Reference Documentation

Version: 2.0.0

provided by

Table of Contents

Preface

Analyzing and reacting to information in real-time oftentimes requires the development of custom applications. Typically these applications must obtain the data to analyze, filter data, derive information and then indicate this information through some form of presentation or communication. Data may arrive with high frequency requiring high throughput processing. And applications may need to be flexible and react to changes in requirements while the data is processed. Esper is an event stream processor that aims to enable a short development cycle from inception to production for these types of applications.

This document is a resource for software developers who develop event driven applications. It also contains information that is useful for business analysts and system architects who are evaluating Esper.

It is assumed that the reader is familiar with the Java programming language.

This document is relevant in all phases of your software development project: from design to deployment and support.

If you are new to Esper, please follow these steps:

1. Read the tutorials, case studies and solution patterns available on the Esper public web site at `http://esper.codehaus.org`
2. Read Section 1.1, “Introduction to CEP and event stream analysis” if you are new to CEP and ESP (complex event processing, event stream processing)
3. Read Chapter 2, *Event Representations* that explains the different ways of representing events to Esper
4. Read Chapter 3, *Processing Model* to gain insight into EPL continuous query results
5. Read Section 4.1, “EPL Introduction” for an introduction to event stream processing via EPL
6. Read Section 5.1, “Event Pattern Overview” for an overview over event patterns
7. Then glance over the examples Section 12.1, “Examples Overview”
8. Finally to test drive Esper performance, read Chapter 13, *Performance*

Chapter 1. Technology Overview

1.1. Introduction to CEP and event stream analysis

The Esper engine has been developed to address the requirements of applications that analyze and react to events. Some typical examples of applications are:

- Business process management and automation (process monitoring, BAM, reporting exceptions)
- Finance (algorithmic trading, fraud detection, risk management)
- Network and application monitoring (intrusion detection, SLA monitoring)
- Sensor network applications (RFID reading, scheduling and control of fabrication lines, air traffic)

What these applications have in common is the requirement to process events (or messages) in real-time or near real-time. This is sometimes referred to as complex event processing (CEP) and event stream analysis. Key considerations for these types of applications are throughput, latency and the complexity of the logic required.

- High throughput - applications that process large volumes of messages (between 1,000 to 100k messages per second)
- Low latency - applications that react in real-time to conditions that occur (from a few milliseconds to a few seconds)
- Complex computations - applications that detect patterns among events (event correlation), filter events, aggregate time or length windows of events, join event streams, trigger based on absence of events etc.

The Esper engine was designed to make it easier to build and extend CEP applications.

1.2. CEP and relational databases

Relational databases and the standard query language (SQL) are designed for applications in which most data is fairly static and complex queries are less frequent. Also, most databases store all data on disks (except for in-memory databases) and are therefore optimized for disk access.

To retrieve data from a database an application must issue a query. If an application need the data 10 times per second it must fire the query 10 times per second. This does not scale well to hundreds or thousands of queries per second.

Database triggers can be used to fire in response to database update events. However database triggers tend to be slow and often cannot easily perform complex condition checking and implement logic to react.

In-memory databases may be better suited to CEP applications then traditional relational database as they generally have good query performance. Yet they are not optimized to provide immediate, real-time query results required for CEP and event stream analysis.

1.3. The Esper engine for CEP

The Esper engine works a bit like a database turned upside-down. Instead of storing the data and running queries against stored data, the Esper engine allows applications to store queries and run the data through. Response from the Esper engine is real-time when conditions occur that match queries. The execution model is thus continuous rather then only when a query is submitted.

Esper provides two principal methods or mechanisms to process events: event patterns and event stream queries.

Esper offers an event pattern language to specify expression-based event pattern matching. Underlying the pattern matching engine is a state machine implementation. This method of event processing matches expected sequences of presence or absence of events or combinations of events. It includes time-based correlation of events.

Esper also offers event stream queries that address the event stream analysis requirements of CEP applications. Event stream queries provide the windows, aggregation, joining and analysis functions for use with streams of events. These queries are following the EPL syntax. EPL has been designed for similarity with the SQL query language but differs from SQL in its use of views rather then tables. Views represent the different operations needed to structure data in an event stream and to derive data from an event stream.

Esper provides these two methods as alternatives through the same API.

1.4. Required 3rd Party Libraries

Esper requires the following 3rd-party libraries at runtime:

- ANTLR is the parser generator used for parsing and parse tree walking of the pattern and EPL syntax. Credit goes to Terence Parr at http://www.antlr.org. The ANTLR license is in the lib directory. The library is required for compile-time only.
- CGLIB is the code generation library for fast method calls. This open source software is under the Apache license. The Apache 2.0 license is in the lib directory.
- LOG4J and Apache commons logging are logging components. This open source software is under the Apache license. The Apache 2.0 license is in the lib directory.

Esper requires the following 3rd-party libraries at compile-time and for running the test suite:

- JUnit is a great unit testing framework. Its license has also been placed in the lib directory. The library is required for build-time only.
- MySQL connector library is used for testing SQL integration and is required for running the automated test suite.

Chapter 2. Event Representations

2.1. Event Underlying Java Objects

An event is an immutable record of a past occurrence of an action or state change. Event properties capture the state information for an event. An event is represented by either a POJO (plain-old Java object), a `java.util.Map` or a XML document via `org.w3c.dom.Node`.

In Esper, an event can be represented by any of the following underlying Java objects:

Table 2.1. Event Underlying Java Objects

Java Class	Description
`java.lang.Object`	Any Java POJO (plain-old java object) with getter methods following JavaBean conventions; Legacy Java classes not following JavaBean conventions can also serve as events .
`java.util.Map`	Map events are key-values pairs
`org.w3c.dom.Node`	XML document object model (DOM)

2.2. Event Properties

Event properties capture the state information for an event. Event properties be simple as well as indexed, mapped and nested event properties. The table below outlines the different types of properties and their syntax in an event expression. This syntax allows statements to query deep JavaBean objects graphs, XML structures and Map events.

Table 2.2. Types of Event Properties

Type	Description	Syntax	Example
Simple	A property that has a single value that may be retrieved.	`name`	`sensorId`
Indexed	An indexed property stores an ordered collection of objects (all of the same type) that can be individually accessed by an integer-valued, non-negative index (or subscript).	`name[`*`index`*`]`	`sensor[0]`
Mapped	A mapped property stores a keyed collection of objects (all of the same type).	`name('`*`key`*`')`	`sensor('light')`
Nested	A nested property is a property that lives within another property of an event.	`name.nestedname`	`sensor.value`

Combinations are also possible. For example, a valid combination could be `per-`

son.address('home').street[0].

2.3. Dynamic Event Properties

Dynamic (unchecked) properties are event properties that need not be known at statement compilation time. Such properties are resolved during runtime.

The idea behind dynamic properties is that for a given underlying event representation we don't always know all properties in advance. An underlying event may have additional properties that are not known at statement compilation time, that we want to query on. The concept is especially useful for events that represent rich, object-oriented domain models.

The syntax of dynamic properties consists of the property name and a question mark. Indexed, mapped and nested properties can also be dynamic properties:

Table 2.3. Types of Event Properties

Type	Syntax
Dynamic Simple	`name?`
Dynamic Indexed	`name[index]?`
Dynamic Mapped	`name('key')?`
Dynamic Nested	`name?.nestedPropertyName`

Dynamic properties always return the `java.lang.Object` type. Also, dynamic properties return a `null` value if the dynamic property does not exist on events processed at runtime.

As an example, consider an OrderEvent event that provides an "item" property. The "item" property is of type `Object` and holds a reference to an instance of either a Service or Product.

Assume that both Service and Product classes provide a property named "price". Via a dynamic property we can specify a query that obtains the price property from either object (Service or Product):

```
select item.price? from OrderEvent
```

As a second example, assume that the Service class contains a "serviceName" property that the Product class does not possess. The following query returns the value of the "serviceName" property for Service objects. It returns a `null`-value for Product objects that do not have the "serviceName" property:

```
select item.serviceName? from OrderEvent
```

Consider the case where OrderEvent has multiple implementation classes, some of which have a "timestamp" property. The next query returns the timestamp property of those implementations of the OrderEvent interface that feature the property:

```
select timestamp? from OrderEvent
```

The query as above returns a single column named "timestamp?" of type `Object`.

When dynamic properties are nested, then all properties under the dynamic property are also considered dynamic properties. In the below example the query asks for the "direction" property of the object returned by the "detail" dynamic property:

```
select detail?.direction from OrderEvent
// equivalent to
select detail?.direction? from OrderEvent
```

The functions that are often useful in conjunction with dynamic properties are:

- The `cast` function casts the value of a dynamic property (or the value of an expression) to a given type.
- The `exists` function checks whether a dynamic property exists. It returns `true` if the event has a property of that name, or false if the property does not exist on that event.
- The `instanceof` function checks whether the value of a dynamic property (or the value of an expression) is of any of the given types.

Dynamic event properties work with all event representations outlined next: Java objects, Map-based and XML DOM-based events.

2.4. Plain-Old Java Object Events

Plain-old Java object events are object instances that expose event properties through JavaBeans-style getter methods. Events classes or interfaces do not have to be fully compliant to the JavaBean specification; however for the Esper engine to obtain event properties, the required JavaBean getter methods must be present.

Esper supports JavaBeans-style event classes that extend a superclass or implement one or more interfaces. Also, Esper event pattern and EPL statements can refer to Java interface classes and abstract classes.

Classes that represent events should be made immutable. As events are recordings of a state change or action that occurred in the past, the relevant event properties should not be changeable. However this is not a hard requirement and the Esper engine accepts events that are mutable as well.

The `hashCode` and `equals` methods do not need to be implemented. The implementation of these methods by a Java event class does not affect the behavior of the engine in any way.

Please see Chapter 10, *Configuration* on options for naming event types represented by Java object event classes.

2.4.1. Java Object Event Properties

As outlined earlier, the different property types are supported by the standard JavaBeans specification, and some of which are uniquely supported by Esper:

- *Simple* properties have a single value that may be retrieved. The underlying property type might be a Java language primitive (such as int, a simple object (such as a java.lang.String), or a more complex object whose class is defined either by the Java language, by the application, or by a class library included with the application.
- *Indexed* - An indexed property stores an ordered collection of objects (all of the same type) that can be indi-

vidually accessed by an integer-valued, non-negative index (or subscript). Alternatively, the entire set of values may be retrieved using an array.
- *Mapped* - As an extension to standard JavaBeans APIs, Esper considers any property that accepts a String-valued key a mapped property.
- *Nested* - A nested property is a property that lives within another Java object which itself is a property of an event.

Assume there is an EmployeeEvent event class as shown below. The mapped and indexed properties in this example return Java objects but could also return Java language primitive types (such as int or String). The Address object and Employee objects can themselves have properties that are nested within them, such as a streetName in the Address object or a name of the employee in the Employee object.

```
public class EmployeeEvent {
        public String getFirstName();
        public Address getAddress(String type);
        public Employee getSubordinate(int index);
        public Employee[] getAllSubordinates();
}
```

Simple event properties require a getter-method that returns the property value. In this example, the `getFirstName` getter method returns the `firstName` event property of type String.

Indexed event properties require either one of the following getter-methods. A method that takes an integer-type key value and returns the property value, such as the `getSubordinate` method. Or a method that returns an array-type such as the `getSubordinates` getter method, which returns an array of Employee. In an EPL or event pattern statement, indexed properties are accessed via the `property[index]` syntax.

Mapped event properties require a getter-method that takes a String-typed key value and returns the property value, such as the `getAddress` method. In an EPL or event pattern statement, mapped properties are accessed via the `property('key')` syntax.

