in the regular expression of the PATTERN clause. As a unique and powerful feature of the pattern matcher, additional functions can be invoked if an incoming event fulfills the condition of a pattern variable. These functions are specified in the DO clause (see the DefineItem clause) and can be used to modify output variables, e.g. to increment or substitute them. The functions can access the attributes of

the input stream and the output variables. In the DO clause the functions are separated by commas.

Summarizing: the pattern matcher delivers the MEASURES variables as a valid match if a sequence of events meets the conditions in the DEFINE statement as specified in the PATTERN statement. Additionally, the sequence must meet the optional time constraints specified in the DURATION and WITHIN clauses.

Notes:

1. A pattern matching query can also be used as a subquery in other queries, i.e., the stream of detected patterns can be processed by other queries.
2. The names of pattern variables and output variables must be unique.
3. As a pattern matching query delivers a chronon stream as its output, you can additionally define a window over the query.

Temporal Shift

As previously described, an event is always attributed with a time interval that defines its validity. In some applications, it is necessary to shift the time intervals. For this time shifting, the FROM clause is extended by the SHIFT BY clause. The temporal offset in this clause is added to the start and end timestamps of each event.

```
SELECT *
FROM Source, Source SHIFT BY 2 HOURS;
```

In the above example, we compare the current behavior of a stream with its behavior two hours ago. Once the query is started, it takes two hours until the first result is produced.

```
SELECT *
FROM Source WINDOW(RANGE 5 MINUTES) SHIFT BY 2 HOURS;
```

In this example, a sliding window of size five minutes and an additional time shift of two hours are applied to the stream. As a result, the validity of chronon input events is extended by five minutes and shifted by two hours. For example, an event valid at 10:00 will become valid from 12:00 to 12:05.

Notes:

1. The usage of temporal shifts over chronon streams requires a careful shift setting as the timestamps of the original stream and the shifted stream may not match.
2. The shift operator can be used in combination with the WINDOW clause, as illustrated in the example above.

Set Membership

With set membership, we provide another powerful SQL extension. Given a stream, you can determine a group of events with attribute values that are in a pre-defined set of values. More precisely, you can specify the number of set values that have to be in the corresponding attribute subset of the group (without ordering). You can then evaluate aggregates over this group of events, or use them in combination with a grouping. Formally, set membership is specified with the SetCondition. The set membership conditions ALL, COUNT *n*, MIN *n*, MAX *n*, BETWEEN *m* AND *n* define how often the different attribute values of the group events have to be in the set:

The following example determines the number of bids for items with automat 3, 42, 65 in the last ten minutes:

```
SELECT COUNT(automat)
FROM Bid WINDOW(RANGE 10 MINUTES)
WHERE automat IN (3, 42, 65) ALL;
```

A crucial aspect to keep in mind is that the group of events that fulfill the set membership condition must have a shared overlap of time intervals.

User-Defined Functions

You can extend the query language by defining your own functions. Such a function is called a **user-defined function** (UDF).

A UDF must be implemented as a public static function method in a Java class. The method must meet the following requirements:

- The method name must be unique, independent of the signature.
- The number of arguments of the method must be fixed, not variable.
- The method must return a value; it cannot be defined as a method with return type "void".

The class is a public class whose name you can choose freely, as long as it is a unique class name. It must extend the class `UserDefinedFunctions`.

Here is an example:

```
package org.demo.company;
...
public class SampleUDF extends UserDefinedFunctions
{

    @UserDefinedFunctionProperties(name = "sum")
     public static Integer sum(Integer operand1, Integer operand2)
     {
```

```
        return operand1 + operand2;
    }
}
```

In this example, the class name is `SampleUDF`. The class defines a method `sum()`, which returns an integer value that is the sum of two input values.

The method must be preceded by the annotation `@UserDefinedFunctionProperties`. You can include the `(name=AssignedName)` property in this annotation to assign an explicit name to the user-defined function. Assigning an explicit name to a UDF enables you to refer to the UDF in a continuous query using the form:

`UDF_AssignedName()`

Note that when referenced in a query, the name of the UDF is always prefixed with the characters "UDF_".

For example, the annotation `@UserDefinedFunctionProperties(name = "sum")` enables you to refer to the user-defined function using the name "UDF_sum", as shown in the query below.

```
SELECT *
FROM inputStream
WHERE UDF_sum(value1, value2) > 1000;
```

If no name is specified in the `@UserDefinedFunctionProperties` annotation, you must refer to the UDF by its fully qualified method name as shown below:

`UDF_PackageName_ClassName_MethodName()`

In this form, each period character (".") in the package name must be replaced by an underscore.

For example, if you omit the "name" property in the sample code above, i.e. you write `@UserDefinedFunctionProperties` instead of `@UserDefinedFunctionProperties(name = "sum")`, you would refer to the UDF in a query statement as shown below:

```
SELECT *
FROM inputStream
WHERE UDF_org_demo_company_SampleUDF_sum(value1, value2) > 1000;
```


Note: If you assign a name to a user-defined function using the `@UserDefinedFunctionProperties` annotation, you *must* use the `UDF_AssignedName()` form when referring to the UDF in a query statement.

User-Defined Aggregates

User-define aggregates enable you to apply customized aggregate functions to event streams. A user-defined aggregate is a Java class that you create to perform an aggregation over the events in a window of time. You create a user-defined aggregate when you need to summarize events in a manner that cannot be accomplished using the built-in aggregate functions provided by EQL.

Within an EQL statement, you can use user-defined aggregates in the same places you can use the built-in aggregate functions. When used in a query, the name of the user-defined aggregate must be prefixed with the characters "UDA_".

In the following example, a user-defined aggregate called "UDA_MEDIAN" evaluates the TxAmt attribute in a stream of sales events and returns the median value of that attribute over a 6-hour window.

```
SELECT UDA_MEDIAN(TxAmt)
FROM Sales WINDOW(RANGE 6 HOURS);
```

For more information about creating user-defined aggregates, see ***Developing User-Defined Aggregates***.

User-Defined Operators

User-define operators enable you to perform custom operations on event streams that Event Server receives. A user-defined operator is a Java class that you create to process an event stream (or multiple streams). You create user-defined operators when you need to analyze a stream of events in a manner that cannot be accomplished using EQL alone.

Within an EQL statement, a user-defined operator appears in the FROM clause (like other event sources). When referenced in a query, the name of the user-defined operator must be prefixed with the characters "UDO_".

In the following example, a user-defined operator called "UDO_SysCheck" analyzes incoming events from several sensors and returns a quality measurement based on the sensor readings.

```
SELECT *
FROM UDO_SysCheck(tempStream, pressureStream, turbidityStream)
WHERE qualityIndex > 1.5;
```

For more information about creating and using user-defined operators, see ***Developing User-Defined Operators***.

Creating and Editing a Query

Creating a Query using EQL

Use the following procedure to create a query using EQL.

Note: Keep in mind that a query specified in EQL cannot be converted to a query model or vice versa. To determine whether you want to create the query using the Continuous Query Modeler instead of specifying it using EQL, see ***Should You Use the Continuous Query Modeler or EQL?***.

To create a new query using EQL

1 In the Continuous Query Development perspective, choose **File > New > Continuous Query**.

2 Complete the **New Continuous Query** dialog as follows:

In this field...	Do the following...
Select the Approach:	Select **Use SQL with temporal extensions to express the query**.
Project	Specify the name of the project in which you want to store the query.
Name	Specify a name for the query. A query name must begin with a letter and include only the following characters: ■ Letters A-Z or a-z ■ Digits 0-9 ■ The following special characters: _ $

3 Click **Finish** to create an empty query.

4 Specify the query statement on the **Query** tab. You can use the content-completion feature to assist you. For information about content completion, see ***Syntax Highlighting and Content Completion***.

- For information about the keywords and clauses that you use to compose a query, see the *EQL Language Reference*.
- For examples of EQL statements, see the *Online Auctions demo* in *Getting Started with Complex Event Processing*.

5 If you want to assign an output type to the query, perform the steps described in ***Assigning an Output Type to a Query***.

> **Note:** You must assign an output type to the query if the query result will be published via JMS or if the query result will be used as an input stream in a query model.

6 Set the output processing parameters for the query as described in ***Specifying the Output Processing Options for a Query.***

7 When you finish defining your query, choose **File > Save** to save it.

> **Note:** Software AG Designer checks your query for errors during the save operation. Syntax errors are indicated as described in ***Highlighting of Syntax Errors.*** Semantic errors and warnings are reported in the **Problems** view.

Syntax Highlighting and Content Completion

In order to improve readability, the query editor highlights keywords.

The query editor also provides content assist, also known as content completion. To use this feature, type the key combination CTRL+SPACE. Based on the context of the current cursor position, the editor offers you a list of all elements that could be inserted here, in the form of a drop-down list. Select the desired entry in the list either with the mouse cursor or by using the up-arrow and down-arrow keys on your keyboard.

Highlighting of Syntax Errors

When you save the query, the query compiler checks whether the query is syntactically valid. If the query contains an invalid keyword or some other inconsistency, the query editor underlines the keyword or inconsistent component. If you place the cursor on the underlined keyword or component, a tooltip message is displayed, indicating the cause of the error.

If the query contains a syntax error, the icon in the header tab of the query editor changes to an error icon, to indicate that an error is present. Also, a description of the error is displayed in the Problems view, and the icons in the Project Explorer for the project, folder and file that contain the error all change into an error icon.

Customizing Syntax Highlighting

You can customize the settings for syntax highlighting in the Query Editor:

To customize the syntax highlighting settings

1 Click **Window** > **Preferences** to start the general preferences dialog.

2 Click **Software AG** > **Continuous Query Development** > **Query Editor**. (Alternatively, you can open this page by entering "Query Editor" in the filter text field.)

3 Click on the appropriate identifier and select the desired style.

The changes are displayed immediately in the preview window below.

4 To apply the changes, click on **Apply** or **OK**.

Assigning an Output Type to the Query

The query's output type describes the events that the query produces. Specifying an output type makes the query's output schema visible to other processes and consumers.

You *must* assign an output type to a query if the query will be used in any of the following ways:

- Its results will be published to a JMS provider.
- The query will represent an Input Stream operation in a query model.

If a query will not be used in any of these ways, you are not required to specify its output type. However, you might consider doing so as a best practice. Assigning an output type to a query increases the ways in which the query's output stream can be used in the design-time and run-time environments. If you do not assign an output stream to a query, Software AG Designer displays a warning in the **Problems** view.

You use the **Output Type** tab to assign an output type to a query. This tab is present both in queries that you specify using EQL and in queries that you model using the Continuous Query Modeler.

When assigning an output type to a query, keep the following points in mind:

- When you define a query using EQL, be sure that the event type you specify on the **Output Type** matches the SELECT clause with respect to the number and type of attributes that the clause specifies. For example, if you have the following query statement:

```
SELECT field1, field2, field3 FROM InStream;
```

The event type must contain exactly three attributes and the data types of those attributes must correspond to the types of `field1`, `field2`, and `field3` in the query. (Note that the names of the attributes in the event type do not need to match those used in the query. Only the number of attributes and their data types must match.) If the output type is not compatible with the query's SELECT clause, Software AG Designer displays an error in the **Problems** view when you save the query.

- When you model a query using the Continuous Query Modeler, the **Output Type** tab reflects the event type of the Output Stream operation in the query model. If you change the event type on the **Output Type** tab, the Continuous Query Modeler changes the type assigned to the Output Stream operation accordingly. If the model does not contain an Output Stream operation, the Continuous Query Modeler adds an Output Stream operation to the design canvas and assigns the selected event type to it. If you delete the event type from the **Output Type** tab, the Output Stream operation in the model becomes untyped.

To assign an output type to a query

1. In the query, display the **Output Type** tab.
2. Click **Browse** and select the event type that represents the schema of the query result.

   Note: If the event type that you want to use does not already exist, you can click **Generate Output Event Type** on the **Output Type** tab to create it.

Specifying the Output Processing Options for the Query

You use the options on the **Output Processing** tab to configure certain characteristics of the query's output stream. Characteristics you can specify include whether to publish events, whether to split events, and whether to include heartbeats in the output stream.

To specify the output processing options for a query

1. In the query, display the **Output Processing** tab.
2. If you want the output from this query to be visible in the **Console** view when you execute the query in Software AG Designer, enable the **Enable Console Output** option.

   Note: Disabling this option is useful when a query produces intermediate results for another query and you only want to see the results from the top-level query in the console.

3. If, you want Event Server to publish the output stream to external clients and subscribers, enable the **Enable query result output** option. You must enable this option when you want to:

- Send query results to JMS topics or NERV-defined endpoints
- Publish results to direct clients of Event Server (i.e., clients that subscribe to events using the Business Events client API)
- Examine query results in the Event Server Admin view

For queries that only provide intermediate results for other queries, you can save performance and bandwidth by disabling query result output. This is particularly relevant if you are running Event Server in high availability mode because in high-availability mode, the master sends all "published" results to the slave for consistency checks.

4 If you enabled the **Enable query result output** option in the step above, perform the following steps to specify the way in which you want Event Server to publish the stream.

1. Choose one of the following options to specify where you want Event Server to publish the output stream.

Select this option...	If you want to publish the output stream to...
Enable JMS output	JMS topics or NERV-defined endpoints. **Note:** Software AG Designer will not let you enable this option unless the output type for the query has been specified on the **Output Type** tab. For information about specifying an output type, see *Assigning an Output Type to a Query*.
Use direct connection of Business Events client API for result output	Clients that consume events using the Business Events client API.

2. If you selected the **Enable JMS output** option above, specify the following parameters to bind the output stream to a JMS topic:

In this field...	Specify...
JMS connection alias	The name of the JMS connection alias the query will use to connect to the JMS provider at run time.
Channel	The name of the JMS topic to which the query will publish its results. The format for a topic name is: `Event::Namespace::EventTypeName` By default, the topic name is derived from the event type specified on the **Output Type** tab. However, you can specify a different name if necessary. **Note:** If you change the output Event Type in the **Output Type** tab, and the JMS binding values are still the default values, the JMS binding values are automatically updated to reflect the new output type.

In this field...	Specify...
	Be aware that the topic name specified in this field must not be a channel that also represents an input stream. If the default name for the output channel is identical to the default name of an input channel (this can happen if input and output channels use the same event type), the automatically generated default name of the output channel is extended to include the query name as shown here: `Event::`*`Namespace`*`::`*`EventTypeName`*`::`*`QueryName`*

Note: If you are using the NERV framework to access the event bus, you can override these settings after the query is deployed. At that time, you can switch to a different JMS binding or to a non-JMS binding using a separate mechanism that is based on the output stream's event type. This mechanism involves including a *custom emit route* in the NERV configuration file. You create a custom emit route by defining a plugin project in Eclipse, and in it you define the required mapping between the initial default binding and the required custom binding. Then, you use the Asset Build Environment framework to build a deployment composite for the plugin project, and you deploy the composite using webMethods Deployer. For information on using the Asset Build Environment and webMethods Deployer, see the *webMethods Deployer User's Guide*. An example of a NERV configuration file is provided in the examples that are supplied with the product delivery.

5 Specify whether you want Event Server to split events in the output stream.

You use splitting to increase the liveliness of an output stream. When you enable output splitting, Event Server splits output events to make partial results available in the output stream as soon as possible. Given a single event, splitting produces multiple semantically equivalent events of shorter validity intervals. For example, an event (value, [start, end)) can be split into the two adjacent events (value, [start, intermediate)), (value, [intermediate, end)). Splitting is particularly intended for increasing the liveliness of output streams from queries that use aggregation, Group By with aggregation, or partitioned windows.

Be aware that when splitting is activated, it is possible that the splitting mechanism might create events in the output stream that have a difference of one millisecond between start and end times, depending on the exact nature of the events in the input stream and the query that processes them.

Note: Splitting increases the liveliness of the stream, but it can also increase the number of output events in the stream. The **minimum split length** enables you to make a tradeoff between liveliness and the output rate. As a general rule, if the system time correlates with application time, the split length should be approximately half as long as the maximum acceptable delay between an event entering the system and the corresponding events occurring in the output stream.

If you...	Do the following...
Do not want Event Server to split events.	Select **Splitting deactivated**.
Want Event Server to split events.	Select the **Minimum split length** option and specify the shortest validity interval into which an event can be split. For example, if you specify 60 seconds as the minimum duration of a split event, the splitting mechanism never creates a split event with a duration less than 60 seconds even if the splitting algorithm would allow creating output events with a shorter duration.

6 Specify whether you want to use heartbeats.

If you...	Do the following...
Do not want Event Server to use heartbeats.	Select **Deactivate heartbeat processing**. **Note:** If heartbeats are deactivated, the publishing of events into the output stream can be delayed.
Want Event Server to include heartbeats.	Select the **Minimum heartbeat interval** option and specify the minimum amount of time that will be allowed between two adjacent heartbeats in the query's output stream. **Note:** Changing the minimum heartbeat interval from the default setting of 1 millisecond can cause the publishing of events into the output stream to be delayed.

Consider performance issues when deciding whether to modify the standard heartbeat generation default. The generation of many heartbeats causes additional computations that are needed to process them. Additionally, if the query's output stream contains many more heartbeats than events, the event bus has to deal with a much higher volume of traffic than necessary, which leads to performance loss. High volumes of heartbeats can also negatively affect the performance of consumers of the query's output stream, since the consumers receive the heartbeats as well as the consumer-relevant events.

Viewing the Output Schema for a Query

You can view the schema of the query's output records in the **Output Schema** tab. The schema is displayed as a table containing the field name and field type for each field in the output stream. Additionally, if the **Required** box is marked for a field, this indicates that a non-null value is required for the field in the output stream. In an EQL query, the **Output Schema** tab reflects the output fields as defined by the SELECT clause in the query statement. In a modeled query, the **Output Schema** tab reflects the output fields as defined by the model's Output Stream operation.

To view the schema of a query's output records

- In the query, display the **Output Schema** tab.

Viewing the XML Serialization for a Query

The **Source** tab contains an XML serialization of the query. This is the form of the query that Event Server executes at run time. Examining the serialized XML can be useful in some troubleshooting situations.

The serialized version of the query is updated as you edit the query and also when you save the query.

The XML that appears on the **Source** tab is read-only.

To view the schema the XML serialization for a query

- In the query, display the **Source** tab.

Viewing or Editing a Query that was Written in EQL

Use the following procedure to view or edit a query.

To view or edit a query

1. In the Project Explorer, double-click the .ceq file that contains the query.
2. Modify the query as required.
3. When you finish editing the query, choose **File > Save** to save your changes.

Note: Software AG Designer checks your query for errors during the save operation. Syntax errors are indicated as described in *Highlighting of Syntax Errors*. Semantic errors and warnings are reported in the **Problems** view.

Copying a Query

Use the following procedure to copy a query.

Important: When you copy a query, Software AG Designer updates internal names that exist within the resulting copy. It also copies the .ceq file and the .cqm file if the query is expressed as a model. To ensure that the copy is generated properly, always use the following procedure to copy a query. Do not use the copy command in the host operating system.

To copy a query model

1 In the Project Explorer, select the .ceq file for the query that you want to copy.

2 Choose **Edit** > **Copy.**

3 Select the Queries folder in the project to which you want to copy the query.

4 Choose **Edit** > **Paste.**

5 Type a new name for the copy. The name of the copied query must begin with a letter and include only the following characters:

- Letters A-Z or a-z
- Digits 0-9
- The following special characters: _ $

6 Click **OK**.

Software AG Designer copies the .ceq file to the selected Queries folder. If the query is expressed as a model, Software AG Designer also copies the .cqm file to the .models sub-folder within the Queries folder. (If the .models folder does not already exist, Software AG Designer will create it.)

Tip: You can also perform the copy and paste operations from the context menu for the query.

Renaming a Query

Use the following procedure to rename a query.

Important: When you rename a query, Software AG Designer updates internal names that exist within the query file. It also renames the .ceq file and the .cqm file if the query is expressed as a model. To ensure that all of these updates are performed, always use the following procedure to rename a query. Do not use the rename command in the host operating system.

To rename a query

1 In the Project Explorer, select the .ceq file for the query that you want to rename.

2 2. Choose **File** > **Rename.**

3 Specify a new name for the query. The query name must begin with a letter and include only the following characters:

- Letters A-Z or a-z
- Digits 0-9
- The following special characters: _ $

Important: Do not modify or delete the file extension. The query file must have the extension ".ceq".

4 Click **OK**.

5 If this query is referenced by other queries in the project, update those queries so that they refer to this query by its new name. (Queries that refer to this query by its old name are listed in the **Problems** view.)

Tip: You can also perform the rename operation from the context menu for the query.

Deleting a Query

Use the following procedure to delete a query. If you use this procedure to delete a query that is expressed as a model, Software AG Designer deletes the .ceq file and the .cqm file associated with the query.

Important: To ensure that query files are removed from a project properly, always use the following procedure to delete a query. Do not use the delete command in the host operating system.

To delete a query model

1 In the Project Explorer, select the .ceq file for the query that you want to delete.

2 Choose **Edit** > **Delete.**

Tip: You can also perform the delete operation from the context menu for the query.

Advanced Query Processing Concepts

This section discusses the following concepts:

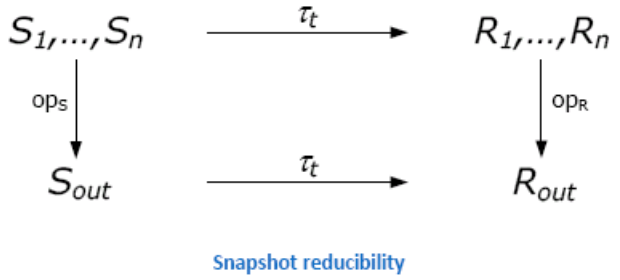
- An Overview of the Event Server's Query Processing Steps
- Theoretical Foundation of Query Semantics
- Plan Generation
- Interpretation of Query Results

An Overview of the Event Server's Query Processing Steps

The Event Server receives an Event Query Language (EQL) statement as input. First, this query is parsed and translated into an initial logical query plan, which is composed of operators from the logical algebra. The translator now translates the final logical query plan into a physical query plan. This is accomplished by replacing each operator in the logical plan by a suitable counterpart of the physical algebra. Note that a single logical operator can have several physical counterparts; for example, a logical join can be replaced by a physical hash or a nested-loops join. Finally, the physical plan is registered with the execution engine. The execution engine integrates the plan into the system's global query graph, which consists of all operative query plans, and initiates data processing.

Theoretical Foundation of Query Semantics

One of our design tenets was to provide well-defined, deterministic query semantics. The logical algebra precisely and directly defines the semantics of each operation by modeling event streams as temporal multisets. In order to preserve the well-understood semantics of the relational algebra as far as possible, the algebra contains a stream counterpart for each operator in the extended relational algebra, except for sorting. Each of these standard operators is snapshot-reducible. We denote a stream operation op_S with inputs $S_1, \ldots, S_n$ as snapshot-reducible to its relational counterpart op_R if, for any time instant `t`, the snapshot at time `t` of the result of op_S is the same as the result of applying op_R to the snapshots of $S_1, \ldots, S_n$ at time `t`. A snapshot of a stream at time `t` can be considered as a relation, since it represents the multiset of all tuples that are valid at this instant. **The figure below** illustrates the temporal concept of snapshot reducibility:



Snapshot reducibility

Thus, the logical algebra guarantees an exact meaning of any continuous query at any instant in time. The physical algebra provides corresponding operator implementations. The physical operators process physical streams, i.e., events with validity intervals, for performance reasons. Evaluating queries for each time instant at the finest granularity would be computationally too expensive, so instead we merge consecutive time instants to an interval.

Plan Generation

As **described before**, we derive a logical plan for a given Event Query Language statement. This process closely resembles query plan construction in conventional database systems. Since the FROM clause has been enhanced significantly to permit the specification of windows, we now clarify how the plan generator fits the novel windowing constructs into the plan. For this reason, we distinguish between standard operators and window operators. All stream operators derived from the extended relational algebra are standard operators. The novel window operator determines the validity of stream events according to its parameters, such as window size (`RANGE`) and advance (`SLIDE`) (see ***Window Specification for Continuous Queries***). Window operators define the scope of the operations downstream. The combination of window operators with stream variants of the relational operators creates their windowed analogs.

A logical plan is constructed as follows:

- At the bottom, we have input streams $S_1, \ldots, S_n$. These input streams are obtained either from raw streams or from derived streams (subqueries).
- Each input stream is processed by a window operator to achieve the desired window semantics.
- The rest of the plan generation process is the same as in a conventional database system. So, for example, the plan generator places above the window operators the standard query plan that would be generated by a database system if no window specification were specified. A standard query plan consists only of standard operators.
- Further queries and sinks that consume the results can be placed above the standard query plan.

Rather than integrating windows directly into operators, we have kept the functionality of the window operator and standard operators separate. As a consequence, we avoid redundancy and facilitate the exchange of window types. Moreover, any optimizations that are applicable in traditional database systems, for example join reordering and selection pushdown, can be applied to standard query plans, since they preserve snapshot reducibility.

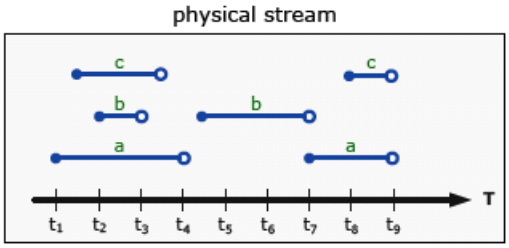
Thereafter, a physical query plan is generated by replacing each logical operator by its physical counterpart. The basic idea of the physical algebra is to use the novel window operator to adjust the validity of tuples, i.e. the time intervals, according to the window specifications, along with appropriately defined standard operators, so that the results of a physical query plan are snapshot-equivalent to its logical counterpart. While stateless operators do not consider the associated time intervals and thus can deal well with potentially infinite windows, time intervals affect stateful operators as follows. An event is relevant to a stateful operator as long as its time interval may overlap with the time interval of any future stream event. This also means that a stateful operator can purge from its state any event whose time interval can no longer intersect with the time interval

of any future incoming stream event. This explains how window specifications can successfully limit the resource demands of stateful operators in the physical algebra.

Interpretation of Query Results

The output of a continuous query corresponds to the output of the topmost physical operator of the corresponding query plan. As each physical operator generates a physical stream as its output, the output of a continuous query is a physical stream and, thus, is ordered by the start timestamps. Each query result (e, [t_S, t_E)) consists of a tuple e and a time interval [t_S, t_E) describing its validity. In other words, the tuple e is valid at each time instant t between time instant t_S (inclusively) and time instant t_E (exclusively). Note that a result stream may consist of duplicates having identical tuples and time intervals.

Arranging the query results and their validity intervals on a timeline provides a snapshot view. **The figure below** illustrates a stream of query results. For each time instant, you can determine the tuples that are valid with respect to the defined outcome of the query.



Example for query result stream

A crucial part of application development is to determine how the computed query results are to be processed in the corresponding sinks. The storage of query results in a log file, table, or database is straightforward. Appropriate result stream visualization, which is a common application requirement, must take into account the fact that one result is valid over a time period, and also that at a given time instant several results may be valid.

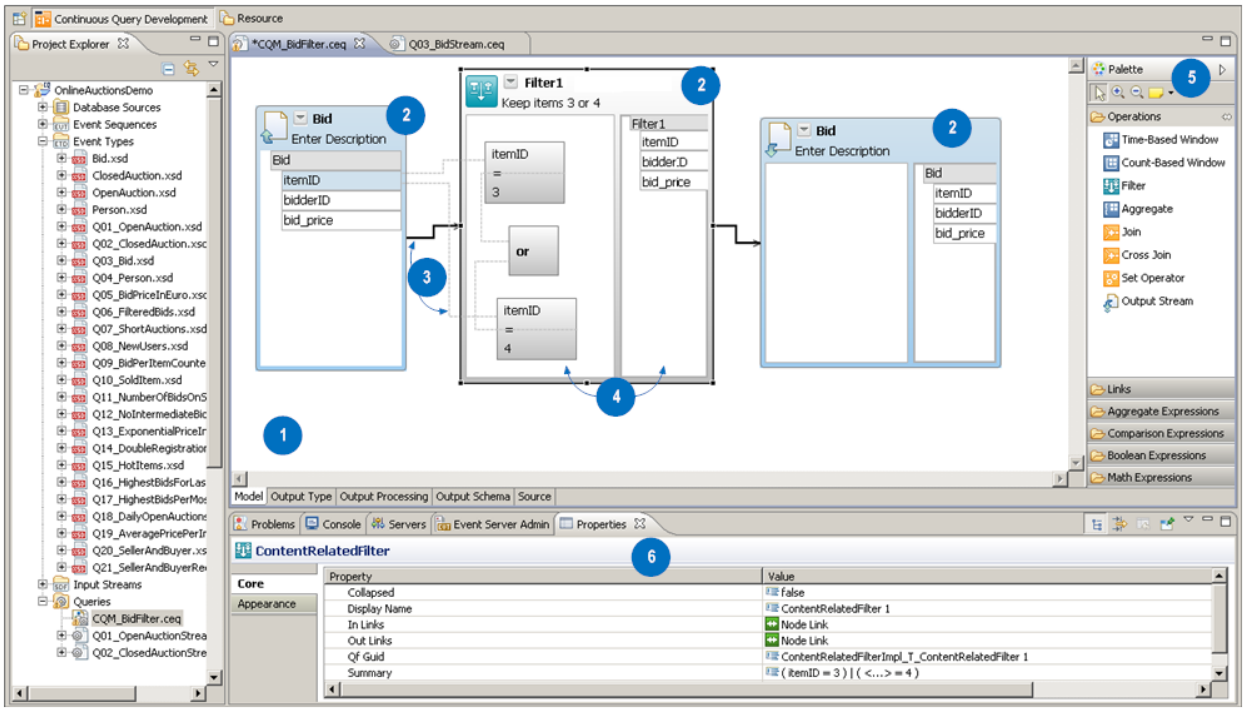
7 Building a Query Using the Continuous Query Modeler

The Continuous Query Modeler

The Continuous Query Modeler is a design-time tool that enables you to compose continuous queries without knowing EQL. When you specify a query using the Continuous Query Modeler, you define the query graphically using simple drag-and-drop techniques.

The **Model** tab in the Continuous Query Development perspective contains the design canvas and the tools you use to compose a query using the Continuous Query Modeler.

The Model tab



#	Feature	
1	**The design canvas.** The *design canvas* is the space on which you compose a query model.	
2	**Operations.** *Operations* represent logic that is to be carried out during the query. An operation represents a single step in the query process. Operations include: ■ Receiving an event from an input stream ■ Applying a filter or a window to an event ■ Joining event streams ■ Publishing an output event	
3	**Links.** *Links* connect operations and expressions within the query.	
		**Operation link.** A solid arrow represents the flow of execution between operations. You use operation links to indicate the order in which the operations in the model are to execute.
		**Expression link**. A dotted line represents a data mapping between or within an operation.

#	Feature
4	**Compartments.** *Compartments* hold data associated with an operation. Most operations contain two compartments: ■ The left compartment is editable. You add expressions to this compartment to configure the behavior of an operation. In a filter operation for example, you use this compartment to specify the filtering criteria. ■ The right compartment displays the output attributes that the operation produces. You cannot edit the contents of this compartment, but you can map data from this compartment to expressions in the next operation. Moreover, if the compartment resides in an Output Stream operation, you can map data from the preceding operation to the attributes in the compartment.
5	**The Palette.** *The Palette* supplies the tools and building blocks that you use to model a query.
6	**The Properties view.** *The Properties view* displays the property settings for the component that is currently selected on the design canvas. Certain properties in this view are editable and others are read-only.

Should You Use the Continuous Query Modeler or EQL?

Creating a query using the Continuous Query Modeler is easy and intuitive. If you are not familiar with SQL, it is often easier to specify queries using the Continuous Query Modeler instead of EQL. However, there are cases when using EQL is a more suitable choice. When deciding whether to compose a query using the Continuous Query Modeler or EQL, consider the following points:

- There are certain aspects of a continuous query that cannot be expressed in a query model. If your query requires these capabilities, you must specify it using EQL. For a list of the capabilities that cannot be modeled using the Continuous Query Modeler, see **Aspects of a Query that You Can Model**.
- Complex queries can consume a large amount of space on the design canvas and might become unwieldy to navigate and edit. If your query involves a large number of operations, consider specifying it in EQL or dividing it into multiple query models.

Important: Keep in mind that you cannot convert a query model to EQL or vice versa.

Aspects of a Query that You Can Model

Using the Continuous Query Modeler, you can model many kinds of continuous queries, including ones that perform joins, filtering, windowing, and aggregation. There are, however, certain aspects of a continuous query that cannot be expressed in a query model (listed below). If your query requires these capabilities, you can use the suggested alternative if one is provided below or use an EQL statement to express your query.

Note: In some cases you can break the work associated with a query into multiple queries and isolate the non-supported aspect of the query in an EQL statement. For example, if you wanted to detect a complex pattern in two joined and filtered event streams, you could join and filter the two streams in a query model. Then you could use an EQL query to perform

a pattern-matching analysis (which is not supported in a model) on the event stream the query model produces.

Query Features that are not Supported by the Continuous Query Modeler

- Pattern matching
- Recursion (that is, the capability provided by the WITH clause in EQL)
- User-defined extensions
- Use of database sources
- Temporal shifting (that is, the capability provided by the SHIFT BY clause in EQL)
- The IN clause (As an alternative, you can model this condition using an expression such as `a=1 OR a=2 OR a=3`)
- The BETWEEN clause (As an alternative, you can model this condition using an expression such as `a>=5 AND a=<10`)
- Use of the IS NULL or IS NOT NULL options
- The CASE function
- The CAST function
- The EXCEPT operator
- Filtering of aggregated groups (that is, the capability provided by the HAVING clause in EQL)
- Use of the DISTINCT option in aggregations
- Use of the DISTINCT option in an INTERSECT operation
- Use of the ALL option in UNION, MINUS and EXCEPT operations
- Use of an SQLExpression as an argument for an aggregation (only a SourceField is allowed)
- Nested queries
- + or - unary operators (As an alternative, you can use an expression such as `a * -1`)
- The String concatenation operator, `||`

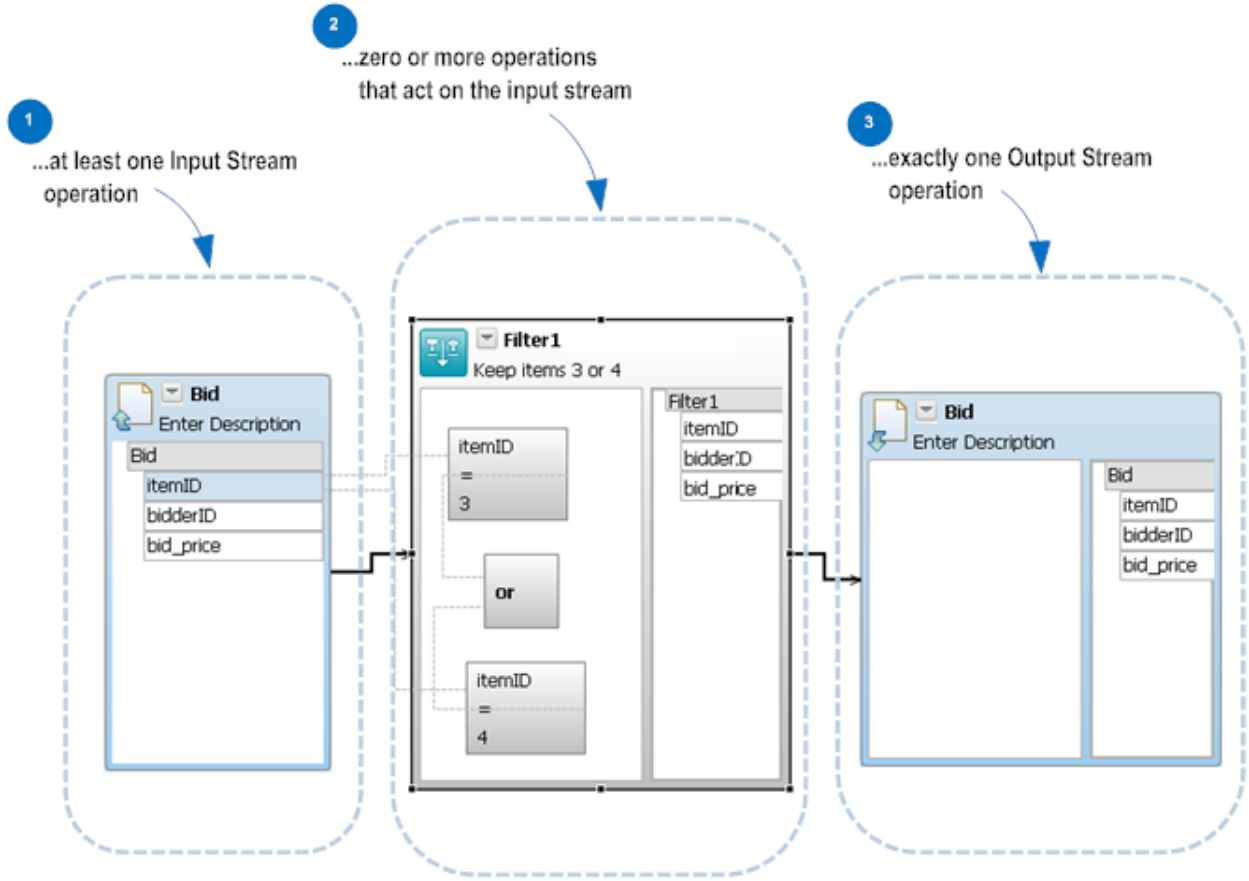
Modeling Basics

- Error Checking in a Query Model

A Basic Model

A query model is a diagram that specifies a sequence of operations that Event Server is to execute against one or more input streams. You arrange the operations in left-to-right fashion on the design canvas and create links to specify the order in which they are to be executed.

A query model must contain:

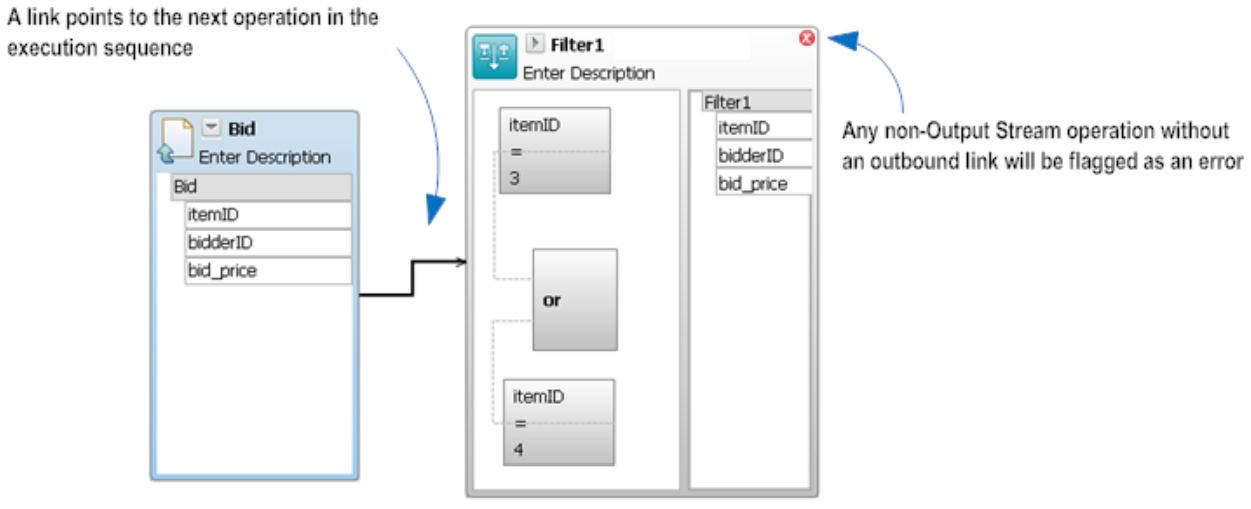


Segment	Description
1	An Input Stream operation identifies an event stream that is to be queried. The stream of events that an Input Stream operation identifies can be an input stream or a query. An Input Stream operation must be the first operation in a query model. If your model queries multiple event streams, it must include an Input Stream operation for each of those streams. For detailed information about adding an Input Stream operation to a model, see *Adding an Input Stream operation to the Design Canvas*.

Segment	Description
2	Following the Input Stream operation, you specify operations that act on the events in the stream. In this segment of the model, you can perform steps such as: ■ Filtering events ■ Aggregating events ■ Applying a window to events ■ Joining event streams ■ Combining or comparing streams using set operators such as UNION and MINUS For detailed information about these operations, see *Modeling Query Operations*.
3	An Output Stream operation returns the output event that the query produces (i.e., the query result). The Output Stream operation is always the last operation in the query model. A query model includes one and only one Output Stream operation. For detailed information about adding an Output Stream operation to a model, see *Adding an Output Stream operation to the Design Canvas*.

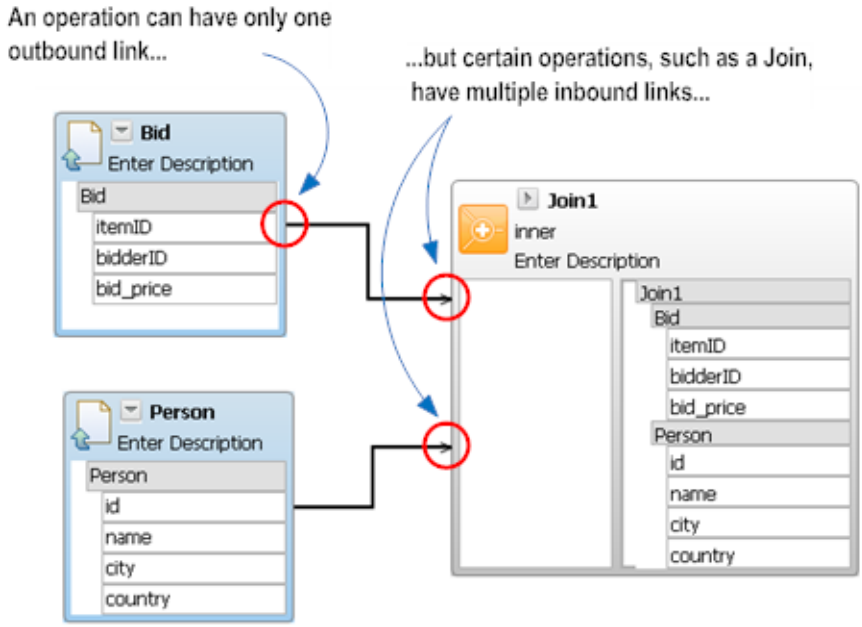
Operation Links

You use links to indicate the order in which operations in the model are to be executed. Except for the Output Stream operation (the last operation in a query model), every operation in a query model must include an outbound link that points to the next operation in the sequence of execution. If an operation is missing such a link, the Continuous Query Modeler will report an error.



An operation can have only one outbound link running from it (that is, it can identify only one operation as the "next step" in the sequence). However, certain types of operations can have multiple

links running into them. A Join operation, for example, has two inbound links, one from each of the input streams involved in the join.



For more information about creating links between operations, see *Linking Operations in a Query Model*.

Distinguishing Stream A from Stream B

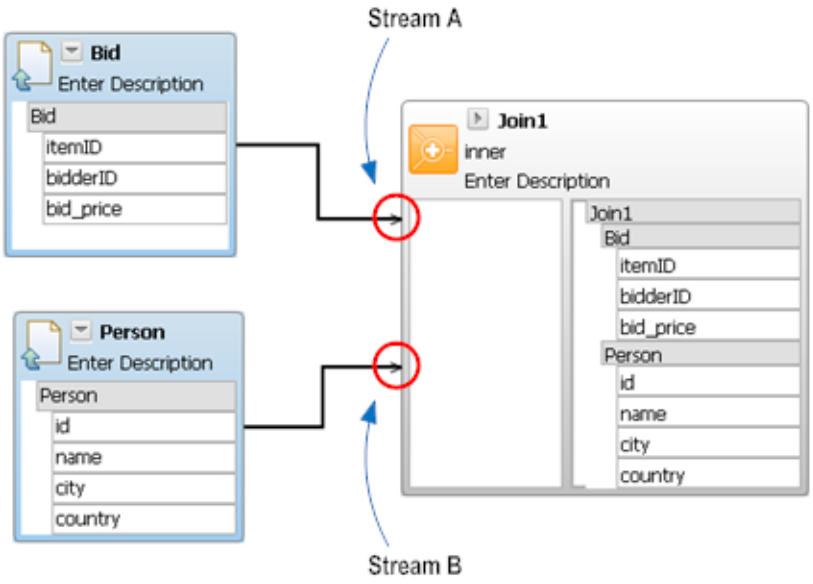
When an operation takes two event streams as input, the order in which the operation processes the streams can be significant. For example, the following operation:

```
StreamA MINUS StreamB ↩
```

produces a different result than the one below:

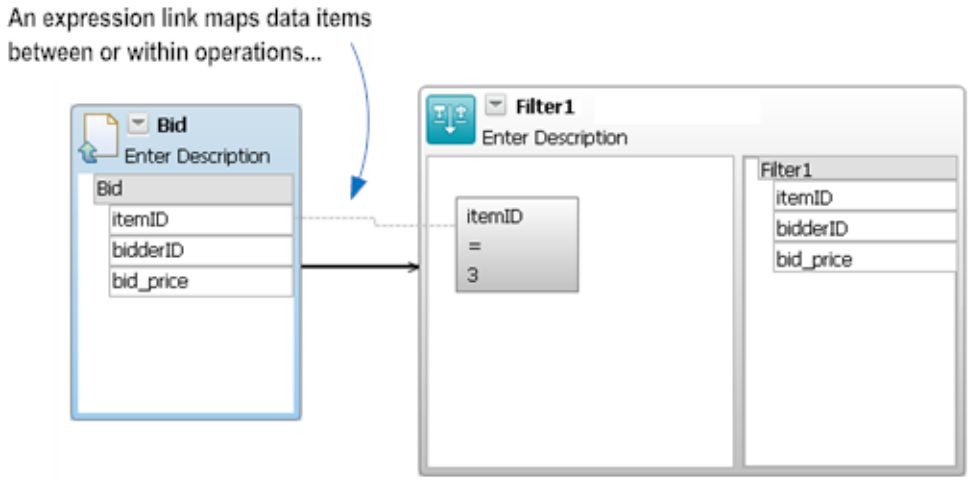
```
StreamB MINUS StreamA ↩
```

When an operation has two inbound links running into it, the Continuous Query Modeler treats the stream attached to the upper anchor as `stream A` and the stream attached to the lower anchor as `stream B`.



Expression Links

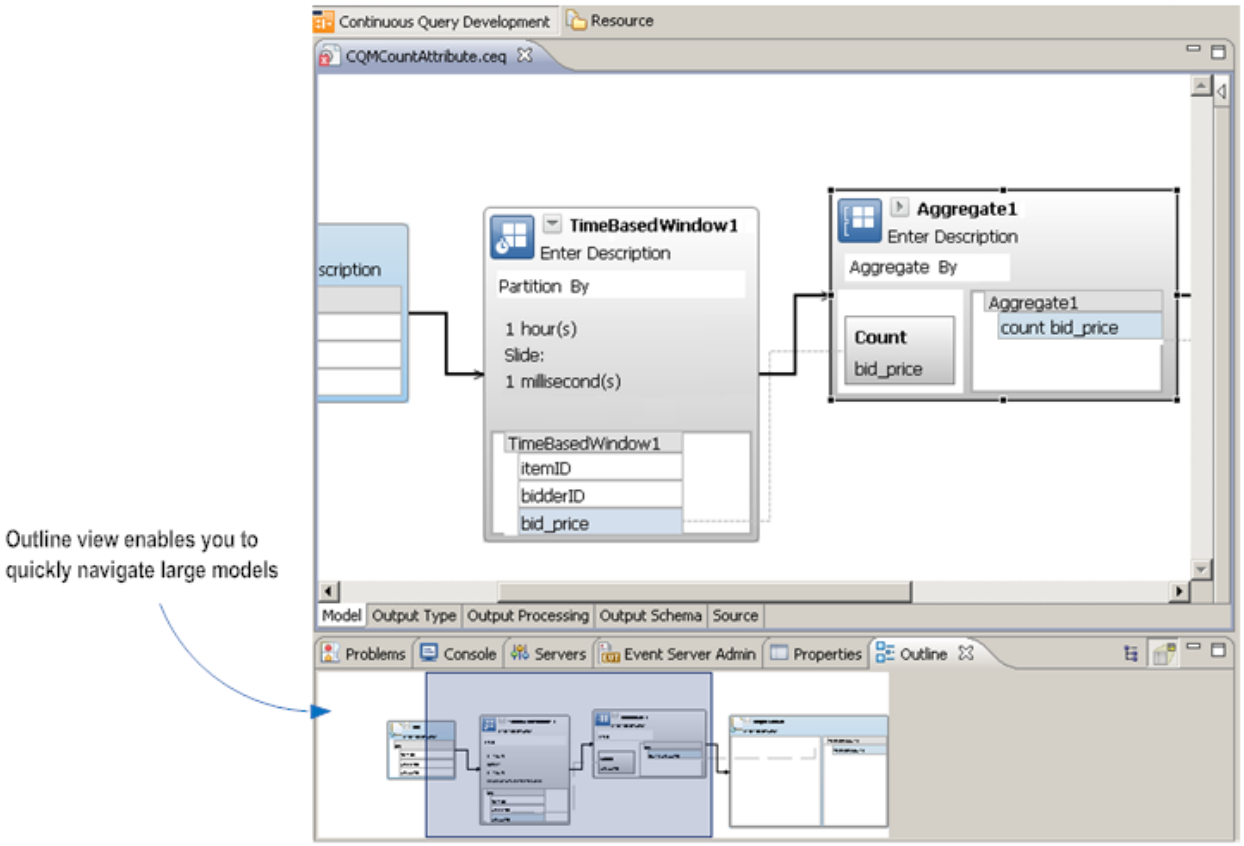
You use expression links to map data to and from fields in the model. When an operation executes, it receives as input the output tuple produced by the preceding operation. Often, you want to map data from this tuple to an expression in the operation that receives it. For example, in the model fragment shown below, an expression link is used to map the itemID attribute produced by the Input Stream operation to a comparison expression in the Filter operation.