Nested event properties require a getter-method that returns the nesting object. The `getAddress` and `getSubordinate` methods are mapped and indexed properties that return a nesting object. In an EPL or event pattern statement, nested properties are accessed via the `property.nestedProperty` syntax.

All event pattern and EPL statements allow the use of indexed, mapped and nested properties (or a combination of these) anywhere where one or more event property names are expected. The below example shows different combinations of indexed, mapped and nested properties in filters of event pattern expressions:

```
every EmployeeEvent(firstName='myName')
every EmployeeEvent(address('home').streetName='Park Avenue')
every EmployeeEvent(subordinate[0].name='anotherName')
every EmployeeEvent(allSubordinates[1].name='thatName')
every EmployeeEvent(subordinate[0].address('home').streetName='Water Street')
```

Similarly, the syntax can be used in EPL statements in all places where an event property name is expected, such as in select lists, where-clauses or join criteria.

```
select firstName, address('work'), subordinate[0].name, subordinate[1].name
from EmployeeEvent
where address('work').streetName = 'Park Ave'
```

Property names follows Java standards: the class `java.beans.Introspector` and method `getBeanInfo` returns the property names as derived from the name of getter methods. In addition, Esper configuration provides a flag to turn off case-sensitive property names. A sample list of getter methods and property names is:

Table 2.4. JavaBeans-style Getter Methods and Property Names

Method	Property Name	Example
`getPrice()`	price	`select price from MyEvent`
`getNAME()`	NAME	`select NAME from MyEvent`
`getItemDesc()`	itemDesc	`select itemDesc from MyEvent`
`getQ()`	q	`select q from MyEvent`
`getQN()`	QN	`select QN from MyEvent`
`getqn()`	qn	`select qn from MyEvent`
`gets()`	s	`select s from MyEvent`

Constants are public static final fields in Java that may also participate in expressions of all kinds, as this example shows:

```
select * from MyEvent where property=MyConstantClass.FIELD_VALUE
```

Event properties that are enumeration values can be compared by their enumeration value:

```
select * from MyEvent where enumProp=EnumClass.ENUM_VALUE_1
```

Alternatively, a static method may be employed on a class, such as the enumeration class 'EnumClass' as below:

```
select * from MyEvent where enumProp=EnumClass.valueOf('ENUM_VALUE_1')
```

Instance methods may also be invoked on event instances by specifying a stream name, as shown below:

```
select myevent.computeSomething() as result from MyEvent as myevent
```

Java classes that do not follow JavaBean conventions, such as legacy Java classes that expose public fields, or methods not following naming conventions, require additional configuration. Via configuration it is also possible to control case sensitivity in property name resolution. The relevant section in the chapter on configuration is Section 10.4.1.3, “Non-JavaBean and Legacy Java Event Classes”.

2.5. `java.util.Map` Events

Events can also be represented by objects that implement the `java.util.Map` interface. Event properties of `Map` events are the values in the map accessible through the `get` method exposed by the `java.util.Map` interface.

The engine can process `java.util.Map` events via the `sendEvent(Map map, String eventTypeAlias)` method on the `EPRuntime` interface. Entries in the Map represent event properties. Keys must be of type `java.util.String` for the engine to be able to look up event property names specified by pattern or EPL statements.

Map event properties can be of any type. Map event properties that are Java application objects or that are of type `java.util.Map` offer additional power:

- Properties that are Java application objects can be queried via the nested, indexed, mapped and dynamic property syntax as outlined earlier.
- Properties that are of type `Map` allow Maps to be nested arbitrarily deep and thus can be used to represent complex domain information. The nested, indexed, mapped and dynamic property syntax can be used to query Maps within Maps..

In order to use `Map` events, the event type name and property names and types must be made known to the engine via Configuration. Please see the examples in Section 10.4.2, “Events represented by java.util.Map”.

The code snippet below creates and processes a `Map` event. The example assumes the `CarLocationUpdateEvent` event type alias has been configured.

```
Map event = new HashMap();
event.put("carId", carId);
event.put("direction", direction);
epRuntime.sendEvent(event, "CarLocUpdateEvent");
```

The `CarLocUpdateEvent` can now be used in a statement:

```
select carId from CarLocUpdateEvent.win:time(1 min) where direction = 1
```

The engine can also query Java objects as values in a `Map` event via the nested property syntax. Thus `Map` events can be used to aggregate multiple data structures into a single event and query the composite information in a convenient way. The example below demonstrates a `Map` event with a transaction and an account object.

```
Map event = new HashMap();
event.put("txn", txn);
event.put("account", account);
epRuntime.sendEvent(event, "TxnEvent");
```

An example statement could look as follows.

```
select account.id, account.rate * txn.amount
from TxnEvent.win:time(60 sec)
group by account.id
```

2.5.1. Map-Within-Map Nested Events

Strongly-typed nested `Map`-within-`Map` events can be used to build rich, type-safe event types on the fly. Use the `addNestableEventTypeAlias` method on `Configuration` or `ConfigurationOperations` for initialization-time and runtime-time type definition.

Noteworthy points are:

- JavaBean (POJO) objects can also appear as properties in `Map`-within-`Map`.
- There is no limit to the number of nesting levels.
- Dynamic properties can be used to query `Map`-within-`Map` keys that may not be known in advance.

- The engine returns a `null` value for properties for which the access path into the nested structure cannot be followed where map entries do not exist.

For demonstration, in this example our top-level event type is an AccountUpdate event, which has an UpdatedField structure as a property. Inside the UpdatedField structure the example defines various fields, as well as a property by name 'history' that holds a JavaBean class 'UpdateHistory' to represent the update history for the account. The code snippet to define the event type is thus:

```
Map<String, Object> updatedFieldDef = new HashMap<String, Object>();
updatedFieldDef.put("name", String.class);
updatedFieldDef.put("addressLine1", String.class);
updatedFieldDef.put("history", UpdateHistory.class);

Map<String, Object> accountUpdateDef = new HashMap<String, Object>();
accountUpdateDef.put("accountId", long.class);
accountUpdateDef.put("fields", updatedFieldDef);

epService.getEPAdministrator().getConfiguration().
    addNestableEventTypeAlias("AccountUpdate", accountUpdateDef);
```

The next code snippet populates a sample event and sends the event into the engine:

```
Map<String, Object> updatedField = new HashMap<String, Object>();
updatedField.put("name", "Joe Doe");
updatedField.put("addressLine1", "40 Popular Street");
updatedField.put("history", new UpdateHistory());

Map<String, Object> accountUpdate = new HashMap<String, Object>();
accountUpdate.put("accountId", 10009901);
accountUpdate.put("fields", updatedField);

epService.getEPRuntime().sendEvent(accountUpdate, "AccountUpdate");
```

Last, a sample query to interrogate AccountUpdate events is as follows:

```
select accountId, fields.name, fields.addressLine1, fields.history.lastUpdate
from AccountUpdate
```

Note that type information for nested maps is only available to the immediately selecting stream. For example, the second select-query does not work:

```
insert into MyStream select fields from NestedMapEvent
// this does not work ... instead select the individual fields in the insert-into statement
select fields.name from MyStream
```

2.6. `org.w3c.dom.Node` XML Events

Events can also be represented as `org.w3c.dom.Node` instances and send into the engine via the `sendEvent` method on `EPRuntime`. Please note that configuration is required for allowing the engine to map the event type alias to `Node` element names. See Chapter 10, *Configuration*.

Esper allows configuring XPath expressions as event properties. You can specify arbitrary XPath functions or expressions and provide a property name by which their result values will be available for use in expressions. For XML documents that follow an XML schema, Esper can load and interrogate your schema and validate event property names and types against the schema information.

Nested, mapped and indexed event properties are also supported in expressions against `org.w3c.dom.Node` events. Thus XML trees can conveniently be interrogated using the existing event property syntax for querying

JavaBean objects, JavaBean object graphs or `java.util.Map` events.

Let's look at how a sample XML document could be queried, given the sample XML below.

```
<?xml version="1.0" encoding="UTF-8"?>
<Sensor>
	<ID>urn:epc:1:4.16.36<ID>
	<Observation Command="READ_PALLET_TAGS_ONLY">
		<ID>00000001<ID>
		<Tag>
			<ID>urn:epc:1:2.24.400<ID>
		</Tag>
		<Tag>
			<ID>urn:epc:1:2.24.401<ID>
		</Tag>
	</Observation>
</Sensor>
```

To configure the engine for processing Sensor documents, simply configure a `SensorEvent` event type alias for the `Sensor` element name via Configuration. Now the document can be queried as below.

```
select ID, Observation.ID, Observation.Command, Observation.Tag[0], countTags
from SensorEvent.win:time(30 sec)
```

The equivalent XPath expressions to each of the properties are listed below.

- The equivalent XPath expression to `Observeration.ID` is `/Sensor/Observation/ID`
- The equivalent XPath expression to `Observeration.Command` is `/Sensor/Observation/@Command`
- The equivalent XPath expression to `Observeration.Tag[0]` is `/Sensor/Observation/Tag[position() = 1]`
- The equivalent XPath expression to `countTags` is `count(/Sensor/Observation/Tag)` for returning a count of tag elements. This assumes the `countTags` property has been configured as an XPath property.

By specifying an event property such below:

```
nestedElement.mappedElement('key').indexedElement[1]
```

The equivalent XPath expression is as follows:

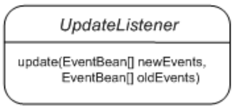
```
/simpleEvent/nestedElement/mappedElement[@id='key']/indexedElement[position() = 2]
```

Chapter 3. Processing Model

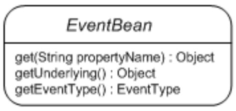
3.1. Introduction

The Esper processing model is continuous: Update listeners to statements receive updated data as soon as the engine processes events for that statement, according to the statement's choice of event streams, views, filters and output rates.

As outlined in Chapter 9, *API Reference* the interface for listeners is `com.espertech.esper.client.UpdateListener`. Implementations must provide a single `update` method that the engine invokes when results become available:



The engine provides statement results to update listeners by placing results in `com.espertech.esper.event.EventBean` instances. A typical listener implementation queries the `EventBean` instances via getter methods to obtain the statement-generated results.



The `get` method on the `EventBean` interface can be used to retrieve result columns by name. The property name supplied to the `get` method can also be used to query nested, indexed or array properties of object graphs as discussed in more detail in Chapter 2, *Event Representations*.

The `getUnderlying` method on the `EventBean` interface allows update listeners to obtain the underlying event object. For wildcard selects, the underlying event is the event object that was sent into the engine via the `sendEvent` method. For joins and select clauses with expressions, the underlying object implements `java.util.Map`.

3.2. Insert Stream

In this section we look at the output of a very simple EPL statement. The statement selects an event stream without using a data window and without applying any filtering, as follows:

```
select * from Withdrawal
```

This statement selects all `Withdrawal` events. Every time the engine processes an event of type `Withdrawal` or any sub-type of `Withdrawal`, it invokes all update listeners, handing the new event to each of the statement's listeners.

The term *insert stream* denotes the new events arriving, and entering a data window or aggregation. The insert stream in this example is the stream of arriving Withdrawal events, and is posted to listeners as new events.

The diagram below shows a series of Withdrawal events 1 to 6 arriving over time. The number in parenthesis is the withdrawal amount, an event property that is used in the examples that discuss filtering.

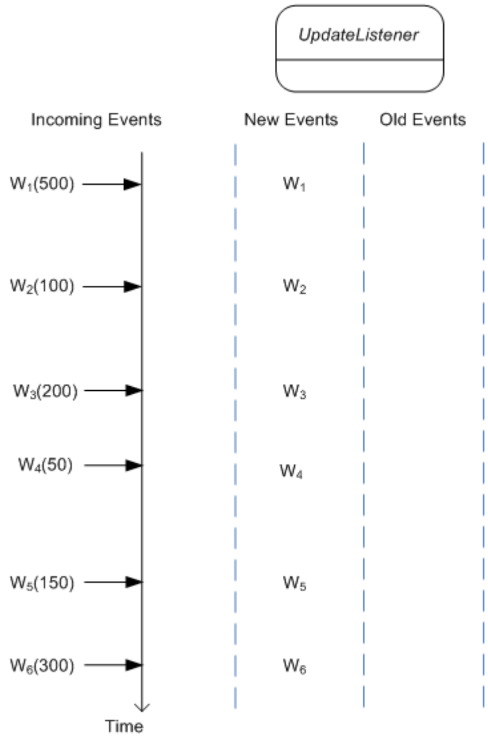


Figure 3.1. Output example for a simple statement

The example statement above results in only new events and no old events posted by the engine to the statement's listeners.

3.3. Insert and Remove Stream

A length window instructs the engine to only keep the last N events for a stream. The next statement applies a length window onto the Withdrawal event stream. The statement serves to illustrate the concept of data window and events entering and leaving a data window:

```
select * from Withdrawal.win:length(5)
```

The size of this statement's length window is five events. The engine enters all arriving Withdrawal events into the length window. When the length window is full, the oldest Withdrawal event is pushed out the window. The engine indicates to listeners all events entering the window as new events, and all events leaving the window as old events.

While the term *insert stream* denotes new events arriving, the term *remove stream* denotes events leaving a data window, or changing aggregation values. In this example, the remove stream is the stream of Withdrawal events that leave the length window, and such events are posted to listeners as old events.

The next diagram illustrates how the length window contents change as events arrive and shows the events posted to an update listener.

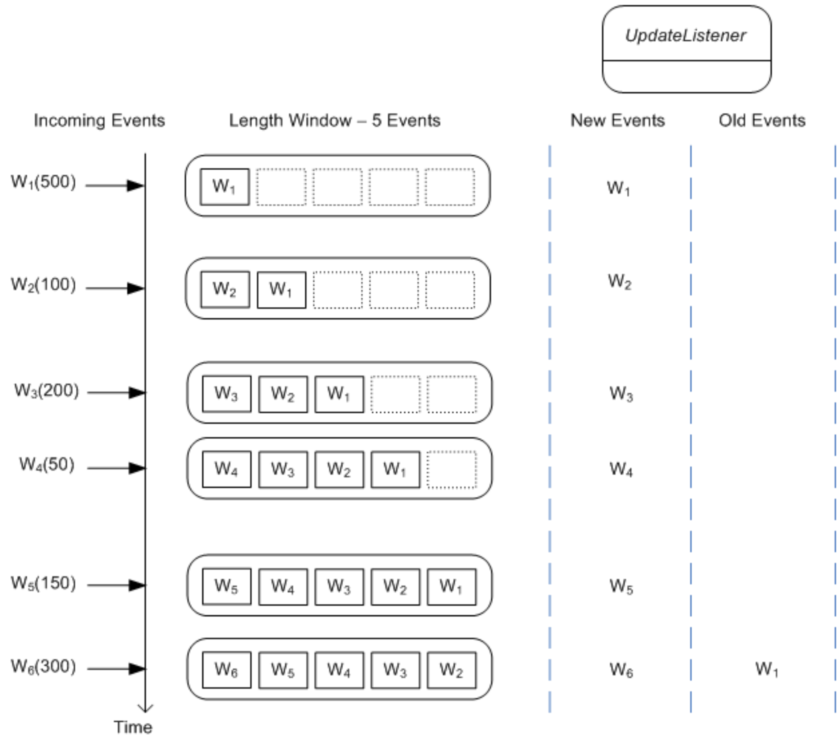


Figure 3.2. Output example for a length window

As before, all arriving events are posted as new events to listeners. In addition, when event W_1 leaves the length window on arrival of event W_6, it is posted as an old event to listeners.

Similar to a length window, a time window also keeps the most recent events up to a given time period. A time window of 5 seconds, for example, keeps the last 5 seconds of events. As seconds pass, the time window actively pushes the oldest events out of the window resulting in one or more old events posted to update listeners.

Note: By default the engine only delivers the insert stream to listeners and observers. EPL supports optional `istream`, `irstream` and `rstream` keywords on select-clauses and on insert-into clauses to control which stream to deliver, see Section 4.3.7, “Selecting insert and remove stream events”. There is also a related, engine-wide configuration setting described in Section 10.4.13, “Engine Settings related to Stream Selection”.

3.4. Filters and Where-clauses

Filters to event streams allow filtering events out of a given stream before events enter a data window. The statement below shows a filter that selects Withdrawal events with an amount value of 200 or more.

```
select * from Withdrawal(amount>=200).win:length(5)
```

With the filter, any Withdrawal events that have an amount of less then 200 do not enter the length window and are therefore not passed to update listeners. Filters are discussed in more detail in Section 4.4.1, “Filter-based Event Streams” and Section 5.4, “Filter Expressions In Patterns”.

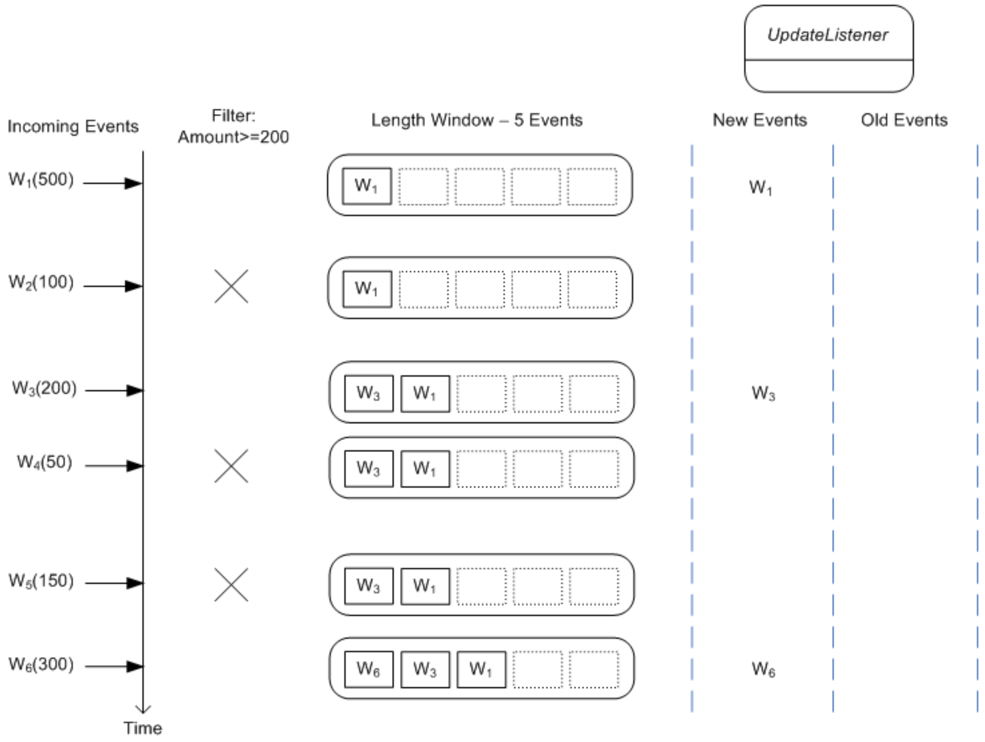


Figure 3.3. Output example for a statement with an event stream filter

The where-clause and having-clause in statements eliminate potential result rows at a later stage in processing, after events have been processed into a statement's data window or other views.

The next statement applies a where-clause to Withdrawal events. Where-clauses are discussed in more detail in Section 4.5, “Specifying Search Conditions: the Where Clause”.

```
select * from Withdrawal.win:length(5) where amount >= 200
```

The where-clause applies to both new events and old events. As the diagram below shows, arriving events enter the window however only events that pass the where-clause are handed to update listeners. Also, as events leave the data window, only those events that pass the conditions in the where-clause are posted to listeners as old events.

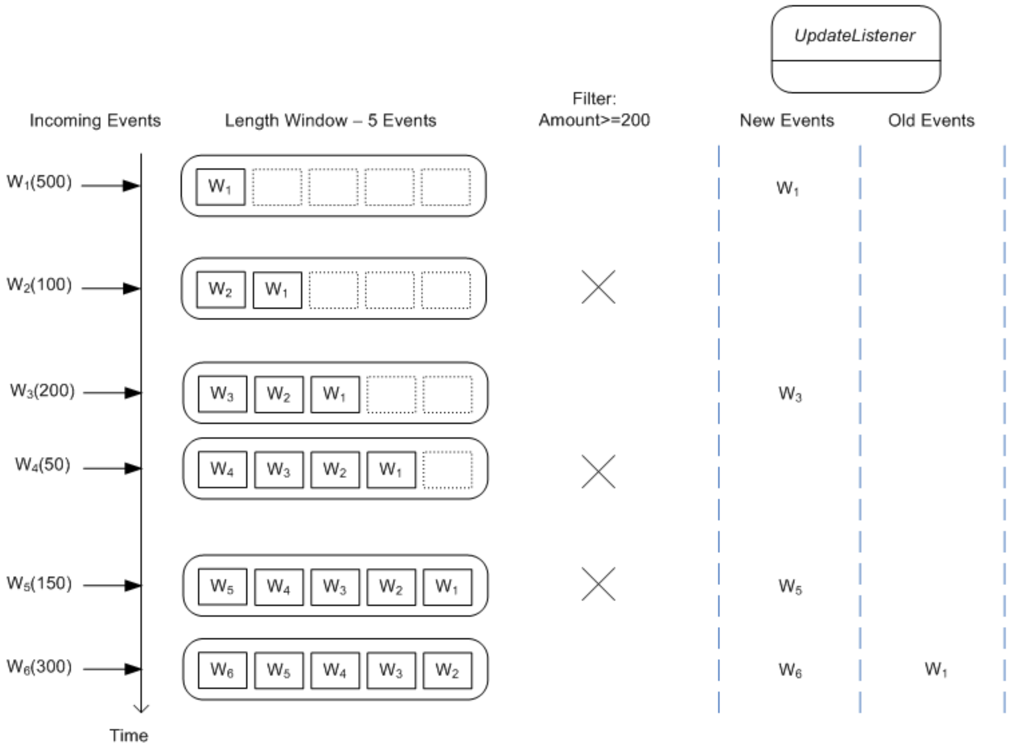


Figure 3.4. Output example for a statement with where-clause

The where-clause can contain complex conditions while event stream filters are more restrictive in the type of filters that can be specified. The next statement's where-clause applies the `ceil` function of the `java.lang.Math` Java library class in the where clause. The insert-into clause makes the results of the first statement available to the second statement:

```
insert into WithdrawalFiltered select * from Withdrawal where Math.ceil(amount) >= 200

select * from WithdrawalFiltered
```

3.5. Time Windows

In this section we explain the output model of statements employing a time window view and a time batch view.

3.5.1. Time Window

A time window is a moving window extending to the specified time interval into the past based on the system time. Time windows enable us to limit the number of events considered by a query, as do length windows.

As a practical example, consider the need to determine all accounts where the average withdrawal amount per account for the last 4 seconds of withdrawals is greater then 1000. The statement to solve this problem is shown below.

```
select account, avg(amount)
from Withdrawal.win:time(4 sec)
group by account
having amount > 1000
```

The next diagram serves to illustrate the functioning of a time window. For the diagram, we assume a query that simply selects the event itself and does not group or filter events.

```
select * from Withdrawal.win:time(4 sec)
```

The diagram starts at a given time `t` and displays the contents of the time window at `t + 4` and `t + 5 seconds` and so on.