For more information about creating expression links, see *Mapping Data Using Expression Links*.

Using Outline View to Navigate Large Models

When a model is large, you can use Outline view to quickly move around the design canvas.



To navigate the model in Outline view

1 If the **Outline** view is not currently displayed, do the following to display it.

1. Go to **Window** > **Show View** > **Other.**
2. In the **Show View** dialog, open the **General** folder and select **Outline**.
3. Click **OK**.

2 In the **Outline** view, slide the blue window to navigate to the section of the model with which you want to work.

Displaying Rulers and Grid Lines on the Design Canvas

Use the procedure below to display rulers and grid lines on the canvas and to enable the snap features.

Note: The following procedure affects settings for an individual model. If you want to enable the rulers, grid lines, and snap features by default (i.e., for all new models), set the **Rulers and Grid** options on the Preferences page. For procedures, see *Setting Preferences for the Continuous Query Modeler*.

To display rulers and grid lines

1 Click any empty area of the design canvas.

2 In the Properties view, select **Rulers & Grid**.

3 To display rulers, do the following:

1. In the **Display** group, enable the **Show Ruler** setting.
2. In the **Measurement** group, set the **Ruler Units** option to specify the scale in which you want to display the rulers.

4 To display grid lines, do the following:

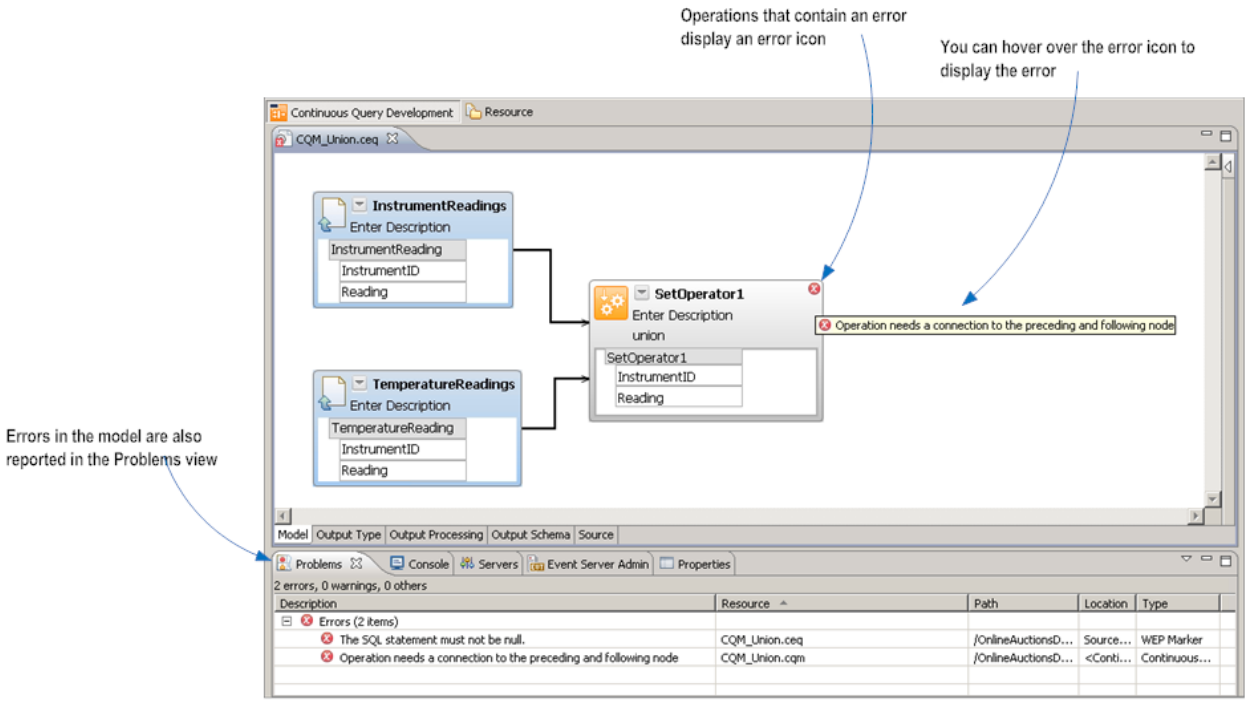
1. In the **Display** group, enable the **Show Grid** setting. If you want to superimpose the grid over the model, enable the **Grid In Front** setting. Disable this setting to display the grid behind the model.
2. In the **Measurement** group, set the **Grid Spacing** option to specify the density of the grid.
3. In the **Grid Line** group, set the **Color** and **Style** options for the grid.

5 To enable the snap features, do the following.

Enable this setting...	To...
Snap to Grid	Align objects (e.g., operations and expressions) with the grid when you move them or resize them. ■ When you move an object, the object's top and left edges snap to the grid. ■ When you resize an object, the edge that you drag with your mouse snaps to the grid.
Snap to Shapes	Display pop-up guidelines to help you align objects relative to one another. When **Snap to Shapes** is enabled, horizontal and vertical guidelines automatically appear when you move an object close to another object on the canvas.

Error Checking in a Query Model

When you save a query model, the Continuous Query Modeler compiles the query model into a form that Event Server can execute. Errors that occur during this process are listed in the **Problems** view. If you double-click an error in the list, the Software AG Designer displays the query model in which the error is located. If the error is associated with a particular operation in the model, an error indicator appears in the upper right corner of the defective operation. You can hover over the indicator to display the error message.



When you examine **Problems** view, be aware that:

- **Problems** view displays errors for all open projects, not just the query model you are viewing. Check the **Resource** column to determine the component to which an error refers.
- The error "The SQL statement must not be null" indicates that your model contains a syntax error. This error appears when you save a query model that is invalid with respect to modeling rules. It also appears before you save your query for the first time.
- Many errors produce two entries in **Problems** view. This is because a query model is composed of two files and the Continuous Query Modeler logs an error against each file. You can double-click either error to open the query model. When you correct the error and save the model, the Continuous Query Modeler will clear both errors from the **Problems** view.

Files Associated with a Query Model

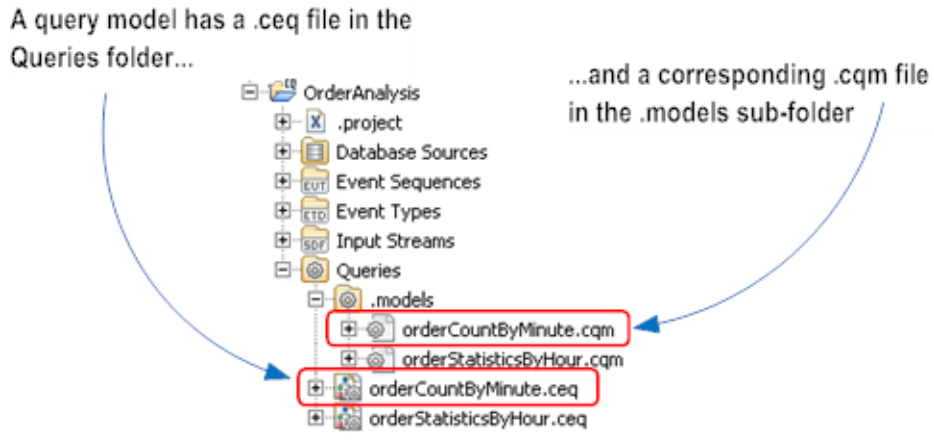
A query model consists of two files:

- The *query file* holds an XML representation of the query that is defined by the model. The query file has a .ceq extension and resides in the Queries folder. This file contains the query instructions that Event Server executes at run time.

  Note: In Project Explorer, different icons are used to distinguish .ceq files for modeled queries from .ceq files for EQL queries.

- The *model file* holds the graphical representation of the query. This file has a .cqm extension and resides in the .models sub-folder. You do not see the .models sub-folder unless you display hidden resources in the Project Explorer. For procedures, see *Displaying the .models Folder*.

Both files are generated automatically when you create a query model.



The ceq file is the primary file in the pair. It is the one you use to open a model and to copy, rename, and delete a model.

Important: When you need to copy, rename, or delete a query model, always use the commands provided by Software AG Designer. Do not use the file commands provided by the host operating system. For information about copying, renaming and deleting query models, see *Copying a Query*, *Renaming a Query*, and *Deleting a Query* .

Displaying the .models Folder

The .models folder contains the .cqm files associated with your query models. Because these files are for internal use only, the folder is hidden. On occasions when you need to examine the folder for troubleshooting purposes, you can use the following procedure to display its contents.

To display the .models sub-folder

1 Click the **View Menu** icon in the upper-right corner of the Project Explorer and select **Customize View** from the menu.

2 On the **Filters** tab, disable the **.* resources** filter.

3 Click **OK**.

Setting Preferences for the Continuous Query Modeler

You use the preference settings to specify default behaviors for the Continuous Query Modeler.

To set preferences for the Continuous Query Modeler

1 Go to **Windows** > **Preferences**.

2 In the navigation tree of the **Preferences** dialog, go to **Software AG** > **Continuous Query Development** > **Continuous Query Modeler**.

3 Set the preferences as follows:

In the navigation tree, select...	To set...
Continuous Query Modeler	Preferences that affect global behaviors of the Continuous Query Modeler. For details about these settings, see *Continuous Query Modeler Preferences*.
Continuous Query Modeler > **Connections**	Preferences relating to links that connect expressions and operations. For details about these settings, see *Connections Preferences*.
Continuous Query Modeler > **Rulers And Grid**	Preferences relating to rulers, gridlines and the snap features. For details about these settings, see *Rulers and Grid Preferences*.

Continuous Query Modeler Preferences

Use the **Continuous Query Modeler** preferences to specify global behaviors of the Continuous Query Modeler. Global settings apply to both existing models and to new models.

Preference	Description
Show connector handles	Specifies whether Continuous Query Modeler displays pop-up arrows when the cursor hovers over a linkable element in the model.
Show popup bars	Specifies whether Continuous Query Modeler displays the pop-up operations menu when the cursor hovers over an empty area of the design canvas.
Enable animated layout	Specifies whether the Continuous Query Modeler moves objects in an animated fashion when you use the **Arrange All** command to reorganize the layout of a model.
Enable animated zoom	Specifies whether the Continuous Query Modeler performs zoom actions in an animated fashion when you use the **Zoom-In** or **Zoom-Out** command to adjust the magnification level of a model.
Enable anti-aliasing	Specifies whether the Continuous Query Modeler uses anti-aliasing to smooth the edges of scalable figures during resizing operations.

Connections Preferences

Use the **Continuous Query Modeler** > **Connections** preferences to specify the line style for operation and expression links.

Note: Changing the **Connections** preferences affects new links that you add to a model. It does not change the appearance of links that you have already created.

<table>
<tr><th>Preference</th><th colspan="2">Description</th></tr>
<tr><td rowspan="3">Line style</td><td colspan="2">Specifies the style of line to use for operation and expression links.</td></tr>
<tr><td>Rectilinear</td><td> 
</td></tr>
<tr><td>Oblique</td><td> 
</td></tr>
</table>

Rulers and Grid Preferences

Use the **Continuous Query Modeler** > **Rulers and Grid** preferences to specify default settings for rulers, grid lines, and the snap features.

Note: Changing the **Rulers and Grid** preferences determines whether Continuous Query Modeler displays rulers and grid lines in new models you create. It does not change the ruler and grid line settings in existing models. To enable or disable rulers and grid lines in an existing model, you can use the **Ruler and Grid** settings in the Properties view. For procedures, see *Displaying Rulers and Grid Lines on the Design Canvas*.

Preference	Description	
Ruler options	Specifies whether rulers are displayed in new models.	
	Setting	**Description**
	Show rulers for new diagram	Specifies whether Continuous Query Modeler displays rulers on the top and left edges the design canvas.
	Ruler Units	Specifies the scale in which the rulers are to be displayed.
Grid options	Specifies whether grid lines are displayed in new models.	
	Setting	**Description**
	Show grid for new diagrams	Specifies whether Continuous Query Modeler displays grid lines on the design canvas.
	Snap to grid for new diagrams	Specifies whether Continuous Query Modeler aligns graphical elements on the grid when you move or resize them.
	Snap to shapes for new diagrams	Specifies whether Continuous Query Modeler displays pop-up guidelines to help you align objects with one another. When enabled, guidelines will automatically appear when you place the edge of one object against another.

Creating a Query Model

- Creating a New Query Using the Continuous Query Modeler
- Adding Operations to a Query Model
- Removing Operations from a Query Model
- Linking Operations in a Query Model
- Changing the Endpoint of an Operation Link or an Expression Link
- Rearranging the Layout of the Query Model
- Resizing an Operation on the Design Canvas
- Adding Expressions to an Operation

- Mapping Data Using Expression Links

Creating a New Query Using the Continuous Query Modeler

Use the following procedure to create a query using the Continuous Query Modeler.

Note: Keep in mind that a query model cannot be converted to EQL or vice versa. To determine whether you want to create the query using EQL instead of the Continuous Query Modeler, see ***Should You Use the Continuous Query Modeler or EQL?***.

To create a new query using the Continuous Query Modeler

Before you begin, make sure that your project contains the input streams and/or queries that your model will use as input.

1 In the Continuous Query Development perspective, choose **File > New > Continuous Query**.

2 Complete the **New Continuous Query** dialog as follows:

In this field...	Do the following...
Select the approach	Select **Use graphical toolset to express the query**.
Project	Specify the name of the project in which you want to store the query.
Name	Specify a name for the query.

3 Click **Finish** to create an empty model.

4 Model your query on the design canvas that is provided on the **Model** tab. For information about how to model a query, see the following topics:

- *Tutorial 2: Creating a Query Using the Continuous Query Modeler* in *Getting Started with Complex Event Processing*.
- ***Modeling Basics***
- ***Adding Operations to a Query Model***
- ***Linking Operations in a Query Model***
- ***Modeling Query Operations***

5 Set the output processing parameters for the query as described in ***Specifying the Output Processing Options for a Query.***

6 When you finish composing your query, choose **File > Save** to save it.

Adding Operations to a Query Model

The procedure you use to add an operation to a query model depends on the type of operation you want to add.

To add...	Use the procedure described in...
An Input Stream operation	*Adding an Input Stream operation to the Design Canvas*
An Output Stream operation	*Adding an Output Stream operation to the Design Canvas*
Any other operation	*Adding Operations to a the Design Canvas*

Adding an Input Stream Operation to the Design Canvas

Use the following procedure to add an Input Stream operation to your query model.

To add an Input Stream operation to the design canvas

1 In the Project Explorer, locate the input stream or query that represents the stream of events that you want to query.

2 Drag the input stream or the query to the **Model** tab and drop it on the design canvas.

Notes:

1. In a model, you can only use input streams and queries that reside in your current project. The Continuous Query Modeler will not allow you to drop input streams and queries from other projects on the design canvas.
2. To use a query as an Input Stream operation, the query must specify its output type (i.e., the query's **Output Type** tab must identify the event type that defines the query's output schema). If a query does not specify its output type, the Continuous Query Modeler will not allow you to drop it on the design canvas. For more information about the **Output Type** tab, see *Assigning an Output Type to the Query*.
3. If the schema of the underlying input stream or query changes after you add an Input Stream operation to a query model, Software AG Designer will report an error the next time the model is saved or rebuilt. You can correct the model by refreshing the Input Stream operation as described in *Refreshing an Input Stream Operation*.

Adding an Output Stream Operation to the Design Canvas

An Output Stream operation can be *typed* or *untyped*.

- *A typed Output Stream operation* has an associated event type. The event type describes the tuple that the query returns.
- *An untyped Output Stream operation* does not have an assigned event type. In other words, it does not declare its schema. An untyped Output Stream operation simply returns the tuple it receives from the preceding operation. Using an untyped Output Stream operation is analogous to using SELECT * in an EQL statement.

If you want the query to return a result whose schema is different from the schema that the query model produces implicitly, use a typed Output Stream operation and map the attributes from the preceding operation to it.

Note: The event type that you assign to the Output Stream operation represents the query's output type and is automatically reflected as such on the **Output Type** tab. When you change the event type for the Output Stream operation, the type assignment on the **Output Type** tab changes accordingly. Similarly, when you specify an event type on the **Output Type** tab, that event type is automatically assigned to the Output Stream operation. For more information about the **Output Type** tab, see *Assigning an Output Type to the Query*.

To add a typed Output Stream operation to the design canvas

1 In the Project Explorer, locate the event type that describes the tuple that you want the query to return.

2 Drag the event type to the **Model** tab and drop it on the design canvas. Note that the schema of the selected event type appears in the output compartment of the resulting Output Stream operation.

Notes:

1. You can only use event types that reside in your current project or reside in a project that is explicitly referenced in Project > Properties > Project References. The Continuous Query Modeler will not allow you to drop event type from unreferenced projects on the design canvas.
2. If the schema of the Output Stream operation differs from the schema of the operation that precedes it, you must map the two schemas. For procedures, see *Mapping Data to the Output Stream Operation*.

To add an untyped Output Stream operation to the design canvas

1 Open the **Operations** folder in the Palette.

2 Drag the Output Stream operation from the **Operations** folder and drop it on the design canvas. Note that the output compartment of the resulting Output Stream operation is empty, indicating that the operation has no associated event type.

To change the event type associated with an Output Stream operation

Use the following procedure to change the type assignment of an Output Stream operation.

1 On the design canvas, select the Output Stream operation.

2 In the **Properties** view, specify the event type in the **Event Type** property. (If you want to switch the Output Stream operation from typed to untyped, simply clear the current value from the **Event Type** property.)

> **Tip:** You can use the **Generate Output Event Type** command to quickly create an event type whose schema corresponds to the output structure of the operation that immediately precedes the Output Stream operation. To use this command, right-click anywhere in header area of the Output Stream operation and select **Generate Output Event Type** from the context menu. The Continuous Query Modeler automatically creates a new event type based on the output schema of the preceding operation and assigns the new event type to the Output Stream operation.

3 If you assigned a different event type to the Output Stream operation in step 2, re-map the operation's attributes as necessary. For procedures, see *Mapping Data to the Output Stream Operation*.

Adding Operations to the Design Canvas

The following procedures describe the ways in which you can add an operation to the design canvas. In all cases, the newly created operation is displayed with a predefined name. You can overwrite the name with a name of your choice, or you can click on an empty part of the design canvas to accept the predefined name.

If you wish to change the name at a later stage, refer to the section *Renaming Operations in a Query Model.*

The name must comply with the naming conventions described in *Naming Conventions for Operation Names*

> **Important:** The following is a general procedure. If you are adding an Input Stream or an Output Stream to the canvas, use the procedures described in *Adding an Input Stream Operation to the Design Canvas* and *Adding an Output Stream Operation to the Design Canvas* instead of the general procedure below.

To add an operation to the design canvas from the Palette

1 Open the **Operations** folder in the Palette and locate the operation that you want to add to your model.

2 Drag the operation from the **Operations** folder and drop it on the design canvas.

OR

Click the operation in the **Operations** folder and then click any empty area of the design canvas.

To add an operation to the design canvas from the pop-up menu

1 Move the mouse pointer to any empty space on the design canvas and wait for the following pop-up menu to appear:

2 Click the icon that represents the operation that you want to add to your model. (Tip: If you hover over an icon, the Continuous Query Modeler will display the name of the associated operation.)

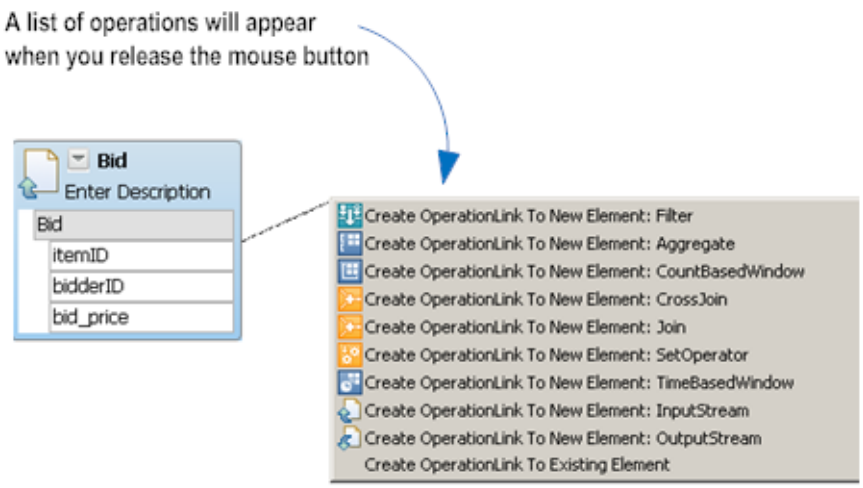
To add an operation to the design canvas using the Operation Link tool

To add an operation using this procedure, your model must already contain the operation that precedes (i.e., will execute immediately prior to) the operation that you want to add.

1 Open the **Links** folder in the Palette and select the **Operation Link** tool.

2 On the design canvas, locate the operation that precedes the operation that you are adding and do the following:

1. Click anywhere in the operation.
2. Drag the cursor to any empty area on the design canvas.
3. Release the mouse button to display the following menu.



3 Select the **Create NodeLink To New Element:...** entry for the type of operation that you want to add to your model.

Naming Conventions for Operation Names

Operation names have the following syntax:

- The name must contain at least one character.
- The first character must be an uppercase [A-Z] or lowercase [a-z] letter.
- Each subsequent character can be an uppercase [A-Z] or lowercase [a-z] letter, a digit [0-9], an underscore "_" or a dollar sign "$".

Renaming Operations in a Query Model

Use the following procedure to rename an existing operation.

To rename an operation

1 On the design canvas, locate the operation whose name you wish to change.

2 Click anywhere in the operation.

3 Open the **Properties** tab.

4 Click on the **Value** field for the **Name** property.

5 Enter the new value as required. This value is the name that is displayed for the operation in the design canvas.

The name must comply with the naming conventions described in *Naming Conventions for Operation Names*

Removing Operations from a Query Model

Use the following procedure to delete an operation from a query model. When you delete an operation, the Continuous Query Modeler will also delete the expressions that the operation contains and all links to and from the operation and its expressions.

To delete an operation

1 On the design canvas, select the operation that you want to delete.

Tip: You can hold the CTRL key down to select multiple operations.

2 Press Delete.

OR

Right-click the selected operation and choose **Delete from Model** from the context menu.

Linking Operations in a Query Model

The following procedures describe the ways in which you can link one operation to another.

Note that when you are linking operations, the shape of the cursor indicates whether the cursor's current position represents a valid endpoint for a link.

When the cursor is in this state...	It indicates that...
	The cursor is properly positioned to establish an endpoint.
	The cursor is not in a position where you can establish an endpoint. This might be because: ■ The cursor is not positioned over an operation. ■ The cursor is not positioned over the proper area within a given operation. (To establish a link to an operation, the cursor must be located over a section of the operation that is not occupied by a compartment.) ■ The given operation already has the maximum number of inbound links running into it.

To link operations using the Operation Link tool

Before you begin, locate the two operations that you want to link and arrange them from left-to-right according to the order in which you want them to execute.

1 Open the **Links** folder in the Palette and select the **Operation Link** tool.

2 Place your cursor anywhere in the header area of the first (i.e., the left) operation in the pair.

Note that the cursor displays the following shape to indicate that it is positioned over a valid endpoint for a link: 

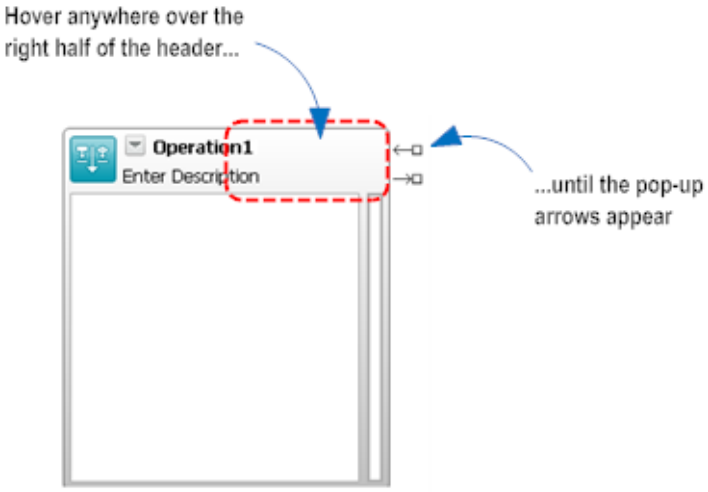
3 Drag the cursor to the header area of the second (i.e., the right) operation in the pair. Watch for the cursor to switch to the valid state and then release your mouse button.

The Continuous Query Modeler connects the two operations with a solid arrow pointing from left to right.

To link operations using the pop-up Links

Before you begin, make sure the two operations that you want to link are visible on the canvas and are arranged in left-to-right order according the sequence in which you want them to execute.

1 Hover over the right side of the header area of the first (i.e., the left) operation and wait for the pop-up arrows to appear.



2 Drag the head of the outbound arrow to the header area of the second operation.

3 When the cursor switches to the following state, release the mouse button:

The Continuous Query Modeler connects the two operations with a solid arrow pointing from left to right.

Changing the Endpoint of an Operation Link or an Expression Link

To route an existing link to a different endpoint, you must delete the existing link and create a new one. You cannot reroute an existing link by dragging its endpoint to a different operation or expression.

Note: When you delete an operation link, the Continuous Query Modeler also deletes all expression links that exist between the two operations.

Rearranging the Layout of the Query Model

After working with a query model for a while, it can become cluttered and difficult to follow. You can use the following procedure to quickly organize the diagram in neat left-to-right order.

To rearrange the layout of the query model

- Click the right mouse button on any empty area in the design canvas and choose **Arrange All** from the context menu.

Resizing an Operation on the Design Canvas

To change the size of an operation on the design canvas, click the operation to display its resize handles (small black squares that appear around the edge of the shape). Then drag a handle to adjust the operation to size you need.

Note that once you enlarge an operation, you cannot reduce its size using the resize handles. If you need to make its shape smaller, use one of the following techniques to return the operation to its default size:

- Collapse the operation using the expand/collapse arrow next to the name of the operation in the header area.

  —OR—

- Right-click the shape and select **Format** > **Auto Size** from the context menu.

Then use the resize handles to adjust the shape to the required size.

Adding Expressions to an Operation

An expression is a literal, function, logical operation, or mathematical computation that resolves to a single value. You use expressions to configure the behavior of certain types of operations.

You create expressions using the following tools on the Palette.

Tool	Description
Aggregate Expressions	An aggregate expression applies an aggregation function to a given attribute in an event stream. You use an aggregate expression with the Aggregate operation. For more information about the Aggregate operation, see ***Returning Aggregate Values from a Query***.
Math Expressions	The math expressions enable you build an expression using mathematic calculations and functions. You can use math expressions with the Filter operation or the Output Stream operation. For information about using the math expressions with the Filter operation, see ***Filtering an Event Stream***. For information about using the math expressions in an Output Stream operation, see ***Mapping Data Using Expression Links***. **Note:** The Math Expressions folder includes the basic arithmetic operators (i.e., addition, subtraction, multiplication, and division) and many mathematic functions. For details about a particular function, see the *EQL Standard Functions Reference*.
Comparison Expressions	A comparison expression compares two values and returns a Boolean value of true or false. You use comparison expressions with the Filter operation. For more information about using the comparison expressions with the Filter operation, see ***Filtering an Event Stream***.
Boolean Expressions	The Boolean expressions enable you to express complex Boolean operations using the logical operators AND, OR, and NOT. You use Boolean expressions with the Filter operation. For more information about using the Boolean expressions with the Filter operation, see ***Filtering an Event Stream***.

To add an expression to an operation

Before you begin, make sure that the operation to which you want to add the expression is visible on the design canvas.

1 In the Palette, select the expression that you want to add to the operation.

2 Move the cursor to the operation to which you want to add the expression and click on any empty area in the left compartment.

Note: If the expression contains input fields that accept literal values, it will display an empty text box for the first input field of that type in the expression. You can optionally type a literal value into that field at this time. Or you can click any empty area in the compartment to close the text box and assign a value to the input field in the next step.

3 To complete the expression, use one of the following techniques to assign values to the input fields in the expression. (Input fields to which you can assign a literal value will display the characters "...").

- If the field accepts a literal value, click the field and type the value. (Note that there is no need to enclose a string literal in quotes of any type. Simply type the string itself directly into the input field.)

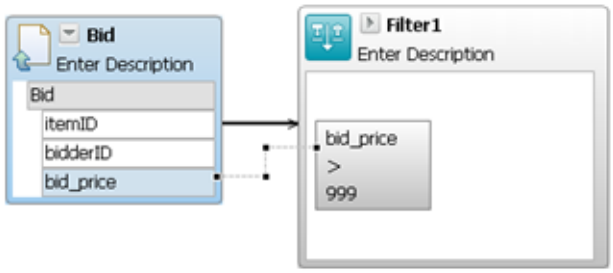
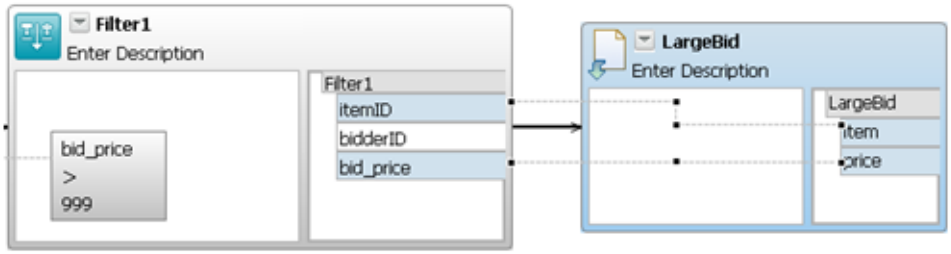
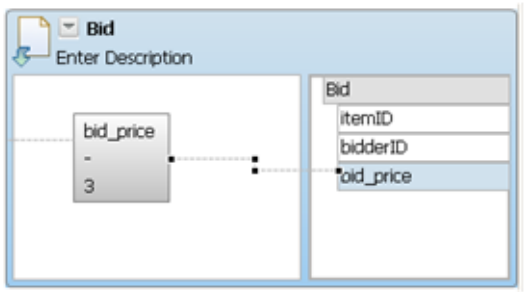
  OR

- Map a value to the input field from the output schema in the previous operation or from another expression in the operation. For information about mapping data values to an expression, see *Mapping Data Using Expression Links*.

Mapping Data Using Expression Links

You use expression links to map data values to and from expressions. As shown below, you can use expression links to map data within a single operation or from one operation to the next.

Note: To map data from one operation to another, the two operations must be connected by an operation link.

You can map...	To...	Example
An output attribute in an operation	A data field in an expression in the following operation	
An output attribute in the operation preceding the Output Stream operation	An output attribute in the Output Stream operation	
The value derived from an expression	A data field in another expression within the same operation	
The value derived from an expression in the Output Stream operation	An output attribute in the Output Stream operation	

When you save a query model, the Continuous Query Modeler validates the model's expression links to ensure that the data types of the linked expressions are compatible. If the model contains invalid expression links, the links are flagged as errors on the design canvas and reported in the **Problems** view. For information about the data types that a particular expression expects as input or the data type that it returns, see the corresponding EQL clause or function in the *EQL Reference Guide* or the *EQL Standard Functions Reference*.

To map data to or from an expression

If you want to map data from one operation to the next, make sure the two operations are connected by an operation link before you perform the following steps.

1 Open the **Links** folder in the Palette and select the **Expression Link** tool.

2 Click the source expression or attribute (i.e., the field that contains the data that you want to map) and then drag the cursor to the target expression or attribute (i.e., the field that is to receive the data).

3 Watch for the cursor to switch to the following state and release your mouse button: ↖

The Continuous Query Modeler connects the two fields with a dotted line.

Modeling Query Operations

Filtering an Event Stream

You use the Filter operation to eliminate events that are of no interest to you. To filter the stream, you use comparison and Boolean expressions to specify the characteristics of the events that you want to select. Only events that satisfy your selection criteria are retained and passed to the next operation in the query model.

A filter operation can consist of a single comparison expression like the one shown below:

Or it can contain a complex Boolean expression that evaluates events for multiple conditions as shown in the following example:

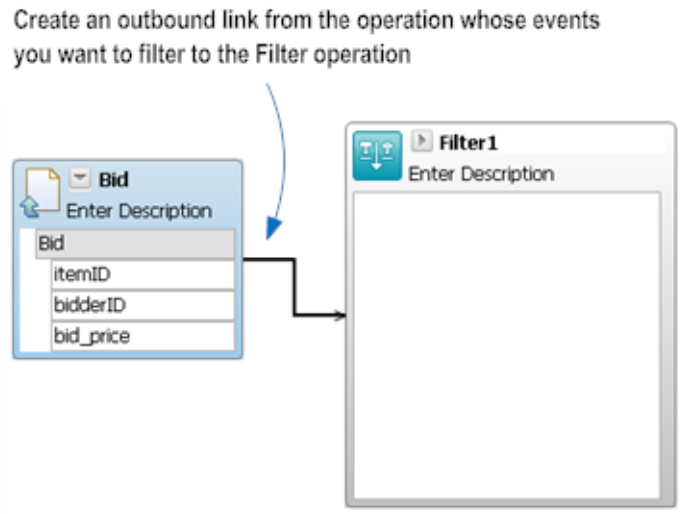
Adding a Filter operation to a Query Model

Use the following procedure to add a Filter operation a query model.

To add a Filter operation to a query model

1 Add the Filter operation to the design canvas. If you need procedures for this step, see *Adding Operations to a Query Model*.

2 Position the filter operation after (i.e., to the right of) the operation that produces the stream of events that you want to filter, and then create an outbound link from that operation to the Filter operation. If you need procedures for this step, see *Linking Operations in a Query Model*.



Note: Although the example above filters events from an Input Stream operation, you can filter the event stream produced by any type of operation except a Output Stream operation. A filter operation does not have to receive its events directly from an Input Stream operation.

3 Within the left compartment of the Filter operation, compose a logical expression that describes the events that you want to select. You can use any of the following expression tools to form the expression, however, the expression you specify must ultimately return a single Boolean value of true or false.

- Comparison Expressions
- Boolean Expressions
- Math Expressions

If you need procedures for this step, see ***Adding Expressions to an Operation***. For examples of expressions that you might use in a Filter operation, see ***Filter Expressions***.

Filter Expressions

- Comparison Expressions
- Boolean Expressions

- Math Expressions

Comparison Expressions

A comparison expression compares two values and returns a value of "true" if the comparison is true, otherwise it returns "false." You use the tools in the Comparison Expressions folder to create a simple comparison expression.

Expression	Description
Equal	Compares two values and returns true if both values are identical.
Greater Than	Compares two values and returns true if the first value is greater than the second.
Greater Than or Equal	Compares two values and returns true if the first value is greater than or equal to the second.
Less Than	Compares two values and returns true if the first value is less than the second.
Less Than or Equal	Compares two values and returns true if the first value is less than or equal to the second.
Like	Compares a String value to a specified pattern string and returns true if the String value matches the given pattern.

Examples

The following example returns true if the bid_price attribute contains a value of 1000 or greater. This expression is equivalent to the EQL clause `WHERE bid_price >= 1000`.

The following example returns true if the value in the name attribute begins with an 'M'. This expression is equivalent to the EQL clause `WHERE name LIKE 'M%'`.

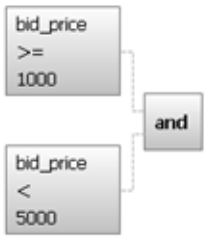
Boolean Expressions

You use the AND, OR, and NOT tools from the Boolean expression folder to compose Boolean expressions.

Expression	Description
AND	Compares the results from two expressions and return true only if both expressions evaluate to true.
OR	Compares the two expressions and return true if either or both expressions evaluate to true.
NOT	The NOT expression reverses the value of a Boolean result. This expression is typically used to retain all events *except* the ones that match a given condition.

Examples

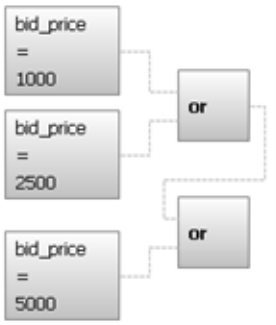
The following example returns true if the value in the bid_price is greater than or equal to 1000 and less than 5000. This expression is equivalent to the EQL clause `WHERE bid_price >= 1000 AND bid_price < 5000`.



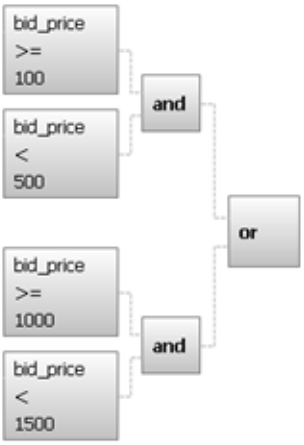
Important: When you link a comparison expression to a Boolean expression, you must drag the link from the comparison expression *to* the Boolean expression, not vice versa.

The following example returns true for those events whose country value is not 'United States'. This expression is equivalent to the EQL clause `WHERE NOT country = 'United States'`. Note that unlike the other Boolean expressions, the NOT expression has only one input value.

The following example evaluates an event to determine whether it satisfies any of three comparison expressions. This expression is equivalent to the EQL clause `WHERE bid_price = 1000 OR bid_price = 2500 OR bid_price = 5000`.



The following example selects an event if its bid price falls within one of two specified ranges. This expression is equivalent to the EQL clause `WHERE (bid_price >= 100 AND bid_price < 500) OR (bid_price >= 1000 AND bid_price < 1500)`.



Math Expressions

Although the expression in a filter operation must ultimately produce a Boolean result, it can include expressions from the Math Expressions folder to compute input values for comparison expressions.

Examples

The following example uses a math expression to compute the total of the bid_price and the shipping_cost attributes. The result is then used as input to a comparison expression that checks whether the total is greater than 99. This expression is equivalent to the EQL clause `WHERE bid_price + shipping_cost > 99`.

Joining Event Streams

A join operation creates a new tuple by combining the tuples from two event streams. You can model the following types of joins:

Type	Description
Inner Join	An inner join combines a tuple from stream A with each tuple in stream B that satisfies a given join condition.
Cross Join	A cross join combines a tuple from stream A with every tuple in stream B (i.e., it produces a Cartesian product).
Self Join	An inner join as described above, but where stream A and stream B are the same stream.

Note: The Join operation evaluates only those events whose time intervals intersect.

For more information about inner joins and cross joins, see the JoinTable construct in the *EQL Reference Guide*.

The following sections provide descriptions and examples of how to model the joins.

- Adding an Inner Join Operation to a Query Model
- Inner Join Examples
- Adding a Cross Join Operation to a Query Model
- Adding a Self Join Operation to a Query Model

Adding an Inner Join Operation to a Query Model

Use the following procedure to include an inner join operation in a query model.

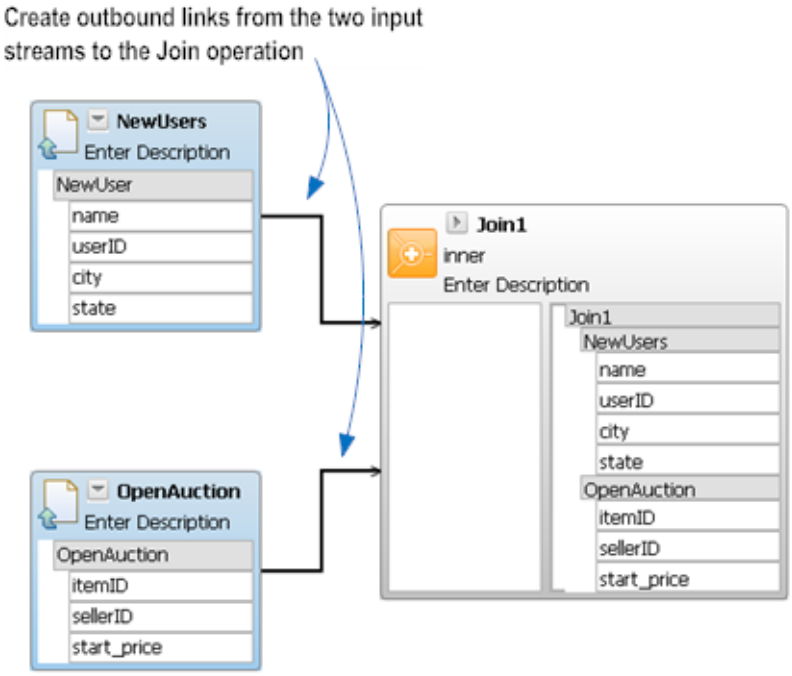
To add an Inner Join operation to a query model

Make sure the model includes the two event streams that you want to join.

1 Add the Join operation to the design canvas. If you need procedures for this step, see ***Adding Operations to a Query Model***.

2 Position the Join operation after (i.e., to the right of) the two operations that produce the streams of events that you want to join. Then create an outbound link from each of those streams to the Join operation. If you need procedures for this step, see ***Linking Operations in a Query Model***.

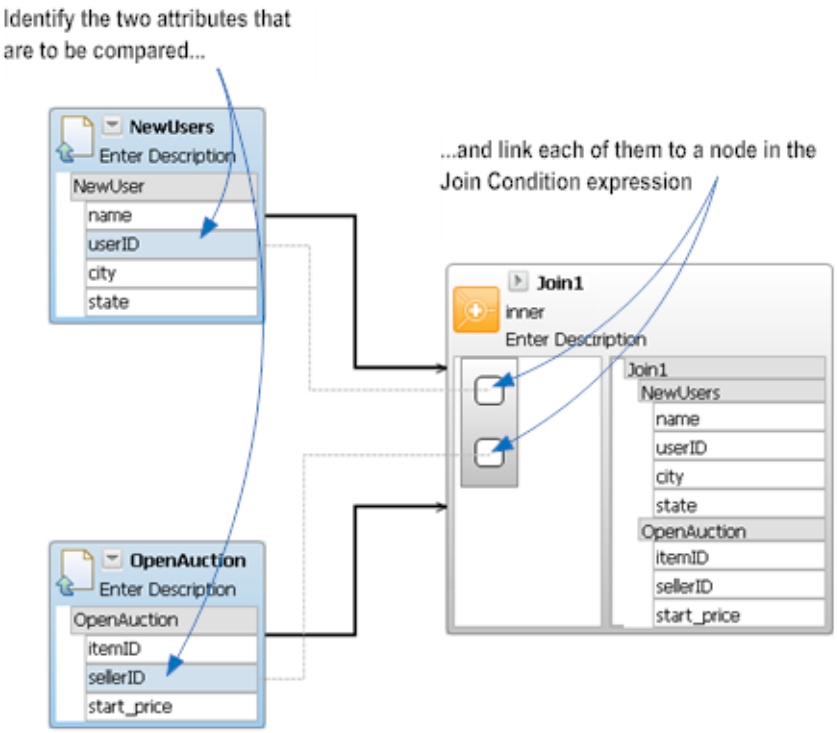
Note: When the Join operation combines two event streams, it joins stream A (the stream attached to the upper anchor point) to stream B (the stream attached to the lower anchor point). If you care about the order in which the streams are combined, be sure to connect the streams to the appropriate anchors. For additional information about how the Continuous Query Modeler uses anchors to differentiate the two input streams, see *Distinguishing Stream A from Stream B*.



Note: Although the example above joins events from two Input Stream operations, you can join streams produced by any operation except an Output Stream operation. A Join operation does not have to receive events directly from an Input Stream operation.

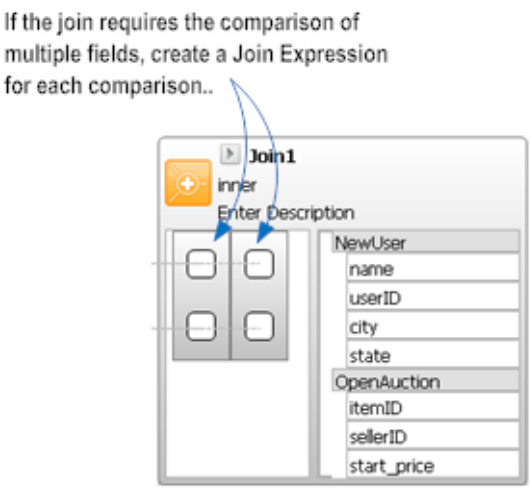
3 Specify the join condition using the following steps.

1. Open the Boolean Expressions folder in the Palette.
2. Select the Join Condition expression and drop it in the left compartment of the Join operation. If you need procedures for this step, see *Adding Expressions to an Operation*.
3. Identify the two attributes whose values are to be compared and use an expression link to link each of them to a node in the Join Condition expression.



If you need procedures for this step, see ***Mapping Data Using Expression Links***.

4. If the join condition requires the comparison of additional attributes in the two streams, add additional Join Condition expressions to the operation. Events from the streams are joined only if they satisfy all of the conditions you specify (i.e., an implicit AND is performed on the specified conditions).



For examples of various Join operations and Join Condition expressions, see ***Inner Join Examples***

Inner Join Examples

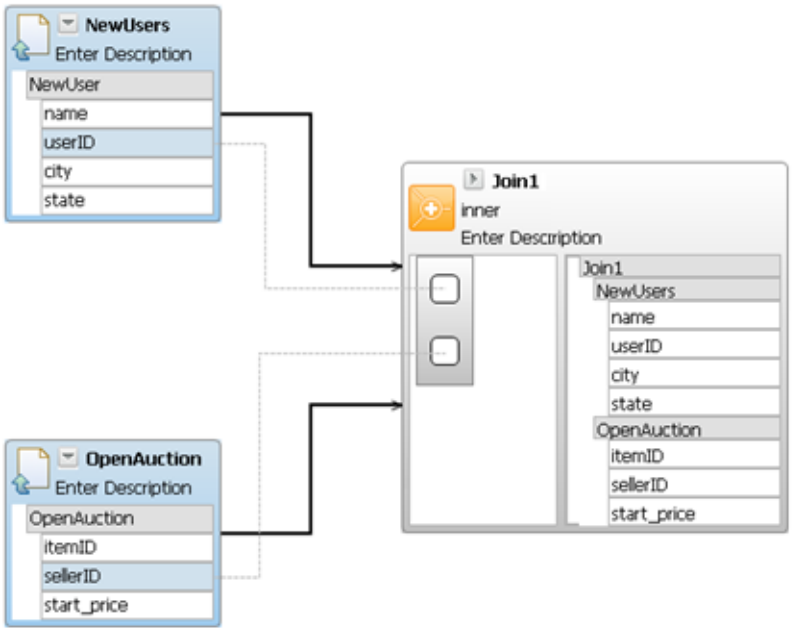
Joining Two Streams

The following example joins the NewUser stream, which contains new-user registrations from an auction site, with the OpenAuction stream, which contains new auctions posted to the site. Each user-registration event has a validity interval of one hour. Each new- auction event has a validity interval of one millisecond.

The Join operation in this example compares the `id` attribute in the NewUser event to the `sellerID` attribute in the OpenAuction event. It produces a joined event only if the two values are equal. Because each event in the NewUser stream has a validity interval of an hour, this join will capture any auctions that a new user opens within one hour of registering on the auction site.

The model fragment shown below is equivalent to the following FROM clause in EQL:

```
FROM NewUsers JOIN OpenAuction
ON NewUsers.userID = OpenAuction.sellerID;
```



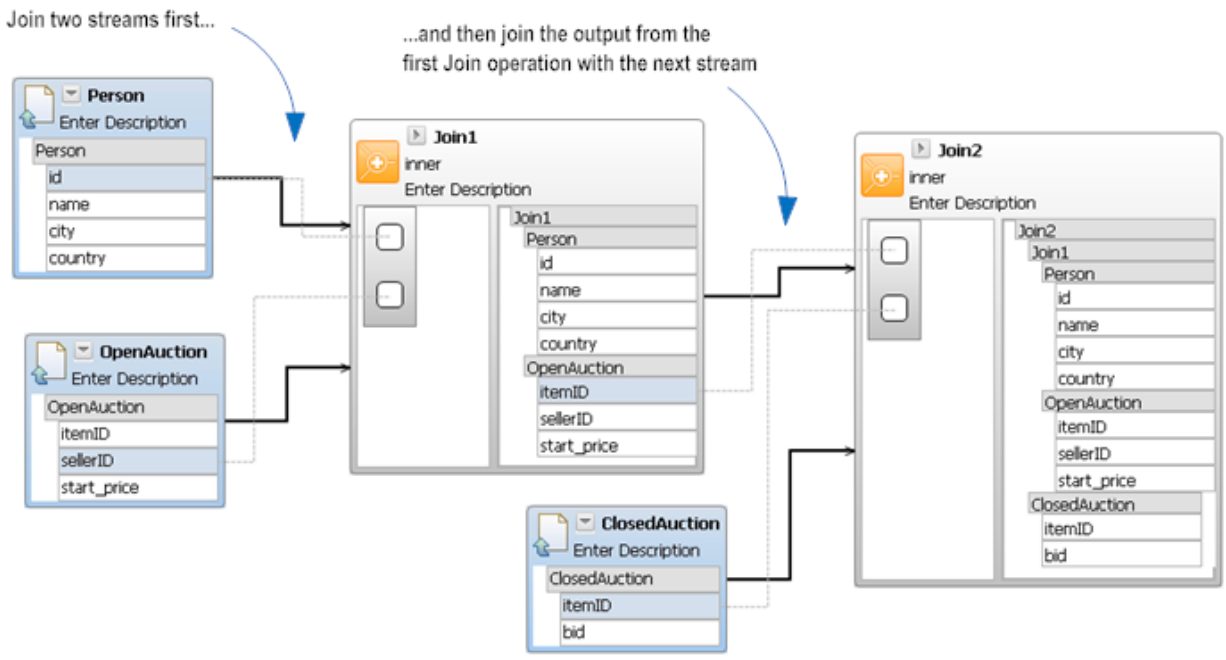
Joining Three Streams

To join more than two streams, you simply insert multiple Join operations into your model. The following model illustrates how to join three event streams. It uses the first Join operation to join two of the three streams. Then, using a second Join operation, it joins the output from the first join with the third event stream. If your join involves more than three streams, simply add additional Join operations in this manner.

Tip: When a model requires many Join operations, consider defining some of the joins as separate queries and referencing those queries as Input Stream operations in your model. Doing this produces a model that is more compact and easier to read.

The model fragment shown below is equivalent to the following FROM clause in EQL:

```
FROM (SELECT *
FROM Person JOIN OpenAuction ON Person.id = OpenAuction.sellerID) AS Join1
    JOIN ClosedAuction ON Join1.itemID = ClosedAuction.itemID;
```

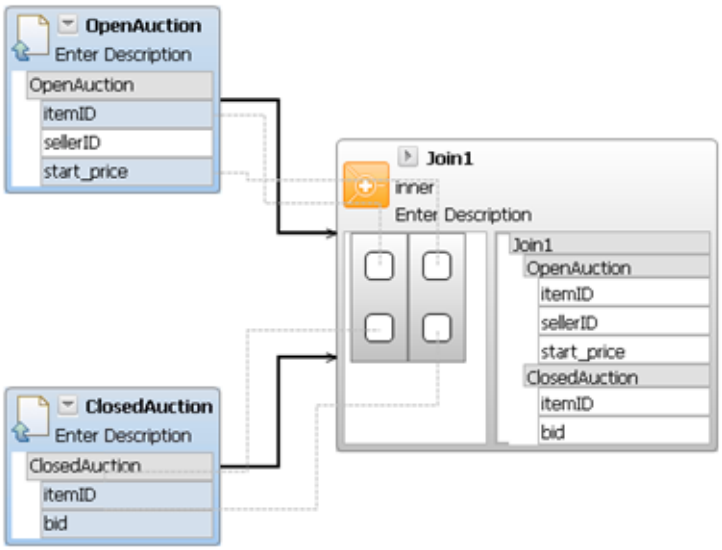


Joining Streams on Multiple Attributes

The following example illustrates how to join streams on multiple condition. In this example, the model joins events from the OpenAuction stream with events from the ClosedAuction stream. A join is performed if events in the two streams are associated with the same item and their opening and closing prices are the same (indicating that the item was sold at its opening price).

The model fragment shown below is equivalent to the following FROM clause in EQL:

```
FROM OpenAuction JOIN ClosedAuction
ON OpenAuction.itemID = ClosedAution.itemID AND OpenAuction.start_price = ↩
ClosedAuction.bid;
```



Adding a Cross Join Operation to a Query Model

Use the following procedure to include a Cross Join operation in a query model.

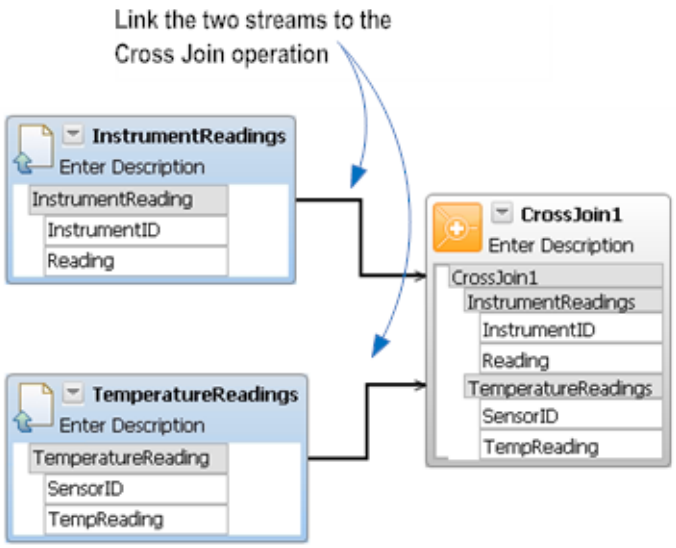
To add an Cross Join operation to a query model

Make sure the model includes the two event streams that you want to join.

1 Add the Cross Join operation to the design canvas. If you need procedures for this step, see *Adding Operations to a Query Model*.

2 Position the Cross Join operation after (i.e., to the right of) the two operations that produce the streams of events that you want to join. Then create an outbound link from each of those streams to the Cross Join operation. If you need procedures for this step, see *Linking Operations in a Query Model*.

> **Note:** When the Cross Join operation combines two event streams, it joins stream A (the stream attached to the upper anchor point on the left side of the Cross Join operation) to stream B (the stream attached to the lower anchor point). If you care about the order in which the streams are combined, be sure to connect the streams to the appropriate anchors. For additional information about how the Continuous Query Modeler uses anchors to differentiate the two input streams, see *Distinguishing Stream A from Stream B*.

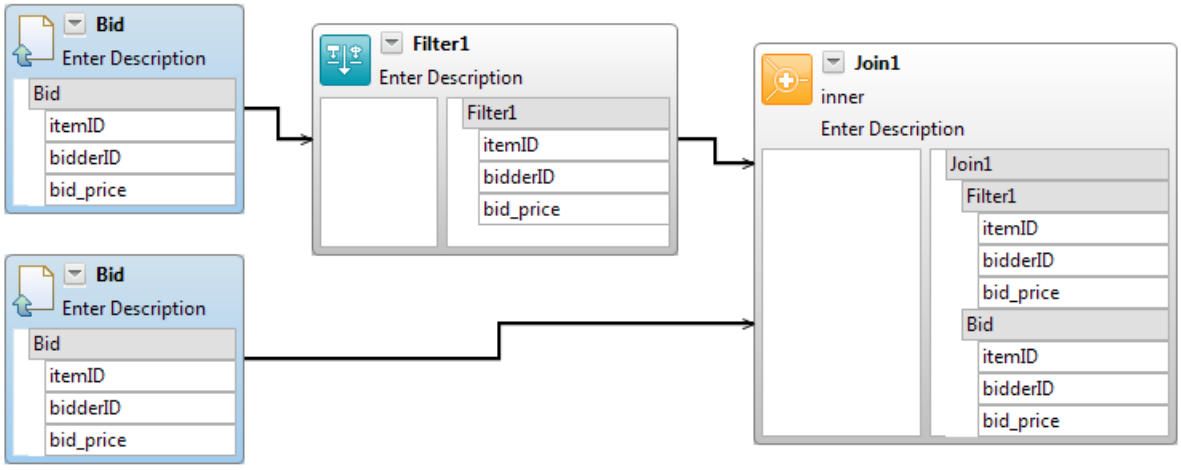


Note: Although the example above joins events from two Input Stream operations, you can join streams produced by any operation except an Output Stream operation. A Cross Join operation does not have to receive events directly from an Input Stream operation.

Adding a Self Join Operation to a Query Model

The Continuous Query Modeler does not allow you to perform a self join (i.e., to join a stream with itself) directly, but you can nevertheless implement a self join in the Continuous Query Modeler by using an empty filter to rename the input stream, then performing a join of the stream with the renamed stream. An empty filter is a filter that specifies no selection criteria, so all incoming events at the filter are passed on directly to the next component in the query.

Here is an example, showing a self join of the input stream Bid. The filter Filter1 is used as an empty filter to rename the first instance of the Bid input stream:



This is similar to using an EQL sequence such as the following to perform a self join:

```
SELECT
    Bid.itemID AS Bid_itemId,
    Filter1.itemID AS Filter1_itemId,
    Bid.bidderID AS Bid_bidderID,
    Filter1.bidderID AS Filter1_bidderID,
    Bid.bid_price AS Bid_bid_price,
    Filter1.bid_price AS Filter1_bid_price
 FROM Bid AS Filter1
      JOIN Bid
 USING (itemId);
```

Applying a Window to an Event Stream

You use window operations to extend the validity interval of a chronon event. Applying a window to an event stream enables you to correlate events based on their proximity in time.

Note: Window operations can only be applied to chronon streams.

The Window Operations

Operation	Description
Time-Based Window	A time-based window adjusts the validity interval of a chronon event by a specified length of time. For general information about time-based windows, see the WindowClause construct in the *EQL Reference Guide*. To create a time-based window in a query model, see the procedure in ***Adding a Time-Based Window Operation to a Query Model***.

Operation	Description
Count-Based Window	A count-based window adjusts the validity interval of a chronon event based on events that surround it in the stream. For general information about count-based windows, see the WindowClause construct in the *EQL Reference Guide*. To create a count-based window in a query model, see the procedure in ***Adding a Count-Based Window Operation to a Query Model***.

Adding a Time-Based Window Operation to a Query Model

Use the following procedure to apply a time-based window to an event stream.

To add a Time-Based Window operation to a query model

1 Add the Time-Based Window operation to the design canvas. If you need procedures for this step, see ***Adding Operations to a Query Model***.

2 Position the Time-Based Window operation after (i.e., to the right of) the operation that produces the stream of events to which you want to apply the window. Then create a link from that operation to the window operation. If you need procedures for this step, see ***Linking Operations in a Query Model***.

3 On the design canvas, select the Time-Based Window operation and specify the following properties in the **Properties** view. If you need additional information about these properties, see the WindowClause construct in the *EQL Reference Guide*.

In this property...	Specify...
Window Range	An integer that specifies the length of the window to apply (as measured in the units of time specified in **Range Granularity**). To specify an unbounded window, set this property to `-1`.
Range Granularity	The units of time represented by the value in **Window Range**.
Window Slide	An integer that specifies the period of time over which the window advances (as measured in the units of time specified in **Slide Granularity**). **Note:** If you want the window to advance each time a new event arrives, set **Window Slide** to 1 and **Slide Granularity** to "milliseconds". If you want to define a tumbling window, set **Window Slide** and **Slide Granularity** to the same values as **Window Range** and **Range Granularity**.
Slide Granularity	The units of time represented by the value in **Window Slide**.
Relative To	The initial point in time from which the interval specified in **Window Slide** and **Slide Granularity** is to be measured.

Note: The Time-Based Window operation includes a **Partition By** field in the header area. In a Count-Based Window operation, this field is used to enable partitioning. In a Time-Based Window, this field is not used.

For examples of various Time-Based Window operations, see *Time-Based Window Examples.*

Time-Based Window Examples

- Time-Based Window that Slides with Each New Event
- Time-Based Window that Slides at a Specified Interval

Time-Based Window that Slides with Each New Event

To apply a window that advances forward each time the Time-Based Window operation receives a new event, specify the size of the window using the **Window Range** and **Range Granularity** properties and leave the **Window Slide** and **Slide Granularity** settings at 1 millisecond. By leaving the slide settings at 1 millisecond, the window operation simply extends each event's validity interval by the amount specified in the range settings. In other words, it uses the range properties to set the event's end timestamp and leaves the event's start timestamp unchanged.

The property settings shown in the table below extend the validity interval of each event by two hours. These property settings are equivalent to the following WINDOW clause in EQL:

```
WINDOW(RANGE 2 HOURS)
```

Set this property...	To...
Window Range	2
Range Granularity	hours
Window Slide	1
Slide Granularity	milliseconds
Relative To	*Leave at the default setting.*

Time-Based Window that Slides at a Specified Interval

You use the slide properties in the Time-Based Window operation to divide the time axis into fixed-length timeframes. The first timeframe in the axis begins at the point you specify in the **Relative To** property. The window operation adjusts the window of each incoming event so that the event is valid during the timeframe in which the event's start timestamp falls.

When you set the **Window Slide** and **Slide Granularity** properties to the same values as the **Window Range** and **Range Granularity** properties, you create a tumbling window, meaning that an event's validity interval will extend over the entire timeframe in which the event occurs.

The property settings shown in the table below apply a six-hour tumbling window to an event stream. These property settings are equivalent to the following WINDOW clause in EQL:

```
WINDOW(RANGE 6 HOURS SLIDE 6 HOURS RELATIVE TO '2010-12-31T00:00:00.000 GMT-00:00')
```

Set this property...	To...
Window Range	6
Range Granularity	hours
Window Slide	6
Slide Granularity	hours
Relative To	2010-12-31T00:00:00.000 GMT-00:00

Adding a Count-Based Window Operation to a Query Model

Use the following procedure to apply a count-based window to an event stream.

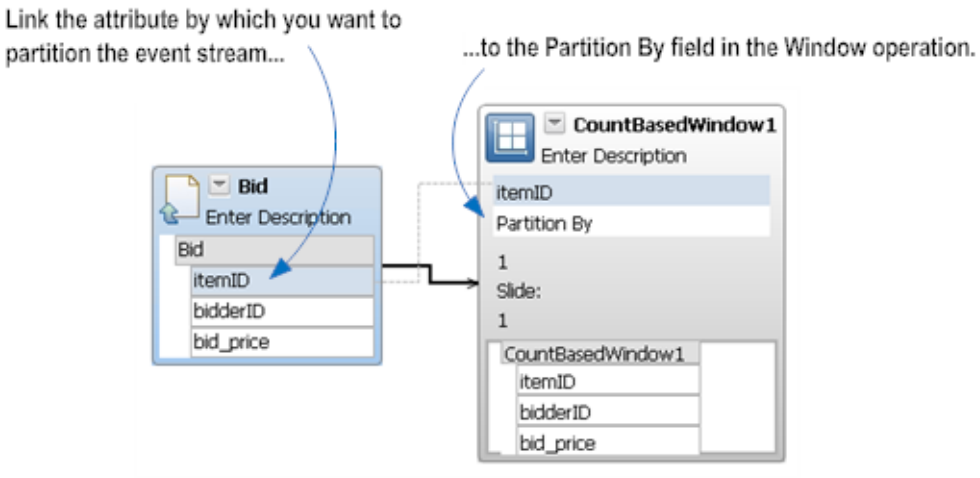
To add a Count-Based Window operation to a query model

1 Add the Count-Based Window operation to the design canvas. If you need procedures for this step, see ***Adding Operations to a Query Model***.

2 Position the Count-Based Window operation after (i.e., to the right of) the operation that produces the stream of events to which you want to apply the window. Then create a link from that operation to the window operation. If you need procedures for this step, see ***Linking Operations in a Query Model***.

3 If you want to partition the event stream, perform the following steps. Partitioning divides events into substreams prior to applying the specified window, then merges the events back into a single stream after the window is applied. For additional information about partitioning an event stream, see the WindowClause construct in the *EQL Reference Guide*.

   1. In the operation that precedes the Count-Based Window operation, locate the attribute by which you want to partition the stream.
   2. Link that attribute to the **Partition By** field in the Count-Based Window operation. If you need procedures for this step, see ***Mapping Data Using Expression Links***.



4 If you want to partition the stream by multiple attributes, repeat step 3 for each of those attributes.

> **Note:** You can add the attributes to the **Partition By** field in any order. The order in which the attributes appear in this field does not affect the way in which events are partitioned.

5 On the design canvas, click the Count-Based Window operation and specify the following properties in the **Properties** view. If you need additional information about these properties, see the WindowClause construct in the *EQL Reference Guide*.

In this property...	Specify...
Window Size	A positive integer that specifies the number of events that defines the size of the window.
Slide Size	A positive integer that specifies the number of events after which the window advances.

For examples of various Count-Based Window operations, see *Count-Based Window Examples*.

Count-Based Window Examples

Count-Based Window with Slide

The count-based window operation specifies a sliding window of the last *N* events. You use the **Window Size** property to specify the number of events to include in the window and the optional **Slide Size** property to specify the number of events after which the window advances.

The property settings shown in the table below uses every third event to determine the end of the window and advances the window after every second event. These property settings are equivalent to the following WINDOW clause in EQL:

```
WINDOW(ROWS 3 SLIDE 2)
```

Set this property...	To...
Window Size	3
Slide Size	2

Returning Aggregate Values

You use the Aggregate operation to produce the following kinds of summary statistics for events in an event stream. An aggregate is evaluated over the set of events whose validity intervals intersect.

Statistic	Description
Average	Computes the average (mean) value for a given numeric attribute in a set of events.
Minimum	Returns the minimum value of a given numeric attribute in a set of events.
Maximum	Returns the maximum value of a given numeric attribute in a set of events.
Count	Returns the number of events that are present in a set of events.
Sum	Computes the sum of a given numeric attribute in a set of events.

For more information about these summary statistics, see the *Standard Aggregate Functions* in the *EQL Standard Functions Reference*.

Note: The Aggregate operation applies the ALL option when it computes a summary statistic. The DISTINCT option is not supported by the Aggregation operation in a query model.

Adding an Aggregation Operation to a Query Model

Use the following procedure to return one or more aggregate values from an event stream.

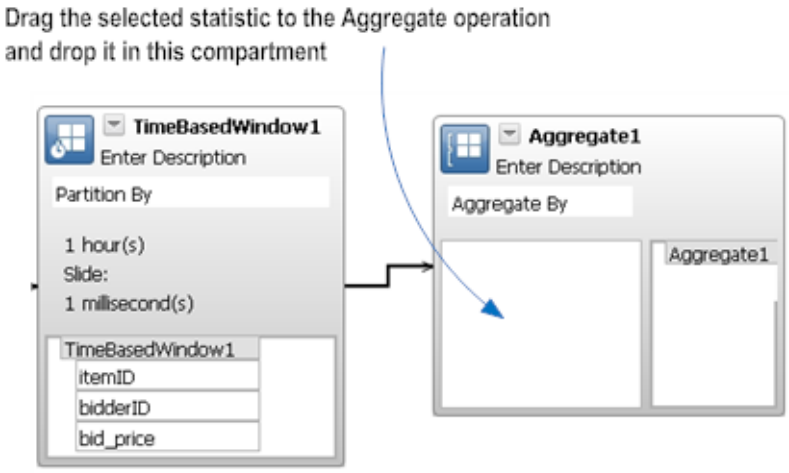
To add an Aggregation operation to a query model

1 Add the Aggregate operation to the design canvas. If you need procedures for this step, see *Adding Operations to a Query Model*.

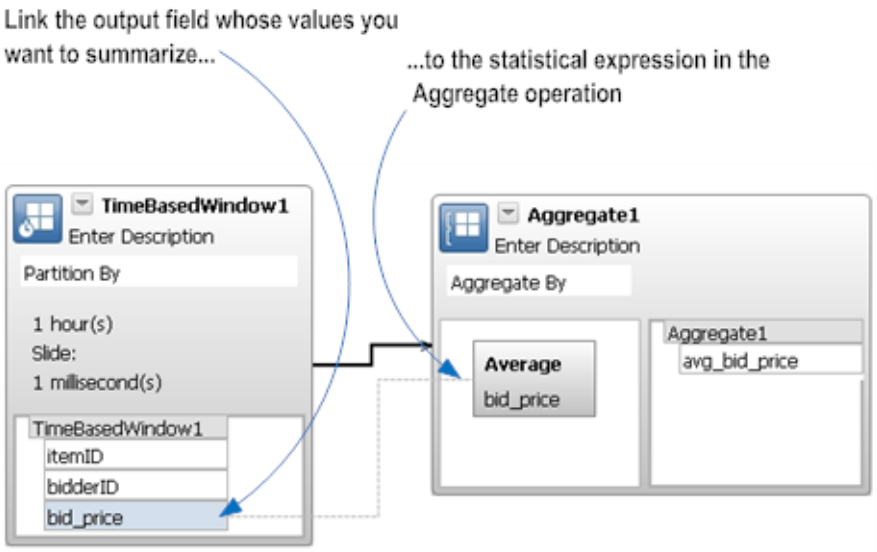
2 Position the Aggregate operation after (i.e., to the right of) the operation that produces the stream of events whose values you want to aggregate. Then create an outbound link from the operation that produces the stream to the Aggregate operation. If you need procedures for this step, see *Linking Operations in a Query Model*.

3 Use the following steps to specify the statistic that you want the operation to produce.

1. Open the Aggregate Expressions folder in the Palette and select the statistic that you want to produce.
2. Drag the selected statistic to the design canvas and drop it in the compartment that appears on the left side of the Aggregate operation.



3. If you added the COUNT statistic to the Aggregate operation in the preceding step and you want to count all events in the set (i.e., perform a COUNT *), skip the remaining sub-steps.
4. In the operation that precedes the Aggregate operation, locate the attribute whose values you want to aggregate.
5. Link that attribute to the statistic expression. If you need procedures for this step, see *Mapping Data Using Expression Links*.



4 Repeat step 3 for each additional statistic that you want the Aggregate operation to produce.

5 If you want to group events based on one or more attributes, perform the following steps to set the **Aggregate By** field. (The **Aggregate By** field is equivalent to the GROUP BY clause in an EQL statement. When you group events, the Aggregate operation divides the event stream into substreams before it performs the aggregation. For example, if you group the Bid stream shown above by itemID, the Aggregate operation will compute the average bid for each item. If you group the stream by itemID and bidderID, the Aggregate operation will compute the average bid per bidder for each item.)

   1. In the operation that precedes the Aggregate operation, locate the attribute by which you want to group the events.
   2. Link that attribute to the **Aggregate By** field in the Aggregate operation. If you need procedures for this step, see *Mapping Data Using Expression Links*.

6 If you want to group events by multiple attributes, repeat step 5 for each of those attributes.

Note: You can add the attributes to the **Aggregate By** field in any order. The order in which the attributes appear in this field does not affect the way in which events are grouped.

For examples of Aggregate operations, see *Aggregate Examples*.

Mapping Aggregate Attributes to an Output Stream Operation

To map the results of an Aggregate operation to the attributes in an Output Stream operation, the data type and cardinality of the attributes in the Output Stream operation must match the type and cardinality of the results produced by the Aggregate operation. The following table describes the requirements for each Aggregate expression.

An attribute that receives a result from this aggregate expression...	Must have the following data type...	And the following cardinality...
Average	Double	0...1
Minimum	Double	0...1
Maximum	Double	0...1
Sum	Double	0...1
Count	Long	1...1 or 0...1

Tip: To create an event type whose attributes match the type and cardinality requirements of the Aggregate operation in your query, do the following: 1) link the Aggregation operation to an untyped Output Stream operation, 2) right-click the Output Stream operation, and 3) select the **Generate Output Event Type** from the context menu. The Continuous Query Modeler produces an event type with the appropriate schema and assigns it to the Output Stream operation. You can edit the event type to change the names and/or order of the attributes as needed.

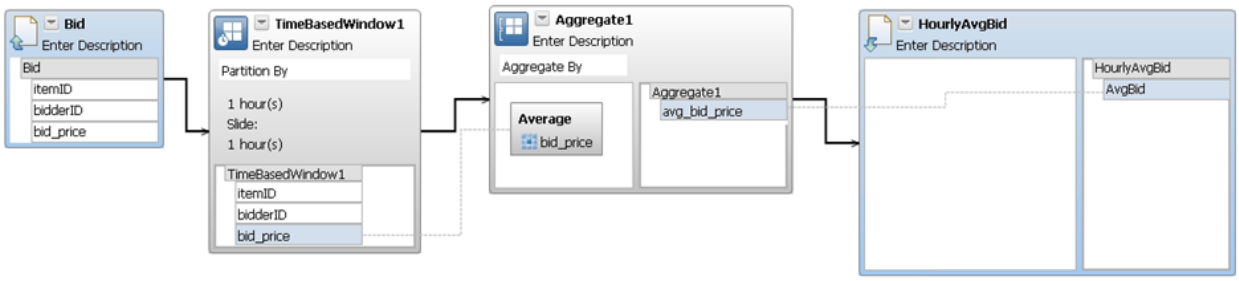
Aggregate Examples

- Single Aggregate Expression
- Multiple Aggregate Expressions
- Counting Occurrences of a Specified Attribute
- Counting All Events
- Grouping Events by a Single Attribute
- Grouping Events by Multiple Attributes

Single Aggregate Expression

The query model shown below returns the hourly average bid price. This model is equivalent to the following EQL statement:

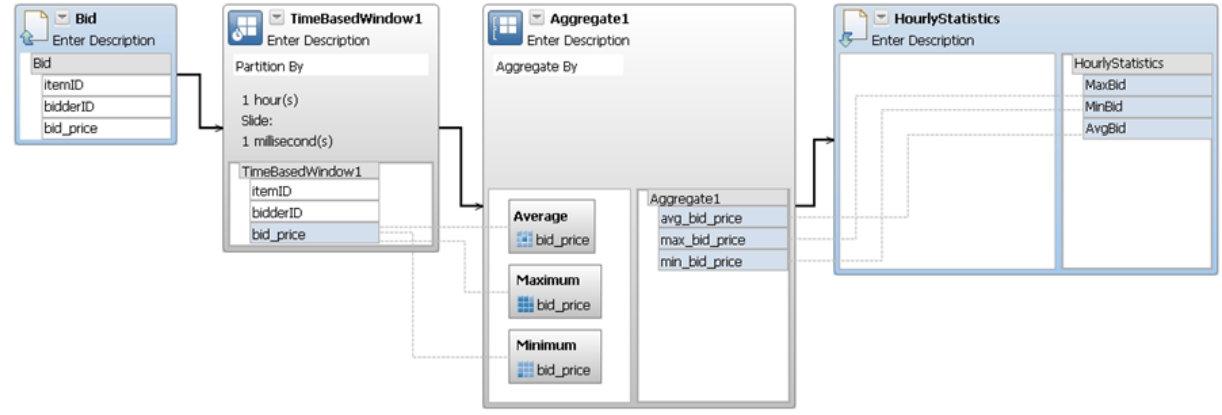
```
SELECT AVG(bid_price)
FROM Bid WINDOW(RANGE 1 HOUR SLIDE 1 HOUR); ↩
```



Multiple Aggregate Expressions

The query model shown below returns the hourly maximum, minimum, and average bid prices. This model is equivalent to the following EQL statement:

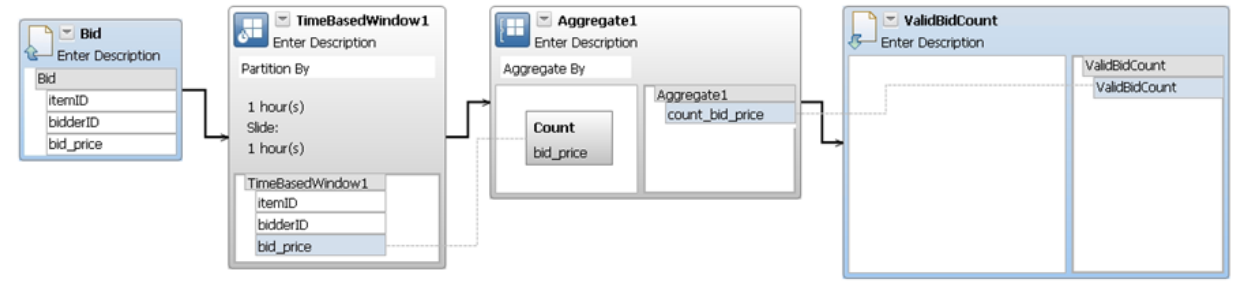
```
SELECT MAX(bid_price), MIN(bid_price), AVG(bid_price)
FROM Bid WINDOW(RANGE 1 HOUR SLIDE 1 HOUR); ↩
```



Counting Occurrences of a Specified Attribute

The query model shown below returns an hourly count of bids with a non-null bid price (In this particular example, we assume that the Bid stream might include bids with a null bid price. These bids would not be counted by the Aggregate operation.) This model is equivalent to the following EQL statement:

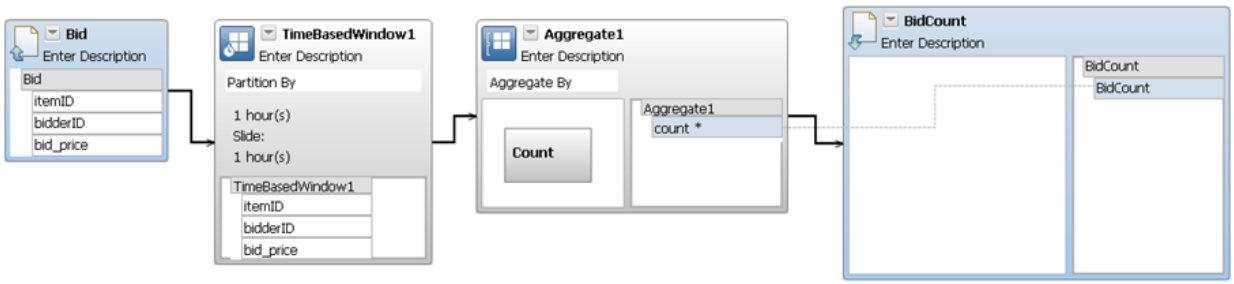
```
SELECT COUNT(bid_price)
FROM Bid WINDOW(RANGE 1 HOUR SLIDE 1 HOUR); ↩
```



Counting All Events

The query model shown below returns an hourly count of all bids in the Bid stream. This model is equivalent to the following EQL statement:

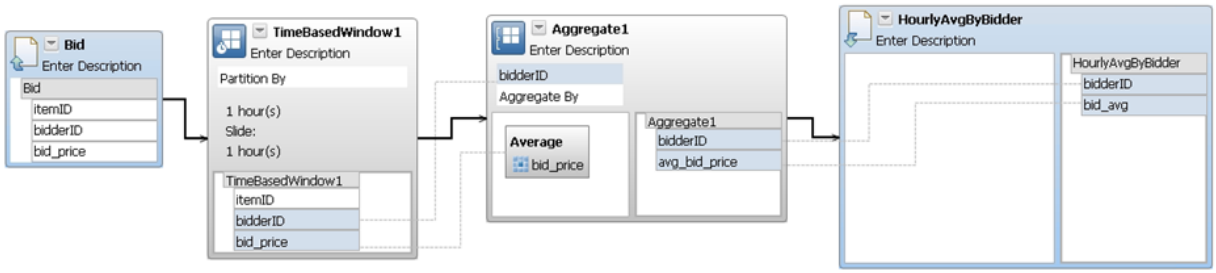
```
SELECT COUNT(*)
FROM Bid WINDOW(RANGE 1 HOUR SLIDE 1 HOUR); ↩
```



Grouping Events by a Single Attribute

The query model shown below returns the average bid for each user who placed a bid during the hour. This model is equivalent to the following EQL statement:

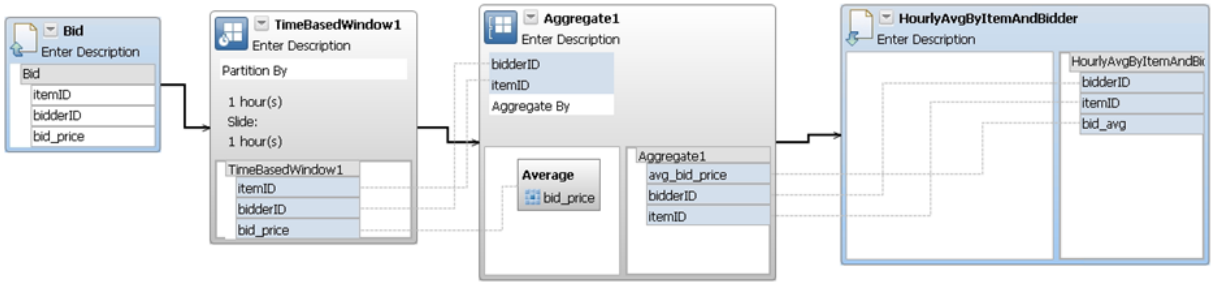
```
SELECT bidderID, AVG(bid_price)
FROM Bid WINDOW(RANGE 1 HOUR SLIDE 1 HOUR)
GROUP BY bidderID; ↩
```



Grouping Events by Multiple Attributes

The query model shown below returns the average bid that each user placed on a particular item during the hour. This model is equivalent to the following EQL statement:

```
SELECT bidderID, itemID, AVG(bid_price)
FROM Bid WINDOW(RANGE 1 HOUR SLIDE 1 HOUR)
GROUP BY bidderID, itemID; ↩
```



Performing Set Operations on Event Streams

You use the Set Operator to combine two event streams using any of the operators in the table below:

Note: Depending on which operator you use, the DISTINCT or ALL option is used to combine event streams. The option that an operator uses is noted in the table below. For more information about the ALL and DISTINCT options, see the SetClause construct in the *EQL Reference Guide*.

Operator	Description
UNION	Produces a single stream that represents the union of two input streams. The UNION operator combines streams using the DISTINCT option.
INTERSECT	Produces a single stream that contains only the events that appear in both streams. The INTERSECT operator combines streams using the ALL option.
EXCEPT	Produces a single stream that contains only events in stream A that are not in stream B. The EXCEPT operator combines streams using the DISTINCT option.
MINUS	Same as EXCEPT.

To combine streams using a set operation, the schemas of the two streams must contain the same number of attributes and the corresponding attributes in the two streams must have the same names and data types. Note, however, that the attributes in the two schemas do not need to be in the same order.

When the schemas of the two streams are compatible, but not identical, the set operation uses the schema of stream A (the stream attached to the upper anchor on the left side of the Set operation) as its output schema. For additional information about how the Continuous Query Modeler uses anchors to differentiate two input streams, see *Distinguishing Stream A from Stream B*.

Adding a Set Operator to a Query Model

Use the following procedure to combine event streams using a set operator.

To add a set operator to a query model

1 Add the Set Operator operation to the design canvas. If you need procedures for this step, see *Adding Operations to a Query Model*.

2 Position the set operation after (i.e., to the right of) the two operations that produce the streams of events that you want to combine.

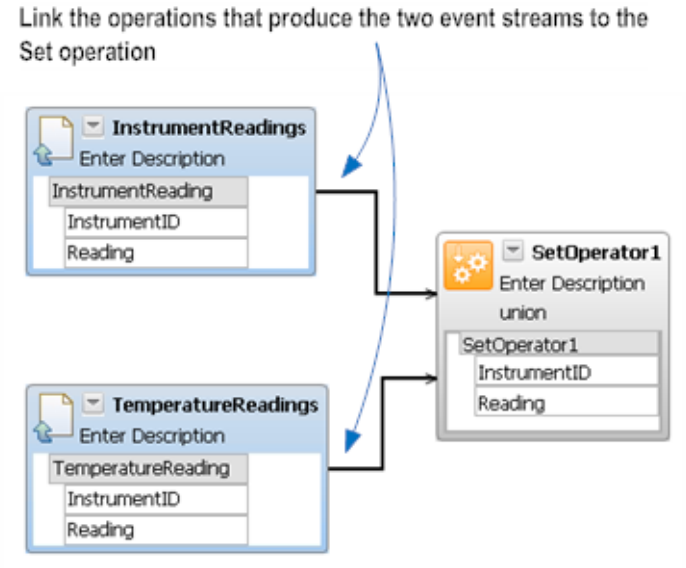
3 Perform the following steps to specify the set operator to apply to the streams:

1. On the design canvas, select the set operation.
2. In the **Operator** property in the **Properties** view, select the set operator that you want to use.

4 Create an outbound link from each of the two input streams to the set operation. If you need procedures for this step, see *Linking Operations in a Query Model*.

Note: When the set operation performs a MINUS or EXCEPT operation, it subtracts the events in stream B (the stream attached to the lower anchor point on the left side of the set operation) from stream A (the stream attached to the upper anchor point). For

correct results from these operations, be sure to connect the input streams to the appropriate anchors. For additional information about how the Continuous Query Modeler uses anchors to differentiate the two input streams, see *Distinguishing Stream A from Stream B*.



Note: Although the example above combines events from two Input Stream operations, you can combine streams produced by any operation except an Output Stream operation. A set operation does not have to receive events directly from an Input Stream operation.

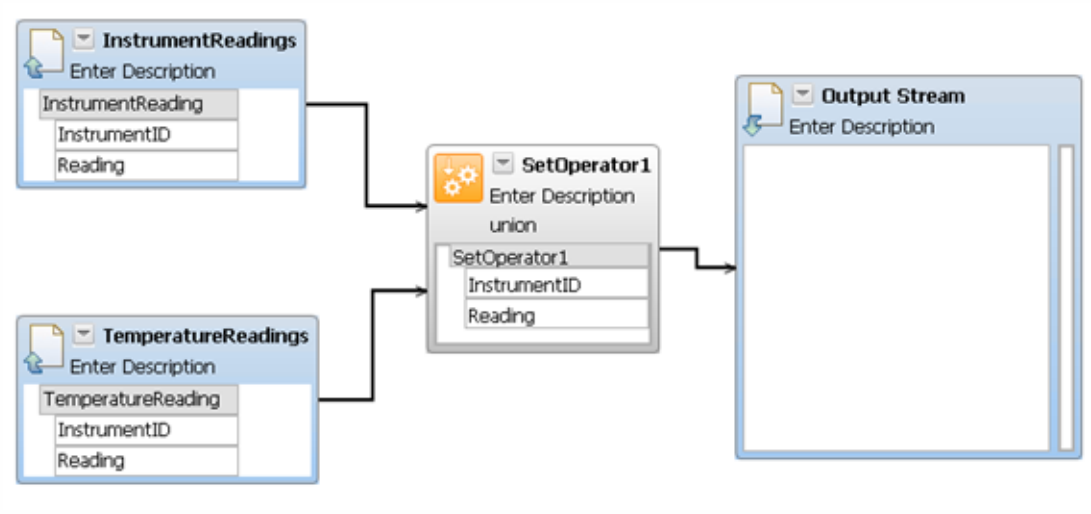
Set Examples

- The UNION operator
- The MINUS Operator

The UNION operator

The following example produces a union of the events in the InstrumentReadings and TemperatureReadings streams. This example is equivalent to the following EQL statement:

```
SELECT * FROM InstrumentReadings
   UNION SELECT * FROM TemperatureReadings;
```

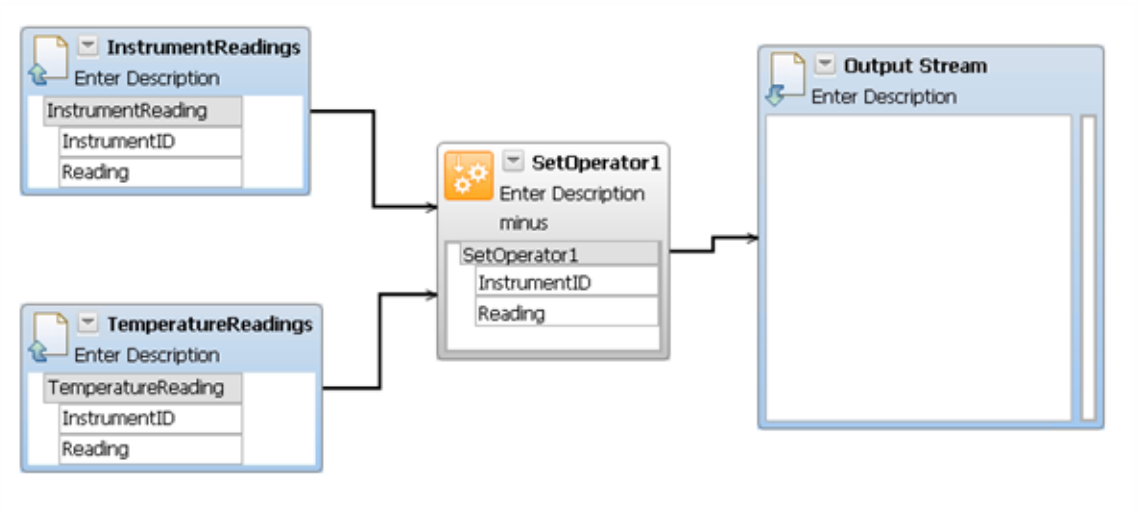


The MINUS Operator

The following example subtracts the events in the TemperatureReadings stream from the InstrumentReadings stream. Only events that occur in the InstrumentReadings stream but not the TemperatureReadings stream are included in the stream produced by this set operation.

This example is equivalent to the following EQL statement:

```
SELECT * FROM InstrumentReadings
   MINUS SELECT * FROM TemperatureReadings;
```



Note: The MINUS operator always subtracts the events in stream B (the stream attached to the lower anchor point on the left side of the set operation) from stream A (the stream attached to the upper anchor point). For additional information about how the Continuous Query Modeler uses anchors to differentiate the two input streams, see *Distinguishing Stream A from Stream B*.

Refreshing an Input Stream Operation

If you modify the schema of an input stream or query that is associated with an Input Stream operation in a query model, Software AG Designer reports an error against the model. To correct the error, open the model and refresh the Input Stream operation as describe in the following procedure.

Note: When you refresh an Input Stream operation, the Continuous Query Modeler propagates the revised schema to the downstream operations and deletes expression links that are no longer valid for the new schema. As part of your editing process, you will need to redefine the links as appropriate for the new schema.

To refresh an Input Stream operation

1 Right-click the errant Input Stream operation and select **Refresh** from the context menu.

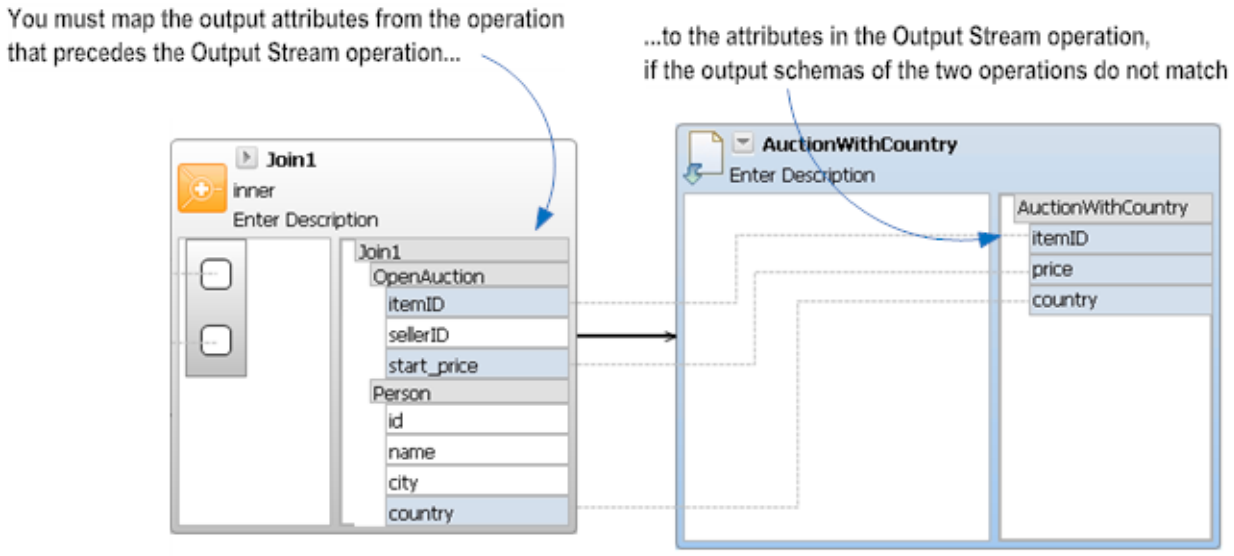
2 Review the model and reinstate any expression links that were dropped due to the schema change. If you need procedures for this step, see *Mapping Data Using Expression Links*.

Mapping Data to the Output Stream Operation

If you use a typed Output Stream operation in your query model and the output schema of the operation that precedes the Output Stream operation does not match the schema of the Output Stream operation, you must use expression links to map the attributes from the preceding operation to the corresponding attributes in the Output Stream operation.

Schemas are considered to "match" if they have the same number of attributes and the attributes appear in the same order and have the same data type and cardinality. (Note that the names of the attributes in the two schemas do not need to be the same for the two schemas to match.)

If you need procedures for mapping data between operations, see *Mapping Data Using Expression Links*.



When mapping data to the attributes in an output stream, keep the following points in mind:

- You must map all of the attributes that the operation produces. If you leave an attribute in the Output Stream operation unmapped, the Continuous Query Modeler will raise an error when you save the model.
- You can only map attributes of the same underlying Java data type. For example, you can map an int to any of the integer types (e.g., positiveInteger, unsignedInt, nonNegativeInteger), but you cannot map it to a non-integer type (e.g., Short, Double, String, Date). You can see the underlying Java type associated with an attribute in the Output Stream operation by viewing the model's **Output Schema** tab. If you map attributes that are not of the same type, the Continuous Query Modeler will raise a "Declared output event type does not match query output record format" error when you save the model.

Viewing or Editing a Query Model

Use the following procedure to view or edit an existing query model.

When editing a query model, keep the following points in mind:

- When you delete an operation link, the Continuous Query Modeler also deletes the expression links between the two operations.
- If you edit the model in a way that alters the schema of an operation, the Continuous Query Modeler propagates the revised schema to the downstream operations and deletes expression links that are no longer valid for the new schema. As part of your editing process, you will need to redefine the links as appropriate for the new schema.

To view or edit a query model

1 In the Project Explorer, double-click the .ceq file for the query model that you want to view or edit.

2 Modify the query model as needed.

3 When you finish editing the query model, choose **File > Save** to save your changes.

Copying a Query Model

See *Copying a Query*.

Renaming a Query Model

See *Renaming a Query*.

Deleting a Query Model

See *Deleting a Query*.

Testing and Deploying a Query Model

Testing and deploying a query model is no different than testing and deploying a query that is expressed in EQL. For information about testing queries, see *Testing a Project in Software AG Designer*. For information about deploying queries, see *Deploying a Continuous Query Application to an Event Server*.

Migrating a Model to the Current Version

When you open a project that contains models that were created in an older version of the Continuous Query Modeler, you may receive error messages in the **Problems** view indicating that the models are not the expected version. Software AG Designer will not allow you to work with the models until you migrate them to the version of the Continuous Query Modeler that you are using.

When you migrate a query model, be aware of the following:

- After you migrate a model, you can no longer edit the model using earlier versions of the Continuous Query Modeler.
- During migration, the operations in a model revert to their default sizes. After migration, you can adjust the size of the operations as needed using the procedure in *Resizing an Operation on the Design Canvas*.

To migrate a model, you can use either of the following procedures.

Note: As a precaution, you might want to create a backup copy of the project before you migrate its queries to the current version.

Migrating a model from the Problems view

1. In the **Problems** view, right click the "Unexpected model version" error message and select **Quick Fix** from the context menu.
2. Select **Migrate Query Model** from the list of fixes and click **Finish.**

   The Continuous Query Modeler migrates the query to the current version.

Migrating a model from the Project Explorer

1. In the Project Explorer, double-click the model that you want to migrate.
2. Click the **Migrate** button that appears in the edit window.

   The Continuous Query Modeler migrates the query to the current version.
3. In Project Explorer, double-click the model again to open it.

8 Developing User-Defined Functions

Introduction

This section explains how to add user-defined functions to your project. You can create user-defined functions to perform calculations that you cannot perform using EQL. For example, you might create a user-defined function to calculate the distance between two objects based on their geographical coordinates.

Creating a User-Defined Function

To create a new user-defined function (UDF), proceed as follows:

To create a new User-Defined Function

1 In the Continuous Query Development, choose **File > New > User Defined Extension**.

2 In the wizard, choose **User-Defined Function** and complete the following fields:

In this field...	Specify...
Project	The project to which you want to add the user-defined function.
Package	The name of the Java package to which the UDF class will belong. The name you enter must be a valid Java package name, as it will be added automatically to the code template that the wizard generates.
Class	The name you want to assign to the UDF Java class. The name you enter must be a valid Java class name, as it will be added automatically to the code template that the wizard generates.

3 Click **Finish**.

A file containing the UDF attributes that you have just specified will be created and stored as a new file in the current project.

At this point, the UDF file exists, but you still need to supply further data to complete the definition. To complete the definition, perform the editing steps described below.

To edit a UDF, proceed as follows:

To edit a UDF

1 Open the editor by doing one of the following:

- Create a new UDF using the wizard as described above. When you close the wizard, the editor opens automatically.

  Or:

- Select the existing UDF definition file from the project tree and open it using any of the standard Eclipse methods.

2 Provide the Java code for the UDF.

  A UDF can have multiple input values and must have one output value. The valid types are listed in the section ***User-Defined Function Parameter and Result Types***

3 Save the Java code.

  If the Java code compiles correctly, a Java nature is added to the project, and the new nodes "src" and "class" are created in the project tree. These contain the Java source and the compiled Java class.