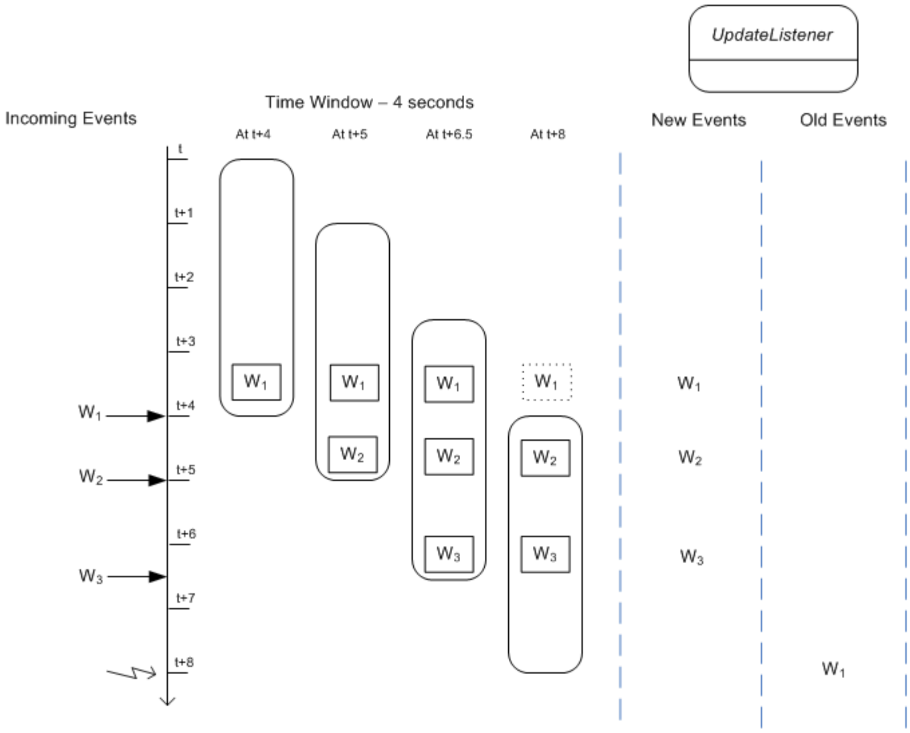


Figure 3.5. Output example for a statement with a time window

The activity as illustrated by the diagram:

1. At time `t + 4 seconds` an event W_1 arrives and enters the time window. The engine reports the new event to update listeners.
2. At time `t + 5 seconds` an event W_2 arrives and enters the time window. The engine reports the new event to update listeners.
3. At time `t + 6.5 seconds` an event W_3 arrives and enters the time window. The engine reports the new event to update listeners.
4. At time `t + 8 seconds` event W_1 leaves the time window. The engine reports the event as an old event to update listeners.

3.5.2. Time Batch

The time batch view buffers events and releases them every specified time interval in one update. Time windows control the evaluation of events, as does the length batch window.

The next diagram serves to illustrate the functioning of a time batch view. For the diagram, we assume a simple query as below:

```
select * from Withdrawal.win:time_batch(4 sec)
```

The diagram starts at a given time `t` and displays the contents of the time window at `t + 4` and `t + 5 seconds` and so on.

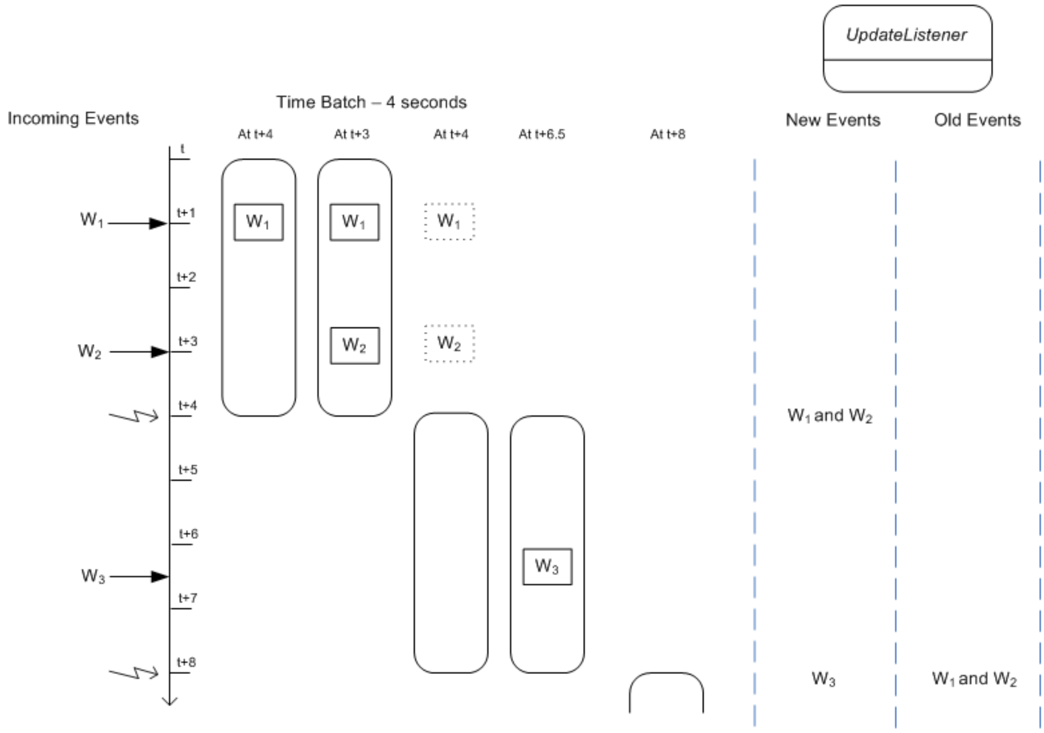


Figure 3.6. Output example for a statement with a time batch view

The activity as illustrated by the diagram:

1. At time `t + 1 seconds` an event W_1 arrives and enters the batch. No call to inform update listeners occurs.
2. At time `t + 3 seconds` an event W_2 arrives and enters the batch. No call to inform update listeners occurs.
3. At time `t + 4 seconds` the engine processes the batched events and a starts a new batch. The engine reports events W_1 and W_2 to update listeners.
4. At time `t + 6.5 seconds` an event W_3 arrives and enters the batch. No call to inform update listeners occurs.
5. At time `t + 8 seconds` the engine processes the batched events and a starts a new batch. The engine reports the event W_3 as new data to update listeners. The engine reports the events W_1 and W_2 as old data (prior batch) to update listeners.

3.6. Batch Windows

The built-in data windows that act on batches of events are the `win:time_batch` and the `win:length_batch` views. The `win:time_batch` data window collects events arriving during a given time interval and posts collected events as a batch to listeners at the end of the time interval. The `win:length_batch` data window collects a given number of events and posts collected events as a batch to listeners when the given number of events has collected.

Let's look at how a time batch window may be used:

```
select account, amount from Withdrawal.win:time_batch(1 sec)
```

The above statement collects events arriving during a one-second interval, at the end of which the engine posts the collected events as new events (insert stream) to each listener. The engine posts the events collected during the prior batch as old events (remove stream). The engine starts posting events to listeners one second after it receives the first event and thereon.

For statements containing aggregation functions and/or a `group by` clause, the engine posts consolidated aggregation results for an event batch. For example, consider the following statement:

```
select sum(amount) as mysum from Withdrawal.win:time_batch(1 sec)
```

Note that output rate limiting also generates batches of events following the output model as discussed here.

3.7. Aggregation and Grouping

3.7.1. Insert and Remove Stream

Statements that aggregate events via aggregation functions also post remove stream events as aggregated values change.

Consider the following statement that alerts when 2 Withdrawal events have been received:

```
select count(*) as mycount from Withdrawal having count(*) = 2
```

When the engine encounters the second withdrawal event, the engine posts a new event to update listeners. The value of the "mycount" property on that new event is 2. Additionally, when the engine encounters the third Withdrawal event, it posts an old event to update listeners containing the prior value of the count. The value of the "mycount" property on that old event is also 2.

The `istream` or `rstream` keyword can be used to eliminate either new events or old events posted to listeners. The next statement uses the `istream` keyword causing the engine to call the listener only once when the second Withdrawal event is received:

```
select istream count(*) as mycount from Withdrawal having count(*) = 2
```

3.7.2. Output for Aggregation and Group-By

Following SQL (Standard Query Language) standards for queries against relational databases, the presence or absence of aggregation functions and the presence or absence of the `group by` clause dictates the number of rows posted by the engine to listeners. The next sections outline the output model for batched events under aggregation and grouping. The examples also apply to data windows that don't batch events and post results continously as events arrive or leave data windows. The examples also apply to patterns providing events when a complete pattern matches.

In summary, as in SQL, if your query only selects aggregation values, the engine provides one row of aggregated values. It provides that row every time the aggregation is updated (insert stream), which is when events arrive or a batch of events gets processed, and when the events leave a data window or a new batch of events arrives. The remove stream then consists of prior aggregation values.

Also as in SQL, if your query selects non-aggregated values along with aggregation values in the select clause, the engine provides a row per event. The insert stream then consists of the aggregation values at the time the event arrives, while the remove stream is the aggregation value at the time the event leaves a data window, if any is defined in your query.

The documentation provides output examples for query types in Appendix A, *Output Reference and Samples*, and the next sections outlines each query type.

Un-aggregated and Un-grouped

An example statement for the un-aggregated and un-grouped case is as follows:

```
select * from Withdrawal.win:time_batch(1 sec)
```

At the end of a time interval, the engine posts to listeners one row for each event arriving during the time interval.

The appendix provides a complete example including input and output events over time at Section A.2, “Output for Un-aggregated and Un-grouped Queries”

Fully Aggregated and Un-grouped

If your statement only selects aggregation values and does not group, your statement may look as the example below:

```
select sum(amount)
from Withdrawal.win:time_batch(1 sec)
```

At the end of a time interval, the engine posts to listeners a single row indicating the aggregation result. The aggregation result aggregates all events collected during the time interval.

The appendix provides a complete example including input and output events over time at Section A.3, “Output for Fully-aggregated and Un-grouped Queries”

Aggregated and Un-Grouped

If your statement selects non-aggregated properties and aggregation values, and does not group, your statement may be similar to this statement:

```
select account, sum(amount)
from Withdrawal.win:time_batch(1 sec)
```

At the end of a time interval, the engine posts to listeners one row per event. The aggregation result aggregates all events collected during the time interval.

The appendix provides a complete example including input and output events over time at Section A.4, “Output for Aggregated and Un-grouped Queries”

Fully Aggregated and Grouped

If your statement selects aggregation values and all non-aggregated properties in the `select` clause are listed in the `group by` clause, then your statement may look similar to this example:

```
select account, sum(amount)
from Withdrawal.win:time_batch(1 sec)
group by account
```

At the end of a time interval, the engine posts to listeners one row per unique account number. The aggregation result aggregates per unique account.

The appendix provides a complete example including input and output events over time at Section A.5, “Output for Fully-aggregated and Grouped Queries”

Aggregated and Grouped

If your statement selects non-aggregated properties and aggregation values, and groups only some properties using the `group by` clause, your statement may look as below:

```
select account, accountName, sum(amount)
from Withdrawal.win:time_batch(1 sec)
group by account
```

At the end of a time interval, the engine posts to listeners one row per event. The aggregation result aggregates per unique account.

The appendix provides a complete example including input and output events over time at Section A.6, “Output for Aggregated and Grouped Queries”

3.8. `EventBean` Query Results

The engine posts events to `UpdateListener` implementations as `com.espertech.esper.event.EventBean` instances. The `EventBean` represents a row (event) in your continuous query's result set.