After you have stored the UDF, it is ready for use.

If you wish to define more than one UDF, you have the following possibilities for adding additional UDFs:

- Add an additional UDF method definition to the existing class. To do this, open the existing Java code in a Java editor and add the method definition.
- Use the Java editor to extend the existing code by creating a new public class that extends `UserDefinedFunctions`, then add your UDF to that class.
- Create a new Java source file by using **File > New > User Defined Function** as described above.

If you have more than one UDF, each UDF definition must include the annotation `@UserDefinedFunctionProperties` followed optionally by "(name=...)".

User-Defined Function Parameter and Result Types

A UDF can have multiple input values and must have one output value. The following input and output value data types are currently supported: Integer, Double, Float, Long, Boolean, Byte, Short, BigDecimal, String.

User-Defined Function Exception Handling

An exception thrown during execution of the UDF method is caught and NULL is returned.

User-Defined Function Implementation Notes

When developing a user-defined function, keep the following points in mind:

- A UDF should be a function in the mathematical sense, i.e. it should return a deterministic result that is only dependent on the parameters that are passed to it. In particular, do not use random numbers. A UDF should be stateless. As only one instance of the UDF class is generated, the corresponding queries share this instance; in the case of stateful UDFs, this would produce side effects.
- Wherever possible, avoid accessing external systems from a UDF.
- When you develop a user defined function, you can extend the classpath of the continuous query project by custom libraries or class folders. Both need to be located inside the continuous query project subtree, so that it can be included in an export archive later on. This means that the Java Buildpath options "**Add external JARS...**" and "**Add External Class Folder...**" must not be used for the project.
- The continuous query interface provides a fixed set of operations (e.g. addition), functions (e.g. absolute value), and aggregates (e.g. average). You use UDFs to create scalar functions. If you need to perform an aggregation function (i.e., compute a result over multiple events), you must create a user-defined aggregate (UDA). For information about implementing a UDA, see ***Developing User-Defined Aggregates***.
- As a UDF can return NULL either by intention or due to an exception, the return type of a UDF is marked as nullable. This becomes relevant if the UDF is used in the uppermost SELECT clause of a query and the associated event type is to be selected from a given set of event types.

Calling an IS Service in a User-Defined Function

You can call an IS service in a UDF. To do so, proceed as follows.

To call an IS service in a UDF

1 Create a UDF, as **described above**.

2 In the project tree, add a *lib* folder, if it is not already present.

Copy the required jar files into the *lib* folder. The required jar files are as follows:

- *wm-isclient.jar* (located in the folder *common/lib* under the product installation folder)
- *mail.jar* (located in the folder *common/lib/ext*)
- *enttoolkit.jar* (located in the folder *common/lib/ext*)

3 Add these jar files to the Java classpath of the project.

4 Implement code as shown in the second example below.

Examples

The following example shows how a user-defined function can be used to compute the Euclidean distance between two points.

```
package test;

import com.softwareag.wep.resource.udf.UserDefinedFunction;
import com.softwareag.wep.resource.udf.UserDefinedFunctions;
import java.lang.Math;

public class Test extends UserDefinedFunctions
{
  @UserDefinedFunctionProperties(name = "distance")
    public static Double distance(Double x0, Double y0, Double x1, Double y1) {
        return Math.sqrt(Math.pow((x0-x1),2) + Math.pow((y0-y1),2));
    }
}
```

The following example shows how to add log messages to the IS Server log file.

```
package test;

import com.softwareag.wep.resource.udf.UserDefinedFunction;
import com.softwareag.wep.resource.udf.UserDefinedFunctions;
import com.wm.app.b2b.client.Context;
import com.wm.app.b2b.client.ServiceException;
import com.wm.data.IData;
import com.wm.data.IDataCursor;
import com.wm.data.IDataFactory;
import com.wm.data.IDataUtil;

public class Test extends UserDefinedFunctions
{
  @UserDefinedFunctionProperties(name = "sampleFunction")
  public static String sampleFunction ()
  {
      String  server = "localhost:5555";
      Context context = new Context();
```

```
      // To use SSL:
      //
      // context.setSecure(true);

      // Optionally send authentication certificates
      //
      // String  cert    = "c:\\myCerts\\cert.der";
      // String  privKey = "c:\\myCerts\\privkey.der";
      // String  cacert  = "c:\\myCerts\\cacert.der";
      // context.setSSLCertificates(cert, privKey, cacert);

      // Set username and password for protected services
      String username = "Administrator";
      String password = "manage";

      try {

          IData input = IDataFactory.create();
          IDataCursor cursor = input.getCursor();
          IDataUtil.put(cursor,"message","TEST INVOKE FROM UDF");
          IDataUtil.put(cursor,"function","UDF");
          IDataUtil.put(cursor,"level","INFO");
          context.connect(server, username, password);
          context.invoke("pub.flow", "debugLog",input);
      } catch (ServiceException e) {
          System.out.println("\n\tCannot connect to server \""+server+"\"");
      }

    return null;
  }
}
```

9 Developing User-Defined Aggregates

What is a User-Defined Aggregate?

A user-defined aggregate (UDA) allows you to use a custom aggregate function in queries. You create a UDA when you need to summarize events in a manner that is not supported by the built-in aggregate functions provided by EQL. You provide the underlying logic for the UDA as a Java program.

An aggregate in EQL is like an aggregate function in SQL. It returns a single output value from the combined values of a specified attribute in a stream of events. Typically, an aggregate returns a statistic that summarizes a particular aspect of an event stream over a period of time. In the following query, for example, a user-defined aggregate called UDA_Median computes the median temperature for the weather events in a 24-hour window.

```
SELECT UDA_Median(Temperature)
FROM ActualWeather WINDOW(RANGE 24 HOURS);
```

Note: When referenced in a query, the name of a user-defined aggregate always begins with the characters "UDA_".

Within an EQL statement, a UDA can be used in the SELECT clause, as shown in the example above, and/or in the HAVING clause. The following is an example of a UDA in the HAVING clause. In this example, UDA_Median is used to return only those locations whose median temperature is greater than 20 degrees.

```
SELECT locID, UDA_Median(Temperature)
FROM ActualWeather WINDOW(RANGE 24 HOURS)
GROUP BY locID HAVING UDA_Median(Temperature) > 20;
```

How is a User-Defined Aggregate Different from a User-Defined Function?

A user-defined aggregate summarizes a specified attribute in a stream. An aggregate takes multiple instances of the attribute as input and returns a single value as output. A user-defined function, in contrast, is a scalar function. It operates against a single event and returns a single value as output.

The following table summarizes the differences between UDAs and UDFs.

A User-Defined Aggregate...	A User-Defined Function...
Takes a specified attribute from multiple events as input.	Takes any combination of constants, attribute values, and/or results from other functions as input.
Returns a single value.	Returns a single value.
Appears in the SELECT clause or the HAVING clause.	Appears in the SELECT clause, WHERE clause or HAVING clause.
Is stateful. A UDA retains information in memory about earlier events that it has processed and it uses this information to produce an aggregated value.	Is stateless. A UDF *must not* retain any information in memory about earlier events that it has processed. Event Server instantiates just one instance of the function class and that instance is used by all queries that invoke the function.

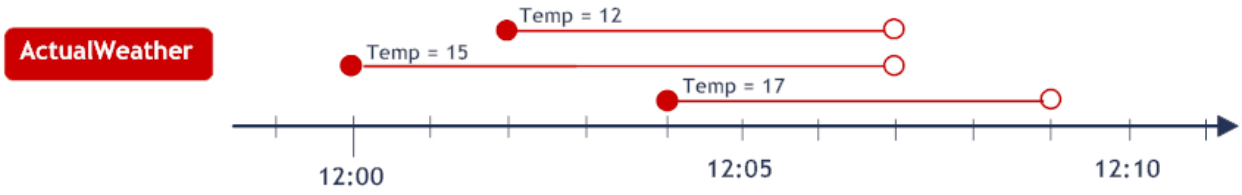
Which Events are Evaluated during an Aggregation?

An aggregate is evaluated over the instances of events whose validity intervals intersect.

For example, if you were to apply the following query to a chronon stream of temperature readings, it would compute the median value for the temperature readings within a six-hour window of time. Because this query specifies a sliding window, the median will be computed each time a new event arrives.

```
SELECT UDA_Median(Temperature)
FROM ActualWeather WINDOW(RANGE 6 HOURS);
```

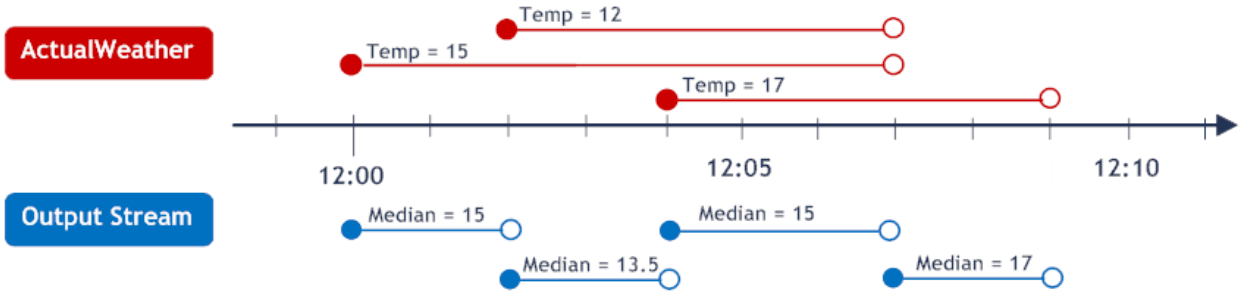
If a query does not include the WINDOW clause, the aggregation operates over the set of events whose validity intervals intersect. For example, let's say your event stream consists of temperature events with the following validity intervals:



If you applied the following query to this stream, the UDA_Median aggregate would compute the median temperature reading for all events that were valid during the same interval in time.

```
SELECT UDA_Median(Temperature)
FROM ActualWeather;
```

As shown below, this query publishes an output event containing the median temperature for every time interval during which at least one temperature event occurred. The timestamp for each output event indicates the validity interval of the reported value. For example, 13.5 was the median value of the events that were valid from 12:02 to 12:04.



Note that when you implement a user-defined aggregate, you only need to provide the logic for computing the aggregation. You do not need to manage the window of time over which the aggregate is applied or specify the validity intervals for the output events that your aggregate produces. The Event Server handles these aspects of the aggregation automatically.

Input and Output Requirements for a User-Defined Aggregate

Within an EQL statement, a UDA specifies one input parameter. This parameter specifies the attribute whose value is to be aggregated. In the sample below, the UDA_Median aggregate specifies the Temperature attribute as its input parameter.

```
SELECT UDA_Median(Temperature)
FROM ActualWeather WINDOW(RANGE 24 HOURS);
```

A user-defined aggregate must produce a return value. The return value must be one of the following types: Integer, Double, Float, Long, Boolean, Byte, Short, BigDecimal, String.

Adding a User-Defined Aggregate to a Continuous Query Project

Use the following procedure to create a new user-defined aggregate.

To create a new User-Defined Aggregate

1 In the Continuous Query Development, choose **File > New > User-Defined Extension**.

2 In the wizard, choose the **User-Defined Aggregation Function** option and complete the following fields:

In this field...	Specify...
Project	The project to which you want to add the user-defined aggregate.
Package	The name of the Java package to which the UDA class will belong. The name you enter must be a valid Java package name, as it will be added automatically to the code template that the wizard generates.
Class	The name you want to assign to the UDA Java class. The name you enter must be a valid Java class name, as it will be added automatically to the code template that the wizard generates.

3 Click **Finish**.

The wizard generates a code template for your UDA class. This class is a public Java class that extends the abstract class UserDefinedAggregationFunction.

4 Code your UDA by implementing the methods in the code template. For information about implementing a UDA, see *Implementing a User-Defined Aggregate*.

Implementing a User-Defined Aggregate

When you add a user-defined aggregate to your project using the **New** > **User-Defined Extension** command, Software AG Designer creates a code template that contains skeletons for the following classes.

Class	Description
The status class	Maintains the state of the aggregation between the events.
The aggregator class	Updates the state of the aggregation when a new event arrives and computes the aggregate value from a given state object.

To create a UDA and access it in a query, you must complete the implementations for these classes.

An Overview of the Aggregation Mechanism

The computation of a user-defined aggregate relies on a user-defined aggregation method. This method is called for an event. The method receives an internally stored state and the attribute value (from the event) that is to be aggregated. The method returns a new state. The actual aggregate value is subsequently derived from the new state.

For example, let's say you want to develop a UDA that computes the mean for a series of events. For this UDA, the stored state would comprise two values: the `sum` and `the number of events`. Given a new event and the previous state, the aggregation method would return a new state that consists of the sum incremented by the event value and the number of events incremented by 1. Given these two statistics, the mean can be determined by dividing the sum by the number of

events. Thus, you have a stored state from which the aggregate value (the mean, in this case) can be derived.

When you implement a UDA, you must specify 1) what the state stores, 2) how a new state is produced given a new event and a previous state, and 3) how to derive the aggregate from a given state.

Sample User-Defined Aggregates

To illustrate the steps involved in creating a user-defined aggregate, this documentation refers to sample code from the following UDAs in the User-Defined Extensions Demo project.

Sample UDA	Description
Average	A basic UDA that computes the mean value for a set of attribute values.
Median	A basic UDA that computes the median value for a set of attribute values.

The source code for these examples is shown in **UDA Sample Code Listings**. If you install the User-Defined Extensions Demo, you can examine the source code in Software AG Designer and run queries that use the sample UDAs. For information about installing and running the User-Defined Extensions Demo, see *Example: User-Defined Extensions* in *Getting Started with Complex Event Processing*.

Implementation Overview

The following sections describe the general steps you must take to complete the implementation for a user-defined aggregate.

Step	See Section
1	*Optional.* **Assigning an Explicit Name to a User-Defined Aggregate**
2	**Specifying the Input and Output Types that the Aggregate Uses**
2	**Implementing the Status Class**
3	**Implementing the Aggregator Class**

Assigning an Explicit Name to a User-Defined Aggregate

By default, the name of a user-defined aggregate consists of the fully qualified name of your aggregator class, delimited with the underscore character instead of a period. For example, the default name for the following UDA is `com_softwareag_demo_uda_Median`.

```
package com.softwareag.demo.uda;
...
public class Median extends UserDefinedAggregationFunction<Double, Median_Status, ↩
Double>
...
}
```

When you reference a UDA in a query statement, you must add the prefix "UDA_" to its name. The following example illustrates how you would reference the UDA shown above in a query statement:

```
SELECT UDA_com_softwareag_demo_uda_Median(Temperature)
FROM ActualWeather WINDOW(RANGE 24 HOURS);
```

You can optionally use the name property in the `@UserDefinedAggregationFunctionProperties` annotation to assign an explicit name to a UDA. In the following example, the UDA has been assigned the name `Median`.

```
package com.softwareag.demo.uda;
...
@UserDefinedAggregationFunctionProperties(name="Median")
public class Median extends UserDefinedAggregationFunction<Double, Median_Status, ↩
Double>  {
...
}
```

When you assign an explicit name to a UDA, you must reference the UDA by that name in a query statement. You can no longer refer to it by its default name. The following illustrates how you would refer to the `Median` aggregate in a query. Note that you must add the "UDA_" prefix to an explicit name, just as you do with the default name.

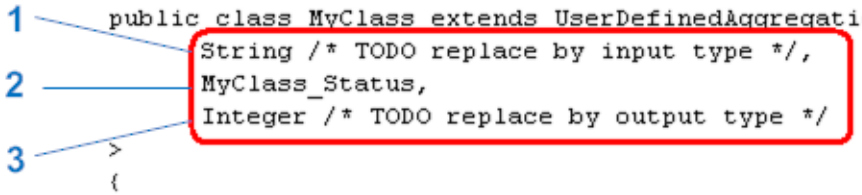
```
SELECT UDA_Median(Temperature)
FROM ActualWeather WINDOW(RANGE 24 HOURS);
```

Specifying the Input and Output Types that the Aggregate Uses

The class declaration for your aggregate includes a list of type parameters that specifies the input and output types used by the aggregator class. This list appears in angle brackets immediately before the body of your aggregator class. In the code skeleton, the list contains the following three parameters:

```
package com.mycompany.aggragates;
.
.
.

*/
@UserDefinedAggregationFunctionProperties(name="aggregateAlias")
public class MyClass extends UserDefinedAggregationFunction<
    String /* TODO replace by input type */,
    MyClass_Status,
    Integer /* TODO replace by output type */
>
{
  public MyClass_Status createInitialState()
  {
    // TODO return initial status object
    return null;
  }
```



You must edit this list to specify the data types that your UDA uses as input and output. For information about the types that an UDA can take as input and return as output, see *Input and Output Requirements for a User-Defined Aggregate*.

In this Position...	Specify...
1	The data type of the attribute that you are aggregating.
2	The data type of the state object for this aggregate.
3	The data type of the output value that the aggregate returns.

The following code snippet shows the parameter list for the Median UDA in the User-Defined Extensions Demo project. This UDA takes a Double attribute as input and returns a Double value as output. The UDA maintains its state in an object of class Median_Status.

```
package com.softwareag.demo.uda;
.
.
.

*/
@UserDefinedAggregationFunctionProperties(name="aggregateAlias")
public class Median extends UserDefinedAggregationFunction<Double, Median_Status, Double>
{
  private static final int EVENDIVISOR = 2;

/**
   * Creates an initial status.
   * @return the initial status
   */
  public Median_Status createInitialState()
  {
    return new Median_Status();
  } }
```

Implementing the Status Class

The status class is the first class that appears in the code template. It has the name *<your UDA class name>*_Status . This class defines the state object that your UDA uses to maintain the state of an aggregation between events. For example, if you have a simple UDA that computes the maximum value for events, its state object will maintain the maximum value that the aggregator has received thus far.

Your status class can contain whatever logic is necessary to maintain the state of your aggregation. However, it typically contains two constructors: one constructor to create a new, empty state object and the other to create an object based on a given previous state. These objects are used by your aggregator class.

The following shows the status class for the Average UDA in the User-Defined Extensions Demo project. To calculate the average over a series of events, the UDA needs 1) the sum of the values that it has received thus far in the series and 2) the number of events that it has received thus far in the series. As shown below, the state object for the Average UDA holds these two values.

```
class Average_Status
{
 int numberOfValues;
 double sumOfValues;

 /**
  * Creates an initial status with sum and counter being zero.
  */
 public Average_Status() {}

 /**
  * Creates a status with given number and sum of values.
  * @param counter the number of values
```

```
  * @param sum the sum of values
  */
 public Average_Status(int counter, double sum)
 {
  this.numberOfValues = counter;
  this.sumOfValues = sum;
 }
}
```

The following shows the status class for the Median UDA in the User-Defined Extensions Demo project. To calculate the median value for a series of events, the UDA must have the entire set of values in the series. Therefore, the state object for the Media UDA contains an array to hold all of the values that the UDA receives.

```
class Median_Status
{

 Double[] medianValues;

 /**
  * Creates an initial status.
  */
 public Median_Status()
 {
  medianValues = new Double[0];
 }

 /**
  * Creates a status with the given value array.
  * @param medianValues the value array
  */
 public Median_Status(Double[] medianValues)
 {
  this.medianValues = medianValues;
 }

}
```


Note: Although a user-defined aggregate must maintain the state of an aggregation between events, you are not required to do this using an explicit class as is done with the status class in the code template. The use of a status class is one suggested approach. You could also embed the logic for maintaining state within the main aggregator class.

Implementing the Aggregator Class

The aggregator class is a public class that extends UserDefinedAggregationFunction. It contains the methods that are used to perform the aggregation. The Event Server calls these methods directly when it evaluates your UDA during query execution.

The aggregator class contains the following methods. You must provide implementations for each of them.

createInitialState()	Returns an empty state object for this aggregator. This state object is used as input to the aggregate() method when the first event arrives.
aggregate()	Updates the state object for this aggregator given an incoming event and the previous state object.
getAggregate()	Computes and returns the aggregate value for a given state object. Event Server calls this method when it requires the aggregated result for a series of events.

Implementing the createInitialState() Method

The createInitialState() method must return an initialized instance of the your aggregate's state object. This initial state object is required to produce a new state object when the first event arrives.

The following snippet shows the implementation of the createInitialState() method for the Median UDA in the User-Defined Extensions Demo project. Note that this method simply returns an empty instance of the Median_Status object.

```
...
  public Median_Status createInitialState()
  {
    return new Median_Status();
  }
...
```

Implementing the aggregate() Method

The aggregate() method produces a new state object containing updated state information. The Event Server calls this method when it receives an event to include in the aggregation. When Event Server calls the method, it passes in the aggregation's previous state object and the value (from the newly received event) that is to be added to the aggregation. The aggregate() method returns a *new* state object containing the updated state.

Important: *The aggregate() method must never directly update the state object that it receives from the Event Server.* It must always create a new instance of the state object and update the new object. If the aggregate() method updates the state object that it receives from Event Server, it will corrupt the state of earlier points in the event stream.

In the code template, the first input parameter and the return value are already defined based on the name of the status class in the code template. The second input parameter in the code template is, by default, defined as a String. If your UDA does not aggregate String values, update the method signature in the code template to specify the correct input type.

```
public myClass_Status aggregate(myClass_Status previousStatus, String /* TODO replace by input type */ value)
 {
   // TODO return new status object based on the previous status where the incoming field value is aggregated
   return null;
 }
```

The following snippet shows the aggregate() method for the Median_Status UDA in the User-Defined Extensions Demo project. Its signature indicates that it takes a Median_Status object and a Double as input and returns a Median_Status object as output. Within its body, the method receives the previous state object, which contains an array of Doubles, and immediately assigns the array to a new variable. Then it adds the new value to the array, sorts the values, and returns the updated array in a new state object.

```
...
public Median_Status aggregate(Median_Status previousStatus, Double value)
{
 Double[] oldValues = previousStatus.medianValues;
 Double[] m_values;

 m_values = new Double[oldValues.length + 1];
 for(int i = 0; i < oldValues.length; i++)
 {
  m_values[i] = oldValues[i];
 }
 m_values[oldValues.length] = value;

 Arrays.sort(m_values);

 return new Median_Status(m_values);
}
...
```

The pattern illustrated above (minus the sorting step, which is a specific requirement for a median computation) is a typical design pattern for a UDA that maintains the entire set of received values in its state object. Other aggregates, such as those that compute average, minimum, and maximum, only need to maintain certain summary values. For an example of a UDA that maintains only summary values in its state object, see the aggregate() method in the *Average UDA source code listing*.

Implementing the getAggregate() Method

The getAggregate() method computes the aggregate value for a given state object. It takes your UDA's state object as input and returns the computed aggregate as output. In the code template, the return value for this method is defined as an Integer. If your aggregate does not produce an Integer, update the method signature to specify the correct return type.

```
...
public Integer /* TODO replace by output type */ getAggregate(myClass_Status status)
{
  // TODO Compute the output value based on the given status of the aggregate
  return null;
}
...
```

The following code snippet shows the getAggregate() method for the Median_Status UDA in the User-Defined Extensions Demo project. This method receives a state object containing a sorted array of values (as generated by the aggregate() method) and it returns the median value from that array.

```
...
 public Double getAggregate(Median_Status status)
  {
 int length = status.medianValues.length;
 if(length % EVENDIVISOR != 0)
 {
  return status.medianValues[length/EVENDIVISOR];
 }
 else
 {
  return 1/EVENDIVISOR * (status.medianValues[length/EVENDIVISOR-1] + ↩
status.medianValues[length/EVENDIVISOR]);
 }
  }
...
```

Handling Exceptions in a User-Defined Aggregate

If exceptions occur at run time, Event Server will catch them and write a notification to the console-view (if the UDA is running in Software AG Designer) or the server log (if the UDA is running on an Event Server and the logging level is at "warn" or higher). For information about viewing the log file, see *Working with the Server Log* in *Administering the Event Server*.

If the exception is caught during execution of one of the aggregates methods, the state object for that aggregation becomes invalid, and results produced from it will be null for the current event as well as any future events that reach it. (Note that if the exception occurs during the aggregation of a group, only the results for that particular group are invalidated.) To ensure meaningful results

from a user-defined aggregate, make certain that its methods process events cleanly and without generating exceptions.

Points to Consider when Implementing a UDA

When developing a user-defined aggregate, keep the following points in mind:

- Wherever possible, avoid accessing external systems from a UDA.
- When you develop a user-defined aggregate, you can extend the classpath of the continuous query project by custom libraries or class folders. Both need to be located inside the continuous query project subtree, so that they can be included in an export archive later on. This means that the Java Buildpath options "**Add external JARS...**" and "**Add External Class Folder...**" must not be used for the project.
- As a UDA can return NULL either by intention or due to an exception, the return type of a UDA is marked as nullable. This becomes relevant if the UDA is used in the uppermost SELECT clause of a query and the associated event type is to be selected from a given set of event types.

UDA Sample Code Listings

Average Example

The following is the complete listing for the Average user-defined aggregate. For more information about this example, see *Example: User-Defined Extensions* in *Getting Started with Complex Event Processing*.

```
package com.softwareag.demo.uda;

import com.softwareag.wep.resource.uda.UserDefinedAggregationFunction;
import com.softwareag.wep.resource.uda.UserDefinedAggregationFunctionProperties;

/**
 * This class stores the status required for computing the average,
 *  i.e. the number and the sum of values.
 */
class Average_Status
{
 int numberOfValues;
 double sumOfValues;

 /**
  * Creates an initial status with sum and counter being zero.
  */
```

```
 public Average_Status() {}

 /**
  * Creates a status with given number and sum of values.
  * @param counter the number of values
  * @param sum the sum of values
  */
 public Average_Status(int counter, double sum)
 {
  this.numberOfValues = counter;
  this.sumOfValues = sum;
 }
}

/**
 * This class provides a User-Defined Aggregation Function that computes the average ↩
of Double values,
 * which is defined as (sum of values)/(number of values).
 * The resulting UDA can be accessed in SQL statements with "UDA_Average".
 */
@UserDefinedAggregationFunctionProperties(name="Average")
public class Average extends ↩
UserDefinedAggregationFunction<Double,Average_Status,Double>
{
  /**
   * Creates an initial status.
   * @return the initial status
   */
  public Average_Status createInitialState()
  {
    return new Average_Status();
  }

  /**
   * Returns a new status by aggregating the previous status with the new value.
   * @param previousStatus the previous status
   * @param value the new value
   * @return the new status
   */
  public Average_Status aggregate(Average_Status previousStatus, Double value)
  {
    return new Average_Status(previousStatus.numberOfValues + 1, ↩
previousStatus.sumOfValues + value);
  }

  /**
   * Returns the average derived from the input status.
   * @param status the status
   * @return the average
   */
  public Double getAggregate(Average_Status status)
  {
```

```
    return status.sumOfValues / status.numberOfValues;
  }
}
```

Median Example

The following is the complete listing for the Median user-defined aggregate. For more information about this example, see *Example: User-Defined Extensions* in *Getting Started with Complex Event Processing*.

```
package com.softwareag.demo.uda;

import java.util.Arrays;

import com.softwareag.wep.resource.uda.UserDefinedAggregationFunction;
import com.softwareag.wep.resource.uda.UserDefinedAggregationFunctionProperties;

/**
 * This class stores the status required for computing the median.  <BR>
 * This status consists of the values at that time instant,
 * which are stored in an array.
 */
class Median_Status
{

 Double[] medianValues;

 /**
  * Creates an initial status.
  */
 public Median_Status()
 {
  medianValues = new Double[0];
 }

 /**
  * Creates a status with the given value array.
  * @param medianValues the value array
  */
 public Median_Status(Double[] medianValues)
 {
  this.medianValues = medianValues;
 }

}

/**
 * This class provides a User-Defined Aggregation Function that computes the median ↩
of Double values.
 * The median is a statistical measure, which is more robust to outliers than the ↩
average.
```

```
 * The median is defined for a sample, i.e. a set of numerical values, and separates ↩
the lower and the higher half.
 * For the computation of the median the sample is sorted first.
 * If the sample size is odd, the value in the middle is returned (e.g. sample ↩
[9,1,4] has median 4).
 * If the sample size is even, the mean of the two values in the middle is returned ↩
(e.g. sample [6,9,1,4] has median 5).
 *
 * The resulting UDA can be accessed in SQL statements with "UDA_Median".
 */
@UserDefinedAggregationFunctionProperties(name="Median")
public class Median extends UserDefinedAggregationFunction<Double, Median_Status, ↩
Double>
{
  private static final int EVENDIVISOR = 2;

/**
   * Creates an initial status.
   * @return the initial status
   */
  public Median_Status createInitialState()
  {
    return new Median_Status();
  }

  /**
   * Returns a new status by aggregating the previous status with the new value.
   * @param previousStatus the previous status
   * @param value the new value
   * @return the new status
   */
  public Median_Status aggregate(Median_Status previousStatus, Double value)
  {
 Double[] oldValues = previousStatus.medianValues;
 Double[] m_values;

 m_values = new Double[oldValues.length + 1];
 for(int i = 0; i < oldValues.length; i++)
 {
  m_values[i] = oldValues[i];
 }
 m_values[oldValues.length] = value;

 Arrays.sort(m_values);

 return new Median_Status(m_values);
  }

  /**
   * Returns the median derived from the status.
   * @param status the status
   * @return the median
```

```
   */
  public Double getAggregate(Median_Status status)
  {
 int length = status.medianValues.length;
 if(length % EVENDIVISOR != 0)
 {
  return status.medianValues[length/EVENDIVISOR];
 }
 else
 {
  return 1/EVENDIVISOR * (status.medianValues[length/EVENDIVISOR-1] + ↩
status.medianValues[length/EVENDIVISOR]);
 }
  }

}
```

10 Developing User-Defined Operators

What is a User-Defined Operator?

There are certain kinds of analyses that are difficult or impossible to accomplish using EQL alone. For example, a plant manager might want to know whether a manufacturing process is operating within an optimal range by analyzing the readings from multiple sensors. This kind of analysis often requires proprietary logic that cannot be expressed using EQL. In cases like this, you can analyze the event streams using customized user-defined operators (UDOs).

A user-defined operator is a Java class that you create to process event streams. It takes one or more event streams as its input and returns an event stream as output. In addition to event streams, a UDO can also take database sources as input.

Within an EQL statement, a user-defined operator appears in the FROM clause (like other event streams). When it is referenced in a query statement, the name of a user-defined operator always begins with the prefix "UDO_".

In the following example, a user-defined operator called "UDO_SysCheck" analyzes incoming events from several sensors and produces a quality measurement based on the sensor readings. The query then correlates the quality reading from the UDO with another input stream called "qosStream".

```
SELECT udoResult.*
FROM UDO_sysCheck(tempStream, pressureStream, turbidityStream) AS udoResult
   JOIN qosStream ON udoResult.qualityIndex < qosStream.requiredQuality;
```

How is a User-Defined Operator Different from a User-Defined Function?

User-defined operators and user-defined functions (UDFs) both enable you to add custom business logic to a query. However, user-defined operators are far more complex than user-defined functions. You use UDOs when your business logic needs to operate across the successive tuples in an event stream and not just on certain attributes within an individual event.

A User-Defined Operator...	A User-Defined Function...
Takes event streams and, optionally, database sources as input.	Takes any combination of constants, attribute values, and/or results from other functions as input.
Returns an event stream.	Returns a single value.
Appears in the FROM clause.	Appears in the SELECT clause, WHERE clause, or HAVING clause (see the GroupByClause).
Is stateful. A UDO can retain information in memory about earlier events that it has processed, and it can apply that information when processing subsequent	Is stateless. A UDF *must not* retain any information in memory about earlier events that it has processed. Event Server instantiates just one instance of the

A User-Defined Operator...	A User-Defined Function...
events. For this reason, Event Server instantiates a separate instance of the operator class for each query that uses the operator.	function class and that instance is used by all queries that invoke the function.

How Event Server Processes a User-Defined Operator

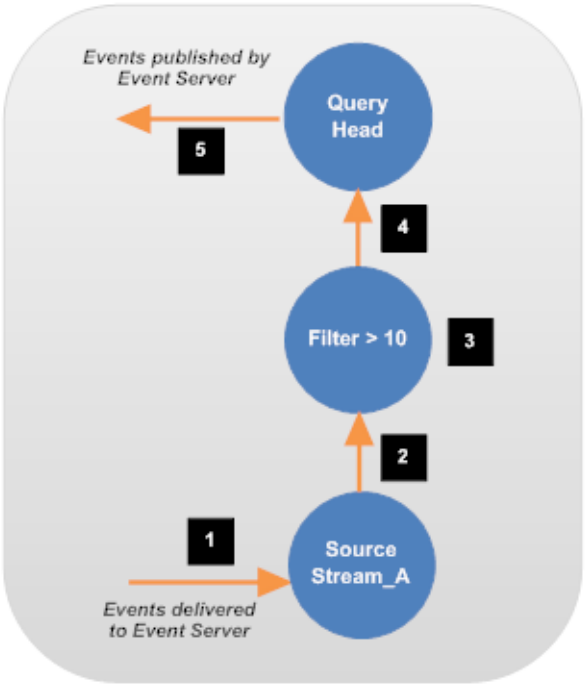
To process a query, Event Server translates your query statement into a query plan. A query plan defines the sequence of operators that Event Server must execute to carry out the query. The diagram below depicts the query plan for the following query statement:

The Query Statement

```
SELECT *
FROM Stream_A
WHERE attribute1 > 10;
```

The Query Plan

Each circle in the diagram represents an operator that executes as part of the query. When an event in Stream_A arrives, each operator executes in turn and pushes its results to the next operator in the sequence.



Note: Query-plan diagrams are read from the bottom up.

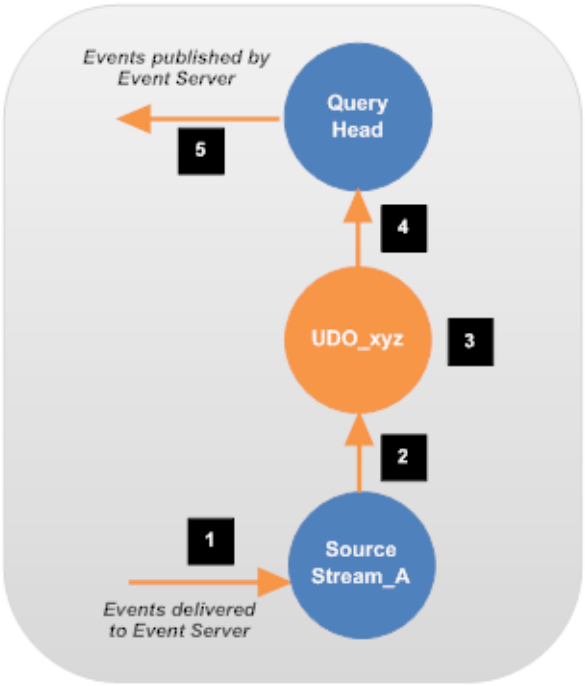
Step	Description
1	Events are received by the Event Server. When an event from Stream_A arrives, it is given to the source operator for Stream_A.
2	The source operator pushes the event to the next operator in the query plan, which in this example is the filter operator.
3	The filter operator receives the event and tests the value of Attribute1. If the value is less than or equal to 10, the event is discarded. If the value is greater than 10, the filter operator pushes the event to the next operator in the query plan.
4	The query-head operator is the last operator in all query plans. When this operator receives an event from an upstream operator, it publishes the event to the Event Bus. **Note:** The query-head operator is an internal operator that Event Server executes implicitly for all queries. The diagram above shows the query-head operator as part of the query plan, however, some query-plan diagrams omit it.

A UDO is simply another operator that can appear in a query plan. In the following query statement, the user-defined operator, UDO_xyz, is used to process Stream_A. As you can see by the query plan, UDO_xyz is treated like any other operator. It receives events from an upstream operator, processes the events in a prescribed manner, and pushes its results to the next operator in the plan.

The Query Statement

```
SELECT *
FROM UDO_xyz(Stream_A);
```

The Query Plan



When Should You Use a User-Defined Operator?

User-defined operators offer a great deal of flexibility and enable you to act upon event streams at a very low level. However, they are complex. A UDO must interact with the Event Server in a prescribed way and deliver a valid stream of output events. Moreover, it must perform many of the behaviors that are implicit in Event Server's own operators. For example, if your operator correlates events between two streams, it is responsible for calculating the intervals where the events intersect in time and assigning the proper timestamps to the resulting output event.

Apart from the complexity associated with implementing a UDO, the following are other points to consider when deciding whether to implement a UDO.

- When you encapsulate event-processing logic in custom code, you lose the advantages of using a standard and well-known query language.
- To be used successfully in a high-availability configuration, the output streams produced by a UDO must satisfy additional requirements relating to result equivalency. In a high-availability environment, the master and the slave servers receive input streams that are equivalent, but not necessarily identical. To run successfully in a high-availability environment, a UDO must be able to produce equivalent results for equivalent input streams. If it cannot do this, the high-availability system will fail.

A user-defined operator is the most complex tool you can include in an event-processing solution. Implement your own operator only when you cannot obtain the analysis you need using EQL (and possibly user-defined functions) alone.

Adding a User-Defined Operator to a Continuous Query Project

Use the following procedure to create a new user-defined operator.

To create a new User-Defined Operator

1 In the Continuous Query Development, choose **File > New > User-Defined Extension**.

2 In the wizard, choose the **User-Defined Operator** option and complete the following fields:

In this field...	Specify...
Project	The project to which you want to add the user-defined operator.
Package	The name of the Java package to which the UDO class will belong. The name you enter must be a valid Java package name, as it will be added automatically to the code template that the wizard generates.
Class	The name you want to assign to the UDO Java class. The name you enter must be a valid Java class name, as it will be added automatically to the code template that the wizard generates.

3 Click **Finish**.

The wizard generates a code template for your UDO class. This class is a public Java class that extends the abstract class UserDefinedOperator.

4 Code your UDO by implementing the methods in the code template. For information about implementing a UDO, see *Implementing a Basic User-Defined Operator*.

Understanding the Lifecycle of a User-Defined Operator

Before you implement a user-defined operator, it is helpful to familiarize yourself with the lifecycle of a UDO so that you understand when the UDO's methods are called.

A user-defined operator is instantiated when Event Server deploys a query that uses the user-defined operator. Instantiation is performed by an internal factory class. The factory produces a separate instance of the UDO class for each reference to the UDO in a query. In other words, if two queries use the same UDO, the factory produces a separate instance of the UDO class for each query.

The lifecycle of a user-defined operator can be broken into three basic stages: initialization, processing, and shutdown. During the initialization stage, Event Server calls initialization methods in the UDO to make it ready for use. When initialization is complete, the UDO enters the processing stage. During this stage, it processes the events and heartbeats that it receives from upstream operators. The UDO continues to process events until the query is undeployed, or, if it is running against an event sequence file in Software AG Designer, the end of the input stream is reached. At this point, the shutdown stage begins. During the shutdown stage, Event Server calls shutdown methods in the UDO so it can close data structures and perform other necessary cleanup procedures.

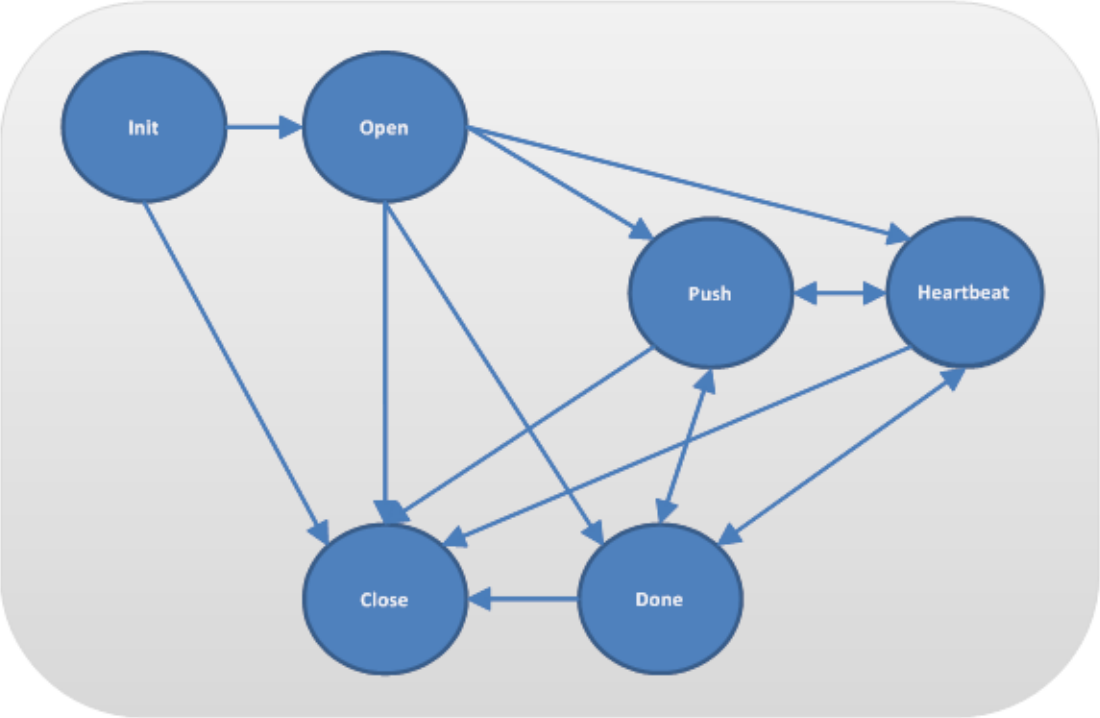
Methods that are Called During the Lifecycle of a UDO

The following table identifies the methods that are called during the stages of a UDO's lifecycle. When you implement a UDO, you provide implementations for these methods that are appropriate for your UDO.

Stage	Method	When Called
Initialization	`init()`	Event Server calls this method when it deploys the query to which the operator belongs. If your operator needs to initialize data structures, it can perform that work in this method.
	`open()`	Event Server calls this method after the `init()` methods on all operators in the query have been executed successfully. Successful execution of the `open()` method indicates that your operator is ready to receive events.
Processing	`push()`	Upstream operators call this method to push an event to your operator. This method should contain the logic that processes an incoming event and pushes resulting output events to the next operator.
	`heartBeat()`	Upstream operators call this method to push a heartbeat to your operator. This method should contain the logic that processes an incoming heartbeat.
	`done()`	Software AG Designer calls this method when it has reached the end of the input stream in an event sequence file. This method should contain the logic that performs any processing that is necessary to process the last event in an input stream.
Shutdown	`close()`	Event Server calls this method when it undeploys a query.

The following diagram illustrates the order in which methods can be called during the lifecycle of a UDO.

Potential paths of execution in the UDO lifecycle



Sample Code Used in the Documentation

To illustrate the steps involved in creating a basic user-defined operator, this documentation uses sample code from the UDO_ItemListSplitter operator in the User-Defined Extensions Demo project. The example reads events from an input stream called ShoppingItemList. The events in this stream represent the shopping carts of individual buyers. The schema for the stream has two fields: *BuyerID* and *ShoppingCart*.

Field Name	Description
BuyerID	An integer that identifies the buyer.
ShoppingCart	A String that contains a list of the items in the cart and their quantities.

Example

BuyerID	ShoppingCart
14408	Pear, 6, Melon, 1, Milk, 10

The UDO_ItemListSplitter operator parses the list of items in the ShoppingCart attribute and produces a separate output event for each item in list. For example, given the example event shown above, UDO_ItemListSplitter will generate the following three output events:

BuyerID	Item	Quantity
14408	Pear	6
14408	Melon	1
14408	Milk	10

The source code for the UDO_ItemListSplitter operator is shown in **The UDO_ListItemSplitter Example (Basic)**. If you install the User-Defined Extensions Demo, you can also examine the source code in Software AG Designer and execute a query that calls the UDO_ListItemSplitter operator. For information about installing and running the User-Defined Extensions Demo, see *Example: User-Defined Extensions* in *Getting Started with Complex Event Processing*.

Implementing a Basic User-Defined Operator

When you add a user-defined operator to your project using the **New** > **User-Defined Extension** command, Software AG Designer creates a skeleton for a public Java class that extends the abstract class `UserDefinedOperator`.

As shown below, the skeleton code includes the annotation `@UserDefinedOperatorProperties` before the class statement. This annotation is required in any Java class that functions as a user-defined operator.

```
package com.softwareag.demo.udo;
...
@UserDefinedOperatorProperties()  // This annotation is required in a UDO ↩
implementation class
public class MyUDO extends UserDefinedOperator {
...
}
```

Within the parenthesis of the annotation, you can specify optional properties relating to your operator. For more information about the properties you can assign in the `UserDefinedOperatorProperties` annotation, see ***Setting Properties in the UserDefinedOperatorProperties Annotation***.

The skeleton code provides default implementations for the methods associated with a user-defined operator. Where appropriate, you replace the default implementations with implementations that are suitable for your UDO.

The following sections describe the general steps you must take to implement a basic user-defined operator.

Step	See Section
1	Optional. **Assigning an Explicit Name to a User-Defined Operator**
2	Optional. **Setting Properties in the UserDefinedOperatorProperties Annotation**
3	**Implementing the `getOutputStreamMetaData()` Method**
4	**Implementing the `requiresGloballyOrderedInputs()` method**
5	**Implementing the `init()` Method**
6	**Implementing the `open()` Method**
7	**Implementing the `push()` Method**
8	**Implementing the `heartBeat()` Method**
9	**Implementing the `done()` Method**
10	**Implementing the `close()` Method**

Assigning an Explicit Name to a User-Defined Operator

By default, the name of a user-defined operator consists of its fully qualified class name delimited with the underscore character instead of a period. For example, the default name for the following UDO is `com_softwareag_demo_udo_ItemListSplitterUdo`.

```
package com.softwareag.demo.udo;
...
public class ItemListSplitterUdo extends UserDefinedOperator {
...
}
```

When you reference a UDO in a query statement, you must add the prefix "UDO_" to its name. The following example illustrates how you would reference the UDO shown above in a query statement:

```
SELECT *
FROM UDO_com_softwareag_demo_udo_ItemListSplitterUdo(ShoppingItemList);
```

You can optionally use the name property in the `UserDefinedOperatorProperties` annotation to assign an explicit name to a UDO. In the following example, the UDO has been assigned the name `ItemListSplitter`.

```
package com.softwareag.demo.udo;
...
@UserDefinedOperatorProperties(name = "ItemListSplitter")
public class ItemListSplitterUdo extends UserDefinedOperator {
...
}
```

For more information about the `UserDefinedOperatorProperties` annotation, see ***Setting Properties in the UserDefinedOperatorProperties Annotation***.

When you assign an explicit name to a UDO, you must reference the UDO by that name in a query statement. You can no longer refer to it by its default name. The following illustrates how you would refer to the `ItemListSplitter` operator in a query. Note that you must add the "UDO_" prefix to an explicit name, just as you do a default name.

```
SELECT *
FROM UDO_ItemListSplitter(ShoppingItemList);
```

Setting Properties in the UserDefinedOperatorProperties Annotation

The `@UserDefinedOperatorProperties` annotation can specify the following optional property settings:

Annotation Element	Meaning
`name`	The name to be assigned to the UDO
`numberOfInputStreams`	The number of input streams that the UDO will process
`numberOfInputCaches`	The number of input database caches that the UDO will process

In the following example, the UDO has been assigned the name `ItemListSplitter`.

```
package com.softwareag.demo.udo;
...
@UserDefinedOperatorProperties(name = "ItemListSplitter")
public class ItemListSplitterUdo extends UserDefinedOperator {
...
}
```

If the elements `numberOfInputStreams` and `numberOfInputCaches` are set, the corresponding values are used when the UDO is instantiated. A check is made before the `getOutputStreamMetaData()` method is called to verify that the number of input streams and caches that are passed as parameters is the same as the numbers that you specified in the annotation. The values of the elements are also used for syntax highlighting in the Software AG Designer; and, if specified, the values of the elements `numberOfInputStreams` and `numberOfInputCaches` are used by the Software AG Designer to generate tool tips showing the number of parameters required for the UDO.

It is good programming practice always to set the three attribute elements `name`, `numberOfInputStreams` and `numberOfInputCaches`, even if, for example, the UDO will not process

any input caches. Setting `numberOfInputCaches = 0` means that the system will check that the UDO call does not include any input caches; in contrast, omitting this element means that the check is not performed. Of course, it is not possible to set `numberOfInputStreams` and `numberOfInputCaches` if your UDO is designed to process a variable number of inputs.

Important: If you choose not to specify `numberOfInputStreams` and `numberOfInputCaches` in the UDO annotation, you should write code in the `getOutputStreamMetaData()` method that checks that the correct number of parameters have been passed.

Index Numbering of Input Streams

The methods in a user-defined operator refer to input streams by index number. The index numbering is 0 based and reflects the order in which the streams appear in the list of streams that the query passes to the UDO. For example, the UDO in the following query takes two event streams, ShoppingItem and ActualWeather, as input. Within the operator's implementation, the ShoppingItem stream will be referenced as index 0 and the ActualWeather stream will be referenced as index 1.

```
SELECT * FROM UDO_TopKAnalyzer(ShoppingItem, ActualWeather);
```

Implementing the getOutputStreamMetaData() Method

Event Server calls the `getOutputStreamMetaData()` method when it parses the query that contains your user-defined operator. During the parsing operation, Event Server uses `getOutputStreamMetaData()` to obtain the schema for the output stream that your operator produces.

Your implementation of `getOutputStreamMetaData()` needs to perform two main tasks:

1. It must verify that the query calling your UDO is using the proper input parameters. Your verification logic should determine whether the UDO can work with the input streams, database sources, and instantiation parameters as given in the query. For example, you might want to check that:
   - The schemas associated with the input streams and database sources have the expected number of attributes and that the attributes are of the correct type.
   - The initialization parameters provided by the query (if any) are of the correct types.
   - The parameters satisfy any additional plausibility requirements. For example, if your operator takes an initialization parameter, you might want to verify that the parameter value given in the query falls within a particular range.

   If any of the validation checks fail, the `getOutputStreamMetaData()` method must throw a UserDefinedOperatorException.
2. It must define the schema of the output events that the operator produces and return the schema as a StreamMetaData object.

The following shows the `getOutputStreamMetaData()` method for the ItemListSplitter operator in the User-Defined Extensions Demo project. This operator receives a single event stream with two attributes, and produces an output stream with three attributes. For details about this example, see ***Sample Code Used in this Documentation***.

```
...
@UserDefinedOperatorProperties(name = "ItemListSplitter", numberOfInputStreams = ↩
1, numberOfInputCaches = 0)
...
public static StreamMetaData getOutputStreamMetaData(
      StreamMetaData[] inputMetaData, RecordMetaData[] cacheMetaData,
      Object[] instanceParameters) throws UserDefinedOperatorException {

  // There is no need to check for the correct number of input stream metadata and ↩
cache metadata,
  // since we defined these numbers in the annotation above and the system ↩
automatically checks for
  // the correct number. Only if we omitted the annotation attributes, we had to ↩
check for the
  // correct number here.

    MetaDataChecks.checkCorrectNumberOfColumns(inputMetaData[0], 2);
    MetaDataChecks.checkCorrectJavaType(inputMetaData[0], SHOPPINGCARTINDEX, ↩
Type.STRING);

    // define output structure
    boolean isChrononStream = inputMetaData[0].isChrononStream();
    FieldMetaData buyerId = MetaDataChecks.getColumnMetaData(inputMetaData[0], ↩
BUYERFIELDINDEX);
    FieldMetaData item = createFieldMetaData(ITEMFIELDDESCRIPTION, Type.STRING, ↩
!IS_NULLABLE, IS_CASE_SENSITIVE);
    FieldMetaData quantity = createFieldMetaData(QUANTITYFIELDDESCRIPTION, ↩
Type.INTEGER, IS_NULLABLE, IS_CASE_SENSITIVE);
    return createStreamMetaData(isChrononStream, buyerId, item, quantity);
  }
... ↩
```

For information about validating the database sources that a query passes to a UDO, see ***Validating the Given Database Sources***.

Implementing the requiresGloballyOrderedInputs() method

The Event Server calls the `requiresGloballyOrderedInputs()` method when it deploys a query that uses your operator. This method specifies whether events are to be delivered to your operator in the order that Event Server receives them or in *global temporal order*. Global temporal ordering ensures that, for any incoming event, cache update, or heartbeat having timestamp *t*, your operator is guaranteed never to receive an event, cache update, or heartbeat from any of its inputs, with a timestamp that is less than *t*.

When you have multiple input streams, receiving events in global temporal order simplifies the work your operator has to do to manage temporal progress of events. However, global ordering adds a small amount of computational overhead, which can affect the performance of your operator. It can additionally reduce the "liveliness" of an operator, because Event Server will not deliver events for a stream unless it has events with an equal or greater timestamp from all of the operator's input sources (i.e., event streams *and* database sources).

The default implementation for this method disables global ordering as shown below. If you want to enable global ordering, change the method's return value to REQUIRES_GLOBALLY_ORDERED_INPUTS.

```
...
 public boolean requiresGloballyOrderedInputs()
  {
    // TODO Change to REQUIRES_GLOBALLY_ORDERED_INPUTS if this operator assumes a ↩
global order on all inputs
    return DOES_NOT_REQUIRE_GLOBALLY_ORDERED_INPUTS;
  }
...
```

Note: The `requiresGloballyOrderedInputs()` is only useful if your operator takes multiple input streams and/or database sources. If your operator takes only one input stream, there is no reason to enable global ordering. The events within a single stream always arrive in non-descending temporal order.

For more information about the use of the `requiresGloballyOrderedInputs()` method, see *Managing Temporal Progress*.

Implementing the init() Method

The Event Server calls your operator's `init()` method when it deploys a query that uses your operator. The `init()` method is the first method executed in the operator's lifecycle and it is executed only once.

Your operator can use the `init()` method to perform any initialization and set-up steps that are needed to prepare your operator for receiving and processing events. If your operator has data structures that need to be constructed and initialized, the `init()` method is the appropriate place to perform that work.

If your user-defined operator does require any special initialization steps, leave the `init()` method empty like the one shown below:

```
...
  public void init() {
  }
...
```

For an example of an `init()` method that performs several initialization steps, see ***The UDO_TopKAnalyzer Example (Advanced)***.

Implementing the open() Method

The Event Server calls your operator's `open()` method after it has successfully instantiated and initialized all of the operators for the query. Successful execution of your operator's `open()` method will indicate to the Event Server that the operator is ready to receive events. The `open()` method is executed only once during the UDO's lifecycle.

When Event Server calls your operator's `open()` method, it passes in an OperatorCallBack object. As shown in the example below, your `open()` method *must store this object*. The OperatorCallBack contains the methods that your operator will use to push events and heartbeats to the next operator in the query plan.

```
...
  public void open(OperatorCallBack callback) {
    this.callback = callback;
  }
...
```

Besides storing the OperatorCallBack object, there is nothing else your `open()` method is required to do. You can perform additional work in this method if your operator requires it. However, the `open()` method *must not* execute any of the methods in the OperatorCallBack object. The methods in this object are meant to be called only after your `open()` method completes its execution.

Implementing the push() Method

Your operator's `push()` method is called each time an event arrives in one of the operator's input streams. The `push()` method is how your operator receives events and where it processes them. When the processing of an event produces a result, your operator uses the `push()` method from its callback object (the object you received and stored in the `open()` method) to push the result to the next operator in the query plan.

Note: There are two `push()` methods in the user-defined operator class. The one you use to process events is the one that takes a Record as input. The other takes a ReadOnlyCache as input and is used when working with a database source. For more information about using a database source with a user-defined operator, see ***Using a Database Source with a UDO***.

The following shows the `push()` method for the ItemListSplitter operator in the User-Defined Extensions Demo project. This operator receives an event containing a list of items in a shopping cart. The operator parses the list and produces one output event for each item in the list. For details about this example, see ***Sample Code Used in this Documentation***.

```
public void push(int index, Record element, long startTimeStamp, long endTimeStamp) {

    // pear, 6, apple, 10 .... separated
    String itemlist = (String) element.getObject(SHOPPINGCARTINDEX);
    String[] split = itemlist.split(DELIMITER);
    /*
     * iterate over the found item quantity pairs <BR>
     * fill the corresponding Item and Quantity field <BR>
     * and finally push each record as operator result
     */
    Object buyerId = element.getObject(BUYERFIELDINDEX);
    for (int I = 0; I < split.length; I = I + 2) {
      Object item = split[i];
      Integer quantity = Integer.valueOf(split[i + 1]);
      Record record = Adapters.createRecord(buyerId, item, quantity);
      callback.push(record, startTimeStamp, endTimeStamp);
    }
  }
```

Note that when your `push()` method is called, it receives the following inputs:

Input	Description
index	An integer that identifies the input stream from which the event originated. **Note:** Index numbering begins with 0. If you have `n` streams, their indices will be 0...(`n`-1). For more information about index numbering, see ***Index Numbering of Input Streams***.
element	A Record containing the event data.
startTimeStamp	The event's start timestamp.
endTimeStamp	The event's end timestamp.

The example above processes events from just one input stream. If your operator processes multiple streams and/or database sources, the logic in your `push()` method will be more complex. For an example of a `push()` method that processes multiple streams and a database source, see ***The UDO_TopKAnalyzer Example (Advanced)*** and the sections under **Advanced Implementation Topics**.

Note: Although the example UDOs both use chronon streams as input, this is not a requirement of user-defined operators. UDOs can use chronon streams or non-chronon streams as input.

When your `push()` method produces an output event, it must push the event to the next operator in the following form:

Output	Description
record	A tuple containing the data associated with the output event.
startTimeStamp	The event's start timestamp.
endTimeStamp	The event's end timestamp.

In the example above, the temporal progress of the output stream matches the temporal progress of the input stream, so the operator simply copies the timestamps from the input event to the output events. However, this behavior would not be appropriate for all UDOs. For example, if your operator correlates events in two streams over a window of time, the timestamps in the output events must indicate the interval of time during which the correlated events intersected. When publishing output events, you must ensure that the operator produces an event stream that is non-descending in time (i.e., it must never publish an event whose starting timestamp is older than the starting timestamp of the last event it published). For more information about producing valid output events, see ***Snapshot Reducibility*** and ***Managing Temporal Progress***.

Points to Consider when Implementing the push() Method

- The `push()` method will be executed repeatedly, so performance is important. If you need to optimize your operator, focus your efforts on this method.
- Your implementation of the `push()` method *should not* include error checking or data-validation logic. Schema validation logic should be performed in the `getOutputStreamMetaData()` method. Write your `push()` method with the assumption that the data is valid and simply process it.
- If your operator will be used in a high-availability environment, it must produce equivalent results for equivalent input streams. In a high-availability environment, the master and the slave servers receive input streams that are equivalent, but not necessarily identical. If a UDO does not produce equivalent results for equivalent input streams, the high-availability system will fail. For more information about this requirement, see ***Ensuring Equivalent Results in a High-Availability Environment***.
- If your UDO involves operations such as joining input streams, enriching events with data from a database, or filtering event streams, consider whether you can use EQL to accomplish that

work instead of implementing it in the UDO. Using EQL can reduce implementation work, however, you will need to determine whether it suits your particular use-case and performance requirements.

Implementing the heartBeat() Method

Your operator's `heartBeat()` method is called each time a heartbeat arrives in one of your operator's input streams.

If your operator has only one input stream and the temporal progress of its output stream corresponds to that of the input stream, you can use the implementation shown below to push the heartbeat to the next operator.

```
...
public void heartBeat(int index, InputType inputType, long timestamp) {
  callback.heartBeat(timestamp);
}
...
```

If your operator monitors the temporal progress of its event streams, it should process the heartbeat as necessary. For example, if your operator evaluates events over a 20-minute window, the arrival of a heartbeat might enable the operator to conclude its processing on the current 20-minute window. For additional details about handling heartbeat events, see ***Processing Heartbeats***. For an example of a `heartBeat()` implementation in an operator that monitors the temporal progress of its input streams, see ***The UDO_TopKAnalyzer Example (Advanced)*** .

Implementing the done() Method

Event Server calls your operator's `done()` method after it reads the last event for one of your operator's input streams from an event sequence file. This method is only called when the query is executed using an event sequence file to simulate an input stream.

The `done()` method enables your operator to perform steps that are needed to conclude processing of the very last event in the file. If your operator processes a single stream of events and the temporal progress of its output stream corresponds with that of its input stream, the `done()` method can simply report that the operator is done as shown in the following implementation:

```
...
public void done(int index, InputType inputType) {
  callback.done();
}
...
```

If your operator monitors the temporal progress of its input streams, the `done()` method might need to include logic to flush out a final result from the events that it has buffered in its memory. For an example of a `done()` implementation in an operator that monitors the temporal progress of its input streams, see ***The UDO_TopKAnalyzer Example (Advanced)*** .

If your operator uses database sources, it must dispose of its caches in the `done()` method. For additional information, see ***Disposing of a Cache***.

Implementing the close() Method

Event Server calls your operator's `close()` method when it undeploys a query. You can use this method to clean up any data structures that you might have created and set them to NULL in preparation for garbage collection. Note that `close()` can be called at any time during the operator's lifecycle. Your implementation of this method should not assume that any other method (including the `init()` method) was called before it.

If your user-defined operator does require any special clean-up steps before it is closed, leave the `close()` method empty like the one shown below:

```
...
  public void close() {
  }
...
```

If your operator uses database sources, it must dispose of its caches in the `close()` method. For additional information, see ***Disposing of a Cache***.

The completeFrom() and maximumPossiblyAffectedTimestamp() Methods

Event Server calls your operator's `completeFrom()` and `maximumPossiblyAffectedTimestamp()` methods when your operator subscribes to a query. You can use the default implementations provided for these methods (shown below) in most cases.

```
...
public long completeFrom(Long[] inputsCompleteFrom) {
    // TODO: Override this method for non-snapshot reducible operators!
    Long max = Long.MIN_VALUE;
    for (int I = 0; I < inputsCompleteFrom.length; I++) {
      if(inputsCompleteFrom[i] != null) {
        max = Math.max(inputsCompleteFrom[i], max);
      }
    }
    return max;
  }


  public Long maximumPossiblyAffectedTimestamp(Long[] inputTimestamps) {
    // TODO: Override, if the given implementation does not match the operator's ↩
semantics
    Long max = null;
    for (Long value : inputTimestamps)
      if (value != null)
        if (max == null)
```

```
          max = value;
        else
          max = Math.max(max, value);
    return max;
  }
...
```

The only time you must re-implement these methods is when the results that your operator returns are not snapshot reducible. For more information about when you must provide your own implementations of these methods, see ***Implementing the Completeness Methods if Your Operator is not Snapshot Reducible***.

Handling Exceptions in a User-Defined Operator

With respect to throwing exceptions, a user-defined operator must comply with the following contract:

- It *must* throw a UserDefinedOperatorException from its `getOutputStreamMetaData()` method if the input parameters given in the query are not valid for the operator. If your operator does not throw an exception at that point, it must be able to be instantiated successfully and operate without throwing exceptions using the parameters that have been given in the query. For an example of what a `getOutputStreamMetaData()` implementation should look like, see ***Implementing the `getOutputStreamMetaData()` Method***.
- It *must not* throw an exception from any method other than `getOutputStreamMetaData()`.

If unhandled exceptions occur at run time, Event Server will catch them and write a notification to the console-view (if the operator is running in Software AG Designer) or the server log (if the operator is running on an Event Server and the logging level is at "warn" or higher). For information about viewing the log file, see *Working with the Server Log* in *Administering the Event Server*.

Note: If an exception occurs repeatedly in the push( ) or heartBeat( ) methods, the log file can grow quite rapidly.

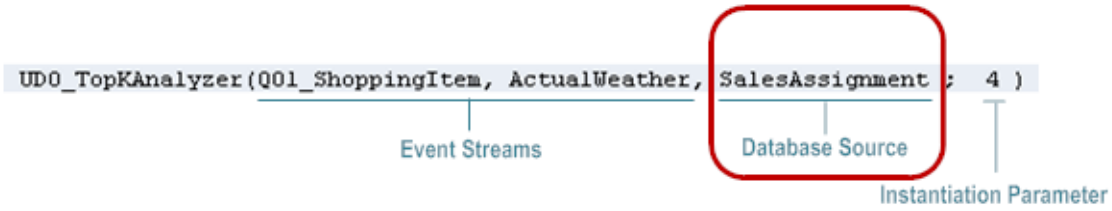
Using Database Sources with a User-Defined Operator

A user-defined operator (UDO) can have zero or more database sources as input. A database source represents a specified table or view in a database system.

Passing Database Sources to a User-Defined Operator

Within the input signature of a UDO, the database sources that a UDO takes as input are given after the list of event streams.

For example, the sample UDO, TopKAnalyzer, installed with Software AG Designer takes two event streams, one database source, and one instantiation parameter as input. When this UDO is used in the sample query, Q02_TopKSales, the query passes the following input parameters to the UDO.



The Caching Requirement for Database Sources Used by User-Defined Operators

In order for a user-defined operator to access the data for a given database source, the database source must be cached. When a database source is cached, the records associated with the database source are copied into memory. A UDO retrieves records from the database by "querying" the cache using the Event Server's ReadOnlyCache API. (For more information about the ReadOnlyCache API, see the Javadoc for the GeneralUserDefinedOperatorAdapter.ReadOnlyCache class.)

Caching of database sources eliminates the need for an operator to interact directly with the underlying database system. It also enables consistent query results in a high-availability environment by ensuring that both master and slave query precisely the same data sets. If the database sources were not cached, the Event Servers might receive different results for the same query, even if they execute the query only milliseconds apart. The difference could potentially cause the Event Servers to produce inconsistent output streams and cause the high-availability system to fail.

Note: An operator can only read records from cache. It cannot write to the cache.

The cache associated with a database source can be a one-time cache or a periodic cache. An operator can use either type, however, the implementation of the operator will vary depending on

whether a cache is periodically refreshed or not. For information about creating a database source that is cached, see *Combining Events with Data from a Database*.

Important: A user-defined operator must always use a cached database source to obtain data from a database. It must not use the JDBC API or any other direct means to interact with a database system. If you need to write data to a back-end database, consider having your operator publish the data in the form of an event and using a service (or other event consumer) to apply the resulting event to the database.

Implementing a User-Defined Operator that Retrieves Records from a Database

The following sections provide information about aspects of implementation that relate to the use of a database by a user-defined operator. In terms of general steps, an operator that uses records from a database must do the following:

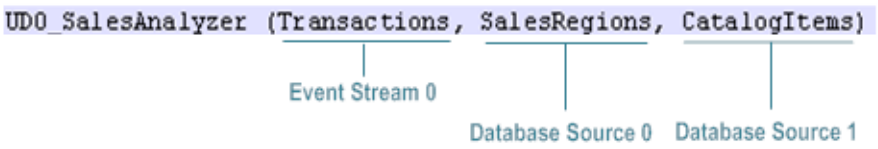
- Verify that it has received the expected database sources as input. For information, see *Validating the Given Database Sources*.
- *Optional*. Generate indicies on caches. For information, see *Generating an Index for a Cached Table or View*.
- Provide an implementation for the push( ) method relating to cache updates. For information, see *Implementing the push( ) Method for Your Caches*.
- Query the cache to obtain the data that it needs. For information, see *Retrieving Data from a Cached Database Source*.
- Dispose of a cache when it is finished using it. For information, see *Disposing of a Cache*.

Before You Begin

Before you begin coding, identify the database tables or views that the operator requires and create corresponding database sources for those tables or views. When you define the database source, be sure to enable one of the caching options. For more information about defining database sources and enabling caching, see *Combining Events with Data from a Database*.

Index Numbering of Database Sources and Their Associated Caches

The methods in a user-defined operator refer to database sources (and their associated caches) by index number. This index numbering is 0 based and reflects the order in which the database sources appear in the list of database sources that the query passes to the UDO. For example, if a user-defined operator has the following signature, it will refer to the `SalesRegions` database source as index 0 and the `CatalogItems` database source as index 1.



Validating the Given Database Sources

If your user-defined operator uses database sources, it must validate the database sources that a query passes to it as input. To do this, your operator must:

- Verify that the query passes in the correct number of database sources.
- Verify that the database sources match the expected schema.

Verifying the Number of Database Sources that a Query Passes In

To verify that your operator receives the correct number of database sources from a query, you specify the `numberOfInputCaches` property in the `@UserDefinedOperatorProperties` annotation. The following example shows the `@UserDefinedOperatorProperties` annotation from the **TopKAnalyzer example**. The TopKAnalyzer operator expects one database source as input, so it sets the `numberOfInputCaches` property to one.

```
...
@UserDefinedOperatorProperties(name = "TopKAnalyzer", numberOfInputStreams = 2, numberOfInputCaches = 1)
public class TopKAnalyzerUdo extends UserDefinedOperator {
...
```

If the `numberOfInputCaches` property is specified in an operator, Event Server automatically checks that a query passes in the required number of database sources when it deploys the query. If the query does not pass in the correct number of database sources, Event Server does not deploy the query. For more information about the `numberOfInputCaches` property, see ***Setting Properties in the UserDefinedOperatorProperties Annotation***.

Note: If your operator does not use the `numberOfInputCaches` property to indicate the number of database sources it requires, you must include code in the `getOutputStreamMetaData()` method to verify that a query passes in the expected number of database sources.

Verifying the Schema of the Database Sources that a Query Passes In

To ensure that your user-defined operator receives the correct database sources at run time, you include code in the `getOutputStreamMetaData()` method to validate the schemas of the databases sources that a query passes in. The code snippet below shows the validation code from the **Top-KAnalyzer example**. This example expects a database source with the following schema:

Schema for the Database Source used by the TopKAnalyzer Example

Attribute	Type	Description
SellerID	String	Salesperson for item.
Item	String	Name of item.

As shown below, the TopKAnalyzer defines the expected schema in EXPECTED_CACHE_META_DATA. Then, in the `getOutputStreamMetaData()` method, it uses the `MetaDataChecks.checkMetaDatasMatch()` method to verify that the database source from the query matches the schema specified in EXPECTED_CACHE_META_DATA. If the database source does not match EXPECTED_CACHE_META_DATA, the method throws a `UserDefinedOperatorException` and the query is not deployed. For more information about the `MetaDataChecks.checkMetaDataMatch()` method, see the Javadoc for the `MetaDataChecks` class.

```
...
 // expected cache structure with SellerID and Item
  private static final RecordMetaData EXPECTED_CACHE_META_DATA =
    createRecordMetaData(
     createFieldMetaData("SellerID", Type.STRING, ! IS_NULLABLE, IS_CASE_SENSITIVE),
      createFieldMetaData("Item", Type.STRING, IS_NULLABLE, IS_CASE_SENSITIVE)
    );

...
public static StreamMetaData getOutputStreamMetaData(StreamMetaData[] inputMetaData,
    RecordMetaData[] cacheMetaData, Object[] instantiationParameters
  ) throws UserDefinedOperatorException {
  MetaDataChecks.checkMetaDatasMatch(
    inputMetaData,
    EXPECTED_SHOPPING_ITEM_STREAM_METADATA,
    EXPECTED_TEMPERATURE_STREAM_METADATA
  );
  MetaDataChecks.checkMetaDatasMatch(
    cacheMetaData,
    EXPECTED_CACHE_META_DATA
  );

  return OUTPUT_STREAM_METADATA;
}
...
```

When you code your operator, its `getOutputStreamMetaData()` method must include similar logic to validate the database sources that its receives from a query.

Generating an Index for the Cached Table or View

Before coding your operator, you must decide whether to index the caches that it uses. Indexing a cache enables faster access to the records in the cache. When you generate an index, you specify the columns by which the records in the cache are to be indexed.

For performance reasons, you should always index the caches that your operator uses. The only reason you would not index a cache is if your operator always retrieves the entire table (all records in the cache), and not just selected records, when it queries the cache.

Important: When you index a cache, you can only retrieve records from the cache by key. For more information about accessing the records in a cache, see *Retrieving Data from a Cached Database Source*.

To index the caches that your operator uses, you must implement the getCacheKeyIndices( ) method. The CEP engine calls this method when it parses a query that uses your UDO. The method is called once for each database source that the query passes to your operator as input.

For each cache that you want to index, your getCacheKeyIndices( ) method must return an integer array that identifies the columns that serve as the key to the records in the cache. The key does not need to be unique.

An cache can have only one index.

The following shows the implementation of the getCacheKeyIndices( ) method in the **TopKAnalyzer** example. The example has one cache, which contains the table of items and their seller assignments (**schema here**). This method indexes the cache by the column that contains the item name.

```
...
public int[] getCacheKeyIndices(int index) {
 return new int [] {CACHE_ITEM_COLUMN_INDEX};
}
...
```

Note: The columns in a cached record are referenced by index numbers. Column numbering begins with 0. If you have a record containing `n` columns, the column indices will be 0...(`n`-1).

If you have multiple database sources, the getCacheKeyIndices( ) method must include logic for each cache that you want to index. For example, in the following implementation, the UDO uses three database sources, and thus has three caches, 0, 1, and 2. It builds indices for caches 0 and 1, but not 2.

```
...
 public int[] getCacheKeyIndices(int index) {
   if(index == 0) return new int[] {0};       // On Cache 0: index column 0
   if(index == 1) return new int[] {1, 2};    // On Cache 1: index columns 1 and 2
   return null;                               // On Cache 2: no index
 }
...
```

Note that the method checks the value of the `index` parameter that it receives from the CEP engine. The `index` parameter identifies the cache for which the method has been called. For more information about index numbering of database sources and their associated caches, see ***Index Numbering of Database Sources and Their Associated Caches***.

If you do not want to index any of the caches that your operator uses, the getCacheKeyIndices( ) method should simply return null. (This is the default implementation.)

Implementing the push( ) Method for Your Caches

A user-defined operator has two push( ) methods. One method takes a Record as input and processes events. The other method takes a ReadOnlyCache as input, and processes a new or updated cache. The Event Server calls the second form of this method when it creates the cache for the UDO. If the cache is periodic, the Event Server also calls this method each time the cache is refreshed.

When Event Server calls the push( ) method for a cache, it passes the following input parameters to your UDO:

Parameter	Description
index	The index number of the cache that Event Server is passing to your UDO. For information about index numbering of caches, see ***Index Numbering of Database Sources and Their Associated Caches***.
cache	The new or updated cache object.
startTimestamp	The time at which the cache became valid.
endTimestamp	The time until which the cache is valid.

Like events in an event stream, records in a cache have a validity interval. The validity interval for a record in a cache is determined by the *startTimestamp* and *endTimestamp* of the cache to which it belongs. When you have a periodic cache, Event Server pushes a new cache to your operator each time the cache is refreshed. The validity intervals for a given cache are always disjoint, so your operator will never receive two instances of a cache with overlapping validity intervals.

Receiving an update from a cache having startTimestamp *t* guarantees that your operator will never again receive an updated instance of this cache having startTimestamp less than *t*. Given this guarantee, your operator might be able to advance temporally and either generate output events or signal its progress with a heartbeat.

The following shows the push( ) method that the **TopKAnalyzer** example uses to process a new or updated cache. The example uses a one-time cache, so this method is only executed when the Event Server creates the cache. Note that the method begins with a set of assertions to determine whether it has received the expected cache. The method then stores the cache object that it receives from the Event Server and calls its temporalProgress method (a private method provided by the example program) to apply the cache to any events that the operator might have received prior to obtaining this cache from Event Server.

```
...
 public void push(int index, ReadOnlyCache cache, long startTimestamp, long ↩
endTimestamp) {
    assert index == CACHE_INPUT_INDEX : "Unexpected cache update from index " + index;
    assert startTimestamp == Long.MIN_VALUE : "Unexpected startTimestamp received: ↩
" + startTimestamp;
    assert endTimestamp == Long.MAX_VALUE : "Unexpected endTimestamp received: " + ↩
endTimestamp;
     assert this.oneTimeCache == null : "Cache has already been initialized!";

     this.oneTimeCache = cache;
     temporalProgress();
 }
...
```

Your operator must include similar logic to receive the caches that Event Server pushes to it. When coding the push method for a cache, keep the following points in mind:

- If your operator uses multiple caches, its push( ) method must include logic to save each cache separately.
- If your operator is snapshot reducible and uses a cache that is periodically refreshed, it needs to maintain multiple instances of that cache in memory so that it can correlate a given event with a record that is valid for the same point in time as the event.
- The method that your operator uses to ascertain temporal progress and generate output events should process new and updated caches in the same manner that it would process a new event arriving on an event stream. That is, the receipt of a new or updated cache should cause your operator to check whether the updated information enables it to publish output events or emit a heartbeat.

Important: When an operator is finished using a cache, it must execute the dispose( ) method on the cache. This method informs Event Server that the cache can be physically removed from the Terracotta cache. For more information about disposing of a cache, see ***Disposing of a Cache***.

Note: After the Event Server loads or refreshes a cache that your operator uses, it pushes a heartbeat from the cache to your operator. For more information about processing heartbeats from a cache, see ***Processing Heartbeats***.

Retrieving Data from a Cached Database Source

To obtain records from a cache, you use the getCachedElements( ) method to query the cache. If the cache is indexed, you must call the getCachedElements( ) method with a key. The method returns an Iterator of Records matching the key. If the cache is not indexed, you use the getCachedElements( ) without a key. In this case, the method returns an Iterator containing all of the Records in the cache.

The following snippet shows the code that the **TopKAnalyzer** example uses to retrieve database records from cache. In this example, the cache contains a table of salespeople and the items they sell (**schema here**). Because this cache is indexed, the operator uses the getCachedElements( ) method with a key (in this case the itemName) to retrieve records from the cache. (In this use case, the key is unique, so the method will return at most one record.)

```
...
  String sellerId = "unknownSellerForItem_"+itemName;
  // query the cache to get the seller for this specific item
  for(Record sellerRecord : oneTimeCache.getCachedElements(itemName)) {
    sellerId = sellerRecord.getString(CACHE_SELLER_ID_COLUMN_INDEX);
    break; // we assume a 1:1 relation between sellers and items!
  }
...
```

The following snippet shows how you would extract the record for a given item if the cache had no index. In this case, you would call the getCachedElements( ) method without a key. Then, you would loop through the Iterator that the method returns (which contains every record in the cache) and retrieve the record you need.

```
...
String sellerId = "unknownSellerForItem_"+itemName;
// query the cache to get the seller for this specific item
for(Record sellerRecord : oneTimeCache.getCachedElements()) {
if(sellerRecord.getString(CACHE_ITEM_COLUMN_INDEX).equals(itemName)) {
   sellerId = sellerRecord.getString(CACHE_SELLER_ID_COLUMN_INDEX);
   break; // we assume a 1:1 relation between sellers and items!
   }
}
...
```

For more information about the getCachedElements( ) method, see the Javadoc for the GeneralUserDefinedOperatorAdapter.ReadOnlyCache class.

Disposing of a Cache

When your operator is finished using a cache, it must dispose of the cache by calling the cache's dispose( ) method. The dispose method informs Event Server that the operator is finished using the cache and indicates that the cache can be deleted from the Terracotta cache.

Note: If the operator is running in a high-availability environment, the cache will not actually be deleted from the Terracotta Server Array until the operators on both Event Servers have disposed of the cache.

All operators should dispose of their caches as part of the clean-up process in the done( ) and close( ) methods.

When you test a query that uses your UDO in Software AG Designer, your UDO's done( ) method is called after your operator receives the last event from an input stream in an event sequence file. The done( ) method is called once for each event stream that the UDO uses. Disposing of a cache in the done( ) method ensures that your UDO's caches are properly disposed of when the operator is executed by a query in Software AG Designer.

The following shows the implementation of the done( ) method from the **TopKAnalyzer** example. Note that it calls the dispose( ) method on the oneTimeCache *before* it clears that cache from memory (by setting the cache to null). This is important, because once the cache object is gone, there is no other way for an operator to dispose of the cache.

```
...
  public void done(int index, InputType inputType) {
    temporalProgress();
    if(callback.getOperatorState().allInputsAreDone()) {
      callback.done();
      itemStreamBuffer = null;
      temperatureStreamBuffer = null;
      oneTimeCache.dispose(); // important!
      oneTimeCache = null;
    }
  }
... ↩
```

Disposing of a cache in the close( ) method ensures that your operator releases the cache when your operator is deregistered from Event Server. Deregistration occurs when a query that uses the UDO is undeployed or redeployed on Event Server, and when the Event Server is shutdown.

The following shows the implementations of the close( ) method from the **TopKAnalyzer** example. In this example, the close( ) method disposes of the cache if the cache has not already been disposed of. (If the operator is running in Software AG Designer, the cache might have already been disposed of by the done( ) method.)

```
...
 public void close (
    if (oneTimeCache != null)  {
       oneTimeCache.dispose();
       oneTimeCache = null;
    }
 }
... ↩
```

In addition to disposing of caches in the done( ) and close( ) methods, your operator's temporal progress logic should routinely dispose of any instance of cache that the operator no longer needs. This step is usually necessary in operators that use periodic caches. With this type of cache, an operator generally maintains multiple instances of the cache, each with different validity intervals, in memory. To prevent the caches from accumulating in memory unnecessarily, your operator should, as part of its temporal progress logic, monitor the validity intervals of its caches and dispose of any cache that it no longer needs.

Important: Be certain that your operator executes the dispose( ) method on an unneeded cache *before* it actually clears or replaces the cache. Once a cache has been cleared or replaced, your operator can no longer dispose of it, and the unused cache will physically remain in the Terracotta cache until the Event Server is restarted. (If the Event Server is running in a high-availability configuration, the cache will not be cleared until both servers are stopped and restarted. During the restart process, the server that is the master deletes the old caches from the Terracotta Server Array.)

For more information about the dispose( ) method, see the Javadoc for the GeneralUserDefined-OperatorAdapter.ReadOnlyCache class.

Advanced Implementation Topics

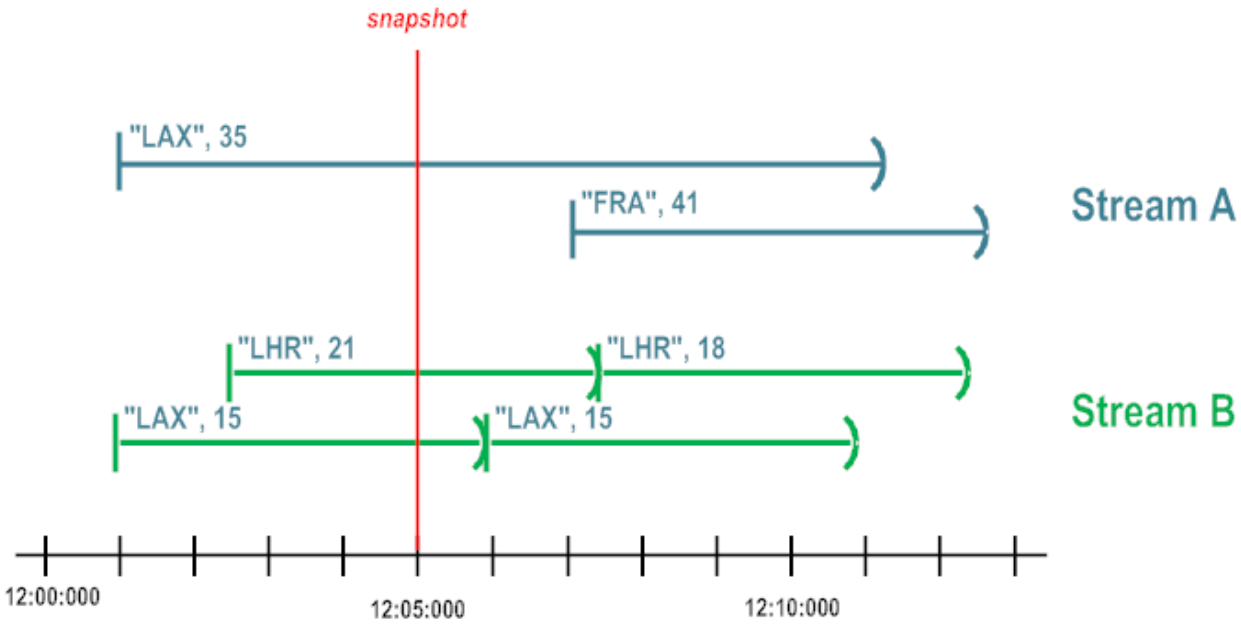
- Snapshot Reducibility
- Result Equivalency
- Equivalency in Input Streams
- Operators that are not Result Equivalent
- Implementing the Completeness Methods if Your Operator is not Snapshot Reducible
- Managing Temporal Progress

- Processing Heartbeats

Snapshot Reducibility

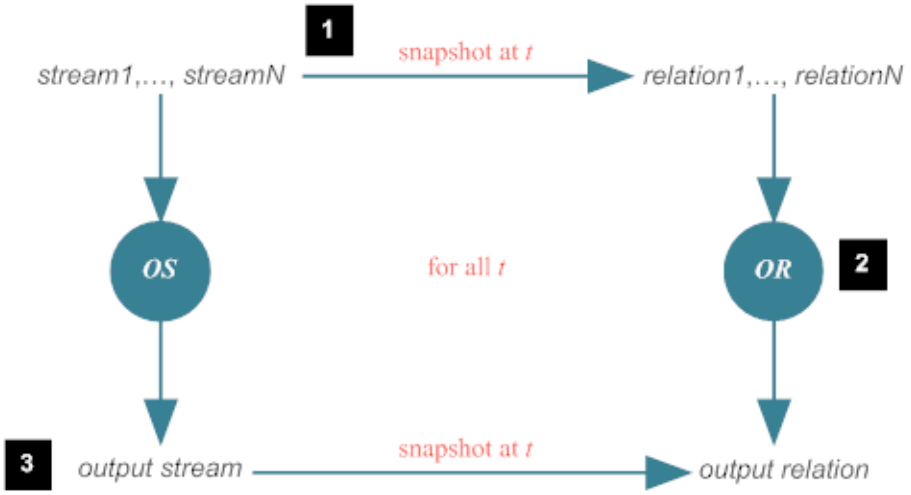
While you can write an operator to perform any kind of calculation you choose, in order to produce results that are determinate and suitable for using in a high-availability environment, an operator must be snapshot reducible or it must at least produce equivalent results for equivalent input streams.

A *snapshot* refers to the multiset of tuples that are valid at a given time instant. In other words, it is a relational representation of an operator's input streams and caches at a particular instant in time. In the following diagram, the red line represents the snapshot of input streams A and B at the instant in time 12:05:000. This snapshot includes one tuple from Stream A and two tuples from Stream B.



A user-defined operator with input streams `stream1`,..., `streamN` is considered to be *snapshot reducible* if, for any time instant `t`, the snapshot at `t` in the result stream that the operator produces is equal to the result that would be produced by applying a relational counterpart of the operator to a database containing the snapshots of `stream1`,..., `streamN` at time `t`.

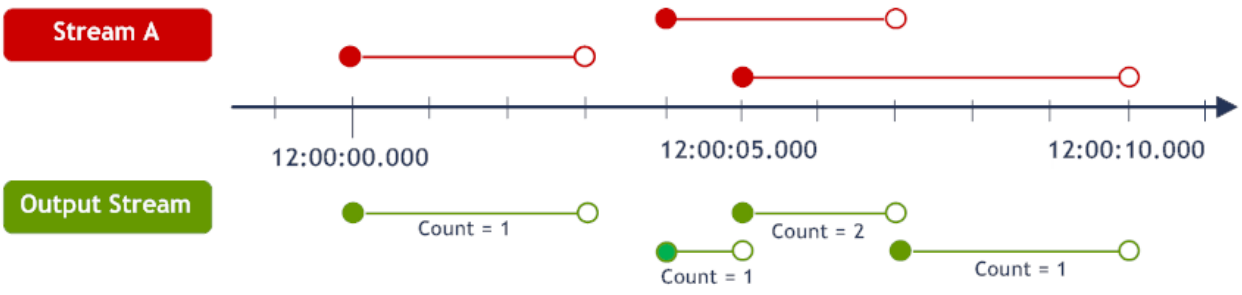
Imagine that you have a stream operator, `OS`, that is snapshot reducible to some database operator, `OR`, as shown below.



1. If you were to record all of the snapshots from the input streams for OS in a relational database,
2. And apply *OR* to any snapshot at time instant *t* in the database,
3. The result produced by *OR* would be the same as the snapshot at time instant *t* in the output stream produced by *OS*

When an operator is snapshot reducible, the information that the operator needs in order to produce a result is provided entirely by the tuples that are valid at the time of the snapshot. A snapshot-reducible operator does not depend on any data from events whose validity interval does not intersect the snapshot. Consequently, an operator that computes results across a series of events whose validity intervals do not intersect (for example, an operator that measures the number of events that occur within a ten-minute period), or monitors an event stream for changes from one event to another (for example, an operator that reports the change in temperature between temperature readings), is not snapshot reducible.

To ensure snapshot reducibility, the validity interval of the output events that an operator produces must reflect the intersection of the validity intervals of the input events from which the results were derived. For example, let's say that your operator reports a count of the events present in an input stream. Given Stream A, below, the operator would produce the four events shown in Output Stream.



The events in this output stream would look as follows:

Start Time	End Time	Count
12:00:00:000	12:00:03:000	1
12:00:04:000	12:00:05:000	1
12:00:05:000	12:00:07:000	2
12:00:07:000	12:00:10:000	1

Note that the start and end time for each output event specifies the complete duration of the time interval for which the reported count is valid. For example, the validity interval for Count = 2 starts at the first chronon where there are two events in the stream and ends at the chronon where the count is no longer two. If this operator did not specify the entire validity interval for the reported count (say that it specified only the start time for each change in count), it would no longer be snapshot reducible.

Operators that are snapshot reducible extend the relational algebra for database systems. They are based on sound semantics and produce results that are deterministic and repeatable. Due to these qualities, an operator that is snapshot reducible is fully compliant with a high-availability environment.

Note: The default implementations of the completeFrom( ) and maximumPossiblyAffected-Timestamp( ) methods are applicable to snapshot-reducible operators. If your operator is snapshot reducible, you do not need to change these methods. If your operator is not snapshot reducible, you must provide suitable implementations for these methods. For additional information, see ***Implementing the completeFrom( ) and maximumPossibleAffectedTimestamp( ) Methods***.

For additional information about snapshot reducibility, see the *Theoretical Foundation of Query Semantics*.

Result Equivalency

Some operators are not snapshot reducible. These include operators that calculate changes between events in an event stream and operators that modify the validity interval of events (e.g., operators that perform windowing or time shifting).

While snapshot-reducible operators produce repeatable results and are compatible with a high-availability environment, operators that are not snapshot reducible do not inherently possess this quality. You must build this quality into your operator by ensuring that it generates *equivalent results for equivalent input streams*.

Important: If an operator does not produce equivalent results for equivalent input streams, *it must never be used in a high-availability configuration*. In a high-availability system, the slave server continually compares the query results that it produces to the results produced by the master server. If the results are not equivalent, the slave detects an error condition and

tries to takeover the master role. If this is not possible, the slave shuts down. When your operator runs in a high-availability configuration, the instances running on the master and the slave will receive event streams that are equivalent, but not necessarily identical. If your operator does not produce equivalent results for equivalent event streams, the high-availability system will fail.

Equivalency in Input Streams

Two event streams do not need to be identical in order to be equivalent. As described below, streams can deliver events in different order or express an event using a different number of tuples, and still be equivalent. Your operator must expect these differences and generate the same (or equivalent) results when they occur.

Equivalent Order within an Event Stream

Events in an event stream are always arranged in non-decreasing order according to the events' start timestamp. However, the order of an event stream is weakly monotonic, meaning that it can contain multiple events with the same start time. When a stream includes a series of events with the same start timestamp, that series can appear in any order. For example, the input streams shown below are equivalent. The streams each contain the same events, but the order of events with the same starting timestamp varies in each case.

Start	Value
10:00:00	XXX
10:00:01	AAA
10:00:01	BBB
10:00:01	CCC
10:00:02	YYY

Start	Value
10:00:00	XXX
10:00:01	AAA
10:00:01	CCC
10:00:01	BBB
10:00:02	YYY

Start	Value
10:00:00	XXX
10:00:01	BBB
10:00:01	AAA
10:00:01	CCC
10:00:02	YYY

Start	Value
10:00:00	XXX
10:00:01	BBB
10:00:01	CCC
10:00:01	AAA
10:00:02	YYY

Start	Value
10:00:00	XXX
10:00:01	CCC
10:00:01	AAA
10:00:01	BBB
10:00:02	YYY

Start	Value
10:00:00	XXX
10:00:01	CCC
10:00:01	BBB
10:00:01	AAA
10:00:02	YYY

To ensure result equivalency, an operator must produce the same (or equivalent) results, regardless of the order in which it receives events having the same start time. In the example above, an operator would need to produce equivalent results for all six permutations of this stream.

Note: If your operator has multiple input streams, it must deliver equivalent results regardless of which stream delivers events to the operator first. For more information about delivering equivalent results when processing multiple event streams, see *Temporal Progress*.

Equivalent Expressions of a Single Event

An event has a value and a validity interval. In an input stream, an event can be represented by one event tuple or by multiple tuples with the same payload and consecutive time intervals. For example, the following are all equivalent expressions of event XXX at time interval 10:00:00 to 10:00:10.

Event XXX 10:00:00 - 10:00:10

Start	End	Value
10:00:00	10:00:10	XXX

Event XXX 10:00:00 - 10:00:10

Start	End	Value
10:00:00	10:00:05	XXX
10:00:05	10:00:10	XXX

Event XXX 10:00:00 - 10:00:10

Start	End	Value
10:00:00	10:00:03	XXX
10:00:03	10:00:07	XXX
10:00:07	10:00:10	XXX

To ensure result equivalency, an operator cannot assume that an event will be represented by a single event record. For example, if an operator receives event XXX with an end timestamp of 10:00:05, the operator cannot know whether it has all of event XXX until it receives an event with a start timestamp greater than 10:00:05 from the same input stream. Your operator must either wait for temporal progress beyond 10:00:05 or be able to produce equivalent results based on 'partial' events.

Operators that depend on knowing the complete duration of an event are expensive performance wise. To know whether an incoming event is new or a continuation of an event that has already been received, an operator must identify and coalesce partial events by comparing the timestamps and payloads of each incoming event with the events it currently has under evaluation. (This is one reason why pattern-matching operators are so expensive compared to other operators.) Such operators generally reduce the "liveliness" of a query as they must wait for all streams to progress beyond the input event's end timestamp before publishing a result. If input events have rather long validity intervals, this delay will significantly impede the pace at which an operator can emit results.

If you want to avoid the performance overhead and liveliness issues associated with coalescing incoming events, you must implement the operator such that it can produce equivalent results based on 'partial' input events.

Operators that are not Result Equivalent

You might occasionally have a use case that does not demand exact or repeatable results. Typically, these use cases involve applications where performance is more important than the exactness or repeatability of a result. For these applications, approximate results are "good enough."

For example, let's say you want to monitor an input stream for certain atypical events and publish the very next event that follows each of these out-of-the-ordinary occurrences. If the purpose of the operator is simply to produce a sample for statistical analysis, the application might not care whether the "next" event is one of many events with the same starting timestamp, or whether the validity interval of that event is incomplete. Simply emitting the "next" event satisfies the functional requirements of the application.

While this type of operator is legitimate in terms of its functionality, it will not produce equivalent results for equivalent input streams. For example, if you created an operator like the one described above, it could publish any of three possible results for the following equivalent streams.

Start	Value
10:00:00	XXX
10:00:01	AAA
10:00:01	BBB
10:00:01	CCC
10:00:02	YYY

Start	Value
10:00:00	XXX
10:00:01	AAA
10:00:01	CCC
10:00:01	BBB
10:00:02	YYY

Start	Value
10:00:00	XXX
10:00:01	BBB
10:00:01	CCC
10:00:01	AAA
10:00:02	YYY

Start	Value
10:00:00	XXX
10:00:01	BBB
10:00:01	AAA
10:00:01	CCC
10:00:02	YYY

Start	Value
10:00:00	XXX
10:00:01	CCC
10:00:01	AAA
10:00:01	BBB
10:00:02	YYY

Start	Value
10:00:00	XXX
10:00:01	CCC
10:00:01	BBB
10:00:01	AAA
10:00:02	YYY

Because the operator does not generate the same result for these equivalent streams, it cannot be used in a high-availability environment.

Implementing the Completeness Methods if Your Operator is not Snapshot Reducible

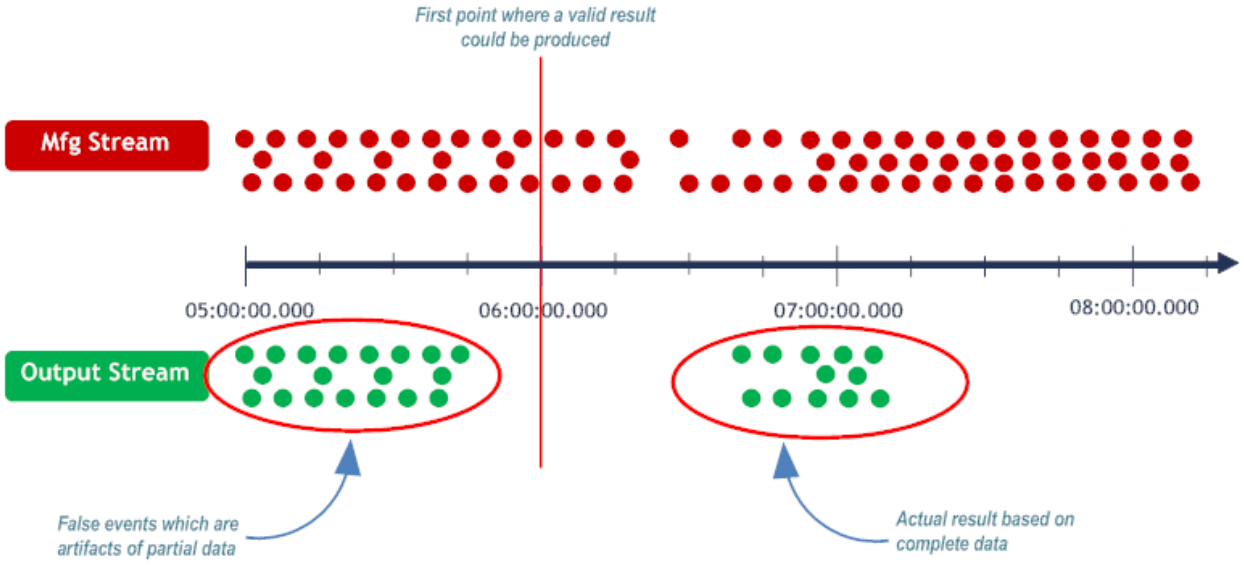
When Event Server deploys a query containing your operator, it executes the operator's `completeFrom()` and `maximumPossiblyAffectedTimestamp()` methods. These methods identify the point in time from which the results produced by the operator are valid. This information is used to ensure that the operator does not provide partial or invalid results to downstream operators. It is also used in a high-availability environment to ensure that the slave does not include any partial or invalid results in the comparisons that it performs with the master server.