Use the `iterator` method on `EPStatement` statements to poll or read data out of statements, if you require read-based access to statement result sets. Statement iterators also return `EventBean` instances.

The `EventBean` interface offers property type metadata via the `getEventType` method returning an `EventType`. The `EventType` provides property name, property type and underlying type information. This information can be useful to dynamically interrogate query results. The underlying event that an `EventBean` represents can be obtained via the `getUnderlying` method. Please see Chapter 2, *Event Representations* for more information on different event underlying objects.

Consider a statement that returns the symbol, count of events per symbol and average price per symbol for tick events. Our sample statement may declare a fully-qualified Java class name as the event type: `org.sample.StockTickEvent`. Assume that this class exists and exposes a `symbol` property of type String, and a `price` property of type (Java primitive) double.

```
select symbol, avg(price) as avgprice, count(*) as mycount
from org.sample.StockTickEvent
group by symbol
```

The next table summarizes the property names and types as posted by the statement above:

Table 3.1. Properties offered by sample statement aggregating price

Name	Type	Description	Java code snippet
`symbol`	java.lang.String	Value of symbol event property	`eventBean.get("symbol")`
`avgprice`	java.lang.Double	Average price per symbol	`eventBean.get("avgprice")`
`mycount`	java.lang.Long	Number of events per symbol	`eventBean.get("mycount")`

A code snippet out of a possible `UpdateListener` implementation to this statement may look as below:

```
String symbol = (String) newEvents[0].get("symbol");
Double price= (Double) newEvents[0].get("avgprice");
Long count= (Long) newEvents[0].get("mycount");
```

The engine supplies the boxed `java.lang.Double` and `java.lang.Long` types as property values rather then primitive Java types. This is because aggregated values can return a `null` value to indicate that no data is available for aggregation. Also, in a select statement that computes expressions, the underlying event objects to `EventBean` instances are of type `java.util.Map`.

Consider the next statement that specifies a wildcard selecting the same type of event:

```
select * from org.sample.StockTickEvent where price > 100
```

The property names and types provided by an `EventBean` query result row, as posted by the statement above are as follows:

Table 3.2. Properties offered by sample wildcard-select statement

Name	Type	Description	Java code snippet
`symbol`	java.lang.String	Value of symbol event property	`eventBean.get("symbol")`
`price`	double	Value of price event property	`eventBean.get("price")`

As an alternative to querying individual event properties via the `get` methods, the `getUnderlying` method on `EventBean` returns the underlying object representing the query result. In the sample statement that features a wildcard-select, the underlying event object is of type `org.sample.StockTickEvent`:

```
StockTickEvent tick = (StockTickEvent) newEvents[0].getUnderlying();
```

Chapter 4. EPL Reference: Clauses

4.1. EPL Introduction

The Event Processing Language (EPL) is a SQL-like language with `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING` and `ORDER BY` clauses. Streams replace tables as the source of data with events replacing rows as the basic unit of data. Since events are composed of data, the SQL concepts of correlation through joins, filtering and aggregation through grouping can be effectively leveraged. The `INSERT INTO` clause is recast as a means of forwarding events to other streams for further downstream processing. External data accessible through JDBC may be queried and joined with the stream data. Additional clauses such as the `PATTERN` and `OUTPUT` clauses are also available to provide the missing SQL language constructs specific to event processing.

EPL statements are used to derive and aggregate information from one or more streams of events, and to join or merge event streams. This section outlines EPL syntax. It also outlines the built-in views, which are the building blocks for deriving and aggregating information from event streams.

EPL statements contain definitions of one or more views. Similar to tables in a SQL statement, views define the data available for querying and filtering. Some views represent windows over a stream of events. Other views derive statistics from event properties, group events or handle unique event property values. Views can be staggered onto each other to build a chain of views. The Esper engine makes sure that views are reused among EPL statements for efficiency.

The built-in set of views is:

1. Data window views: `win:length`, `win:length_batch`, `win:time`, `win:time_batch`, `win:time_length_batch`, `win:time_accum`, `win:ext_timed`, `ext:sort_window`, `ext:time_order`, `std:unique`, `std:groupby`, `std:lastevent`
2. Views that derive statistics: `std:size`, `stat:uni`, `stat:linest`, `stat:correl`, `stat:weighted_avg`, `stat:cube`

EPL provides the concept of *named window*. Named windows are data windows that can be inserted-into and deleted-from by one or more statements, and that can queried by one or more statements. Named windows have a global character, being visible and shared across an engine instance beyond a single statement. Use the `CREATE WINDOW` clause to create named windows. Use the `INSERT INTO` clause to insert data into a named window, the `ON DELETE` clause to remove events from a named window, and the `ON SELECT` clause to perform a non-continuous fire-once query on a named window. Finally, the name of the named window can occur in a statement's `FROM` clause to query a named window or include the named window in a join or subquery.

Variables can come in handy to parameterize statements and change parameters on-the-fly and in response to events. Variables can be used in an expression anywhere in a statement as well as in the output clause for dynamic control of output rates.

Esper can be extended by plugging-in custom developed views and aggregation functions.

4.2. EPL Syntax

EPL queries are created and stored in the engine, and publish results to listeners as events are received by the engine or timer events occur that match the criteria specified in the query. Events can also be obtained from running EPL queries via the `safeIterator` and `iterator` methods that provide a pull-data API.

The `select` clause in an EPL query specifies the event properties or events to retrieve. The `from` clause in an EPL query specifies the event stream definitions and stream names to use. The `where` clause in an EPL query specifies search conditions that specify which event or event combination to search for. For example, the following statement returns the average price for IBM stock ticks in the last 30 seconds.

```
select avg(price) from StockTick.win:time(30 sec) where symbol='IBM'
```

EPL queries follow the below syntax. EPL queries can be simple queries or more complex queries. A simple select contains only a `select` clause and a single stream definition. Complex EPL queries can be build that feature a more elaborate select list utilizing expressions, may join multiple streams, may contain a `where` clause with search conditions and so on.

```
[insert into insert_into_def]
select select_list
from stream_def [as name] [, stream_def [as name]] [,...]
[where search_conditions]
[group by grouping_expression_list]
[having grouping_search_conditions]
[output output_specification]
[order by order_by_expression_list]
```

4.2.1. Specifying Time Periods

Time-based windows as well as pattern observers and guards take a time period as a parameter. Time periods follow the syntax below.

```
time-period : [day-part] [hour-part] [minute-part] [seconds-part] [milliseconds-part]

day-part : number ("days" | "day")
hour-part : number ("hours" | "hour")
minute-part : number ("minutes" | "minute" | "min")
seconds-part : number ("seconds" | "second" | "sec")
milliseconds-part : number ("milliseconds" | "millisecond" | "msec")
```

Some examples of time periods are:

```
10 seconds
10 minutes 30 seconds
20 sec 100 msec
1 day 2 hours 20 minutes 15 seconds 110 milliseconds
0.5 minutes
```

4.2.2. Using Comments

Comments can appear anywhere in the EPL or pattern statement text where whitespace is allowed. Comments can be written in two ways: slash-slash (`// ...`) comments and slash-star (`/* ... */`) comments.

Slash-slash comments extend to the end of the line:

```
// This comment extends to the end of the line.
// Two forward slashes with no whitespace between them begin such comments.

select * from MyEvent  // this is a slash-slash comment

// All of this text together is a valid statement.
```

Slash-star comments can span multiple lines:

```
/* This comment is a "slash-star" comment that spans multiple lines.
 * It begins with the slash-star sequence with no space between the '/' and '*' characters.
 * By convention, subsequent lines can begin with a star and are aligned, but this is
 * not required.
 */
select * from MyEvent  /* this also works */
```

Comments styles can also be mixed:

```
select field1, // first comment
  /* second comment*/  field2
  from MyEvent
```

4.3. Choosing Event Properties And Events: the *Select* Clause

The `select` clause is required in all EPL statements. The `select` clause can be used to select all properties via the wildcard `*`, or to specify a list of event properties and expressions. The `select` clause defines the event type (event property names and types) of the resulting events published by the statement, or pulled from the statement via the iterator methods.

The `select` clause also offers optional `istream`, `irstream` and `rstream` keywords to control whether input stream, remove stream or input and remove stream events are posted to `UpdateListener` instances and observers to a statement. By default, the engine provides only the insert stream to listener and observers. See Section 10.4.13, "Engine Settings related to Stream Selection" on how to change the default.

The syntax for the `select` clause is summarized below.

```
select [istream | irstream | rstream] * | expression_list ...
```

The `istream` keyword is the default, and indicates that the engine only delivers insert stream events to listeners and observers. The `irstream` keyword indicates that the engine delivers both insert and remove stream. Finally, the `rstream` keyword tells the engine to deliver only the remove stream.

4.3.1. Choosing all event properties: select *

The syntax for selecting all event properties in a stream is:

```
select * from stream_def
```

The following statement selects StockTick events for the last 30 seconds of IBM stock ticks.

```
select * from StockTick(symbol='IBM').win:time(30 sec)
```

The `*` wildcard and expressions can also be combined in a `select` clause. The combination selects all event properties and in addition the computed values as specified by any additional expressions that are part of the `select` clause. Here is an example that selects all properties of stock tick events plus a computed product of price and volume that the statement names 'pricevolume':

```
select *, price * volume as pricevolume from StockTick(symbol='IBM')
```

When using wildcard (*), Esper does not actually copy your event properties out of your event or events. It simply wraps your native type in an `EventBean` interface. Your application has access to the underlying event object through the `getUnderlying` method and has access to the property values through the `get` method.

In a join statement, using the `select *` syntax selects one event property per stream to hold the event for that stream. The property name is the stream alias name in the `from` clause.

4.3.2. Choosing specific event properties

To choose the particular event properties to return:

```
select event_property [, event_property] [, ...] from stream_def
```

The following statement simply selects the symbol and price properties of stock ticks, and the total volume for stock tick events in a 60-second time window.

```
select symbol, price, sum(volume) from StockTick(symbol='IBM').win:time(60 sec)
```

The following statement declares a further view onto the event stream of stock ticks: the univariate statistics view (`stat:uni`). The statement selects the properties that this view derives from the stream, for the last 100 events of IBM stock ticks in the length window.

```
select datapoints, total, average, variance, stdev, stdevpa
from StockTick(symbol='IBM').win:length(100).stat:uni(volume)
```

4.3.3. Expressions

The `select` clause can contain one or more expressions.

```
select expression [, expression] [, ...] from stream_def
```

The following statement selects the volume multiplied by price for a time batch of the last 30 seconds of stock tick events.

```
select volume * price from StockTick.win:time_batch(30 sec)
```

4.3.4. Renaming event properties

Event properties and expressions can be renamed using below syntax.

```
select [event property | expression] as identifier [, ...]
```

The following statement selects volume multiplied by price and specifies the name *volPrice* for the event property.

```
select volume * price as volPrice from StockTick.win:length(100)
```

4.3.5. Choosing event properties and events in a join

If your statement is joining multiple streams, your may specify property names that are unique among the joined streams, or use wildcard (*) as explained earlier.

In case the property name in your `select` or other clauses is not unique considering all joined streams, you will need to use the alias name of the stream as a prefix to the property.