The completeness methods are required to eliminate incorrect results that an operator produces when it begins receiving events from a stream. Because an operator does not have data for events that occurred prior to the first event it receives, it might not produce a valid result until it reaches a certain point in the event stream. The completeness methods identify the point where an operator's results become reliable.

The default implementations provided for the `completeFrom()` and `maximumPossiblyAffectedTimestamp()` methods can be used for any snapshot-reducible operator. If your operator is not snapshot reducible, you must provide suitable implementations for these methods.

Implementing the completeFrom( ) method

The `completeFrom()` method identifies the point in time when an operator will begin to produce reliable results. For example, let's say your operator counts the number of outputs from a manufacturing process over a one-hour sliding window and generates an output event when the count falls below 20. When the operator is deployed, it will begin receiving events. However, it will not be able to produce valid results until it has received at least an hour's worth of events from the stream. During the first hour, the operator will produce numerous false events, because of the artificially low counts that result from not having any event data for the previous hour.



To indicate that the first hour of output events are invalid, this operator uses the `completeFrom()` method to specify that its output stream is valid from *`startOfStream`* + 1 hour.

When Event Server calls the operator's `completeFrom()` method, it passes in an array containing the complete-from times for each of the operator's input streams and caches. These times are supplied by the `completeFrom()` methods of the upstream operators that provide the input streams and caches that the operator uses. The time specified in each index of the array represents the "complete-from" time for the corresponding input stream or database source. Given these times, the operator must return its own complete-from time. This value is then passed to the `completeFrom()` methods in the downstream operators that use the operator's output stream.

Note: The Event Server calls your operator's `completeFrom()` method after it has obtained the complete-from times for all of the operator's input streams and caches. This might occur immediately when the query starts, at some point in time after the query starts, or possibly never (e.g., if an event never arrives on one of the operator's input streams).

The following snippet shows an implementation of the `completeFrom()` method for the operator with the sliding one-hour window described above:

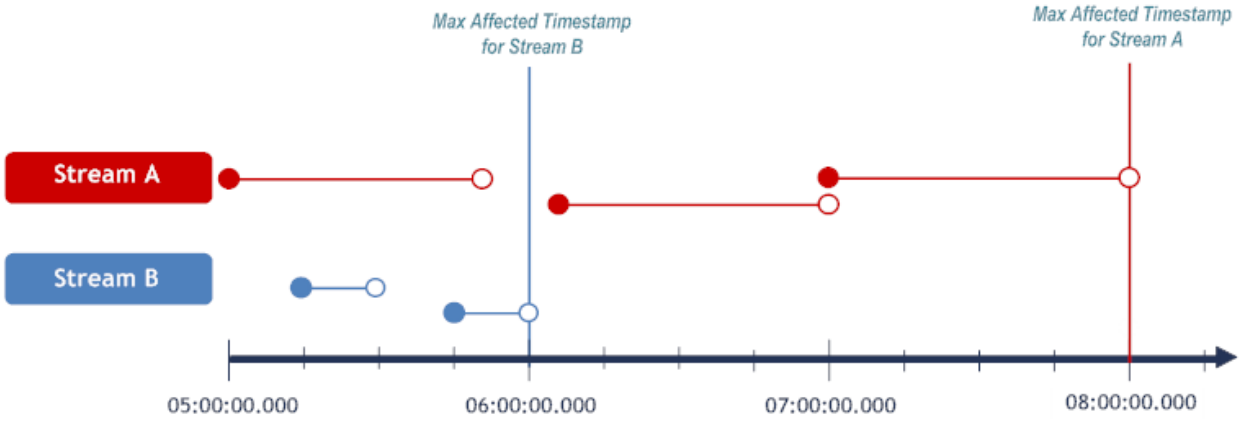
```
...
 public long completeFrom(Long[] inputsCompleteFrom) {
    Long outputValidFrom = inputsCompleteFrom[0] + (60 * 60 * 1000);
    return outputValidFrom;
 }
...
```

Important: There are operators for which you cannot precisely identify the point in time when its output stream will become valid. For example, if an operator produces a result after it detects a specified number of events, there is no way to determine when exactly the first result will occur. If an operator cannot accurately identify the point from which it is complete, it must not be used in a high-availability environment. The master and slave will not be able to successfully synchronize the output stream produced by the operator, and the high-availability system will fail. (Often, this kind of problem is an indication that you should consider an alternative design for your operator. For example, you might consider applying windows to the operator's input streams to make it snapshot reducible.)

For additional information about the `completeFrom()` method, see the Javadoc for the GeneralUser-DefinedOperatorAdapter interface.

Implementing the maximumPossiblyAffectedTimestamp( ) method

The *maximum affected timestamp* refers to the largest timestamp that is present in the validity interval of any event in an event stream. The following figure depicts the maximum affected timestamp for Stream A (08:00:00) and Stream B (06:00:00).



The `maximumPossiblyAffectedTimestamp()` method requires an operator to identify the maximum affected timestamp that it could produce given the maximum affected timestamps for its input streams. Event Server calls this method when it starts the operator.

When Event Server calls the `maximumPossiblyAffectedTimestamp()` method, it passes in an array containing the maximum affected timestamp for each of the operator's input streams and caches. These timestamps are supplied by the `maximumPossiblyAffectedTimestamp()` methods of the upstream operators that provide input streams and caches to the operator. The timestamp given in each index represents the maximum affected timestamp for the corresponding input stream or database source. Given these timestamps, an operator must return the largest possible timestamp that it could have produce in its output stream if it received events withe the affected timestamps.

For example. say you have a snapshot-reducible operator that correlates events across Stream A and Stream B, and the operator receives the maximum affected timestamps shown above (i.e., 08:00:00 for Stream A and 06:00:00 for Stream B). The largest possible timestamp that this operator could produce is the smaller of these two values (i.e., 06:00:00 in this case), because it is the latest point in time to which both streams have progressed.

An operator with different semantics might return a different maximum affected time stamp. For example, if the operator performs a union of Streams A and Stream B, its `maximumPossiblyAffectedTimestamp()` method would return the larger of the two streams' maximum affected timestamps (i.e., 08:00:00 given the streams shown above).

Note that unlike the `completeFrom()` method, the input array passed to the `maximumPossiblyAffectedTimestamp()` method can include nulls (meaning that the corresponding input stream or cache does not yet have any associated elements). The operator's `maximumPossiblyAffectedTimestamp()` method can also return null. For example, an operator

that receives an input array containing only null values would usually return null for `maximumPossiblyAffectedTimestamp()`.

For additional information about the `maximumPossiblyAffectedTimestamp()` method, see the Javadoc for the GeneralUserDefinedOperatorAdapter interface.

Managing Temporal Progress

If your operator correlates events across two or more input streams, it must synchronize the streams temporally and correlate events whose validity intervals intersect. For example, the TopKAnalyzer operator correlates a stream of purchased items with a stream of temperature readings. It then publishes the top-selling items for the validity interval associated with each temperature reading. The events in both streams are aligned on six-hour tumbling windows, which preserves snapshot reducibility and simplifies temporal management. (To understand how these windows have been applied to these streams, install the **TopKAnalyzer example** and examine the queries associated with the W_ShoppingItem and W_ActualWeather input streams. For information about installing the TopKAnalyzer example, see the user-defined extensions demo in *Getting Started with Complex Event Processing*.)

To calculate a result, the TopKAnalyzer monitors the two streams to determine whether it has received a temperature reading and one or more purchased items within the same six-hour interval.

Managing temporal progress generally involves the following high-level steps:

- Storing incoming events in memory.
- Evaluating the stored events to ascertain whether a result (or a heartbeat) can be published.
- Identifying events that are no longer needed and clearing them from memory.

Storing Incoming Events

Unlike the operators in a relational database system, which can re-read records at any time, continuous query operators have only one opportunity to see a particular record (i.e., event) in an input stream. If an operator needs to refer to information about an event after the event has been received, it must store that information in memory.

The most common way for an operator to retain event data for future reference is to establish an input buffer for each input stream. As shown in the following snippet, the TopKAnalyzer establishes a buffer for the Temperature stream and a buffer for the Items stream. Each buffer is a Queue of StreamElements. A StreamElement is a data-transfer object that enables an operator to easily extract the timestamps or the payload from an event. For more information about StreamElements, see the Javadoc for the Adapters.StreamElement class.

```
...
private Queue<StreamElement> temperatureStreamBuffer;
private Queue<StreamElement> itemStreamBuffer;
...
```

When Event Server pushes an event to the TopKAnalyzer operator, the operator's push( ) method receives the incoming event and stores it in the appropriate buffer.

```
...
public void push(int index, Record record, long startTimestamp, long endTimestamp) {
  assert index == ITEM_STREAM_INPUT_INDEX || index == TEMPERATURE_STREAM_INPUT_INDEX ↩
: "Unexpected index " + index;

  StreamElement element = Adapters.createStreamElement(record, startTimestamp, ↩
endTimestamp);
  if(index == TEMPERATURE_STREAM_INPUT_INDEX) {
      temperatureStreamBuffer.offer(element);
  }
  else {
    itemStreamBuffer.offer(element);
  }
  temporalProgress();
}
...
```

Note: It is not necessary to retain all of the data associated with an incoming event. In general, an operator should keep the event's start and end timestamps and any portion of the payload that it needs in order to compute a potential result.

Evaluating Events and Publishing Results

When you work with multiple input streams, it is important to remember that, *by default*, the events are not globally ordered (temporally) across the streams. Event Server delivers events to the operator as they arrive in the streams. For example, if you have two streams, A and B, your operator might receive the following events from stream A:

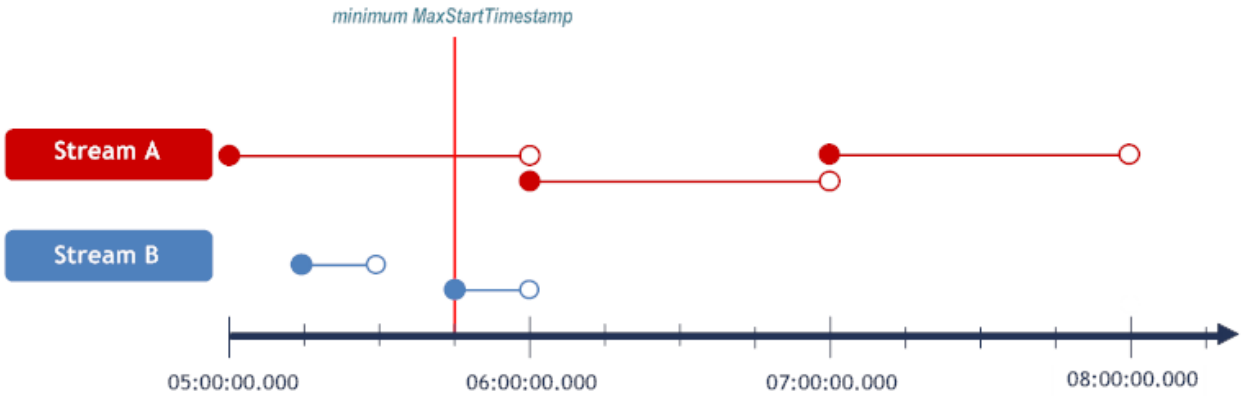
```
Stream_A, 2011-06-17T 05:00:00.000Z, 2011-06-17T 06:00:00.000Z, …
Stream_A, 2011-06-17T 06:00:00.000Z, 2011-07-17T 06:00:00.000Z, …
Stream_A, 2011-06-17T 07:00:00.000Z, 2011-08-17T 06:00:00.000Z, …
```

And later receive the following events from stream B:

```
Stream_B, 2011-06-17T 05:30:00.000Z, 2011-06-17T 05:45:00.000Z, …
Stream_B, 2011-06-17T 05:45:00.000Z, 2011-06-17T 06:00:00.000Z, …
```

So, while stream A has progressed to the 07:00 instant in time, stream B has only progressed to 05:45. The two streams advance independently.

The largest starting timestamp that has been received in an event stream is referred to as the stream's *maxStartTimestamp*. The smallest maxStartTimestamp across all streams is referred to as the *minimum maxStartTimestamp*. The minimum maxStartTimestamp represents the point in time to which all of the streams have mutually progressed.



Given the minimum maxStartTimestamp, an operator knows that it will never receive an event, from any of its input sources (i.e., streams or caches), with a starting timestamp that is less than the value of the minimum maxStartTimestamp. This guarantee assures an operator that is has all of the input for the time period prior to the minimum maxStartTimestamp. Given this assurance, an operator can evaluate events in its buffers with starting timestamps that are less than or equal to the minimum maxStartTimestamp and publish results as appropriate.

The temporalProgress( ) Method from the TopKAnalyzer Operator

The following code snippet shows the method that the TopKAnalyzer uses to manage the temporal progress of its input streams. The TopKAnalyzer calls this method each time it receives an event, a heartbeat, or a cache update.

```
...
private void temporalProgress() {
```

1
```
// retrieve the most recent timestamp that we received an event or cache ↩
update upon on each input
    long minMaxStartTimestamp = ↩
callback.getOperatorState().getMinMaxStartTimeStamp();
```

2
```
// compute and publish the elements of the output stream up to timestamp ↩
minMaxTimestamp
    while(!temperatureStreamBuffer.isEmpty() && ↩
temperatureStreamBuffer.peek().getEndTimestamp() <= minMaxStartTimestamp) {
     StreamElement temperatureElement = temperatureStreamBuffer.poll();  ↩

      double temperature = ↩
temperatureElement.getRecord().getDouble(STREAM_TEMPERATURE_COLUMN_INDEX);
```

3
```
// remove all items from the item buffer that do not intersect the current ↩
temperatureElement
      while(!itemStreamBuffer.isEmpty() && ↩
itemStreamBuffer.peek().getEndTimestamp() < ↩
temperatureElement.getStartTimestamp())
       itemStreamBuffer.remove();
```

4
```
Map<String, Integer> totalQuantitiesBySeller = new HashMap<String, ↩
Integer>();
      // determine all items that temporarily intersect the ↩
temperature-element's timestamp,
     // get the corresponding sellerId for that item from the cache, and ↩
aggregate the sold quantities
      // of this seller
      while(!itemStreamBuffer.isEmpty()
        && itemStreamBuffer.peek().getStartTimestamp() < ↩
temperatureElement.getEndTimestamp()
        && itemStreamBuffer.peek().getEndTimestamp() > ↩
temperatureElement.getStartTimestamp()) {
        StreamElement itemElement = itemStreamBuffer.poll();
        assert itemElement.getStartTimestamp() >= ↩
temperatureElement.getStartTimestamp(); ↩
```

5
```
String itemName = ↩
itemElement.getRecord().getString(STREAM_ITEM_COLUMN_INDEX);
        Integer quantity = ↩
itemElement.getRecord().getInteger(STREAM_QUANTITY_COLUMN_INDEX);

        String sellerId = "unknownSellerForItem_"+itemName;
        // query the cache to get the seller for this specific item
       for(Record sellerRecord : oneTimeCache.getCachedElements(itemName)) {
          sellerId = sellerRecord.getString(CACHE_SELLER_ID_COLUMN_INDEX);
          break; // we assume a 1:1 relation between sellers and items!
        }
        // aggregate
        Integer totalQuantity = quantity + ↩
(totalQuantitiesBySeller.containsKey(sellerId) ? ↩
```

```
totalQuantitiesBySeller.get(sellerId) : 0);
        totalQuantitiesBySeller.put(sellerId, totalQuantity);
      }
      // sort the set of sellers based on sold quantities
      List<Entry<String, Integer>> sellersSortedByQuantity = ↩
sortDescendingByValue(totalQuantitiesBySeller);
      // publish an element for each of the top-k sellers
      int rank = 1;
      for(Entry<String, Integer> topSeller : sellersSortedByQuantity) {
        callback.push(
         Adapters.createRecord(topSeller.getKey(), topSeller.getValue(), ↩
rank, temperature),
          temperatureElement.getStartTimestamp(),
          temperatureElement.getEndTimestamp()
        );
        if(++rank > topK)
          break;
      }
    }
```

```
}
...
```

The following provides additional information about the code snippet shown above.

Segment	Description
1	To begin, the method uses the `getMinMaxStartTimestamp( )` method to get the minimum MaxStartTimestamp for the combined streams and caches. (If the operator has not yet received an event from a particular stream, `getMinMaxStartTimestamp( )` returns `Long.MIN_VALUE )`. For more information about the `getMinMaxStartTimestamp( )` method, see the Java doc for the GeneralUserDefinedOperatorAdapter.OperatorState interface.
2	The operator checks to see whether the end timestamp of the oldest event in the temperature buffer is smaller than the minMaxStartTimestamp value. When this condition is true, it indicates that all input streams have progressed beyond the duration of the given temperature event, and the operator now has all of the information it needs to produce results for the time interval associated with that temperature event.
3	The operator first eliminates all items that are older than the given temperature event (i.e., it removes all item events whose validity intervals end before the temperature event begins).
4	The operator dequeues each item whose validity interval intersects with the temperature event, retrieves the seller ID for the item from cache, and stores the seller ID and associated running quantity in a hash map. (For more information about how the operator retrieves information from cache, see ***Retrieving Data from a Cached Database Source***.) **Important:** An operator should only retain those elements that could potentially contribute to the computation of a future result. All other elements should be cleared from its input buffers using logic similar to that shown in segments 3 and 4 in **The temporalProgress( ) Method from the TopKAnalyzer Operator**.

Segment	Description
5	After all item events whose validity interval intersects the temperature event have been processed, the operator sorts the hash map by item quantity (to place the items in top-selling order) and uses the callback object's push( ) method to publish an output event for each of the top four entries in the list. The operator uses the Adapaters.createRecord method to produce an event record containing the following information: ■ Seller ID ■ Total quantity sold during interval ■ Seller's rank for the interval ■ Temperature during the interval Note that the operator gives the output event the same start and end timestamps as the temperature event. It can do this because the events in both streams have identical validity intervals.

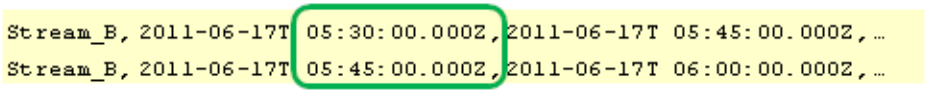
Managing Temporal Progress When Streams are Globally Ordered

You can optionally use the `requiresGloballyOrderedInputs()` method to enable global ordering for your operator. When you enable global ordering, your operator automatically receives events in temporal order across streams. To deliver events in global order, Event Server deliberately withholds events from the operator until it has events with the same or greater timestamps from all of the operator's input streams. For example, let's say your operator takes Stream A and Stream B as input, and Event Serverreceives the following events for these streams:

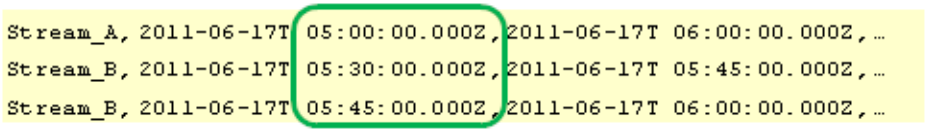
Stream A

```
Stream_A, 2011-06-17T 05:00:00.000Z, 2011-06-17T 06:00:00.000Z, …
Stream_A, 2011-06-17T 06:00:00.000Z, 2011-07-17T 06:00:00.000Z, …
Stream_A, 2011-06-17T 07:00:00.000Z, 2011-08-17T 06:00:00.000Z, …
```

Stream B



Event Server will not deliver the 05:00 event from Stream A until it receives the 05:30 event from Stream B. Similarly, Event Server will not deliver the 06:00 event from Stream A until it receives an event from Stream B with a start timestamp that is greater than or equal to 06:00. Given the state of the streams above, Event Server would deliver the following events to the operator:



Receiving events in global order simplifies the work an operator must do to manage temporal progress of its event streams. An operator can simply process the events that it has in memory knowing that it will not receive any additional elements with starting timestamps smaller than the events it has already received.

Although global ordering of events can simplify the management of temporal progress, it can reduce the "liveliness" of an operator. Liveliness refers to the latency between an event streaming in and the complex events derived from that event streaming out.

When events are globally ordered, your operator might not always have the most recent events for a given stream. For some operators, not having immediate access to events may not matter, since they cannot actually operate on those events until the events from the other streams arrive. However, other operators can benefit from receiving events as soon as they arrive.

Producing a Valid Output Stream

Your operator is responsible for delivering its results in valid temporal order, meaning that it must deliver events in *non-descending order by start timestamp.*

The TopKAnalyzer operator, for example, produces zero or more output events for each temperature event that it receives. The validity interval (i.e., the start and end timestamp) for each output event corresponds to the temperature event for which the output was computed. The TopKAnalyzer does not need to take any special measures to ensure the temporal order of its output streams, because its output stream implicitly follows the order of the Temperature stream (which is already in order by start timestamp).

Not all operators will have an output stream that is implicitly ordered like the one that the TopKAnalyzer produces. In such cases, the operator should establish an output queue, write output events to that queue, and order the queue by start timestamp. When the operator advances tem-

porally and is guaranteed to not produce any result having start timestamp less than t, it can push out any events from the output queue having a start timestamp less than t.

Clearing Outdated Events from the Input Buffers

The logic that your operator uses to manage the temporal progress of events should include steps to clear events from the buffer when they are no longer needed.

In general, an operator can clear any event whose end timestamp is smaller than the start timestamp of the event that is currently under evaluation. The TopKAnalyzer operator does this in segment 3 of **The temporalProgress( ) Method from the TopKAnalyzer Operator**.

Processing Heartbeats

An operator can receive heartbeats from its input streams and its caches. A heartbeat has a timestamp, but no payload. A heartbeat is metadata that indicates the progress of time for a given event stream or cache. It is considered a form of "punctuation" for an event stream.

The receipt of a heartbeat indicates to your operator that it has received all events for a given event stream or all updates from a given cache for the period of time prior to the heartbeat. For example, if an operator receives a heartbeat with the timestamp Oct 31, 2011 13:00 from Stream A, it is guaranteed not to receive any additional events on Stream A for any instant in time prior to Oct 31, 2011 13:00. Given this guarantee, the operator can advance the temporal progress of Stream A and potentially publish output events.

Heartbeats are passed to an operator by the operator's `heartBeat()` method. An operator must provide an implementation of this method to process the heartbeats that it receives from its input streams and caches. When the `heartBeat()` method in your operator is called, it receives the following input parameters:

Parameter	Description
`index`	The index number of the event stream or cache that emitted the heartbeat. For information about index numbering of input streams and caches, see ***Index Numbering of Input Streams*** and ***Index Numbering of Database Sources and Their Associated Caches***.
`inputType`	A flag indicating whether the heartbeat was emitted by an event stream or a cache.
`timestamp`	The heartbeat's timestamp.

Processing the Heartbeats from an Event Stream or a Cache

If your operator involves a simple mapping-type operation on a single input stream, wherein each event in the output stream is derived from exactly one event from the input stream and has exactly the same validity interval as the input event from which it was derived, the `heartBeat()` method can simply push incoming heartbeats to the next operator as shown in the implementation below.

```
public void heartBeat(int index, InputType inputType, long timestamp) {
    callback.heartBeat(timestamp);
  }
```

The **ItemListSplitter UDO** uses this implementation of the `heartBeat()` method.

If your operator has multiple input streams or uses periodic caches, you use heartbeats to update the operator's temporal progress. The following shows the `heartBeat()` method from the **TopKAnalyzer UDO**. In this implementation, the `heartBeat()` method calls the operator's `temporalProgress( )` method each time a heartbeat is received. The `temporalProgress( )` method is the private method that the operator uses to manage the temporal progress of its event streams and publish output events.

```
...
 public void heartBeat(int index, InputType inputType, long timestamp) {
   temporalProgress();
   callback.heartBeat(callback.getOperatorState().getMinMaxStartTimeStamp());
 }
...
```

Note that after the `temporalProgress( )` method executes, this implementation of the `heartBeat()` method emits a heartbeat whose timestamp reflects the current minimum maxStartTimestamp. The minimum maxStartTimestamp represents the point in time to which all of the operator's streams have mutually progressed. By emitting this heartbeat, the operator signals that is has published all possible results up to the minimum maxStartTimestamp, and guarantees that it will never again publish a result whose start timestamp is less than the timestamp in this heartbeat. For more information about the minimum maxStartTimestamp, see ***Managing Temporal Progress***.

As a general rule, an operator should emit a heartbeat for time instance `t` when it can guarantee that it will produce no further results with a timestamp less than `t`. In its temporal progress method, for example, an operator should check whether it can compute results each time the minimum maxStartTimestamp advances. After it computes all possible results up to the minimum maxStartTimestamp, the operator should emit a heartbeat with a timestamp equal to that minimum maxStartTimestamp.

Note: Due of the particular characteristics of the **TopKAnalyzer operator** and its input streams, the TopKAnalyzer inherently produces a result each time it receives an event that advances the minimum maxStartTimestamp. Therefore, the `temporalProgress( )` method in the TopKAnalyzer does not explicitly issue a heartbeat each time the minimum maxStart-Timestamp advances.

Emitting heartbeats enables downstream operators that use your operator's output stream to advance their temporal progress and deliver results more quickly. As a developer, you do not need to worry about emitting too many heartbeats. Event Server automatically suppresses duplicate heartbeats, heartbeats that have the same timestamp as an event that has already been published, and any heartbeat that occurs within the minimum heartbeat interval of another heartbeat or output event. By default, the minimum heartbeat interval is one millisecond, but it is configurable per query.

UDO Sample Code Listings

The UDO_ItemListSplitter Example (Basic)

The following is the complete listing for the ItemListSplitter user-defined operator. For details about this example, see ***Sample Code Used in this Documentation***.

```
package com.softwareag.demo.udo;

import static de.rtm.push.adapters.Adapters.IS_CASE_SENSITIVE;
import static de.rtm.push.adapters.Adapters.IS_NULLABLE;
import static de.rtm.push.adapters.Adapters.createFieldMetaData;
import static de.rtm.push.adapters.Adapters.createStreamMetaData;

import com.softwareag.wep.resource.udo.UserDefinedOperator;
import com.softwareag.wep.resource.udo.UserDefinedOperatorProperties;

import de.rtm.push.adapters.Adapters;
import de.rtm.push.adapters.MetaDataChecks;
import de.rtm.push.adapters.StreamMetaData;
import de.rtm.util.exception.UserDefinedOperatorException;
import de.rtm.util.records.data.Record;
import de.rtm.util.records.metadata.FieldMetaData;
import de.rtm.util.records.metadata.RecordMetaData;
import de.rtm.util.records.types.JavaTypes.Type;

/**
 * This example separates one event <BR>
 * buyer's shopping item list (as a string "pear, 6, apples, 10")  <BR>
 * into 2 events containing  BuyerId, Item and Quantity
 */
@UserDefinedOperatorProperties(name = "ItemListSplitter", numberOfInputStreams = ↩
1, numberOfInputCaches = 0)
public class ItemListSplitter extends UserDefinedOperator {
  private static final String ITEMFIELDDESCRIPTION = "Item";
  private static final String QUANTITYFIELDDESCRIPTION = "Quantity";
```

```
  private static final String DELIMITER = ",";
  private static final int SHOPPINGCARTINDEX = 1;
  private static final int BUYERFIELDINDEX = 0;

  private OperatorCallBack callback;

  /**
   * Check the passed method parameters <BR><UL>
   * <LI>check the expected number of incoming streams and database caches</LI>
   * <LI>Check schema and/or field names in expected form </LI>
   * <LI>define the output structure</LI>  </UL> <BR>
   * starting with the decision whether the result of the operator is chronon stream, ↩
<BR>
   * followed by the field structure.<BR><UL>
   * <LI>BuyerId is taken from the 1. field of the 1. input stream</LI>
   * <LI>Item as String, not nullable and case sensitive </LI>
   * <LI>Quantity as Integer, nullable and not case sensitive</LI>  </UL>
   */
  public static StreamMetaData getOutputStreamMetaData(
      StreamMetaData[] inputMetaData, RecordMetaData[] cacheMetaData,
      Object[] instanceParameters) throws UserDefinedOperatorException {

 // There is no need to check for the correct number of input stream metadata and ↩
cache metadata,
 // since we defined these numbers in the annotation above and the system ↩
automatically checks for
 // the correct number. Only if we ommitted the annotation attributes, we had to ↩
check for the
 // correct number here.

    MetaDataChecks.checkCorrectNumberOfColumns(inputMetaData[0], 2);
    MetaDataChecks.checkCorrectJavaType(inputMetaData[0], SHOPPINGCARTINDEX, ↩
Type.STRING);

    // define output structure
    boolean isChrononStream = inputMetaData[0].isChrononStream();
    FieldMetaData buyerId = MetaDataChecks.getColumnMetaData(inputMetaData[0], ↩
BUYERFIELDINDEX);
    FieldMetaData item = createFieldMetaData(ITEMFIELDDESCRIPTION, Type.STRING, ↩
!IS_NULLABLE, IS_CASE_SENSITIVE);
    FieldMetaData quantity = createFieldMetaData(QUANTITYFIELDDESCRIPTION, ↩
Type.INTEGER, IS_NULLABLE, IS_CASE_SENSITIVE);
    return createStreamMetaData(isChrononStream, buyerId, item, quantity);
  }

  /**
   * Returns whether this operator requires a global order of all input streams.
   * @return true, if this operator requires a global order of all input streams, ↩
false otherwise
   */
  public boolean requiresGloballyOrderedInputs() {
    return DOES_NOT_REQUIRE_GLOBALLY_ORDERED_INPUTS;
```

```
  }

  /**
   * no special coding needed in that example
   *
   * @see de.rtm.push.adapters.GeneralUserDefinedOperatorAdapter#init()
   */
  @Override
  public void init() {
  }

  /**
   * save the callback Operator for later use
   *
   * @see
   * de.rtm.push.adapters.UserDefinedOperatorAdapter#open(de.rtm.push.adapters
   * .UserDefinedOperatorAdapter.OperatorCallBack)
   */
  @Override
  public void open(OperatorCallBack callback) {
    this.callback = callback;
  }

  /**
   * the incoming event stream is processed in separating the SHOPPINGCART
   * item list by DELIMITER. The separated Item and Quantity pair are
   * published as event stream.<BR>
   * This method is called permanently, therefore Type checks should be handled in
   * getOutputStreamMetaData
   *
   * @see de.rtm.push.adapters.UserDefinedOperatorAdapter#push(int,
   * de.rtm.util.records.data.Record, long, long)
   */
  public void push(int index, Record element, long startTimeStamp, long endTimeStamp) ↩
{

    // pear, 6, apple, 10 .... separated
    String itemlist = (String) element.getObject(SHOPPINGCARTINDEX);
    String[] split = itemlist.split(DELIMITER);
    /**
     * iterate over the found item quantity pairs <BR>
     * fill the corresponding Item and Quantity field <BR>
     * and finally push each record as operator result
     */
    Object buyerId = element.getObject(BUYERFIELDINDEX);
    for (int i = 0; i < split.length; i = i + 2) {
      Object item = split[i];
      Integer quantity = Integer.valueOf(split[i + 1]);
      Record record = Adapters.createRecord(buyerId, item, quantity);
      callback.push(record, startTimeStamp, endTimeStamp);
    }
  }
```

```
/**
 * no usage of database cache access in this example
 *
 * @see de.rtm.push.adapters.UserDefinedOperatorAdapter#push(int,
 * de.rtm.push.adapters.GeneralUserDefinedOperatorAdapter.ReadOnlyCache,
 * long, long)
 */
@Override
public void push(int arg0, ReadOnlyCache arg1, long arg2, long arg3) {
}

/**
 * no special handling for heartBeats
 *
 * (non-Javadoc)
 * @see de.rtm.push.adapters.GeneralUserDefinedOperatorAdapter#heartBeat(int,
 *  de.rtm.push.adapters.GeneralUserDefinedOperatorAdapter.InputType, long)
 */
@Override
public void heartBeat(int index, InputType inputType, long timestamp) {
  callback.heartBeat(timestamp);
}

/**
 * inform callback that the operator is done
 *
 * (non-Javadoc)
 * @see de.rtm.push.adapters.GeneralUserDefinedOperatorAdapter#done(int,
 *  de.rtm.push.adapters.GeneralUserDefinedOperatorAdapter.InputType)
 */
@Override
public void done(int index, InputType inputType) {
  callback.done();
}

/**
 * no special coding needed in that example
 *
 * @see de.rtm.push.adapters.GeneralUserDefinedOperatorAdapter#close()
 */
@Override
public void close() {
}

/**
 * no database cache
 *
 * @see
 * de.rtm.push.adapters.GeneralUserDefinedOperatorAdapter#getCacheKeyIndices
 * (int)
 */
```

```
  @Override
  public int[] getCacheKeyIndices(int arg0) {
    return null;
  }

  /**
   * (non-Javadoc)
   *
   * @see
   * de.rtm.push.adapters.GeneralUserDefinedOperatorAdapter#completeFrom(java
   * .lang.Long[])
   */
  @Override
  public long completeFrom(Long[] inputsCompleteFrom) {
    long max = Long.MIN_VALUE;
    for (int i = 0; i < inputsCompleteFrom.length; i++)
      if (inputsCompleteFrom[i] != null)
        max = Math.max(max, inputsCompleteFrom[i]);
    return max;
  }

  /**
   * (non-Javadoc)
   *
   * @see de.rtm.push.adapters.GeneralUserDefinedOperatorAdapter#
   * maximumPossiblyAffectedTimestamp(java.lang.Long[])
   */
  @Override
  public Long maximumPossiblyAffectedTimestamp(Long[] inputTimestamps) {
    Long max = null;
    for (Long value : inputTimestamps)
      if (value != null)
        if (max == null)
          max = value;
        else
          max = Math.max(max, value);
    return max;
  }
}
```

The UDO_TopKAnalyzer Example (Advanced)

The following is the complete listing for the TopKAnalyzer user-defined operator. For more information about this example, see *Example: User-Defined Extensions* in *Getting Started with Complex Event Processing*.

```
package com.softwareag.demo.udo;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Queue;

import com.softwareag.wep.resource.udo.UserDefinedOperator;
import com.softwareag.wep.resource.udo.UserDefinedOperatorProperties;

import de.rtm.push.adapters.Adapters;
import de.rtm.push.adapters.Adapters.StreamElement;
import de.rtm.push.adapters.MetaDataChecks;
import de.rtm.push.adapters.StreamMetaData;
import de.rtm.util.exception.UserDefinedOperatorException;
import de.rtm.util.records.data.Record;
import de.rtm.util.records.metadata.RecordMetaData;
import de.rtm.util.records.types.JavaTypes.Type;

import static de.rtm.push.adapters.Adapters.IS_CASE_SENSITIVE;
import static de.rtm.push.adapters.Adapters.IS_NULLABLE;
import static de.rtm.push.adapters.Adapters.IS_CHRONON_STREAM;
import static de.rtm.push.adapters.Adapters.createStreamMetaData;
import static de.rtm.push.adapters.Adapters.createRecordMetaData;
import static de.rtm.push.adapters.Adapters.createFieldMetaData;

/**
 * The purpose of this example is to demonstrate the different possibilities ↩
available with User Defined Operators (UDO).
 * <B> It is strongly recommended to use EQL operators when ever possible. This UDO ↩
example could more easy expressed with EQL operators,
 *  but for demonstrations purpose, it is implemented as UDO</B>
 * <P> This UDO example demonstrate the use of 2 event streams, 1 database cache ↩
access and passing 1 initial parameters. <BR>
 * <UL>
 * <LI>The item from the first UDO example and an ActualWeather stream are the two ↩
Event streams. <BR> </LI>
 * <LI>SalesAssignment is the database cache combining seller and item<BR> </LI>
 * <LI>The parameter K is used for the top k seller</LI>
 * </UL>
 * In the example each item is related to a seller.
 * The item quantities for a seller are summarized in the window of two temperature ↩
events
 * When the temperature rises or drops the top K seller are listed
 */
@UserDefinedOperatorProperties(name = "TopKAnalyzer", numberOfInputStreams = 2, ↩
numberOfInputCaches = 1)
```

```
public class TopKAnalyzer extends UserDefinedOperator {

  private static final int ITEM_STREAM_INPUT_INDEX = 0;
  private static final int TEMPERATURE_STREAM_INPUT_INDEX = 1;
  private static final int CACHE_INPUT_INDEX = 0;

  //expected stream structure from QO1_ShoppingItem
  private static final StreamMetaData EXPECTED_SHOPPING_ITEM_STREAM_METADATA =
    createStreamMetaData(
      ! IS_CHRONON_STREAM,
     createFieldMetaData("BuyerID", Type.INTEGER, ! IS_NULLABLE, IS_CASE_SENSITIVE),
      createFieldMetaData("Item", Type.STRING, ! IS_NULLABLE, IS_CASE_SENSITIVE),
     createFieldMetaData("Quantity", Type.INTEGER, IS_NULLABLE, IS_CASE_SENSITIVE)
    );

  //expected stream structure from ActualWeatherStream with Temperature and Humidity
  private static final StreamMetaData EXPECTED_TEMPERATURE_STREAM_METADATA =
    createStreamMetaData(
      ! IS_CHRONON_STREAM,
      createFieldMetaData("Temperature", Type.DOUBLE, ! IS_NULLABLE, ↩
IS_CASE_SENSITIVE),
     createFieldMetaData("Humidity", Type.INTEGER, ! IS_NULLABLE, IS_CASE_SENSITIVE)
    );

  // expected cache structure with SellerID and Item
  private static final RecordMetaData EXPECTED_CACHE_META_DATA =
    createRecordMetaData(
     createFieldMetaData("SellerID", Type.STRING, ! IS_NULLABLE, IS_CASE_SENSITIVE),
      createFieldMetaData("Item", Type.STRING, IS_NULLABLE, IS_CASE_SENSITIVE)
    );

  // output structure of UDO with SellerID, Amount (Aggregated item quantity),
  // Rank, Temperature (last measured) and Temperature change
  private static final StreamMetaData OUTPUT_STREAM_METADATA = ↩
Adapters.createStreamMetaData (
    ! IS_CHRONON_STREAM,
    createFieldMetaData("SellerId", Type.STRING, IS_NULLABLE, IS_CASE_SENSITIVE),
    createFieldMetaData("Amount", Type.INTEGER, IS_NULLABLE, ! IS_CASE_SENSITIVE),
    createFieldMetaData("Rank", Type.INTEGER, IS_NULLABLE, ! IS_CASE_SENSITIVE),
   createFieldMetaData("Temperature", Type.DOUBLE, IS_NULLABLE, ! IS_CASE_SENSITIVE) ↩

  );

  private static final int STREAM_ITEM_COLUMN_INDEX =
    MetaDataChecks.getColumnIndex(EXPECTED_SHOPPING_ITEM_STREAM_METADATA, "Item");
  private static final int STREAM_QUANTITY_COLUMN_INDEX =
   MetaDataChecks.getColumnIndex(EXPECTED_SHOPPING_ITEM_STREAM_METADATA, "Quantity");
  private static final int STREAM_TEMPERATURE_COLUMN_INDEX =
    MetaDataChecks.getColumnIndex(EXPECTED_TEMPERATURE_STREAM_METADATA, ↩
"Temperature");
  private static final int CACHE_SELLER_ID_COLUMN_INDEX =
    MetaDataChecks.getColumnIndex(EXPECTED_CACHE_META_DATA, "SellerID");
```

```
  private static final int CACHE_ITEM_COLUMN_INDEX =
    MetaDataChecks.getColumnIndex(EXPECTED_CACHE_META_DATA, "Item");

  /**
   * The purpose of this static method is twofold:
   *  <ol>
   *    <li>Check, whether the given metadata, that is the number of input streams ↩
and caches,
   *    their respective schemas and the given instantiation parameters are suitable ↩
for
   *     user-defined operator instances of this class. Throw an {@link ↩
UserDefinedOperatorException},
   *     if this is not the case.</li>
   *     <li> Given the metadata, compute and return the metadata of the output ↩
stream produced by
   *     an instance of this class.</li>
   *  </ol>
   *
   * @param inputMetaData For each input stream this array has an entry describing ↩
the characteristics
   *                      of the stream and the schema of the stream elements.
   * @param cacheMetaData For each cache this array has an entry describing the ↩
schema of the tuples
   *                      in this cache.
   * @param instantiationParameters Additional user-defined parameters needed for ↩
instantiating an
   *                      an object of this class.
   * @return A {@link StreamMetaData} object describing the characteristics of the ↩
output-stream
   *                    produced by this operator and defining the schema of this ↩
output-streams elements.
   * @throws UserDefinedOperatorException If - with the given parameters - it is ↩
impossible to instantiate
   *                      an operator of this class
   */
  public static StreamMetaData getOutputStreamMetaData(StreamMetaData[] ↩
inputMetaData,
      RecordMetaData[] cacheMetaData, Object[] instantiationParameters
    ) throws UserDefinedOperatorException {
    MetaDataChecks.checkMetaDatasMatch(
      inputMetaData,
      EXPECTED_SHOPPING_ITEM_STREAM_METADATA,
      EXPECTED_TEMPERATURE_STREAM_METADATA
    );
    MetaDataChecks.checkMetaDatasMatch(
      cacheMetaData,
      EXPECTED_CACHE_META_DATA
    );

    return OUTPUT_STREAM_METADATA;
  }
```

```
  private OperatorCallBack callback;
  private Integer topK;

  private Queue<StreamElement> temperatureStreamBuffer;
  private Queue<StreamElement> itemStreamBuffer;
  private ReadOnlyCache oneTimeCache;

  /**
   * The one and only constructor used to instantiate this user-defined
   * operator with the passed parameter
   *
   * @param topK  how many top seller should be listed
   */
  public TopKAnalyzer(Integer topK) {
    this.topK = topK;
  }

  /**
   * Returns whether this operator requires a global order of all input streams.
   * @return true, if this operator requires a global order of all input streams, ↩
false otherwise
   */
  public boolean requiresGloballyOrderedInputs() {
    return DOES_NOT_REQUIRE_GLOBALLY_ORDERED_INPUTS;
  }

  /**
   * Initializes this operator and acquires any resources needed for its
   * computations. This method is called prior to {@link #open(OperatorCallBack)}.
   * Prepare buffering input streams and max values for calculation of temporal ↩
progress
   */
  public void init() {
    temperatureStreamBuffer = new LinkedList<StreamElement>();
    itemStreamBuffer = new LinkedList<StreamElement>();
  }

  /**
   * Opens this operator, i.e., the operator starts to process incoming elements. ↩
Using the given
   * <code>callback</code> the operator can publish its output elements and signal ↩
its processing
   * state to its sinks.
   *
   * @param callback A callback providing the operator methods to publish its output ↩
elements and
   *        signal its processing state to its sinks.
   */
  public void open(OperatorCallBack callback) {
    this.callback = callback;
  }
```

```
  /**
   * Processes the incoming element from input <code>index</code>. The element is ↩
a record
   * whose schema is defined by {@link getInputStreamMetaData()}[index] and whose ↩
temporal
   * validity is defined by the interval <code>[startTimestamp, endTimestamp)</code>. ↩
An element
   * from input <code>index</code> having <code>startTimestamp</code> <it>t</it> ↩
guarantees that the operator will
   * never again receive an element (or a heartbeat) from this input having a ↩
<code>startTimestamp</code>
   * less than <it>t</it>. <br />
   * Given this guarantee, the operator itself might advance temporally and either ↩
publishes new
   * output elements using {@link OperatorCallBack#push(Record, long, long)} or ↩
signal its
   * temporal progress by means of heartbeats in its output-stream using {@link ↩
OperatorCallBack#heartBeat(long)}
   * (with the heartbeats timestamp being the greatest timestamp that is lower than ↩
any elements'
   * or heartbeats' timestamp computed in the future).
   *
   * @param index The index of the input stream that sent the given element to this ↩
operator (0 ≤ <code>index</code> < {@link #getNumberOfStreamInputs()}).
   * @param element The element to be processed by this operator.
   * @param startTimestamp The start timestamp of the elements validity interval.
   * @param endTimestamp The end timestamp of the elements validity interval.
   */
  public void push(int index, Record record, long startTimestamp, long endTimestamp) ↩
{
    assert index == ITEM_STREAM_INPUT_INDEX || index == ↩
TEMPERATURE_STREAM_INPUT_INDEX : "Unexpected index " + index;

    StreamElement element = Adapters.createStreamElement(record, startTimestamp, ↩
endTimestamp);
    if(index == TEMPERATURE_STREAM_INPUT_INDEX) {
        temperatureStreamBuffer.offer(element);
    }
    else {
      itemStreamBuffer.offer(element);
    }
    temporalProgress();
  }

  private void temporalProgress() {
    // retrieve the most recent timestamp that we received an event or cache update ↩
upon on each input
   long minMaxStartTimestamp = callback.getOperatorState().getMinMaxStartTimeStamp();

    // compute and publish the elements of the output stream up to timestamp ↩
minMaxTimestamp
    while(!temperatureStreamBuffer.isEmpty() && ↩
```

```
temperatureStreamBuffer.peek().getEndTimestamp() <= minMaxStartTimestamp) {
      StreamElement temperatureElement = temperatureStreamBuffer.poll();
      double temperature = ↩
temperatureElement.getRecord().getDouble(STREAM_TEMPERATURE_COLUMN_INDEX);

      // remove all items from the item buffer that do not intersect the current ↩
temperatureElement
     while(!itemStreamBuffer.isEmpty() && itemStreamBuffer.peek().getEndTimestamp() ↩
< temperatureElement.getStartTimestamp())
       itemStreamBuffer.remove();

     Map<String, Integer> totalQuantitiesBySeller = new HashMap<String, Integer>();
      // determine all items that temporarily intersect the temperature-element's ↩
timestamp,
     // get the corresponding sellerId for that item from the cache, and aggregate ↩
the sold quantities
      // of this seller
      while(!itemStreamBuffer.isEmpty()
        && itemStreamBuffer.peek().getStartTimestamp() < ↩
temperatureElement.getEndTimestamp()
        && itemStreamBuffer.peek().getEndTimestamp() > ↩
temperatureElement.getStartTimestamp()) {
        StreamElement itemElement = itemStreamBuffer.poll();
        assert itemElement.getStartTimestamp() >= ↩
temperatureElement.getStartTimestamp();

       String itemName = itemElement.getRecord().getString(STREAM_ITEM_COLUMN_INDEX);
        Integer quantity = ↩
itemElement.getRecord().getInteger(STREAM_QUANTITY_COLUMN_INDEX);

        String sellerId = "unknownSellerForItem_"+itemName;
        // query the cache to get the seller for this specific item
        for(Record sellerRecord : oneTimeCache.getCachedElements(itemName)) {
          sellerId = sellerRecord.getString(CACHE_SELLER_ID_COLUMN_INDEX);
          break; // we assume a 1:1 relation between sellers and items!
        }

        // aggregate
        Integer totalQuantity = quantity + ↩
(totalQuantitiesBySeller.containsKey(sellerId) ? ↩
totalQuantitiesBySeller.get(sellerId) : 0);
        totalQuantitiesBySeller.put(sellerId, totalQuantity);
      }

      // sort the set of sellers based on sold quantities
      List<Entry<String, Integer>> sellersSortedByQuantity = ↩
sortDescendingByValue(totalQuantitiesBySeller);
      // publish an element for each of the top-k sellers
      int rank = 1;
      for(Entry<String, Integer> topSeller : sellersSortedByQuantity) {
        callback.push(
          Adapters.createRecord(topSeller.getKey(), topSeller.getValue(), rank, ↩
```

```
temperature),
          temperatureElement.getStartTimestamp(),
          temperatureElement.getEndTimestamp()
        );
        if(++rank > topK)
          break;
      }
    }
  }

  // utility method to sort the entries of a map in descending order of their values
  private static <K,V extends Comparable<V>> List<Entry<K, V>> ↩
sortDescendingByValue(Map<K,V> map) {
    List<Entry<K,V>> list = new ArrayList<Entry<K,V>>(map.entrySet());
    Collections.sort(list, new Comparator<Entry<K,V>>() {
      public int compare(Entry<K,V> e1, Entry<K,V> e2) {
        return - e1.getValue().compareTo(e2.getValue());
      }
    });
    return list;
  }

  /**
   * Processes the incoming update from cache <code>index</code>. The update ↩
comprises a reference
   * to a cache and a validity interval.
   * All the records in the cache referenced by <code>cache</code> share the same ↩
schema as specified
   * by {@link getCacheMetaData()}<code>[index]</code> and have a temporal validity ↩
as defined by the interval
   * <code>[startTimestamp, endTimestamp)</code>. An update from cache ↩
<code>index</code> having
   * <code>startTimestamp</code> <it>t</it> guarantees that the operator will never ↩
again receive an update
   * (or a heartbeat) from this cache having a <code>startTimestamp</code> less ↩
than <it>t</it>. <br />
   * Given this guarantee, the operator itself might advance temporally and either ↩
publish new
   * output elements using {@link OperatorCallBack#push(Record, long, long)} or ↩
signal its
   * temporal progress by means of heartbeats in its output-stream using {@link ↩
OperatorCallBack#heartBeat(long)}
   * (with the heartbeats' timestamp being the greatest timestamp that is lower ↩
than any elements
   * or heartbeats timestamp computed in the future).
   *
   * @param index The index of the cache that sent the given element to this operator ↩
(0 ≤ <code>index</code> < {@link UserDefinedOperator#getNumberOfCacheInputs()}).
   * @param cache The element to be processed by this operator.
   * @param startTimestamp The start timestamp of the elements validity interval.
   * @param endTimestamp The end timestamp of the elements validity interval.
   */
```

```
  public void push(int index, ReadOnlyCache cache, long startTimestamp, long ↩
endTimestamp) {
    assert index == CACHE_INPUT_INDEX : "Unexpected cache update from index " + index;
    assert startTimestamp == Long.MIN_VALUE : "Unexpected startTimestamp received: ↩
" + startTimestamp;
    assert endTimestamp == Long.MAX_VALUE : "Unexpected endTimestamp received: " + ↩
endTimestamp;
     assert this.oneTimeCache == null : "Cache has already been initialized!";

     this.oneTimeCache = cache;
     temporalProgress();
  }

  /**
   * Signals the operator, that the input specified by <code>index</code> is done ↩
and does not
   * deliver any further elements. If all inputs are done and the operator can not ↩
publish any
   * further elements to its output stream, the operator should announce its own ↩
state using
   * <code>callback.done()</code>.
   *
   * @param index The index of the input that is done, ranging from 0 to {@link ↩
UserDefinedOperator#getNumberOfStreamInputs()}
   *  or 0 to {@link UserDefinedOperator#getNumberOfCacheInputs()} depending on the ↩
<code>inputType</code>
   * @param inputType a flag indicating whether the signal was sent by a stream ↩
input or a cache input
   */
  public void done(int index, InputType inputType) {
    temporalProgress();
    if(callback.getOperatorState().allInputsAreDone()) {
      callback.done();
      itemStreamBuffer = null;
      temperatureStreamBuffer = null;
      oneTimeCache.dispose(); // important!
      oneTimeCache = null;
    }
  }

  /**
   * Closes the operator and releases any resources used.
   */
  public void close() {
    if(oneTimeCache != null) {
      oneTimeCache.dispose();
      oneTimeCache = null;
    }
  }

  /**
   * send a heartbeat, if the operator could advance temorarily in reaction to a
```

```
   * heartbeat sent by one of its inputs
   *
   * Any input - be it an input stream or a cache input - can notify this operator ↩
of its temporal
   * progress by means of an heartbeat. A heartbeat from input <code>index</code> ↩
guarantees
   * that the operator will never again receive a stream element (or a cache update) ↩
from this
   * input having a start-timestamp less than the timestamp given by that heartbeat. ↩
<br />
   * Given this guarantee, the operator itself might advance temporally and either ↩
publishes new
   * output elements using {@link OperatorCallBack#push(Record, long, long)} or ↩
signal its
   * temporal progress by means of heartbeats in its output-stream using {@link ↩
OperatorCallBack#heartBeat(long)}
   * (with the heartbeats' timestamp being the greatest timestamp that is lower ↩
than any elements
   * or heartbeats timestamp computed in the future).
   *
   * @param index The index of the input that sent this heartbeat, ranging from 0 ↩
to {@link UserDefinedOperator#getNumberOfStreamInputs()}
   *  or 0 to {@link UserDefinedOperator#getNumberOfCacheInputs()} depending on the ↩
<code>inputType</code>
   * @param inputType a flag indicating whether the signal was sent by a stream ↩
input or a cache input
   * @param timestamp The timestamp of the heartbeat.
   */
  public void heartBeat(int index, InputType inputType, long timestamp) {
    temporalProgress();
    callback.heartBeat(callback.getOperatorState().getMinMaxStartTimeStamp());
  }

  /**
   * For each cache this method defines the key columns of the cached records that ↩
are used to
   * create an index for faster look-up. Assume the following example for ↩
clarification:<br />
   * An operator uses three caches with the schemas
   *  <ul><li><code>C0(id: integer, name:String)</code>,</li>
   *     <li><code>C1(id:integer, department:String, location:String)</code> and</li>
   *      <li><code>C2(value:integer)</code>.</li>
   *  </ul>
   * In the processing phase the operator needs to look-up records in <code>C0</code> ↩
by
   * <code>id</code>, records in <code>C1</code> by <code>department</code> and
   * <code>location</code> and always accesses all records in <code>C2</code> (no ↩
key required). <br />
   * In order to build the indexes accordingly, this method would have to be ↩
implemented like that:
   * <pre>
   * public int[] getCacheKeyIndices(int index) {
```

```
   *   if(index == 0) return new int[] {0};     // C0: index of column id
   *   if(index == 1) return new int[] {1, 2};  // C1: indices of columns department ↩
and location
   *   return null;                 // C2: no index
   * }
   * </pre>
   * in this example the index of the item column is returned <br />
   * <b>Note:</b>Having an index on a cache available, querying the cache always ↩
requires a key to be specified (see {@link ReadOnlyCache#getCachedElements(Object[])}.
   *
   * @param index The index of the cache whose key columns shall be defined.
   * @return An array containing all column indices of the key columns used to build ↩
an index.
   */
  public int[] getCacheKeyIndices(int index) {
    return new int [] {CACHE_ITEM_COLUMN_INDEX};
  }

  /**
   * This method computes the timestamp from which on this operator may produce ↩
snapshot reducible results
   * as the maximum of all timestamps from the inputs.
   * Note that this method should only regard the information from which timestamp ↩
on the inputs are complete.
   *
   * @param inputsCompleteFrom the timestamps from which on the inputs are complete
   * @return the timestamp from which on this operator can produce snapshot reducible ↩
results
   */
  public long completeFrom(Long[] inputsCompleteFrom) {
    long max = Long.MIN_VALUE;
    for (int i = 0; i < inputsCompleteFrom.length; i++)
      if (inputsCompleteFrom[i] != null)
        max = Math.max(max, inputsCompleteFrom[i]);
    return max;
  }

  /**
   * Given the affected timestamps for all inputs (or null if no timestamp is ↩
affected yet),
   * this method computes which is the maximum possibly affected timestamp for the ↩
output as the maximum
   * of all given timestamps.
   *
   * @param inputTimestamps the maximum affected timestamps for all inputs
   * @return the maximum possibly affected timestamp for the output
   */
  public Long maximumPossiblyAffectedTimestamp(Long[] inputTimestamps) {
    Long max = null;
    for (Long value : inputTimestamps)
      if (value != null)
        if (max == null)
```

```
          max = value;
        else
          max = Math.max(max, value);
    return max;
  }
}
```

11 Using Variable XML Schema Components

Overview

The event query language (EQL) used by the Event Server requires that all events in a given stream have the same tuple structure: the number of data fields, the data type of each data field and the order in which the data fields appear are the same in each event. This is consistent with the SQL approach, but does not allow the following variable structures known from XML Schema Definition (abbreviated as XSD), as defined by the W3C at **http://www.w3.org/TR/xmlschema-ref/**:

xs:any
A wildcard allows an element with an arbitrary name and data type to be present.

xs:anyAttribute
An attribute wildcard allows one or more attributes of any data type to be present.

Untyped elements
Allows an element of any data type (text content, attributes or child nodes) to be present.

Cardinality
Occurrences greater than one (maxOccurs > 1) of elements or enclosing groups (sequence, choice or all).

Recursive elements
Allows an element to be used recursively.

If an input stream is associated with an event type schema which contains any of the variable structures listed above, instances of the element cannot be processed directly by the continuous query application. Nevertheless, you can use the mechanisms described below to process these variable schema elements in the Event Server.

Processing XSD structures in the Event Server

Since the continuous query application expects input and output events to be flat structures with no variable parts, such variable structures in the input stream must be converted to flat structures. This conversion takes place automatically, according to the event type definition for the input stream.

Similarly a conversion from a flat structure to a variable structure takes place automatically in the output stream, according to the event type definition for the output stream.

Note that there are only partial validation steps for the XML fragment created or inserted at the input or output side, respectively.

The conversion takes place for any element in the event type definition that contains one or more of the following structures:

- The XSD element "xs:any"
- The XSD element "xs:anyAttribute"
- Untyped elements
- Recursive elements

Internally, a flat structure resulting from this conversion is represented as a document object model (DOM) using the JDOM API (org.jdom.Document). JDOM is a Java library for processing XML structures. For general information about JDOM, see the external site **http://www.jdom.org/**.

To derive or extract a value of any datatype from an incoming DOM-valued field, you can implement a user defined extension (for example, a user defined function).

Similarly, to convert from a flat structure to a DOM which is mapped to an XML fragment directed to the output stream, you can implement a user defined extension (for example, a user defined function) to generate the variable portion of the event.

Variable Structures in Events in the Input Stream

Variable structures in the input stream can be presented to the continuous query application in an XML document that complies with the schema structure given in the stream's event type.

Assume for example this schema snippet defining elements E0 (a floating point number), E1 that contains variable components (xs:any, xs:anyAttribute), and E2 (a string):

```
<xs:element name="E0" type="xs:float" />
<xs:element name="E1">
    <xs:ComplexType>
        <xs:sequence>
            <xs:any  ... />
        </xs:sequence>
        <xs:attribute name="..." />
        <xs:anyAttribute ... />
    </xs:ComplexType>
</xs:element>
<xs:element name="E2" type="xs:string" />
```

The continuous query application would expect to receive events with just three input fields E0, E1 and E2, using data types supported by continuous queries. E0 and E2 are standard data types (xs:float and xs:string). However, the definition of E1 is a variable structure (it contains xs:any and xs:anyAttribute), so in the event type definition for the input stream, you omit the data type, since the Event Server will convert the element E1 to a DOM structure.

For example, consider an event that conforms to the above schema and contains the following data:

```
<root xmlns:p = "myURI">
    <E0>3.14159</E0>
    <E1 A1="..." A2="...">
        <X>ABCDEFG</X>
        <Y>Hello</Y>
        <p:Z>1234.5678</p:Z>
    </E1>
    <E2>Hello World</E2>
</root>
```

This event is presented to the continuous query application as a tuple consisting of the following 3 data fields:

- The field E0, containing the floating point number 3.14159.
- The field E1, passed as the DOM representation of the following well-formed XML fragment:

```
'<E1 xmlns:p = "myURI" A1="..." A2="..."> <X>ABCDEFG</X> <Y>Hello</Y> ↩
<p:Z>1234.5678</p:Z> </E1>'
```

- The field E2, containing the string 'Hello World'

In the continuous query application, the element E1 arrives as a DOM. You can use a user defined extension in a continuous query application to extract or derive simple-typed values from the XML structure and make it available to be queried in a SELECT statement.

The product delivery contains a sample UDF `UDF_XPATH` that allows you to access any part of the DOM structure using an XPath expression. See the section **General-Purpose User-Defined Extensions for handling XML Fragments** below for details.

Variable Structures in Events in the Output Stream

Similarly, the query in a continuous query application can send an XML structure to the query output stream. The XML structure must correspond to a variable structure in the output schema. The XML structure in the SELECT statement must be provided as a DOM.

To create the DOM, you can use the UDF `UDF_CreateJDOM`, which is available as part of the demo project ProcessMonitoringDemo. See the section **General-Purpose User-Defined Extensions for handling XML Fragments** below for details.

General-Purpose User-Defined Extensions for handling XML Fragments

The Process Monitoring demo contains general purpose user defined extensions for handling XML fragments (for example, the UDFs `UDF_XPATH` and `UDF_CreateJDOM` mentioned above), so you might want to use these UDFs as a starting point for any UDF that you wish to develop for your own needs.

The demo "Process Monitoring Example" uses predefined event types to define the structure of events in the input and output streams, including event types with variable XML schema structures. For example, the schema for the input stream used in the Process Monitoring Example is *ProcessStepInstanceChange*.

If you want to view the predefined event types in the Software AG Designer, follow the instructions in the section **Working with the Event Type Store**.

For further information about how to access and run the product's demo applications, refer to the document "*Getting Started with Complex Event Processing*".

- User defined function "UDF_XPATH"
- User defined function "UDF_CreateJDOM"

User defined function "UDF_XPATH"

The "Process Monitoring Example" demo contains a set of queries, some of which use UDFs. For example, the query *TransformAggregateTop.ceq* uses the user defined function UDF_XPATH in the SELECT clause.

UDF_XPATH takes two input parameters, namely:

- An XPath statement that addresses an end node in an XML structure.
- An object that represents the XML structure to be parsed.

It returns the value of the required end node of the XML structure. The Java code of the UDF is located in the folder *ProcessMonitoringDemo/src* in the package *com.softwareag.demo.udf*.

```
@UserDefinedFunctionProperties(name = "XPATH")
  public static String xpath (String xPath, org.jdom.Document doc)
                throws JDOMException, FileNotFoundException
  {
   XPath  jdomXPath = getCompiledQueryJDOM(xPath);
                String result = jdomXPath.valueOf(doc);
                return result;
  }
```

Example:

A query statement that would select the <Y> element inside the <E1> element could be:

```
SELECT * FROM MyInputStream WHERE UDF_XPATH ('/E1/Y', E1) = 'Hello';
```

The result returned by the UDF is then the string value of element <Y>. Hence, this would select all events in which the value 'Hello' is contained in the element Y in the element E1. In this example, the user defined function UDF_XPATH takes two input parameters, namely the XPath statement (/E1/Y) identifying the end node whose value is to be compared, and the element to be searched (E1).

Points to note:

- For general information about user defined functions, refer to the section ***Developing User-Defined Functions***.
- The input XML structure is processed as a JDOM 1.1 object. In JDOM, the root object is an object of type org.jdom.Document.
- JDOM 1.1 provides a class `org.jdom.xpath.XPath` with a method `valueOf()` that allows you to return the value of a DOM structure's end node that is addressed by an XPath statement.
- The XPath statement needs to take namespace definitions in the input XML structure into account. For example, if you specify a search pattern "Customer/Name", this will work if the input structure is `<Customer><Name>...` but not if the input structure is `<Customer xmlns="..."><Name>...`. Here you would need to use an XPath statement "x:Customer/x:Name", and the Java code would need to be adapted as follows (the example assumes that the namespace is "http://namespaces.softwareag.com/EDA/WebM/Process/1.0") :

```
Namespace defaultnamespace = Namespace.getNamespace("x",
  "http://namespaces.softwareag.com/EDA/WebM/Process/1.0");
   jdomXPath.addNamespace(defaultnamespace);
```

User defined function "UDF_CreateJDOM"

The "Process Monitoring Example" demo contains the user defined function "UDF_CreateJDOM". This UDF creates a DOM structure from an input XML structure.

UDF_CreateJDOM takes one parameter, namely a string containing the required XML structure.

Example:

```
SELECT 42.0, UDF_CreateJDOM('<E1>Test structure</E1>'), E2 FROM MyInputStream;
```

In this example, each tuple in the output stream will consist of (1) the floating constant 42.0, (2) the XML structure '<E1>Test structure</E1>', and (3) the value from the E2 element of events in the input stream.

12 Combining Events with Data from a Database

Introduction

There are two places where you might refer to the contents of an external database in a query statement. First, you might use the JOIN construct to join a database table with an input stream.

Here is an example:

```
SELECT * FROM inputStream JOIN dbsource123
USING (locationID);
```

In this example, `inputStream` is an input stream that is defined in the current continuous query project, and `dbsource123` is a database source object that references a database table. The stream and the table are joined on a common attribute, which is present in both the input stream and the table. In this example, they are joined on an attributed called `locationID`. The output event type is composed of the required fields of the input stream and the required columns of the database table. When `SELECT *` is specified, the query returns all of the fields of the input stream and all of the columns of the database table.

Note: The reference to the input stream is required since the Event Server's implementation of the SELECT statement requires temporal information that a database alone does not provide. In other words, you cannot create a query that includes *only* database source in the FROM clause.

The second place where you might refer to an external database in a query statement is in a user-defined operator. A user-defined operator is a Java class that you create to process event streams. A user-defined operator can use database sources as input in addition to event streams. For example, the following statement includes a user-defined operator called `UDO_MyOperator`. As input, this operator uses the event stream, `myInputStream` and the database source, `myDBSource`.

```
SELECT * FROM UDO_myOperator(myInputStream, myDBSource);
```

Important: Any database source that is used as input for a user-defined operator must be cached.

For more information about user-defined operators, see ***Developing User-Defined Operators***.

To use a database in a query, the following conditions must exist:

- The database must already exist and contain the data that is required by the query statement.
- You must create a *database connection profile* that references the required database. For procedures, see ***Creating a Database Connection Profile***.
- You must create a *database source object* that specifies the database table or view that the query statement requires. For procedures, see ***Creating and Editing a Database Source***.

Creating a Database Connection Profile

To create the database connection profile, you use an Eclipse perspective that is available in Software AG Designer as a pre-installed plug-in. The plug-in is provided by the Eclipse Foundation (**http://www.eclipse.org/**) as part of the Eclipse Data Tools Platform (DTP) project.

To create the database connection profile, proceed as follows:

To create the database connection profile

1 In Software AG Designer, open the perspective **Database Development**. Do this by selecting **Window > Open Perspective > Other...** and then select **Database Development**.

2 Start the **New Connection Profile** wizard. You can do this by selecting **New** from the context menu of the **Database Connections** node, or by clicking the **New Connection Profile** icon.

3 Supply the information required for connecting to the database. The required information varies, depending on the type of database you are accessing, but generally the following information is needed:

- JDBC driver for the specific database.
- The host name and port number for the database.
- Database logon credentials (user name and password).
- Depending on the database, you might also need the database name, ID, etc.

Please refer to the Eclipse Web site mentioned above for documentation on how to set up the specific type of database that you are using.

4 You can check that the database connection information you have provided is correct by using the **Test Connection** button. This attempts to connect to the database that you have specified, and informs you if the database is reachable.

5 After you have supplied all of the required information in the wizard, close the wizard in order to store the connection profile.

To access the connection profile in your continuous query project, you need to create a database source object in the Continuous Query Development perspective and add a reference from the database source object to the connection profile.

Creating and Editing a Database Source

To create the database source object, perform the following steps. If you want to cache the database table, after creating the database source object, edit it to indicate you want to use caching. For more information, see *Editing a Database Source Object*.

Important: If you are creating a database source that will be used by a user-defined operator *or* if the query using the database source will be executed by an Event Server that is running in high-availability mode, you *must* enable caching in the database source object. For more information about this requirement, see *Editing a Database Source Object*.

To create the database source object

1 In the Continuous Query Development perspective, choose **File > New > Database Source**.

This opens the wizard for creating a new database source.

2 In the **Project** field, select the project where you wish to store the database source object. By default, the current project is selected.

3 In the **Name** field, provide a name of your choice for the new database source object. The name of a database source must begin with a letter and include only the following characters:

- Letters A-Z or a-z
- Digits 0-9
- The following special characters: _ $

Note: The Event Query Language (EQL) does not directly support the use of EQL keywords or Java type names as database source names. If you wish to use an EQL keyword or a Java type name as a database source name in an EQL query, you must enclose the database source name in quotation marks in the query. See the description of the ObjectName statement in the *EQL Reference Guide* document for more information.

4 In the **Profile** field, select the name of the database connection profile that you created previously.

The drop-down menu of the **Profile** field lists the names of all currently defined database connection profiles. The list of connection profiles can be a subset of the profiles shown in the **Database Development** perspective; the list only shows connection profiles for databases whose proxy drivers are currently supported by webMethods products. See the section *List of Supported Database Drivers* below for a list of the supported databases.

5 If the database uses catalogs, the editor activates the **Catalog** field. Select the required catalog from the drop-down menu. When you have done this, the editor activates the **Schema** field.

If the database does not use catalogs, the **Catalog** field remains inactive and the editor activates the **Schema** field directly.

Note: Not all databases support catalogs, and even if a database supports catalogs, it does not necessarily use them.

6 In the **Schema** field, select the required schema.

7 In the **Tables and Views** field, specify the name of the database table that will provide input data for the query statement.

8 Click **Finish** to store the new database source object.

The file containing the database source object will be created and stored as a new file in the current project.

Note: The password that is specified in the database source is stored automatically in encrypted form in the Eclipse secure storage. See the section *Password Handling* below for more information.

You can now include references to the database in a query statements, as in the example shown above.

Editing a Database Source Object

If you want to change values defined for a database source object, including indicating that you want to cache the database tables, edit the object.

To edit a database source object, proceed as follows:

To edit a database source object

1 Open the editor by doing one of the following:

- Create a new database source object using the wizard as described above. When you close the wizard, the editor opens automatically.

  Or:

- Select the existing database source object file from the project tree and open it using any of the standard Eclipse methods.

2 Select the **General** tab, if it is not selected already. This tab contains the values that you specified in the wizard when you created the database object. Ensure that the values specified in this tab are correct for your application.

3 Select the **Tuple Schema** tab. Here you see the name and data type of each column of the table that you have specified. Each of the column names you see here can be used as an argu-

ment of the SELECT statement of a query that uses the database source. A marked checkbox under the **Required** heading indicates that the corresponding table column is not marked as NULLABLE, so it will always have a value assigned in the database.

4 Select the **Caching** tab. Use this tab to specify whether you want to use caching, and if so, how often you want the cache refreshed.

When you select to use cache, Event Server stores a copy of the database table contents in the cache. It then uses the cached database table for the computation of the results of your queries. As a result, your queries run faster. For information about the supported methods for caching, see ***Caching Methods***.

Be aware that when you use caching, if changes are made to the external database, those changes are not reflected in query results until the cache is refreshed.

Important: There are two cases when you *must* enable caching for a database source. 1) If the database source will be used by a user-defined operator or 2) if the query will be executed by an Event Server that is running in high availability-mode. If either of these conditions exist, you must enable caching (either permanent or with a refresh interval) in the database source object. If the Event Server is running in high-availability mode, you must additionally configure the Event Server so that it uses a Terracotta Server Array for caching. For more information about high availability, see *Running Event Server in High Availability Mode* in *Administering the Event Server*. For a description of using Terracotta Server Array for caching, see ***Caching Methods*** below. For information about configuring Event Server to use a Terracotta Server Array for caching, see *Configuring Event Server* in *Administering the Event Server*.

To specify whether you want to use caching and how often the cache is refreshed, select one of the following options:

Caching Mode	Description
No caching	Select this option if you do not want to use caching. When you use this option, each execution of a continuous query that uses a database source will access the external database table.
One-time caching	Select this option for a permanent cache. Event Server makes an initial copy of the database table and always works with the copy. With this option, the cache is never refreshed. If the contents of the database table do not change, use this option to avoid repeated access to the database table.
Periodic cache refreshing	Select this option if you want to make a copy of the database table and update the copy at regular intervals. Use this option if the rate at which the database contents change does not significantly affect the accuracy of the continuous query that accesses the database. Use the **Start date and time** and the **Refresh Interval** fields to set up a schedule for when and how often you want Event Server to refresh the cache. Note that the **Start date and time** field does not indicate the first time to copy the database table into the cache. Rather Event Server uses the values from this field along with the value in the

Caching Mode	Description
	Refresh Interval field to establish the refresh schedule. Event Server creates the initial cache when the query starts, then refreshes based on the schedule. For more information about how to define the caching schedule, see *Caching at Intervals*. Event Server maintains a cache validity. When Event Server refreshes a cache, it does not overwrite an existing cache. Rather, it maintains one or more copies of the cache, each with a separate validity interval. When a cache becomes obsolete, Event Server deletes it. For more information, see *Cache Validity* and *Matching Incoming Events to Caches and Deleting Caches*.

5 You can see the XML serialization of the database source object by selecting the **Source** tab. The contents of this display are read-only.

Software AG Designer uses the **Source** display for highlighting errors identified at a later stage by the builder.

Caching Methods

Event Server supports the following types of database cache:

Method	Advantages	Disadvantages
Local Cache Data is stored within the heap of the Java Virtual Machine (JVM). This is also known as on-heap cache. This is the default type of cache.	■ Requires no additional installation or configuration.	■ Limits the size of the database tables you can use because Event Server stores a copy of the entire database table in the cache. You are constrained by the overall heap limit of the underlying machine and operating system. It is recommended that you use the JVM heap for only caches up to 2 GB. ■ Can cause performance issues. The JVM heap is subject to the Java garbage collection. Depending on your machine size and operating system, garbage collection can introduce pauses in processing when large heaps are in use.
Off-Heap Cache (BigMemory) Ehcache BigMemory is used for cache. This is also known as off-heap cache.	■ Access to data is faster than via JVM heap when the cache size exceeds the size of main memory, which causes memory	■ Requires additional Event Server configuration to specify you are using BigMemory.

Method	Advantages	Disadvantages
When using BigMemory, Event Server can store data outside the JVM heap. When the cache that is allocated to the JVM heap is filled, the cache "overflows" to the off-heap storage. For more information about BigMemory, see the *Ehcache User Guide Version 2.6* at ***http://ehcache.org/documentation/2.6***.	swapping when using the JVM heap for caching. ■ Allows larger caches (e.g., more than 2 GB of data), up to 1 TB. ■ Allows you to use larger database tables. ■ Is not subject to the Java garbage collection process. This eliminates interruptions caused by garbage collection and allows the cache to perform more predictably and consistently. ■ Allows for easier management of the cache because there is no need to tune the garbage collector.	■ Requires Terracotta license from Software AG. ■ Requires installation. configuration, and setup of Terracotta Ehcache.
Distributed Caching (Terracotta Server Array) Data is stored in a Terracotta Server Array. A Terracotta Server Array consists of one or more Terracotta Servers. The data in a distributed cache is spread across the Terracotta Servers using a technique called "striping." For more information about distributed caching using a Terracotta Server Array, see the *Ehcache Distributed Cache User Guide Version 2.6* at ***http://ehcache.org/documentation/2.6*** **Note:** When running Event Server in high-availability mode, Event Server uses a Terracotta Server Array to store synchronization information. Event Server uses the same Terracotta Server Array for database caching.	■ Access to data can be faster than via JVM heap when the cache size exceeds the size of main memory, which causes memory swapping when using the JVM heap for caching. ■ Allows larger caches (e.g., more than 2 GB of data). ■ Provides failover and continuous uptime for the cache by setting up the Terracotta Server Array to use a hot standby or multiple active servers. ■ When running Event Server in high availability mode, the Terracotta Server Array keeps the cached database information in sync between the master and slave Event Servers.	■ Access to data can be slower than using off-heap cache (i.e., BigMemory) because the servers within the Terracotta Server Array have to be synchronized. ■ Requires additional Event Server configuration to identify that a Terracotta Server Array is being used for cache. ■ Requires Terracotta license from Software AG. ■ Requires installation, configuration, and setup of the Terracotta Server Array.

If you want to use the local cache, that is the JVM heap, for database caching, no action is required. However, if you want to use either BigMemory or a Terracotta Server Array, you need to configure the Event Server to indicate the method of caching you want to use. Additionally, you must install and configure the appropriate additional Terracotta product. For information about how to:

- Configure Event Server to use BigMemory or a Terracotta Server Array, see information about the `<ehcache-config>` parameter in *Configuring Event Server* in *Administering the Event Server*.
- Install and setup Ehcache to use BigMemory, see the *Ehcache User Guide Version 2.6* at ***http://ehcache.org/documentation/2.6***.
- Install and setup a Terracotta Server Array, see the *Installing webMethods Products* guide.

Caching at Intervals

When you specify caching for a database source object and you select the **Periodic cache refreshing** option, you use the **Start date and time** and **Refresh Interval** fields to define a schedule for how often Event Server refreshes the cache.

Note: When a query uses database sources with periodic cache refreshing, it is important to consider the time difference between the system time of the hosting computer (which is used to calculate timestamps of cache versions) and the application time. If, for example, the application time is before the system time of the hosting computer, such a query will not produce any output (apart from heartbeats) for the time difference because the input stream using application time "waits" for caches with corresponding timestamps.

The **Start date and time** do not indicate a specific date and time when to start caching. Rather, Event Server uses the values you specify to determine the caching schedule. Regardless of the values you specify, when a query starts, Event Server stores the initial copy of the database table into cache. After the initial copy is cached, Event Server continues to refresh the cache based on the refresh schedule.

Examples

- The following settings indicate that Event Server is to refresh the cache at 00:00:00 GMT, 02:00:00 GMT, 04:00:00 GMT, and so on.

Start date and time	2011-08-29 00:00:00
Start time zone	GMT
Refresh Interval	2 hours

  - When a query starts on 2011-08-30 at 01:00:00 GMT, Event Server creates the initial cache. The next scheduled refresh is on 2011-08-30 at 02:00:00 GMT.
  - When a query starts on 2011-07-30 at 02:30:00 GMT, Event Server creates the initial cache. The fact that the query start time precedes the start date specified when defining the cache refresh schedule does not prevent Event Server from starting the refresh schedule immediately. The next scheduled cache refresh is on 2011-07-30 at 04:00:00 GMT.
  - When a query starts on 2011-08-29 at 20:00:00 US/Pacific, Event Server creates the initial cache. The query start time is in US/Pacific time. The cache schedule is specified in GMT time. 2011-

08-29 at 20:00:00 US/Pacific is 2011-08-30 at 03:00:00 GMT. The next scheduled cache refresh is on 2011-08-30 at 04:00:00 GMT (i.e., 2011-08-29 at 21:00:00 US/Pacific).

- The following setting indicate that Event Server is to refresh the cache each day at 00:00:00 GMT.

Start date and time	2011-08-29 00:00:00
Start time zone	GMT
Refresh Interval	1 day

  - When a query starts on 2011-08-30 at 01:00:00 GMT, Event Server creates the initial cache. The next scheduled refresh is on 2011-08-31 at 00:00:00 GMT.
  - When a query starts on 2011-07-30 at 02:30:00 GMT, Event Server creates the initial cache. The next scheduled cache refresh is on 2011-07-31 at 00:00:00 GMT.
  - When a query starts on 2011-08-29 at 20:00:00 US/Pacific, Event Server creates the initial cache. 2011-08-29 at 20:00:00 US/Pacific is 2011-08-30 at 03:00:00 GMT. The next scheduled cache refresh is on 2011-08-31 at 00:00:00 GMT (i.e., 2011-08-30 at 17:00:00 US/Pacific).

Cache Validity

Event Server maintains a cache validity. When Event Server refreshes a cache, it does not overwrite an existing cache. Rather, it maintains one or more copies of the cache, each with a separate validity interval. When Event Server executes a query, it matches the validity of the incoming event against the cache validities to determine the appropriate cache to use for the query. Because incoming events are ordered based on their timestamp, it is possible for Event Server to determine when a cache is no longer useful. When an event arrives that is beyond the validity of an old cache, Event Server deletes the obsolete cache.

When Event Server initially caches a database table, the cache is valid from all time up to the next refresh time. At the next refresh time, Event Server creates another cache that is valid from when it is created until the next refresh time, and so on. For example, consider the following cache interval settings:

Start date and time	2011-08-29 00:00:00
Start time zone	GMT
Refresh Interval	1 day

- When a query is started on 2011-09-02 at 11:38:34 GMT, Event Server creates the initial cache. The validity of the initial cache is from all previous time until the next refresh, that is until 2011-09-03 at 00:00:00 GMT.
- At the next refresh time (i.e., 2011-09-03 00:00:00 GMT), Event Server refreshes the cache by making a new copy of the database table without overwriting the previous cache. This new cache has a validity from the time of this refresh until the next refresh, that is from 2011-09-03 at 00:00:00 GMT to 2011-09-04 at 00:00:00 GMT.

- At the next refresh time (i.e., 2011-09-04 at 00:00:00 GMT), Event Server refreshes the cache by making a new copy of the database table without overwriting the previous caches. This new cache has a validity from the time of this refresh until the next refresh, that is from 2011-09-04 at 00:00:00 GMT to 2011-09-05 at 00:00:00 GMT.

Event Server continues in this manner.

Matching Incoming Events to Caches and Deleting Caches

When Event Server receives an incoming event, it checks the event validity against the validities of its caches.

- When the validity of an event falls completely within the validity of a single cache, the event is joined against the cache that has the common validity.
- When the validity of an event spans multiple cache validities, Event Server splits up the event based on cache validities. Each part of the event is joined against the cache that has the same validity.

For example, consider the following cache validities:

cache 1	Valid from all time up to 2011-09-03 at 00:00:00 GMT
cache 2	Valid from 2011-09-03 at 00:00:00 GMT to 2011-09-04 at 00:00:00 GMT
cache 3	Valid from 2011-09-04 at 00:00:00 GMT to 2011-09-05 at 00:00:00 GMT

- An incoming event (Event 1) is valid from 2011-09-02 at 11:00:00 GMT to 2011-09-04 at 17:00:00 GMT.

  The validity of Event 1 spans cache 1, cache 2, and cache 3. As a result, Event 1 joins cache 1, cache 2, and cache 3.

- An incoming event (Event 2) is valid from 2011-09-03 at 03:00:00 GMT to 2011-09-03 at 16:00:00 GMT.

  The validity of Event 2 falls completely within the validity of cache 2. As a result, Event 2 joins only cache 2.

Event Server keeps track of the progress of event streams joined against cached database tables. Because events are ordered based on their start timestamp, Event Server can detect when a cached database table has become obsolete. That is, when a stream delivers an event that has a start timestamp that is later than the validity of a cache, the cache has becomes obsolete, and as a result, Event Server deletes it. In the example above when Event 2 is delivered, it triggers the deletion of cache 1 because every succeeding event will have a start timestamp equal to or later than the start timestamp of Event 2.

Because Event Server can maintain multiple versions of the caches, you should set up your queries and/or refresh intervals to avoid exceeding the size of main memory. If your query uses a temporal

window that is quite large, you should consider increasing the refresh interval so that Event Server does not need to keep as many copies of the database table in memory. Alternatively, you can reduce the window size defined in the query so that caches can become obsolete more frequently so that Event Server maintains fewer cache versions.

List of Supported Database Drivers

When you create a database source object, you can use an Eclipse DTP connection profile that is based on any webMethods DataDirect JDBC driver.

The supported webMethods DataDirect JDBC drivers are:

- webMethods DataDirect DB2 8.1 Driver
- webMethods DataDirect DB2 8.2 Driver
- webMethods DataDirect DB2 9.1 Driver
- webMethods DataDirect Oracle 8 Driver
- webMethods DataDirect Oracle 9 Driver
- webMethods DataDirect Oracle 10 Driver
- webMethods DataDirect SQL Server 2000
- webMethods DataDirect SQL Server 2005
- webMethods DataDirect Sybase 12.x
- webMethods DataDirect Sybase 15.0

In addition, the following Apache Derby JDBC drivers are supported:

- Derby Client JDBC Driver 10.1 Default
- Derby Client JDBC Driver 10.2 Default

Renaming a Database Source

Use the following procedure to rename a database source

To rename a database source

1 In the Project Explorer, right-click the database source.

2 From the context menu, select **Rename**.

3 Specify a new name for the database source. The name of a database source must begin with a letter and include only the following characters:

- Letters A-Z or a-z
- Digits 0-9
- The following special characters: _ $

Important: Do not modify or delete the file extension. A database source file must have the extension ".dbsource".

4 Click **OK**.

5 If this database source is referenced by queries in the project, update those queries so that they refer to this database source by its new name. (Queries that refer to the database source by its old name are listed in the **Problems** view.)

Tip: You can also perform the rename operation from the context menu for the database source.

Password Handling

When you create a database source object for a database connection, Software AG Designer takes the user name and password that you specified for the database connection and stores them in the Eclipse secure storage. The password is stored in encrypted form. These values are required when the database source is deployed to the Event Server, so that the Event Server can access the corresponding database using the user name and password.

You can check that user names and passwords have been assigned to your database source objects as follows:

To check that a user name and password are assigned to a database source

1 In Software AG Designer, open **Windows > Preferences**.

2 Expand **General > Security > Secure Storage** and select the **Contents** tab.

3 Expand the node **com.softwareag.wep**.

The user name of your database source should now be visible. The password field is shown but its value is hidden.

Eclipse uses a master password to encrypt password information in the Eclipse secure storage for the workspace. Your Software AG Designer environment can include several Software AG products that use the Eclipse secure storage. The master password is generated automatically by Eclipse using an internal algorithm when you create the first encrypted password in the Eclipse secure

storage. In the webMethods Business Events environment, the password mechanism in secure storage is used only when dealing with database sources, so if the master password has not already been generated in your Software AG Designer environment, the master password will be generated when you create the first database source. When this happens, a dialog appears, informing you that a master password has been generated. Normally you do not need to change this password, since Eclipse uses it for its own internal processing, so we suggest that you to leave this value unchanged. For more information about master passwords, refer to the standard Eclipse documentation.

Notes on Usage

Notes:

1. If you create a database source object from a connection profile, then change the connection profile, the database source object is not updated automatically. To use the new connection profile, you must create a new database source object. If you create a new database source object with the same name as an existing database source object, the existing version will be overwritten (you will be prompted for confirmation before the object is overwritten).
2. If you edit a database source which is already used in a query and change the table, it will probably happen that the new table has different columns from the original one. This would lead to an error in the query as this changes the output of the query; for example, fields might no longer be available, or new fields might be added to the output.
3. Currently, the user name and password for a database source are stored in the Eclipse secure storage and also in the database source itself. At a later stage, the user name and password will only be stored in the Eclipse secure storage.
4. The password stored in the Eclipse secure storage for a database source is currently not modifiable.

13 Testing a Project in Software AG Designer

Introduction

This section describes how you create an event sequence file and use it to simulate an input stream for a continuous query application. It also describes how to create a launch configuration to test a continuous query application using a specified event sequence file.

Creating Event Sequences

You can use event sequences to simulate events in event streams, and therefore to test continuous query applications. An event sequence is a sequence of event instances based on event type definitions. Each event type definition describes a type of event that can occur in an input stream, for example: a new bid on the item with item ID 5 arrives for the bid stream. The event sequence is stored in a text file, and each line of the file contains a single event instance. Each event instance is represented by a list of comma-separated values (CSVs), i.e. each event instance is a CSV list.

Example of an Event Sequence

As an example, consider the following sample event sequence:

```
EVENT,OpenAuctionStream,2010-04-01T08:15:59.001Z,2010-04-01T08:15:59.005Z,"myStuff"
EVENT,BidStream,2010-04-01T08:16:00Z,,"b1007-1","hugo",100.00
EVENT,OpenAuctionStream,2010-04-01T08:17:12Z,,"otherStuff"
EVENT,BidStream,2010-04-01T08:17:12Z,"b1007-2","clara",123.45
EVENT,ClosedAuctionStream,2010-04-01T08:17:59Z,, 123.45
HEARTBEAT,BidStream,2010-04-01T08:18:00Z
```

This example shows:

- An event on the stream *OpenAuctionStream* with an explicit end timestamp.
- A set of events for miscellaneous streams which have no explicit end timestamp and hence have an implicit validity duration of 1 chronon (1 millisecond), i.e. the end timestamp equals the start timestamp plus one millisecond.
- A heartbeat on the stream *BidStream*.

Syntax of Event Sequences

The event sequence contains EVENT statements and HEARTBEAT statements with the following syntax:

```
EVENT,<streamName>,<startTimestamp>,[<endTimestamp>],<column1-value>,<column2-value>, ↩
...

HEARTBEAT,<streamName>,<timestamp>
```

The following guidelines apply for the syntax:

- For each EVENT statement, you must specify the stream name and start timestamp. You can supply the end timestamp; if you omit it, a duration of 1 chronon is assumed. The column values <column1-value>, <column2-value> etc. must correspond in number and data type to the values defined in the schema of the stream.
- For each HEARTBEAT statement, you must supply the stream name and timestamp. By definition, a heartbeat identifies a single point in time, so there is a single timestamp for the statement rather than a start and end timestamp.
- The field delimiter in each event definition is the comma character ",".
- Leading and trailing whitespace characters around field delimiters is allowed, e.g. `, test ,` `test ,` and will be ignored during processing.
- To declare a field value as NULL, omit the value. For example, `attr1,,attr3` means that the value of attr2 is null. You can also use an empty string, indicated by two adjacent double quotes, to indicate a null value, for example `attr1,"",attr3`. Note that the field must be defined as nullable in the input stream definition if you want to supply a NULL value in the event sequence.
- The quoting of strings with double quotes is optional except for the following cases where it is mandatory:
  - If the string contains the field delimiter character, e.g. `test,test` then it must be set as `"test,test"`.
  - The empty string must be set as `""`.
  - If the string contains double quotes, they must be substituted by a pair of consecutive double quotes, e.g. `test" test` must be set as `"test"" test"`.
  - If the string starts or ends with whitespace characters.
- If a string is quoted, only whitespace characters are allowed between the delimiter character and the quoted sequence as well as between the quoted sequence and the delimiter character, e.g. `,a "test" b,` is not allowed.
- If an input stream is defined as a chronon stream, the end timestamp must be 1 millisecond after the start timestamp.

- The timestamp format relies on the W3C date and time format approach; more precisely the format is date plus hours, minutes, seconds and an optional decimal fraction of a second, followed by a required time zone designator:

  ```
  YYYY-MM-DDThh:mm:ss.sssTZD
  ```

  The time zone designator (indicated by `TZD`) can be either "Z" for indicating UTC, or "+hh:mm" or "-hh:mm" to indicate an offset from UTC.

  The decimal fraction of a second can be omitted, or can contain any number of digits; if it contains more than 3 digits, the value will be rounded to the nearest millisecond.

  Example of a timestamp: `2011-04-28T08:15:59.001Z`
- The start timestamp of an event must not be before the start timestamp of the preceding event.

In addition, comments are allowed. Any line beginning with the "#" character is considered to be a comment line.

Conversion from Schema Syntax to Event Sequence Syntax

Field order

As mentioned above in the section **Syntax of Event Sequences**, the <column-value> entries in the EVENT statements correspond to definitions in the event type schemas for the various input streams.

For example, here is the schema for a sample input stream *Person*:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:eda="http://namespaces.softwareag.com/EDA/Event"
  xmlns:p="http://namespaces.softwareag.com/EDA/MyFolder"
  targetNamespace="http://namespaces.softwareag.com/EDA/MyFolder"
  elementFormDefault="qualified">
  <xs:import namespace="http://namespaces.softwareag.com/EDA/Event" ↩
schemaLocation="../Event/Envelope.xsd"/>
  <xs:element name="MyEventType" type="p:MyEventTypeType" ↩
substitutionGroup="eda:Payload">
    <xs:complexType name="MyEventTypeType">
      <xs:sequence>
        <xs:element name="id" type="xs:integer"/>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="emailAddress" type="xs:string"/>
        <xs:element name="city" type="xs:string"/>
        <xs:element name="country" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
```

```
  </xs:element>
</xs:schema>
```

This corresponds to the following serialized field definitions:

```
id :           INTEGER
name :         STRING
emailAddress : STRING
city :         STRING
country :      STRING
```

The parameters of each EVENT statement for the input stream *Person* must contain values for these fields in the order shown. Values for fields defined with the attribute `minOccurs="0"` in the schema may be left blank, but the comma must still be retained as a field separator.

An event in the sample input stream *Person* could look like this:

```
<Person>
    <id>24479</id>
    <name>Lang</name>
    <emailAddress>MyName@MyCompany.com</emailAddress>
    <city>Frankfurt</city>
    <country>Germany</country>
</Person>
```

A corresponding EVENT statement could look like this:

```
EVENT,Person,2010-06-04T08:15:59.001Z,,24479,"Lang","MyName@MyCompany.com","Frankfurt","Germany"
```

In this EVENT statement, the optional end time has been omitted.

A query statement that selects the fields *id* and *name* from the *Person* input stream would look like this:

```
SELECT id, name FROM Person
```

Complex schema structures

The schema for the input stream can define complex structures (hierarchies of elements nested within elements). For example, a schema for an input stream *Client* might allow an event instance such as:

```
<Client>
  <id>123</id>
  <person>
    <name>Petit</name>
    <address>
      <city>Paris</city>
      <country>France</country>
    </address>
  </person>
</Client>
```

In this case, the serialized field definition would use the dollar character ("$") to indicate the hierarchy of the fields:

```
id :                    123
person$name :           Petit
person$address$city:    Paris
person$address$country: France
```

A query statement that selects the fields *id* and *city* from the *Client* input stream would look like this:

```
SELECT id, person$address$city FROM Client
```

Element attributes

The schema for the input stream can define element attributes. For example, this schema defines the attributes `mode` and `type` for the input stream *deviceStatus*:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  xmlns:eda="http://namespaces.softwareag.com/EDA/Event"
  <import namespace="http://namespaces.softwareag.com/EDA/Event"
         schemaLocation="../Event/Envelope.xsd"/>
  <element name="deviceStatus" substitutionGroup="eda:Payload">
    <complexType>
      <sequence>
        <element name="id" type="string"/>
        <element name="status" type="boolean"/>
        <element name="address" minOccurs="0" type="string"/>
      </sequence>
      <attribute name="mode" type="string"/>
      <attribute name="type" type="long" use="required"/>
    </complexType>
  </element>
</schema>
```

This corresponds to the serialized field definitions:

```
id :      STRING
status :  BOOLEAN
address : STRING NULLABLE
@mode :   STRING NULLABLE
@type :   LONG
```

A query statement that selects the field *mode* from the *deviceStatus* input stream would look like this:

```
SELECT "@mode" FROM deviceStatus
```

The quotation marks round "@mode" are required, since field names containing "@" are not valid standard identifiers.

Field datatypes

The data type of each field is converted according to the following table:

Schema datatype	Event sequence datatype
xsd:decimal	BIG_DECIMAL (java.math.BigDecimal)
xsd:boolean	BOOLEAN (java.lang.Boolean)
xsd:byte	BYTE (java.lang.Byte)
xsd:double	DOUBLE (java.lang.Double)
xsd:float	FLOAT (java.lang.Float)
xsd:integer	INTEGER (java.lang.Integer)
xsd:long	LONG (java.lang.Long)
xsd:short	SHORT (java.lang.Short)
xsd:string	STRING (java.lang.String)
xsd:date	DATE (java.sql.Date)
xsd:time	TIME (java.sql.Time)
xsd:dateTime	TIMESTAMP (java.sql.Timestamp)

Note: For the <column-value> entries in the EVENT statements, the datatypes `DATE`, `TIME` and `TIMESTAMP` can optionally include timezone information; if no timezone is set, the time is interpreted as UTC. For the <startTimestamp> and <endTimestamp> entries in the EVENT statements, the timezone information is mandatory.

Manually Creating an Event Sequence File in Software AG Designer

Use the following procedure to create a new event sequence.

To create an event sequence

1. In the Continuous Query Development, choose **File > New > Event Sequence**.
2. In the **Project** field, specify the name of the project in which you want to store the event sequence.
3. In the **Name** field, specify a name for the new event sequence.
4. Click **Finish**.
5. In the text editor view, enter your event definitions.
6. When you are finished, use CTRL_S to save the sequence.

Using an Event Log File to Test a Query in Software AG Designer

If you have an Event Server that receives the type of events that your query consumes, you can configure the Event Server to log those events to the event log file. The event log file has the same structure as an event sequence file, which enables you to use it for testing and debugging a continuous query application in Software AG Designer

For information about how you configure an Event Server to log certain types of events, see *Logging Query Output Streams*.

Be aware that an event log file can contain events from many different input streams, not just the one(s) your project consumes. An event log file can also include events from output streams that continuous query applications running on the Event Server generate. If, when testing your project, Continuous Query Development encounters an output event in the log file or an event for a different project, it issues a warning and continues to the next event in the log file.

Using a Launch Configuration for Local Testing

A launch configuration enables you to start your continuous query application in Software AG Designer with predefined start conditions. A Continuous Query Development launch configuration defines the following general types of information:

- The project for which the launch configuration applies.
- The event sequence that will be processed during the run to simulate one or more input event streams.
- The query or queries to use to process the event sequence.

- Various run options.

Launch configurations can be used to test your Continuous Query Development applications.

When you run a launch configuration, the events in the event sequence are processed in sequential order by the local event processing engine. When an event in the event sequence matches the selection criteria in a query, the fields identified by the query are written to the output stream.

Creating a Launch Configuration

Use the following procedure to create a launch configuration:

To create a launch configuration

1 Choose **Run Configurations...** from the toolbar.

This opens the Run Configurations dialog. The dialog shows a pane where the available configuration types are displayed, including the configuration type **Continuous Query Application**.

2 Select **Continuous Query Application** in the configuration type pane.

If you have already defined any Continuous Query Application configurations, the names of these configurations will be shown under the node **Continuous Query Application**.

3 If you wish to create a new configuration, click the New icon above the configuration type pane.

Alternatively, you can choose **New** from the context menu of the node **Continuous Query Application**.

4 In the field **Name**, enter a name for the configuration you want to create.

5 Select the **Main** tab.

Specify the name of the project that contains the definitions (event sequences, queries) that you want to use in the run configuration.