This example is a join between the two streams StockTick and News, respectively named as 'tick' and 'news'. The example selects from the StockTick event the symbol value using the 'tick' stream alias as a prefix:

```
select tick.symbol from StockTick.win:time(10) as tick, News.win:time(10) as news
```

Use the wildcard (*) selector in a join to generate a property for each stream, with the property value being the event itself. The output events of the statement below have two properties: the 'tick' property holds the StockTick event and the 'news' property holds the News event:

```
select * from StockTick.win:time(10) as tick, News.win:time(10) as news
```

The following syntax can also be used to specify what stream's properties to select:

```
select stream_name.* [as alias] from ...
```

The selection of `tick.*` selects the StockTick stream events only:

```
select tick.* from StockTick.win:time(10) as tick, News.win:time(10) as news
where tick.symbol = news.symbol
```

The next example uses the `as` keyword to name each stream's joined events. This instructs the engine to create a property for each named event:

```
select tick.* as stocktick, news.* as news
from StockTick.win:time(10) as tick, News.win:time(10) as news
where stock.symbol = news.symbol
```

The output events of the above example have two properties 'stocktick' and 'news' that are the StockTick and News events.

4.3.6. Choosing event properties and events from a pattern

If your statement employs pattern expressions, then your pattern expression tags events with a tag name. Each tag name becomes available for use as a property in the `select` clause and all other clauses.

For example, here is a very simple pattern that matches on every StockTick event received within 30 seconds after start of the statement. The sample selects the symbol and price properties of the matching events:

```
select tick.symbol as symbol, tick.price as price
from pattern[every tick=StockTick where timer:within(10 sec)]
```

The use of the wildcard selector, as shown in the next statement, creates a property for each tagged event in the output. The next statement outputs events that hold a single 'tick' property whose value is the event itself:

```
select * from pattern[every tick=StockTick where timer:within(10 sec)]
```

You may also select the matching event itself using the `tick.*` syntax. The engine outputs the StockTick event itself to listeners:

```
select tick.* from pattern[every tick=StockTick where timer:within(10 sec)]
```

4.3.7. Selecting `insert` and `remove` stream events

The optional `istream`, `irstream` and `rstream` keywords in the `select` clause control the event streams posted to listeners and observers to a statement.

If neither keyword is specified, and in the default engine configuration, the engine posts only insert stream events via the `newEvents` parameter to the `update` method of `UpdateListener` instances listening to the statement. The engine does not post remove stream events, by default.

The insert stream consists of the events entering the respective window(s) or stream(s) or aggregations, while the remove stream consists of the events leaving the respective window(s) or the changed aggregation result. See Chapter 3, *Processing Model* for more information on insert and remove streams.

The engine posts remove stream events to the `oldEvents` parameter of the `update` method only if either the `irstream` or the `rstream` keyword occurs in the `select` clause. This behavior can be changed via engine-wide configuration as described in Section 10.4.13, “Engine Settings related to Stream Selection”.

By specifying the `istream` keyword you can instruct the engine to only post insert stream events via the `newEvents` parameter to the `update` method on listeners. The engine will then not post any remove stream events, and the `oldEvents` parameter is always a null value.

By specifying the `irstream` keyword you can instruct the engine to post both insert stream and remove stream events.

By specifying the `rstream` keyword you can instruct the engine to only post remove stream events via the `newEvents` parameter to the `update` method on listeners. The engine will then not post any insert stream events, and the `oldEvents` parameter is also always a null value.

The following statement selects only the events that are leaving the 30 second time window.

```
select rstream * from StockTick.win:time(30 sec)
```

The `istream` and `rstream` keywords in the `select` clause are matched by same-name keywords available in the `insert into` clause. While the keywords in the `select` clause control the event stream posted to listeners to the statement, the same keywords in the `insert into` clause specify the event stream that the engine makes available to other statements.

4.4. Specifying Event Streams: the *From* Clause

The `from` clause is required in all EPL statements. It specifies one or more event streams or named windows. Each event stream or named window can optionally be given a name by means of the `as` syntax.

```
from stream_def [as name] [, stream_def [as stream_name]] [, ...]
```

The event stream definition *stream_def* as shown in the syntax above can consists of either a filter-based event stream definition or a pattern-based event stream definition.

For joins and outer joins, specify two or more event streams. Joins between pattern-based and filter-based event streams are also supported. Joins are described in more detail in Section 4.10, “Joining Event Streams”.

Esper supports joins against relational databases for access to historical or reference data as explained in Section 4.14, “Joining Relational Data via SQL”. Esper can also join results returned by an arbitrary method invocation, as discussed in Section 4.15, “Joining Non-Relational Data via Method Invocation”.

The *stream_name* is an optional identifier assigned to the stream. The stream name can itself occur in any expression and provides access to the event itself from the named stream. Also, a stream name may be combined with a method name to invoke instance methods on events of that stream.

4.4.1. Filter-based Event Streams

For filter-based event streams, the event stream definition *stream_def* as shown in the `from` clause syntax consists of an event type, optional filter expressions and an optional list of views that derive data from a stream. The syntax for a filter-based event stream is as below:

```
event_type ( [filter_criteria] ) [.view_spec] [.view_spec] [...]
```

The following EPL statement shows event type, filter criteria and views combined in one statement. It selects all event properties for the last 100 events of IBM stock ticks for volume. In the example, the event type is the fully qualified Java class name `org.esper.example.StockTick`. The expression filters for events where the property `symbol` has a value of "IBM". The optional view specifications for deriving data from the StockTick events are a length window and a view for computing statistics on volume. The name for the event stream is "volumeStats".

```
select * from
  org.esper.example.StockTick(symbol='IBM').win:length(100).stat:uni(volume) as volumeStats
```

Esper filters out events in an event stream as defined by filter criteria before it sends events to subsequent views. Thus, compared to search conditions in a `where` clause, filter criteria remove unneeded events early. In the above example, events with a symbol other then IBM do not enter the time window.

Specifying an event type

The simplest form of filter is a filter for events of a given type without any conditions on the event property values. This filter matches any event of that type regardless of the event's properties. The example below is such a filter.

```
select * from com.mypackage.myevents.RfidEvent
```

Instead of the fully-qualified Java class name any other event name can be mapped via Configuration to a Java class, making the resulting statement more readable:

```
select * from RfidEvent
```

Interfaces and superclasses are also supported as event types. In the below example `IRfidReadable` is an interface class.

```
select * from org.myorg.rfid.IRfidReadable
```

Specifying filter criteria

The filtering criteria to filter for events with certain event property values are placed within parenthesis after the event type name:

```
select * from RfidEvent(category="Perishable")
```

All expressions can be used in filters, including static methods that return a boolean value:

```
select * from com.mycompany.RfidEvent(MyRFIDLib.isInRange(x, y) or (x < 0 and y < 0))
```

Filter expressions can be separated via a single comma ','. The comma represents a logical AND between filter expressions:

```
select * from RfidEvent(zone=1, category=10)
...is equivalent to...
select * from RfidEvent(zone=1 and category=10)
```

The following operators are highly optimized through indexing and are the preferred means of filtering in high-volume event streams:

- equals `=`
- not equals `!=`
- comparison operators `< , > , >=, <=`
- ranges
  - use the `between` keyword for a closed range where both endpoints are included
  - use the `in` keyword and round `()` or square brackets `[]` to control how endpoints are included
  - for inverted ranges use the `not` keyword and the `between` or `in` keywords
- list-of-values checks using the `in` keyword or the `not in` keywords followed by a comma-separated list of values

At compile time as well as at run time, the engine scans new filter expressions for sub-expressions that can be indexed. Indexing filter values to match event properties of incoming events enables the engine to match incoming events faster. The above list of operators represents the set of operators that the engine can best convert into indexes. The use of comma or logical `and` in filter expressions does not impact optimizations by the engine.

Filtering Ranges

Ranges come in the following 4 varieties. The use of round `()` or square `[]` bracket dictates whether an endpoint is included or excluded. The low point and the high-point of the range are separated by the colon `:` character.

- Open ranges that contain neither endpoint `(low:high)`
- Closed ranges that contain both endpoints `[low:high]`. The equivalent 'between' keyword also defines a closed range.
- Half-open ranges that contain the low endpoint but not the high endpoint `[low:high)`
- Half-closed ranges that contain the high endpoint but not the low endpoint `(low:high]`

The next statement shows a filter specifying a range for `x` and `y` values of RFID events. The range includes both endpoints therefore uses `[]` hard brackets.

```
mypackage.RfidEvent(x in [100:200], y in [0:100])
```

The `between` keyword is equivalent for closed ranges. The same filter using the `between` keyword is:

```
mypackage.RfidEvent(x between 100 and 200, y between 0 and 50)
```

The `not` keyword can be used to determine if a value falls outside a given range:

```
mypackage.RfidEvent(x not in [0:100])
```

The equivalent statement using the `between` keyword is:

```
mypackage.RfidEvent(x not between 0 and 100)
```

Filtering Sets of Values

The `in` keyword for filter criteria determines if a given value matches any value in a list of values.

In this example we are interested in RFID events where the category matches any of the given values:

```
mypackage.RfidEvent(category in ('Perishable', 'Container'))
```

By using the `not in` keywords we can filter events with a property value that does not match any of the values in a list of values:

```
mypackage.RfidEvent(category not in ('Household', 'Electrical'))
```

Filter Limitations

The following restrictions apply to filter criteria:

- Range and comparison operators require the event property to be of a numeric type.
- Aggregation functions are not allowed within filter expressions.
- The `prev` previous event function and the `prior` prior event function cannot be used in filter expressions.

4.4.2. Pattern-based Event Streams

Event pattern expressions can also be used to specify one or more event streams in an EPL statement. For pattern-based event streams, the event stream definition *stream_def* consists of the keyword `pattern` and a pattern expression in brackets `[]`. The syntax for an event stream definition using a pattern expression is below. As in filter-based event streams, an optional list of views that derive data from the stream can be supplied.

```
pattern [pattern_expression] [.view_spec] [.view_spec] [...]
```

The next statement specifies an event stream that consists of both stock tick events and trade events. The example tags stock tick events with the name "tick" and trade events with the name "trade".

```
select * from pattern [every tick=StockTickEvent or every trade=TradeEvent]
```

This statement generates an event every time the engine receives either one of the event types. The generated events resemble a map with "tick" and "trade" keys. For stock tick events, the "tick" key value is the underlying stock tick event, and the "trade" key value is a null value. For trade events, the "trade" key value is the underlying trade event, and the "tick" key value is a null value.

Lets further refine this statement adding a view the gives us the last 30 seconds of either stock tick or trade events. Lets also select prices and a price total.

```
select tick.price as tickPrice, trade.price as tradePrice,
       sum(tick.price) + sum(trade.price) as total
  from pattern [every tick=StockTickEvent or every trade=TradeEvent].win:time(30 sec)
```

Note that in the statement above `tickPrice` and `tradePrice` can each be null values depending on the event processed. Therefore, an aggregation function such as `sum(tick.price + trade.price))` would always return null values as either of the two price properties are always a null value for any event matching the pattern. Use the `coalesce` function to handle null values, for example: `sum(coalesce(tick.price, 0) + coalesce(trade.price, 0))`.

4.4.3. Specifying Views

Views are used to derive or aggregate data. Views can be staggered onto each other. See the section Chapter 8, *EPL Reference: Views* on the views available.

Views can optionally take one or more parameters. These parameters can consist of primitive constants such as String, boolean or numeric types. Arrays are also supported as a view parameter types.

The below example serves to show views and staggering of views. It uses a car location event that contains information about the location of a car on a highway.

The first view `std:groupby(carId)` groups car location events by car id. The second view `win:length(4)` keeps a length window of the 4 last events, with one length window for each car id. The next view `std:groupby(expressway, direction, segment)` groups each event by its expressway, direction and segment property values. Again, the grouping is done for each car id considering the last 4 events only. The last view `std:size()` is used to report the number of events. Thus the below example reports the number of events per car id and per expressway, direction and segment considering the last 4 events for each car id only.

```
select * from CarLocEvent.std:groupby(carId).win:length(4).
  std:groupby(expressway, direction, segment).std:size()
```

4.4.4. Using the Stream Name

Your `from` clause may assign a name to each stream. This assigned stream name can serve any of the following purposes.