6 Select the **Event Sequence** tab.

In the field **Select Event Sequence File**, specify the name of the file that contains the required event sequence. You can use the drop down list to view the event sequences defined in your workspace.

You can also use the **Import** button to copy an event sequence file from your file system to your workspace. When you import a file in this way, the newly imported file appears at the start of the drop-down list.

7 Select the **Queries** tab.

In this tab, choose the stored queries that you want to use for the launch configuration. These are the queries that will be used to query the input event stream when you activate the run configuration at a later stage.

Initially, the display lists all queries defined in the workspace. Mark the checkboxes of the queries you wish to use for the current launch configuration, then click **Apply**.

The queries you specify may depend on other queries. To ensure that all dependencies between queries are resolved, use the **Add required** button.

Here is an example of dependencies between queries: If you have a query named *AllBids* defined as `SELECT * FROM Bidstream` and a second query defined as `SELECT * FROM AllBids WHERE sellerID = 'SomeName'`, then the second query depends on *Allbids* defined in the first query. If only the second query and not the first query is supplied in the run configuration, this would not be a valid run configuration, since the definition of *Allbids* would be missing. Using the **Add required** button would automatically add the *Allbids* query to the list.

8 Select the **Common** tab.

This is a standard tab that appears in the Run Configurations dialog of many Eclipse perspectives. In this tab you can specify various options that apply to the launch configuration.

If you wish to store the launch configuration in your workspace, mark the radio button **Local file** in the section **Save as**. If you want to save the run configuration in your file system, mark **Shared file** and specify the required location.

By default, all output that is created by running the launch configuration is sent to the console. You can route the console output to a file by marking the **File** checkbox in the panel **Standard Input and Output** and specifying the name of a file on the file system. You can suppress the output by deselecting the checkbox **Allocate Console**.

Running a Launch Configuration

When you run a launch configuration, you perform a local test that checks whether the queries you have specified return the expected records from the test records in the event sequence. When you are satisfied that the queries are ready for use in a production environment, you can deploy them to the target configuration. For information about deploying a continuous query application, see *Deploying a Continuous Query Application to an Event Server*.

To run a launch configuration for local testing, proceed as follows:

To run a launch configuration

1 Choose **Run Configurations...** from the toolbar.

This opens the Run Configurations dialog. The dialog shows a pane where the available configuration types are displayed, including the configuration type **Continuous Query Application**.

2 Select **Continuous Query Application** in the configuration type pane.

If you have already defined any Continuous Query Application configurations, the names of these configurations will be shown under the node **Continuous Query Application**.

3 Click the name of the configuration you wish to run.

4 Click **Run**.

When you run a launch configuration, automatic validity checks are made to ensure that all dependencies defined in the configuration settings can be resolved. These validity checks include the following:

- All input streams required by the configuration's queries must be already defined in the project;
- All queries that are used as an input stream for another query in the launch configuration must be present in the set of queries specified in the launch configuration;
- All of the configuration's event sequences must be already defined in the project.

If the launch configuration is not valid, the launch configuration cannot be run, and appropriate error messages are displayed. If this happens, correct the problem and run the launch configuration again.

The validity checks apply only to the current launch configuration that you wish to run; they do not apply to any other launch configurations or any other components defined in the project.

Using The Output Stream

Overview of the output stream

The output stream is displayed in the Console tab in the Continuous Query Development perspective.

The output stream is structured so that it can be used as the input event sequence of any other launch configuration. Therefore the records in the output stream are extended automatically to contain not only the fields of the selected events, but also the syntax elements required for an event sequence.

If you wish to use the output stream as a new event sequence for another launch configuration, create a new event sequence and copy the contents of the Console tab into it.

The following formatting guidelines apply to the output stream:

- A timestamp is formatted as `YYYY-MM-DDThh:mm:ss.sssZ` with "Z" as the UTC designator.
- A string is quoted in the following cases:
  - if it is empty;
  - if it contains the delimiter character;
  - if it starts or ends with whitespace characters;
  - if it contains quotes (which are automatically substituted by a double quote pair).

Logging event sequences in the console output

By default, input stream events in an event sequence file are also logged in the console output.

Logged events from the input stream have the same layout as logged events from the output stream, but have a hash sign (“#”) in the first column:

```
#EVENT, application.stream, ...
```

If you want to use the console output as a new event sequence file, all lines starting with the hash character are treated as commented lines, and are therefore ignored.

You can switch off the logging of input events as follows:

1. Open the file *eclipse.ini*, that is located in a version-specific subdirectory of *<SAGInstallDir>/eclipse*, for example *C:\SoftwareAG\eclipse\v43_95*.
2. At the end of the file (in particular, after the line containing "–vmargs"), add the line:

   ```
   -Dcom.softwareag.wep.plugin.launch.log.input=false
   ```

3. Restart the Software AG Designer.

You can turn logging back on again by replacing "false" by "true", or by removing this line from the file, then restarting the Software AG Designer.

14 Deploying a Continuous Query Application to an Event Server

Overview of the Deployment Process

When you are ready to test your continuous query application in a "live" environment or put the application in production, you *deploy* the application to an Event Server.

The deployment process installs your continuous query application on an Event Server. If the deployment process is successful, the Event Server launches the application automatically.

You can deploy a continuous query application using the two methods described below. The method you use depends on the following:

- Whether you are deploying the application to a production or non-production server
- Whether you are running a standalone Event Server or running Event Server in high availability mode

Method	Description	For Additional Information and Procedures
The CAR-file method	This method involves archiving your continuous query application to a CAR (Continuous Query ARchive) file and copying the CAR file to the Event Server's deployment folder. This method is easy to use but it does not allow you to alter properties of the application that were specified at design-time. This method is suitable for deploying continuous query applications into non-production environments such as those used for demonstration, development and testing. You cannot use a CAR file for deployment when Event Server is running in high availability mode.	See ***Deploying a Continuous Query Application from a CAR File***.
The webMethods Deployer method	This method involves several steps: ■ If your application requires event types that are not already present in the Event Type Store, you must deploy the event types to the Event Type Store before you deploy your continuous query application. ■ For your continuous query application, you first create an ACDL (Asset Composite Description Language) file using the Asset Build Environment. The ACDL file contains a list of the assets (resources) associated with your continuous query application, including details about their properties and dependencies. ■ Next, you use webMethods Deployer, the ACDL file and the deployment archive to deploy the application to an Event Server. The advantages of using the Deployer method are that 1) it supports deployment of assets from a version control system (VCS), 2) it enables	See ***Deploying a Continuous Query Application using webMethods Deployer***.

Method	Description	For Additional Information and Procedures
	you to deploy a continuous query application to multiple Event Servers in one step and 3) it supports property substitution. Use the webMethods Deployer method when you are ready to deploy an application into a production environment. You must use webMethods Deployer for deployment when Event Server is running in high availability mode.	

Important: Do not mix the two deployment methods on the same Event Server. Mixing the two methods can potentially result in a situation wherein you have two instances of the same continuous query application running on the same server.

The deployed application typically refers to event types that describe the data structure of events in the application's input and output streams. Using the CAR-file method, the event types are included in the CAR file. Using the webMethods Deployer method, these event types must be deployed to the Event Type Store in a separate step before the application is deployed.

What Happens When You Deploy a Continuous Query Application to Event Server?

This section lists the actions that Event Server performs during deployment of a continuous query application when you are using either deployment method.

If you are running Event Server in high availability mode, the Event Servers forming the high-availability cluster ensure that the deployment archives received by each server in the cluster are consistent before they perform the actions listed below. To do so, each Event Server computes a checksum for the deployment archives received from Deployer. The Event Servers then compare the checksums to ensure the results are equal. If the results differ, the Event Servers reject the deployment. For more information about running Event Server in high availability mode, see *Running Event Server in High Availability Mode* in *Administering Event Server*.

During deployment of a continuous query application, Event Server extracts the artifacts from the deployment archive file and does the following:

- Compiles the Java code associated with any user-defined aggregates (UDAs), user-defined functions (UDFs), and user-defined operators (UDOs) that are associated with the application (webMethods Deployer method only).
- Registers the application's input streams with the event engine in the Event Server.
- Registers and initializes the database sources, if any. To complete this step, the specified database must be running and it must be accessible using the connection credentials given at deployment

time. For more information about how the deployment process passes the connection credentials to the Event Server, see ***Connection Parameters for Database Sources***.

Additionally, if you have configured Event Server to use Ehcache BigMemory or a Terracotta Server Array, the following also must be true:

- For Enterprise Ehcache BigMemory, Ehcache must be installed and configured.
- For Terracotta Server Array, the Terracotta Server must be configured and started.

For more information, see information about the `<ehcache-config>` parameter in the Configuring Event Server section of *Administering the Event Server*.

- Registers any user-defined extensions (UDAs, UDFs, or UDOs) that are associated with the application.
- Starts the application's queries.

  If you are running Event Server in high availability mode, each Event Server synchronously starts the queries.
- For compatibility with previous product releases: In earlier releases, the deployment composite created using the webMethods Deployer method also contained the event types required by the continuous query application. If you wish to deploy such a composite and the compatibility mode is active, the event types in the composite are copied to the Event Type Store, otherwise the deployment is rejected. You can activate the compatibility mode using the property `eventserver_deployment.skip.types` in the *build.properties* file.

Additionally, if you are using Integration Server (as opposed to the NERV framework) to connect to the Event Bus, the deployment does the following:

- If the auto-gen-topics option is enabled on the Event Server, the deployment process checks whether a corresponding topic exists in the JNDI directory for each channel the application defines. If a corresponding topic does not exist for a channel, the deployment process creates the topic. For more information about the auto-gen-topics option, see *JMS Topic Generation* in *Administering webMethods Event Server*.
- Creates a JMS trigger (on the Integration Server) for each input stream that the application consumes. The trigger receives events of a specified type from the Event Bus and passes them to the continuous query application on the Event Server.

The deployment process also extracts the resources from the archive file and places them in the following directory on the Event Server:

SAGInstallDir/EventServer/WEPApps/deployed/*projectName*

For example, if you deploy a project named *MyProject* on Event Server, the deployment process extracts the file's contents to:

SAGInstallDir/EventServer/WEPApps/deployed/*MyProject*

Note: When you start Event Server, it automatically restarts the continuous query applications that have already been deployed on the server. It does this by redeploying the applications whose deployment archives are present in the WEPApps directory. If an undeployed CAR file is present in the WEPApps directory, (i.e., a CAR file whose application has not yet been deployed on the server) the application in that CAR file will be deployed when the server begins polling the WEPApps directory for new applications.

The deployment.xml File

The directory into which an application is deployed (i.e., the *SAGInstallDir*/EventServer/WEPApps/deployed/*projectName*) will contain a file called *deployment.xml*. This file contains messages that are logged by the deployment process for that application. You can examine this file to determine when the application was last deployed and whether it was deployed successfully. If a deployment process fails, this file will include error messages that indicate the cause of the failure.

Things to Check Before You Create the Deployment Archive

Before you create the deployment archive for either deployment method, verify that the following conditions are satisfied:

- Be sure you are starting with a project that builds without any errors in the Continuous Query Development perspective in Software AG Designer.
- If you are using Integration Server, verify that the **JMS connection alias** to which each input or output stream is bound exists on the Integration Server (i.e., make sure the Integration Server has all of the JMS connection aliases that the application specifies).

If you are running Event Server in high availability mode, verify the following is true for the Event Servers belonging to the high availability cluster:

- The Event Servers all receive the same input events.
- The Event Servers are all connected to the same Terracotta Server Array.

  The Event Servers use of the same Terracotta Server Array for:

  - Synchronization of the Event Servers within the cluster
  - Detection of high availability failover scenarios
  - Database caching
- The Event Servers all use the same data from external databases.

  To ensure this, you must use database caching. The database caches are shared between all Event Servers via the Terracotta Server Array. For more information, see **Combining Events with Data from a Database** and Configuring Event Server in *Administering Event Server*.

- The user-defined functions for the Event Servers are stateless and their results do not depend on any external data.

Things to Check Before You Deploy Your Continuous Query Application

Before you deploy the archive, verify that the following conditions are satisfied:

- Ensure that all of the required components in the runtime environment are running. Depending on your setup, based in some cases on the configuration options you chose when you installed the product, some components might be started automatically (for example, as Windows services) when you start your machine, whereas you might have to start others manually.

  In particular, check that the webMethods Universal Messaging or webMethods Broker (or a 3rd party product) that you are using as your JMS provider is running. Check the appropriate product documentation for information about how to start or restart your chosen JMS provider.
- If you are using Integration Server:
  - If the <auto-gen-topics> option is *not enabled* on Event Server, verify that the JMS topic (channel) to which each input or output stream is bound exists on the JNDI naming server.
  - If the <auto-gen-topics> option is *enabled* on Event Server, verify that the Universal Messaging realm server or webMethods Broker that serves as your JNDI naming service and JMS provider is running.
- If the application includes database sources, verify that the specified databases are running. Also verify that the connection credentials that the application specifies (or the ones you intend to pass to the Event Server through property substitution) are valid for the databases to which the application will connect.
- If the application includes database sources that are defined to use caching and the caching method is:
  - Ehcache BigMemory, ensure Ehcache is installed and configured.
  - Terracotta Server Array, ensure that the Terracotta Server is configured and started.
- If you use the webMethods Deployer method, all event types that the continuous query application requires should already be deployed and available in the Event Type Store.

Failure to meet these conditions can cause the deployment process to fail. For information about how a deployment failure is handled, see ***What Happens if the Deployment Process Fails***.

What Happens if the Deployment Process Fails

If the deployment of a continuous query application using the webMethods Deployer method fails, the deployment process can automatically roll back any assets that it registered on the Event Server for the application. This includes queries, database sources, and input streams. Assets that the deployment process created on any other components do not get rolled back. For example, the deployment process does not roll back JMS topics that it added to the Event Bus.

Similarly, if the deployment of an event type project fails, the deployment process can roll back any event types that it registered in the Event Type Store.

You can specify whether or not a failed deployment will cause an automatic rollback by selecting the option "Rollback on Error" in the Deployer dialog when you build the deployment archive. See the *Deployer User Guide* for details of automatic rollback.

The deployment process records the failure in the following log files:

- In the server log
- In the **deployment.xml** file (located in *SAGInstallDir*/EventServer/WEPApps/deployed/*projectName*/)

Connection Parameters for Database Sources

If a continuous query application includes a database source, the application that you deploy to the Event Server must include connection parameters for the appropriate database. For example, when you build and test a continuous query application, you usually configure the database source to log on to a database in your test environment. When you deploy the application to a production environment, you must configure the database source to log on to the production database.

The way in which you specify the database connection parameters when you deploy an application depends on which deployment method you are using.

- *If you are performing a CAR-file deployment*, the deployment process simply copies the database sources and their associated connection profiles to the Event Server, exactly as they are specified in the CAR file. Therefore, before you generate the CAR file, you must make sure that the connection profiles in the project are configured for the databases that you want the application to use at run time.
- *If you are using webMethods Deployer*, you specify the application's database connection parameters at deployment time. When you deploy a continuous query application using webMethods Deployer, you use the Deployer's property-substitution feature to enter the appropriate database URL, user name, and password for each of the database sources in the application. When the

application is deployed on Event Server, the deployment process replaces the connection parameters that were specified at design time with the ones you specified in webMethods Deployer.

Security Considerations Relating to Deployment

The credentials that you need in order to deploy a continuous query application depends on the deployment method you are using.

- *If you are performing a CAR-file deployment*, you must have access to the file system on the machine where Event Server is installed. You must also have permission to write files to the *SAGInstallDir*/EventServer/WEBApps/ directory.
- *If you are using webMethods Deployer*, you must have a user account that belongs to the Administrators group on the Integration Server that runs in the OSGi container with the Event Server.

If the continuous query application includes database sources, consider taking the following precautions to protect the passwords for those databases.

- *When you perform a CAR-file deployment*, the passwords associated with the database sources are included in the CAR file. The passwords are encrypted, so they are not human readable, however, the encryption level is not strong. Because CAR files can potentially contain passwords, you must not handle them carelessly. Always store them in a secure directory and take steps to protect them when transmitting them from one machine to another.
- *If you are using webMethods Deployer*, the passwords associated with the data sources are never included in the *composites*, which are compressed files containing the assets to deploy. Thus, the archive file contains no password information and does not need to be secured in the same way as a CAR file.

  However, when you specify database passwords in webMethods Deployer using the property-substitution feature, the Deployer transmits those passwords as part of the ACDL file that it sends to Integration Server. Although the passwords are encrypted, the encryption level is not strong. To ensure that the encrypted passwords cannot be picked up during transmission, we recommend that you configure an HTTPS port on Integration Server and use that port for deploying continuous query applications from webMethods Deployer (instead of using Integration Server's default HTTP port, 5555). For information about configuring ports on Integration Server, see “Configuring Ports” in *Administering webMethods Integration Server*.

Deploying a Continuous Query Application from a CAR File

Note: You cannot use a CAR file for deployment when Event Server is running in high availability mode.

This section provides information and procedures for deploying a continuous query application from a CAR file.

The CAR File

A CAR file is a deployment archive that contains the resources necessary to produce a runnable instance of a continuous query application on an Event Server. A CAR file contains the following resources from a continuous query project:

- The project name (as derived from the root folder name within the CAR file)
- Queries
- Event types

  Note: The CAR file will contain the event types that you explicitly included in the continuous query project, as well as any event types or component schemas that the project referenced from other continuous query projects or event type projects.

- Input streams
- Database sources (if the project includes them)
- User-defined aggregates, user-defined functions, and user-defined operators (if the project includes them)

  Important: Any classes and custom libraries (JARs) that are associated with the aforementioned user-defined extensions but are not explicitly included in the project subtree *are not* included in the CAR file. In particular, if you have defined custom libraries in the classpath of the project but the libraries are not located in the project subtree, those libraries will not be included in the export. To ensure that the custom libraries are included in the CAR file and deployed with the continuous query application, you must explicitly include those libraries in your project.

Note that a CAR file does not include every artifact related to a continuous query project in Software AG Designer. Resources that are relevant only to the design-time environment, such as event sequence files, are not included in the CAR file.

Steps in a CAR-File Deployment

To deploy a continuous query application using a CAR file, you must do the following:

1. Export your continuous query project using Software AG Designer. The export process collects the run-time related resources associated with your project and copies them to a CAR file. Procedures for this step are provided in *Creating a CAR File*.
2. Copy the CAR file to the deployment directory (*SAGInstallDir*/EventServer/WEPApps/) on the Event Server.

   The Event Server polls its deployment directory every ten seconds. When it detects the presence of a new CAR file, Event Server deploys the application that the CAR file contains. Procedures for this step are located in *Deploying a CAR File*.

Important: Do not use the CAR file deployment method to deploy a continuous query application to an Event Server that hosts applications that were deployed to it using webMethods Deployer. To determine how the applications on an Event Server were deployed, examine the server's *SAGInstallDir*/EventServer/WEPApps/ directory. If it contains deployment archives that are zip files, you must use webMethods Deployer to deploy additional applications on that server. For procedures, see *Deploying a Continuous Query Application using webMethods Deployer*.

Creating a CAR File

Use the following procedure to create a CAR file.

Important: Before you create your CAR file, be sure that the conditions in **Things to Check Before You Create the Deployment Archive** are satisfied.

To create a CAR file

1 In Software AG Designer, go to **File > Export**.

2 From the **Software AG** entry, choose **Continuous Query Application**.

3 Select the continuous query project that you want to export and click **Next**.

4 If Software AG Designer prompts you to assign a JMS connection alias to an input steam, specify the appropriate alias and click **Next**. (Software AG Designer will prompt you once for each input stream whose **JMS connection alias** property is unassigned.)

Note: When you assign a JMS connection alias to an input stream, you can optionally change the **Channel** settings at the same time. If you make changes to the **JMS connec-**

tion alias or the **Channel** properties in this dialog box, Software AG Designer applies those changes both to the exported project in the archive file and to the underlying project in Software AG Designer.

5 In the **Archive File** panel, specify the fully qualified name of the file to which you want the project exported or click **Browse** to select a directory and specify a file name.

- You can assign any name you like to the CAR file, but it is a good practice give the CAR file the same name as the project.
- If you manually type a file name, be sure to include the .car file extension.

6 Click **Finish**.

7 Deploy the resulting CAR file on Event Server using the procedure described in *Deploying a CAR File*.

Deploying a CAR File

Use the following procedure to deploy a CAR file on an Event Server.

Important: Before you deploy your CAR file, be sure that the conditions in **Things to Check Before You Deploy Your Continuous Query Application** are satisfied.

To deploy a CAR file on Event Server

1 On the Event Server, copy the CAR file to the following directory:

SAGInstallDir/EventServer/WEPApps/

Event Server will deploy the CAR file in its next polling cycle (which is every ten seconds).

2 To verify that the application has deployed successfully, examine the application's **deployment.xml** file as described in *How to Determine Whether a Continuous Query Application Deployed Successfully*.

Deploying a Continuous Query Application using webMethods Deployer

webMethods Deployer is a tool that enables you to deploy many different types of webMethods assets. You can use webMethods Deployer to deploy continuous query applications to Event Servers, and to deploy event types to the Event Type Store.

webMethods Deployer offers several advantages over CAR files for deployment into a production environment. For example:

- It enables you to deploy assets for other webMethods components at the same time you deploy a continuous query application. For example, if your continuous query application triggers the execution of services on Integration Server, you can deploy those services to the Integration Server at the same time that you deploy the continuous query application to the Event Server.
- It enables you to deploy assets in a distributed environment where target runtimes like Event Servers are located on different host machines.
- It enables you to perform *property substitution* at deployment time. Property substitution lets you modify the values of certain properties in the application before you deploy it. For example, if the application's database parameters currently point to a test database, Deployer enables you to replace those parameters at deployment time with ones that point to the production database.
- It enables you to deploy an asset (e.g., one continuous query application) to multiple target environments in one step.

For more information about the webMethods Deployer tool, see the *webMethods Deployer User's Guide*.

Note: You must use webMethods Deployer for deployment when Event Server is running in high availability mode.

An Overview of a Deployer-Based Deployment

To deploy a continuous query application using webMethods Deployer, you must use Deployer's *repository-based deployment*. In a repository-based deployment, you use Ant scripts to build a *composite file*, a compressed file containing the continuous query applications you want to deploy. The composite includes a deployment archive (i.e., a zip file) containing the assets that make up an application, and an ACDL file containing metadata about the assets in the zip file (including asset dependencies and substitutable properties). After you build the composite, you deploy it to an Event Server using webMethods Deployer.

You also use this approach to build composites for deploying event type projects. Any event types required by a continuous query application must be deployed before the continuous query application. Refer to the *Deployer User Guide* for related information.

Note: webMethods Deployer supports other modes of deployment. However, you can deploy continuous query applications only using the Deployer's repository-based deployment.

To build a composite, your continuous query project or event type project must reside where you have installed the *webMethods Asset Build Environment*. The Asset Build Environment provides Ant scripts that you use to build composites for Event Server and other webMethods components.

If you install the Asset Build Environment on the same computer that you use to develop continuous query applications, you can create composites on that computer. However, many organizations install the Asset Build Environment on a separate computer, known as a *build machine*, and they use this computer to build composites for webMethods components.

When you use a build machine to build the composite, developers check their project files into a version control system (VCS) when their continuous query applications are ready to be deployed. When you want to build the composite, you check the project files out of the VCS and execute the build script. You might even configure the build script to check the project files out from the VCS automatically.

When you run the build script for a continuous query application, the build process generates a composite containing your application. You deploy the composite to a specified Event Server using webMethods Deployer.

When you run the build script for an event type project, the build process generates a composite containing your event types. You deploy the composite to a specified Event Type Store using webMethods Deployer.

Deploying an Application Using webMethods Deployer

Important: Do not use webMethods Deployer to deploy a continuous query application to an Event Server that currently hosts applications that were deployed to it using CAR files. To determine how the applications on an Event Server were originally deployed, examine the server's *SAGInstallDir*/EventServer/WEPApps/ directory. If it contains CAR files, you must use the CAR file deployment method to deploy additional applications to that server. For procedures, see ***Deploying a Continuous Query Application from a CAR File***.

To deploy a continuous query application using webMethods Deployer, you must perform the following high-level steps.

1. Deploy all event types required by the application. For information, see **Deploying Event Types Using webMethods Deployer**.
2. Place the project files on a machine that is equipped with the Asset Build Environment. See **Placing the Project Files in the Asset Build Environment**.
3. Configure the *build.properties* file and run the build. For information, see **Configuring the Build Environment and Running the Build to Create a Composite**.
4. Define the Event Servers to which you want to deploy continuous query applications. For more information, see **Setting Up Connections to Target Event Servers**.
5. If you are running Event Server in high availability mode, create a target group that contains the Event Servers you use for high availability. If you are running Event Server in high availability mode, you have to deploy a project simultaneously to all Event Servers belonging to the high-availability cluster. As a result, it is recommended that you define a target group that contains the Event Servers. For more information, see **Creating a Target Group for Deployment When Running in High Availability Mode**.
6. Use the Deployer's repository-based deployment to deploy the composite that the build produced. For information, see **Deploying the Application using webMethods Deployer**.

Deploying Event Types Using webMethods Deployer

Before you deploy a continuous query application, you must deploy the event types that the continuous query application uses.

When you deploy event types, the deployment process copies the event types to the Event Type Store.

The method you use to deploy event types is similar to the method for deploying continuous query applications: You create the event type definitions in an Eclipse project (of type "Plug-In Project"), then you run a build script in the Asset Build Environment to create a deployment composite that contains the event types, then you use the Deployer to copy the deployment composite to the target machine.

Refer to the *Deployer User Guide* for details.

Placing the Project Files in the Asset Build Environment

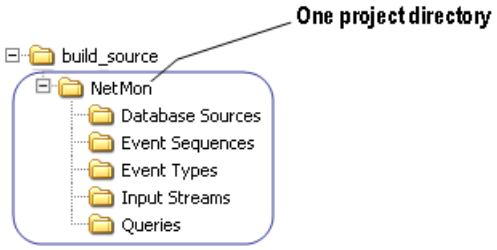
If the Asset Build Environment is not installed on your development machine, you must copy your continuous query project to a machine where the Asset Build Environment is installed.

Usually, you transfer files to the build machine through a version control system (VCS), for example, Concurrent Versions System (CVS) or Subversion (SVN). That is, you check your project files into a VCS and then check them out on the build machine. However, you can copy them to the build machine using any mechanism you choose.

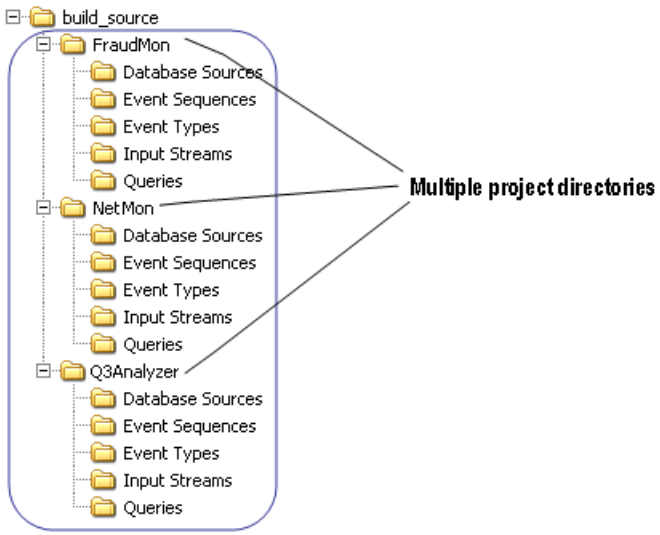
Important: Before you begin the build process, be sure that your project files satisfy the conditions listed in **Things to Check Before You Create the Deployment Archive**.

When you copy your project files to the build machine, keep the following points in mind:

- Be sure the copy includes the project's top-level directory and all of the project's subdirectories. Your project directory should look similar to the example below:



- If you want to deploy several continuous query projects at one time, place all of the projects under one top-level directory as shown in the example below:



- If your project includes event sequence files, you can include them on the build machine with the other project files. However, the build process will not include them in the composite because event sequence files are not used in the run-time environment.

Configuring the Build Properties and Running the Build Script to Create a Composite

You run build scripts in the Asset Build Environment to create a composite. Before you run the build script, you must configure the build.properties file. This file specifies:

- The types of target runtimes for which you are building a composite.
- The location of the Event Server project files that you want to deploy (that is, the build *source* directory).
- The repository location to which you want to write the resulting composite (that is, the build *output* directory).

Note: If you need additional information about configuring the build.properties file or running the build, see the *webMethods Deployer User's Guide*.

Important: When specifying a file path in the properties file, use the slash "/" character as the path delimiter, not the backslash "\", even if you are running the build on a Windows machine.

To configure the build.properties file and run the build script

1 Open the following file in a text editor:

`SAGInstallDir`/common/AssetBuildEnvironment/master_build/build.properties

2 Specify the location of the project (or projects) for which you want to build a composite.

If you are deploying...	In this parameter...	Specify...
One continuous query project	`build.source.project.dir`	The fully qualified name of the project directory that contains the Event Server project that you want to deploy. **Example** `build.source.project.dir=D:/build_source/NetMon`
Multiple continuous query projects	`build.source.dir`	The fully qualified name of the top-level directory that contains all of the Event Server projects that you want to deploy. **Example** `build.source.dir=D:/build_source`

3 Specify the following additional parameters.

In this parameter...	Specify...
`sag.install.dir`	The fully qualified name of the Software AG installation directory. **Example** `sag.install.dir=D:/SoftwareAG`
`build.output.dir`	The fully qualified name of the directory in which you want the build process to create the resulting composite(s). **Note:** The build process will create this directory if it does not already exist. **Example** `build.output.dir=D:/build_output`
`enable.build.EventServer`	true **Example** `enable.build.EventServer=true`
`eventserver_deployment.skip.types`	Whether to include event types in the composite. In previous product versions, there was no separate deployment procedure for the event types; instead, the event types were included in the deployment composite for the continuous query application. For compatibility reasons, the Event Server can still accept deployment composites that contain a combination of application and event type components. If you set the property to false (this is the default value), the Event Server will accept composites that include both application and event type

In this parameter...	Specify...
	components. If you set this property to true, the Event Server will accept the deployed application but ignore any event types that are included in the composite. **Example** `eventserver_deployment.skip.types=true`

4 Save the *build.properties* file.

5 From the command line, do the following:

1. Navigate to the following directory:

   SAGInstallDir/common/AssetBuildEnvironment/master_build

2. Run the following command:

   ```
   ant build
   ```

6 When the build is complete, go to the directory that you specified in the `build.output.dir` parameter. If the build was successful, this directory will contain a subdirectory named EventServer. The EventServer directory will contain a *projectName*.zip and a *projectName*.acdl file for each project in the source folder hierarchy.

Setting Up Connections to Target Event Servers

From webMethods Deployer, you must define connections to the target Event Servers to which you want to deploy continuous query applications. When defining a target Event Server, you are asked to provide the information in the table below. For instructions about how to create a connection to a target Event Server, see the *webMethods Deployer User's Guide*.

For this parameter...	Specify...
Host	The DNS name or IP address of the computer on which the Event Server to which you want to deploy the continuous query application is running.
Port	The port (on Integration Server) from which you want the Deployer to invoke the Event Server deployment service. To use the Integration Server default HTTP port, specify port 5555. **Note:** If you are deploying a continuous query application that includes database sources, consider using an SSL port to invoke the deployment service on Integration Server. Doing this ensures that the database passwords are transmitted over the network securely. For more information about password security during deployment, see **Security Considerations Relating to Deployment**.

For this parameter...	Specify...
User Name	The user name that Deployer will submit to Integration Server when it invokes the Event Server deployment service. This user name must be a member of the Administrators group on the Integration Server that is bundled with the Event Server. **Note:** If you have not changed the password for the Administrator account that is installed with Integration Server, you can use the user name, Administrator, and the password, manage. (These credentials are case-sensitive.)
Password	The password for the user specified in **User Name.**
Version	The version number of the target Event Server. In general, this version number will be the same as the version number of the Deployer. If the Deployer supports older versions of the Event Server, these versions can be selected also. The version number determines the scope of functionality supported by the target Event Server. The Deployer will deploy the composite according to the functionality supported by the target Event Server.
Use SSL	Enable this option if you want the Deployer to use SSL to deploy the continuous query to the target server that you specified in **Host** and **Port**. If you enable this option, the port that you specify in **Port** must be SSL enabled.

Creating a Target Group for Deployment When Running in High Availability Mode

If you are running Event Server in high availability mode, create a Deployer target group that contains the Event Servers that you are using for high availability. After the target group is created, you can then deploy to the target group, rather than to the individual servers. By deploying to the target group, you ensure that both instances of the Event Server have the same set of deployed queries. For instructions about how to create a target group, see the *webMethods Deployer User's Guide*.

Deploying the Application Using Deployer

After you have built the composite for you continuous query application and have identified the target servers, you can use Deployer to deploy the continuous query application.

Important: Before you begin the deployment process, verify that the Event Server to which you want to deploy the application is running. If you are running Event Server in high availability mode, ensure both Event Servers are running. Additionally, be sure that the conditions in **Things to Check Before You Deploy Your Continuous Query Application** are satisfied.

To deploy the continuous query application, follow the procedures for a repository-based deployment described in the *webMethods Deployer User's Guide*.

- Create a deployment project to define the continuous query applications in the repository that you want to deploy. To identify the applications you want to deploy, you define deployment sets for the project. When identifying the source for the deployment sets, specify the repository where you created the composite.
- Create a deployment map to identify where to deploy each deployment set in the project. If you are running Event Server:
  - Standalone, identify the target Event Server.
  - High availability mode, identify the target group, which specifies the two Event Servers you use for high availability.
- If the application includes database source definitions, you must configure the deployment map and specify the appropriate connection parameters for each database source in the composite. The properties that you must specify are shown below:

Property	Specify ...
JdbcConnectionUrl	The URL to be used to connect to the database.
Username	The user name to be used to connect to the database.
Password	The password for the user specified in **Username**.

A database source in the composite file has the same URL and user name values that it did at design time. These values become the default values for the **JdbcConnectionUrl** and **Username** properties at deployment time .If you leave the **JdbcConnectionUrl** and **Username** property fields empty at deployment time, the default values (i.e., the values that were assigned to the database source at design-time) are assigned to the database source when it is deployed.

The password for a database source, however, is never included in the composite file. Therefore, the **Password** property does not have an associated default value that can be used for deployment. Because it has no default value, you must explicitly assign a value to the **Password** property for each database source in the composite file.

Note: If multiple database sources use the same **Password** value, select all of them in the **Configurable Components** panel, specify the password value and then click **Save Substitutions**. Doing this will assign the specified password to all of the selected database sources in one step.

- Deploy the project.

How to Determine Whether a Continuous Query Application Deployed Successfully

To determine whether a continuous query application was deployed successfully, examine the application's **deployment.xml** file. This file contains messages that are logged by the deployment process. The messages in this file will indicate whether the application was deployed successfully. If the deployment fails, the log will provide information about the reason for the failure.

The deployment log is located here:

SAGInstallDir/EventServer/WEPApps/deployed/*projectName*/deployment.xml

See also, ***What Happens if the Deployment Process Fails***.

If you deployed the continuous query application using webMethods Deployer, you will also receive a deployment report. The report will indicate whether Deployer was able to deploy the application successfully.

Undeploying a Continuous Query Application

To undeploy a continuous query application, you simply delete the application's deployment archive from the following directory on the Event Server:

```
SAGInstallDir/EventServer/WEPApps/
```

For example, if the application had been deployed using a CAR file, you would delete the following deployment archive:

```
SAGInstallDir/EventServer/WEPApps/projectName.car
```

If the application had been deployed using webMethods Deployer, you would delete the following deployment archive:

```
SAGInstallDir/EventServer/WEPApps/projectName.zip
```

Important: If you are running Event Server in high availability mode, be sure to delete the deployment archive from both Event Servers that you are using for high availability. Deleting the deployment archive from only one of the Event Servers can cause problems for the high availability system.

At the next polling interval, Event Server will detect that the deployment archive is gone and it will do the following:

- If you are using Integration Server: Delete the JMS triggers that are associated with the application
- Deregister the application's queries, database sources, user-defined extensions (UDAs, UDFs, and UDOs), and input streams from the Event Server

The following objects *are not* affected when you undeploy an application. These are shared objects that might be used by other components.

- The JMS topics and their corresponding JNDI directory entries
- Event types in the Event Type Store

Redeploying a Continuous Query Application

If you make changes to a continuous query project, and you want to deploy it on the same Event Server where it is currently running, simply deploy the updated project using the regular deployment step described in ***Deploying a Continuous Query Application from a CAR File*** or ***Deploying a Continuous Query Application using webMethods Deployer***.

Important: You must redeploy the application using the same deployment method that you used to deploy the application originally.

Note that there is no need to stop the Event Server or to undeploy the existing application before deploying the updated project. For example, if you wanted to redeploy an application that you had originally deployed from a CAR file, you would:

1. Generate a new CAR file containing the updated project files.
2. Copy the new CAR file to the `SAGInstallDir`/EventServer/WEPApps/ directory on the Event Server (overwriting the one that exists there already).

When Event Server detects the presence of the updated CAR file, it will automatically redeploy the project from the updated deployment archive.

15 Testing, Debugging, and Troubleshooting a Deployed Application

Introduction

This section describes methods you can employ to test, debug, and troubleshoot your continuous query applications. It describes how you can use:

- **Event Server Admin** view in Software AG Designer to:
  - Examine a continuous query application that is running on Event Server. For more information, see *Displaying Administrative Information about an Event Server*.
  - Log event streams for analysis. For more information, see *Logging Query Output Streams*.

  Note: Before you can use **Event Server Admin** view, you must first identify the server to Software AG Designer. For more information, see *Working with Event Servers in Software AG Designer*.

- Event Generator to generate a simulated input stream to test an application in the run-time environment. For more information, see *Simulating an Input Stream to Test a Continuous Query Application in the Run-Time Environment*.
- Event Bus Console, Event Server logs, and Integration Server logs to obtain information that you can use to troubleshoot an application that is not running properly. For more information, see *Viewing Events Flowing Across the Event Bus* and *Viewing the Event Server's Log*.
- Terracotta Developer Console to view information about cache if you use BigMemory or a Terracotta Server Array for your database cache. For more information, see *Viewing Terracotta Cache Information*.

Working with Event Servers in Software AG Designer

Many of the testing and debugging procedures in this section use the features of the **Event Server Admin** view in Software AG Designer. To use this view with a particular Event Server, you must first add that server to the **Servers** view in your workspace. After you add an Event Server to the **Servers** view, you can connect to that Event Server when you work in the Continuous Query Development perspective. For example, if you are running Event Server in high availability mode, add all the Event Servers in the high availability cluster.

The following sections describe how to add Event Servers to your workspace, edit the properties of Event Servers that are in your workspace, and remove Event Servers from your workspace.

Adding an Event Server to Your Software AG Designer Workspace

Use the following procedure to add an Event Server to the Continuous Query Development perspective in Software AG Designer. When you complete this procedure, Software AG Designer lists the Event Server in the **Servers** view, and you can monitor and administer that Event Server in the **Event Server Admin** view.

Important: If you configure an Event Server instance so that remote administration is not allowed, you cannot add the Event Server to the **Event Server Admin** view. To ensure the Event Server allows remote administration, verify that the `server-port` setting in the EventServerCfg file is set to a valid port number and is not set to 0. For more information, see the section *Configuring Event Server* in *Administering the Event Server* for details.

Note: Although the **Servers** view is a standard Eclipse view that provides many standard controls for starting a server, stopping a server, publishing to a server, and so forth, you cannot use these controls for Event Server. For example, you cannot start and stop the Event Server from the **Servers** view.

To add an Event Server to Software AG Designer

1 In the Continuous Query Development perspective, open the **Servers** view.

2 From the context menu in the **Servers** view, select **New > Server**.

3 From the list in the **Define a New Server** panel, select the **Software AG > Event Server** node and complete the following fields.

In this field...	Specify...
Server's host name	The DNS name or IP address of the machine on which the Event Server is running.
Server name	The display name that you want to use for this Event Server. The display name is the name that Software AG Designer uses to represent this server in the user interface. (This is the name that you will see for this server in the **Servers** view.) By default, Software AG Designer suggests the default name "Event Server at *ServerHostName*", where *ServerHostName* is the DNS name or IP address you specified in **Server's host name**. However, you can specify any name to represent the specified Event Server.
Server runtime environment	**Event Server**.

4 If the Event Server is configured to listen for administrative requests on port 7867, click **Finish**.

5 If the Event Server is configured to listen for administrative requests on any port other than 7867, click **Next** to display the **Connection Settings for Event Server** panel. Then do the following:

   1. In the **Port** field, specify the port number on which this Event Server listens for administrative requests.
   2. Click **Finish.**

Removing an Event Server from Your Software AG Designer Workspace

Use the following procedure to remove an Event Server to the Continuous Query Development perspective in your Software AG Designer workspace.

To remove an Event Server from Software AG Designer

1 In the Continuous Query Development perspective, open the **Servers** view.

2 Open the context menu for the Event Server that you want to remove from the Continuous Query Development perspective and select **Delete**.

3 Set the options in the **Delete Server** dialog, as follows:

   1. Enable the **Delete running server(s)** option.
   2. Disable the **Stop server(s) before deleting** option.

4 Click **OK**.

Note: The **Stop Server(s) before deleting** option is not applicable for Event Server. If enabled, the Event Server is not stopped. When you enable this option, Eclipse attempts to stop the Event Server causing a lengthy delay before it removes the Event Server from the **Servers** view. When you disable the option, Eclipse immediately removes the Event Server from the view.

Changing the Name or Connection Properties of an Event Server Displayed in Your Software AG Designer Workspace

Use the following procedure to change connection properties and/or the display name for an Event Server listed in the **Servers** view.

To modify the properties of an Event Server listed in the Servers view

1 In the Continuous Query Development perspective, open the **Servers** view.

2 Open the context menu for the Event Server whose properties you want to change and select **Open**.

3 In the **General Information** section of the **Overview** window, modify the following parameters as necessary:

In this field...	Specify...
Server name	The display name that you want to use for this Event Server. The display name is the name that Software AG Designer uses to represent this server in the user interface. (This is the name that you will see for this server in the **Servers** view.) By default, Software AG Designer suggests the default name "Event Server at *ServerHostName*", where *ServerHostName* is the DNS name or IP address you specified in **Server's host name**. However, you can specify any name to represent the specified Event Server.
Host name	The DNS name or IP address of the machine on which the Event Server is running.
Server runtime environment	**Event Server**.

Note: Ignore the **Open Launch Configuration** link that appears in the **General Information** section. This option is not applicable to Event Server. Clicking this link generates an error message.

4 In the **Connection Information** section of the **Overview** window, specify the port on which this Event Server listens for administrative requests.

Note: The properties in the **Publishing** and **Timeouts** sections are not applicable to Event Server.

5 Click the Close icon on the **Overview** window.

6 When Software AG Designer asks whether you want to save your changes, click **Yes.**

Displaying Administrative Information about an Event Server

After you have added Event Servers to your Software AG Designer workspace, you can use the **Event Server Admin** view to display information about the Event Servers. To view information about an Event Server, select the server in the **Event Server Admin** view.

The **Event Server Admin** view displays the continuous query applications that are currently deployed on each server. For each deployed application, the **Event Server Admin** view displays the application's input streams, queries, database sources, and user-defined extensions.

Displaying the Continuous Query Applications

Use the following procedure to view the list of continuous query applications that are deployed on an Event Server.

To display the continuous query applications deployed on an Event Server

1 In the Continuous Query Development perspective, open the **Event Server Admin** view.

2 Locate the Event Server for which you want to view information. If necessary, expand the server node to display the continuous query applications running on it.

3 Expand the application node to display the input streams, queries, database sources and user-defined extensions associated with the application.

Logging Query Output Streams

You can log query output streams to troubleshoot an application that is not working the way you expect. Logging events enables you to inspect a stream of events that the application is receiving or producing. It also allows you to capture a stream of events and use it for additional testing and debugging in Software AG Designer.

Event logging is activated via the **Event Server Admin** view in the Software AG Designer. You can log output events from queries. Note that logging is only supported for queries with query output enabled. For input streams, logging is generally not supported.

When you enable logging for a query, Event Server writes every query output event to the standard event log file. There is only one log file. If you enable the logging for more than one query simultaneously, Event Server writes the events of all of these queries to the same log file. Also, if you have several applications enabled simultaneously, and each application has its own query output streams, Event Server logs all events to the same log file.

To differentiate between streams from different applications, Event Server writes the events to the log file using the following format:

```
EVENT, application.stream, ...
```

Note: When Event Server is running in high availability mode, only the server acting as the master logs query output events to the event log. The server acting as the slave does not log events. If the slave detects that the query results that it computed differs from those that the master computed, it logs this difference information to the event log. For more information, see *Running Event Server in High Availability Mode* in *Administering Event Server*.

The event log uses the same format as the event sequence files that you can use in Software AG Designer for continuous query development. As a result, you can use an event log file for the de-

velopment or improvement of continuous queries. For details about the structure of an Event Sequence File, see *Syntax of an Event Sequence*.

You specify the location of the event log file using the `<eventlog-path>` parameter in the EventServerCfg file. For more information about setting this parameter, see *Configuring Event Server* in *Administering the Event Server*.

Logging is useful for debugging purposes, and also for replaying event sequences at a later stage. For related information, see *Using an Event Log File to Test a Query in Software AG Designer*.

Use the following procedure to enable or disable the logging for a query output stream.

To enable or disable the logging for a query output stream

1 In the Continuous Query Development perspective, display the **Event Server Admin** view.

2 Locate the server for which you want to enable or disable the recording of an event stream and expand that server node to display the continuous query applications on that server.

Note: If the server whose query output streams you want to record does not appear in the **Event Server Admin** view, you need to first add the server to Software AG Designer. For procedures, see *Adding an Event Server to Your Software AG Designer Workspace*.

3 If you want to enable or disable recording of a query output stream, perform the following steps:

1. Expand the continuous query application whose output stream(s) you want to enable or disable.
2. Under **Queries**, locate the query for which you want to enable or disable logging. You can select multiple queries.
3. From the context menu, choose **Enable Logging** or **Disable Logging** as appropriate.

When logging is enabled for a query output stream, the label "recording..." appears beside the name of the stream.

Simulating an Input Stream to Test a Continuous Query Application in the Run-Time Environment

You can test a continuous query application using an input stream that you simulate using the Event Generator.

The Event Generator is a non-supported tool that is made available on the Software AG community Web site. You can use this tool to publish a stream of events to a particular topic on the Event Bus, including topics that represent input streams for a continuous query application.

The Event Generator allows you to specify the characteristics of the event stream, such as the rate at which events are to be submitted and the exact attributes that you want it to submit for each event. You can also assign behavioral characteristics to individual attributes. For example, you might want a particular attribute to trend up for five minutes with a variance of three percent. Or you might want to produce random numbers against a baseline of 100 with a variance of ten percent. Or you might want an attribute to simply iterate through a list of values over time.

For more information about the Event Generator, see the **Software AG community Web site**.

Checking Activity on the Event Bus

Viewing Events Flowing Across the Event Bus

Often it is useful to confirm whether events are flowing to a particular topic on the Event Bus. For example, if a consumer is not receiving events from a topic that a continuous query application publishes, you might want to examine the topic to determine whether the query application is actually delivering any events to it. To determine whether a particular topic is actually receiving any events, you can use the Event Bus Console utility to monitor the topic in Software AG Designer.

The Event Bus Console is a Java program that enables you to subscribe to topics on the Event Bus and display a trace of the topic's event traffic in the Eclipse console. For more information about using the Event Bus Console, see *Viewing Events Using the Event Bus Console* in *Event Type Development Help*.

Checking the status of the Event Bus

If your application does not appear to be receiving events from the event bus or sending events to the event bus, check that the webMethods Universal Messaging or webMethods Broker (or a 3rd party product) that you are using as your JMS provider is running. Depending on the way your chosen JMS provider is configured, you might have to start or restart the JMS provider manually. Check the product documentation of the JMS provider for details.

Viewing the Event Server's Log

The Event Server logs errors that occur while a continuous query application is processing. The errors are written to the server log, which, by default, resides in *SAGInstallDir*/profiles/IS/logs/sag_osgi.log. You can view the log by opening it in a text editor.

Note: The location of the server log is configurable. The server log on your Event Server might reside in a different location. If necessary, consult the administrator of your Event Server to get the log's exact location.

An administrator can configure the amount and type of information that the Event Server records in the server log. This flexibility is useful for troubleshooting. For example, you might temporarily increase the level of detail written to the server log to help uncover the cause of an application error or other problem, and return to a lower level after the problem is resolved. For information about configuring the logging level for the server log, see *Working with the Server Log* in *Administering the Event Server*.

Viewing the Integration Server's Log

You can also find helpful information in the Integration Server's log, particularly when troubleshooting issues that involve JMS.

To view the server log for Integration Server, open the Integration Server Administrator and go to the **Logs** > **Server** page. For more information about the server log for Integration Server, see the *webMethods Integration Server Administrator's Guide*.

Viewing Terracotta Cache Information

If you use Ehcache BigMemory or a Terracotta Server Array, you can use the Terracotta Developer Console to examine the cache.