First, the stream name can be used to disambiguate property names. The `stream_name.property_name` syntax uniquely identifies which property to select if property names overlap between streams. Here is an example:

```
select prod.productId, ord.productId from ProductEvent as prod, OrderEvent as ord
```

Second, the stream name can be used with a wildcard (*) character to select events in a join, or assign new names to the streams in a join:

```
// Select ProductEvent only
select prod.* from ProductEvent as prod, OrderEvent

// Assign column names 'product' and 'order' to each event
select prod.* as product, ord.* as order from ProductEvent as prod, OrderEvent as ord
```

Further, the stream name by itself can occur in any expression: The engine passes the event itself to that expression. For example, the engine passes the ProductEvent and the OrderEvent to the user-defined function 'checkOrder':

```
select prod.productId, MyFunc.checkOrder(prod, ord)
from ProductEvent as prod, OrderEvent as ord
```

Last, you may invoke an instance method on each event of a stream, and pass parameters to the instance method as well. Instance method calls are allowed anywhere in an expression.

The next statement demonstrates this capability by invoking a method 'computeTotal' on OrderEvent events and a method 'getMultiplier' on ProductEvent events:

```
select ord.computeTotal(prod.getMultiplier()) from ProductEvent as prod, OrderEvent as ord
```

4.5. Specifying Search Conditions: the *Where* Clause

The `where` clause is an optional clause in EPL statements. Via the `where` clause event streams can be joined and events can be filtered.

Comparison operators `=, < , > , >=, <=, !=, <>, is null, is not null` and logical combinations via `and` and `or` are supported in the `where` clause. The `where` clause can also introduce join conditions as outlined in Section 4.10, "Joining Event Streams". `where` clauses can also contain expressions. Some examples are listed below.

```
...where fraud.severity = 5 and amount > 500
...where (orderItem.orderId is null) or (orderItem.class != 10)
...where (orderItem.orderId = null) or (orderItem.class <> 10)
...where itemCount / packageCount > 10
```

4.6. Aggregates and grouping: the *Group-by* Clause and the *Having* Clause

4.6.1. Using aggregate functions

The aggregate functions are `sum, avg, count, max, min, median, stddev, avedev`. You can use aggregate functions to calculate and summarize data from event properties. For example, to find out the total price for all stock tick events in the last 30 seconds, type:

```
select sum(price) from StockTickEvent.win:time(30 sec)
```

Here is the syntax for aggregate functions:

```
aggregate_function( [all | distinct] expression)
```

You can apply aggregate functions to all events in an event stream window or other view, or to one or more groups of events. From each set of events to which an aggregate function is applied, Esper generates a single value.

`Expression` is usually an event property name. However it can also be a constant, function, or any combination of event property names, constants, and functions connected by arithmetic operators.

For example, to find out the average price for all stock tick events in the last 30 seconds if the price was doubled:

```
select avg(price * 2) from StockTickEvent.win:time(30 seconds)
```

You can use the optional keyword `distinct` with all aggregate functions to eliminate duplicate values before the aggregate function is applied. The optional keyword `all` which performs the operation on all events is the default.

You can use aggregation functions in a `select` clause and in a `having` clause. You cannot use aggregate functions in a `where` clause, but you can use the `where` clause to restrict the events to which the aggregate is applied. The next query computes the average and sum of the price of stock tick events for the symbol IBM only, for the last 10 stock tick events regardless of their symbol.

```
select 'IBM stats' as title, avg(price) as avgPrice, sum(price) as sumPrice
```

```
from StockTickEvent.win:length(10)
where symbol='IBM'
```

In the above example the length window of 10 elements is not affected by the `where` clause, i.e. all events enter and leave the length window regardless of their symbol. If we only care about the last 10 IBM events, we need to add filter criteria as below.

```
select 'IBM stats' as title, avg(price) as avgPrice, sum(price) as sumPrice
from StockTickEvent(symbol='IBM').win:length(10)
where symbol='IBM'
```

You can use aggregate functions with any type of event property or expression, with the following exceptions:

1. You can use `sum, avg, median, stddev, avedev` with numeric event properties only

Esper ignores any null values returned by the event property or expression on which the aggregate function is operating, except for the `count(*)` function, which counts null values as well. All aggregate functions return null if the data set contains no events, or if all events in the data set contain only null values for the aggregated expression.

4.6.2. Organizing statement results into groups: the *Group-by* clause

The `group by` clause is optional in all EPL statements. The `group by` clause divides the output of an EPL statement into groups. You can group by one or more event property names, or by the result of computed expressions. When used with aggregate functions, `group by` retrieves the calculations in each subgroup. You can use `group by` without aggregate functions, but generally that can produce confusing results.

For example, the below statement returns the total price per symbol for all stock tick events in the last 30 seconds:

```
select symbol, sum(price) from StockTickEvent.win:time(30 sec) group by symbol
```

The syntax of the `group by` clause is:

```
group by arregate_free_expression [, arregate_free_expression] [, ...]
```

Esper places the following restrictions on expressions in the `group by` clause:

1. Expressions in the `group by` cannot contain aggregate functions
2. Event properties that are used within aggregate functions in the `select` clause cannot also be used in a `group by` expression

You can list more then one expression in the `group by` clause to nest groups. Once the sets are established with `group by` the aggregation functions are applied. This statement posts the median volume for all stock tick events in the last 30 seconds per symbol and tick data feed. Esper posts one event for each group to statement listeners:

```
select symbol, tickDataFeed, median(volume)
from StockTickEvent.win:time(30 sec)
group by symbol, tickDataFeed
```

In the statement above the event properties in the `select` list (symbol, tickDataFeed) are also listed in the `group by` clause. The statement thus follows the SQL standard which prescribes that non-aggregated event properties in the `select` list must match the `group by` columns.

Esper also supports statements in which one or more event properties in the `select` list are not listed in the `group by` clause. The statement below demonstrates this case. It calculates the standard deviation for the last 30 seconds of stock ticks aggregating by symbol and posting for each event the symbol, tickDataFeed and the standard deviation on price.

```
select symbol, tickDataFeed, stddev(price) from StockTickEvent.win:time(30 sec) group by symbol
```

The above example still aggregates the `price` event property based on the `symbol`, but produces one event per incoming event, not one event per group.

Additionally, Esper supports statements in which one or more event properties in the `group by` clause are not listed in the `select` list. This is an example that calculates the mean deviation per `symbol` and `tickDataFeed` and posts one event per group with `symbol` and mean deviation of price in the generated events. Since tickDataFeed is not in the posted results, this can potentially be confusing.

```
select symbol, avedev(price)
from StockTickEvent.win:time(30 sec)
group by symbol, tickDataFeed
```

Expressions are also allowed in the `group by` list:

```
select symbol * price, count(*) from StockTickEvent.win:time(30 sec) group by symbol * price
```

If the `group by` expression resulted in a null value, the null value becomes its own group. All null values are aggregated into the same group. If you are using the `count(expression)` aggregate function which does not count null values, the count returns zero if only null values are encountered.

You can use a `where` clause in a statement with `group by`. Events that do not satisfy the conditions in the `where` clause are eliminated before any grouping is done. For example, the statement below posts the number of stock ticks in the last 30 seconds with a volume larger then 100, posting one event per group (symbol).

```
select symbol, count(*) from StockTickEvent.win:time(30 sec) where volume > 100 group by symbol
```

4.6.3. Selecting groups of events: the *Having* clause

Use the `having` clause to pass or reject events defined by the `group-by` clause. The `having` clause sets conditions for the `group by` clause in the same way `where` sets conditions for the `select` clause, except `where` cannot include aggregate functions, while `having` often does.

This statement is an example of a `having` clause with an aggregate function. It posts the total price per symbol for the last 30 seconds of stock tick events for only those symbols in which the total price exceeds 1000. The `having` clause eliminates all symbols where the total price is equal or less then 1000.

```
select symbol, sum(price)
from StockTickEvent.win:time(30 sec)
group by symbol
having sum(price) > 1000
```

To include more then one condition in the `having` clause combine the conditions with `and`, `or` or `not`. This is shown in the statement below which selects only groups with a total price greater then 1000 and an average volume less then 500.

```
select symbol, sum(price), avg(volume)
from StockTickEvent.win:time(30 sec)
group by symbol
```

```
having sum(price) > 1000 and avg(volume) < 500
```

Esper places the following restrictions on expressions in the `having` clause:

1. Any expressions that contain aggregate functions must also occur in the `select` clause

A statement with the `having` clause should also have a `group by` clause. If you omit `group-by`, all the events not excluded by the `where` clause return as a single group. In that case `having` acts like a `where` except that `having` can have aggregate functions.

The `having` clause can also be used without `group by` clause as the below example shows. The example below posts events where the price is less then the current running average price of all stock tick events in the last 30 seconds.

```
select symbol, price, avg(price)
from StockTickEvent.win:time(30 sec)
having price < avg(price)
```

4.6.4. How the stream filter, *Where*, *Group By* and *Having* clauses interact

When you include filters, the `where` condition, the `group by` clause and the `having` condition in an EPL statement the sequence in which each clause affects events determines the final result:

1. The event stream's filter condition, if present, dictates which events enter a window (if one is used). The filter discards any events not meeting filter criteria.
2. The `where` clause excludes events that do not meet its search condition.
3. Aggregate functions in the select list calculate summary values for each group.
4. The `having` clause excludes events from the final results that do not meet its search condition.

The following query illustrates the use of filter, `where`, `group by` and `having` clauses in one statement with a `select` clause containing an aggregate function.

```
select tickDataFeed, stddev(price)
from StockTickEvent(symbol='IBM').win:length(10)
where volume > 1000
group by tickDataFeed
having stddev(price) > 0.8
```

Esper filters events using the filter criteria for the event stream `StockTickEvent`. In the example above only events with symbol IBM enter the length window over the last 10 events, all other events are simply discarded. The `where` clause removes any events posted by the length window (events entering the window and event leaving the window) that do not match the condition of volume greater then 1000. Remaining events are applied to the `stddev` standard deviation aggregate function for each tick data feed as specified in the `group by` clause. Each `tickDataFeed` value generates one event. Esper applies the `having` clause and only lets events pass for `tickDataFeed` groups with a standard deviation of price greater then 0.8.

4.6.5. Comparing the *Group By* clause and the *std:groupby* view

The *group by* clause as well as the built-in *std:groupby* view are similar in their ability to group events. This section explains the key differences in their behavior and use.

The *group by* clause works together with aggregation functions in your statement to produce an aggregation result per group. In greater detail, this means that when a new event arrives, the engine applies the expressions in the *group by* clause to determine a grouping key. If the engine has not encountered that grouping key before (a new group), the engine creates a set of new aggregation results for that grouping key and performs the aggregation changing that new set of aggregation results. If the grouping key points to an existing set of prior aggregation results (an existing group), the engine performs the aggregation changing the prior set of aggregation results for that group.

The *std:groupby* view is a built-in view that also groups events. The view is described in greater detail in Section 8.2.2, “Group-By (std:groupby)”. Its primary use is to create a separate data window per group, or more generally to create separate instances of all its sub-views for each grouping key encountered.

The next example shows two queries that produce equivalent results. The query using the *group by* clause is generally preferable as is easier to read. The second form introduces the `stat:uni` view which computes univariate statistics for a given property:

```
select symbol, avg(price) from StockTickEvent group by symbol
// ... is equivalent to ...
select symbol, average from StockTickEvent.std:groupby(symbol).stat:uni(price)
```

The next example shows two queries that are NOT equivalent as the length window is ungrouped in the first query, and grouped in the second query:

```
select symbol, sum(price) from StockTickEvent.win:length(10) group by symbol
// ... NOT equivalent to ...
select symbol, sum(price) from StockTickEvent.std:groupby(symbol).win:length(10)
```

The key difference between the two statements is that in the first statement the length window is ungrouped and applies to all events regardless of group. While in the second query each group gets its own instance of a length window. For example, in the second query events arriving for symbol "ABC" get a length window of 10 events, and events arriving for symbol "DEF" get their own length window of 10 events.

4.7. Stabilizing and Limiting Output: the *Output* Clause

4.7.1. Output Clause Options

The `output` clause is optional in Esper and is used to control or stabilize the rate at which events are output. For example, the following statement outputs, every 60 seconds, the total price for all orders in the 30-minute time window:

```
select sum(price) from OrderEvent.win:time(30 min) output snapshot every 60 seconds
```

Here is the syntax for output rate limiting:

```
output [all | first | last | snapshot] every output_rate [minutes | seconds | events]
```

The `all` keyword is the default and specifies that all events in a batch should be output, each incoming row in the batch producing an output row. Note that for statements that group via the `group by` clause, the `all` keyword provides special behavior as below.

The `first` keyword specifies that only the first event in an output batch is to be output. Using the `first` keyword instructs the engine to output the first matching event as soon as it arrives, and then ignores matching events for the time interval or number of events specified. After the time interval elapsed, or the number of matching events has been reached, the next first matching event is output again and the following interval the engine again ignores matching events.

The `last` keyword specifies to only output the last event at the end of the given time interval or after the given number of matching events have been accumulated. Again, for statements that group via the `group by` clause the `last` keyword provides special behavior as below.

The `snapshot` keyword indicates that the engine output current computation results considering all events as per views specified and/or current aggregation results. While the other keywords control how a batch of events between output intervals is being considered, the `snapshot` keyword outputs all current state of a statement independent of the last batch. It's output is equivalent to the `iterator` method provided by a statement.

The *output_rate* is the frequency at which the engine outputs events. It can be specified in terms of time or number of events. The value can be a number to denote a fixed output rate, or the name of a variable whose value is the output rate. By means of a variable the output rate can be controlled externally and changed dynamically at runtime.

Please consult the Appendix A, *Output Reference and Samples* for detailed information on insert and remove stream output for the various output rate limiting keywords.

The time interval can also be specified in terms of minutes; the following statement is identical to the first one.

```
select * from StockTickEvent.win:length(5) output every 1.5 minutes
```

A second way that output can be stabilized is by batching events until a certain number of events have been collected. The next statement only outputs when either 5 (or more) new or 5 (or more) old events have been batched.

```
select * from StockTickEvent.win:time(30 sec) output every 5 events
```

Additionally, event output can be further modified by the optional `last` keyword, which causes output of only the last event to arrive into an output batch.

```
select * from StockTickEvent.win:time(30 sec) output last every 5 events
```

Using the `first` keyword you can be notified at the start of the interval. The allows to watch for situations such as a rate falling below a threshold and only be informed every now and again after the specified output interval, but be informed the moment it first happens.

```
select * from TickRate.win:time(30 seconds) where rate<100 output first every 60 seconds
```

4.7.2. Aggregation, Group By, Having and Output clause interaction

Remove stream events can also useful in conjunction with aggregation and output rate limiting: When the engine posts remove stream events for fully-aggregated queries, it presents the aggregation state before the expiring event leaves the data window. Your application can thus easily obtain a delta between the new aggregation value and the prior aggregation value.

The engine evaluates the having-clause at the granularity of the data posted by views. That is, if you utilize a time window and output every 10 events, the `having` clause applies to each individual event or events entering and leaving the time window (and not once per batch of 10 events).

The `output` clause interacts in two ways with the `group by` and `having` clauses. First, in the `output every n events` case, the number `n` refers to the number of events arriving into the `group by clause`. That is, if the `group by` clause outputs only 1 event per group, or if the arriving events don't satisfy the `having` clause, then the actual number of events output by the statement could be fewer than `n`.

Second, the `last` and `all` keywords have special meanings when used in a statement with aggregate functions and the `group by` clause:

- When no keyword is specified, the engine produces an output row for each row in the batch.
- The `all` keyword (the default) specifies that the most recent data for *all* groups seen so far should be output, whether or not these groups' aggregate values have just been updated
- The `last` keyword specifies that only groups whose aggregate values have been updated with the most recent batch of events should be output.

Please consult the Appendix A, *Output Reference and Samples* for detailed information on insert and remove stream output for aggregation and group-by.

By adding an output rate limiting clause to a statement that contains a *group by* clause we can control output of groups to obtain one row for each group, generating an event per group at the given output frequency:

```
select symbol, sum(price) from StockTickEvent group by symbol output all every 5 seconds
```

4.7.3. Runtime Considerations

Output rate limiting provides output events to your application in regular intervals. Between intervals, the engine uses a buffer to hold events until the output condition is reached. If your application has high-volume streams, you may need to be mindful of the memory needs for output rates.

The `output` clause with the `snapshot` keyword does not require a buffer, all other output keywords do consume memory until the output condition is reached.

4.8. Sorting Output: the *Order By* Clause

The `order by` clause is optional. It is used for ordering output events by their properties, or by expressions involving those properties. .

For example, the following statement outputs batches of 5 or more stock tick events that are sorted first by price ascending and then by volume ascending:

```
select symbol from StockTickEvent.win:time(60 sec)
output every 5 events
order by price, volume
```

Here is the syntax for the `order by` clause:

```
order by expression [asc | desc] [, expression [asc | desc]] [, ...]
```

If the `order by` clause is absent then the engine still makes certain guarantees about the ordering of output:

- If the statement is not a join, does not group via `group by` clause and does not declare grouped data windows via `std:groupby` view, the order in which events are delivered to listeners and through the `iterator` pull API is the order of event arrival.
- If the statement is a join or outer join, or groups, then the order in which events are delivered to listeners and through the `iterator` pull API is not well-defined. Use the `order by` clause if your application requires events to be delivered in a well-defined order.

Esper places the following restrictions on the expressions in the `order by` clause:

1. All aggregate functions that appear in the `order by` clause must also appear in the `select` expression.

Otherwise, any kind of expression that can appear in the `select` clause, as well as any alias defined in the `select` clause, is also valid in the order by clause.

4.9. Merging Streams and Continuous Insertion: the *Insert Into* Clause

The `insert into` clause is optional in Esper. The clause can be specified to make the results of a statement available as an event stream for use in further statements, or to insert events into a named window. The clause can also be used to merge multiple event streams to form a single stream of events.

The syntax for the `insert into` clause is as follows:

```
insert [istream | rstream] into event_stream_name  [ (property_name [, property_name] ) ]
```

The `istream` (default) and `rstream` keywords are optional. If no keyword or the `istream` keyword is specified, the engine supplies the insert stream events generated by the statement. The insert stream consists of the events entering the respective window(s) or stream(s). If the `rstream` keyword is specified, the engine supplies the remove stream events generated by the statement. The remove stream consists of the events leaving the respective window(s).

The `event_stream_name` is an identifier that names the event stream (and also implicitly names the types of events in the stream) generated by the engine. The identifier can be used in further statements to filter and process events of that event stream. The `insert into` clause can consist of just an event stream name, or an event stream name and one or more property names.

The engine also allows listeners to be attached to a statement that contain an `insert into` clause. Listeners receive all events posted to the event stream.

To merge event streams, simply use the same `event_stream_name` identifier in all EPL statements that merge their result event streams. Make sure to use the same number and names of event properties and event property types match up.

Esper places the following restrictions on the `insert into` clause:

1. The number of elements in the `select` clause must match the number of elements in the `insert into` clause if the clause specifies a list of event property names
2. If the event stream name has already been defined by a prior statement or configuration, and the event property names and/or event types do not match, an exception is thrown at statement creation time.

The following sample inserts into an event stream by name CombinedEvent:

```
insert into CombinedEvent
select A.customerId as custId, A.timestamp - B.timestamp as latency
  from EventA.win:time(30 min) A, EventB.win:time(30 min) B
 where A.txnId = B.txnId
```

Each event in the `CombinedEvent` event stream has two event properties named "custId" and "latency". The events generated by the above statement can be used in further statements, such as shown in the next statement:

```
select custId, sum(latency)
  from CombinedEvent.win:time(30 min)
 group by custId
```

The example statement below shows the alternative form of the `insert into` clause that explicitly defines the property names to use.

```
insert into CombinedEvent (custId, latency)
select A.customerId, A.timestamp - B.timestamp
...
```

The `rstream` keyword can be useful to indicate to the engine to generate only remove stream events. This can be useful if we want to trigger actions when events leave a window rather then when events enter a window. The statement below generates `CombinedEvent` events when EventA and EventB leave the window after 30 minutes (1800 seconds).

```
insert rstream into CombinedEvent
select A.customerId as custId, A.timestamp - B.timestamp as latency
  from EventA.win:time(30 min) A, EventB.win:time(30 min) B
 where A.txnId = B.txnId
```

The `insert into` clause can be used in connection with patterns to provide pattern results to further statements for analysis:

```
insert into ReUpEvent
select linkUp.ip as ip
from pattern [every linkDown=LinkDownEvent -> linkUp=LinkUpEvent(ip=linkDown.ip)]
```

Merging Streams By Event Type

The `insert into` clause allows to merge multiple event streams into a event single stream. The clause names an event stream to insert into by specifing an *event_stream_name*. The first statement that inserts into the named stream defines the stream's event types. Further statements that insert into the same event stream must match the type of events inserted into the stream as declared by the first statement.

One approach to merging event streams specifies individual colum names either in the `select` clause or in the `insert into` clause of the statement. This approach has been shown in earlier examples.

Another approach to merging event streams specifies the wildcard (*) in the `select` clause (or the stream wildcard) to select the underlying event. The events in the event stream must then have the same event type as generated by the `from` clause.

Assume a statement creates an event stream named MergedStream by selecting OrderEvent events:

```
insert into MergedStream select * from OrderEvent
```

A statement can use the stream wildcard selector to select only OrderEvent events in a join:

```
insert into MergedStream select ord.* from ItemScanEvent, OrderEvent as ord
```

And a statement may also use an application-supplied user-defined function to convert events to OrderEvent instances:

```
insert into MergedStream select MyLib.convert(item) from ItemScanEvent as item
```

Esper specifically recognizes a conversion function: A conversion function must be the only selected column, and it must return either a Java object or `java.util.Map`.

4.10. Joining Event Streams

Two or more event streams can be part of the `from` clause and thus both (all) streams determine the resulting events. The `where` clause lists the join conditions that Esper uses to relate events in the two or more streams. Reference and historical data such as stored in your relational database, and data returned by a method invocation, can also be included in joins. Please see Section 4.14, “Joining Relational Data via SQL” and Section 4.15, “Joining Non-Relational Data via Method Invocation” for details.

Each point in time that an event arrives to one of the event streams, the two event streams are joined and output events are produced according to the `where` clause.

This example joins 2 event streams. The first event stream consists of fraud warning events for which we keep the last 30 minutes (1800 seconds). The second stream is withdrawal events for which we consider the last 30 seconds. The streams are joined on account number.

```
select fraud.accountNumber as accntNum, fraud.warning as warn, withdraw.amount as amount,
       max(fraud.timestamp, withdraw.timestamp) as timestamp, 'withdrawlFraud' as desc
  from com.espertech.esper.example.atm.FraudWarningEvent.win:time(30 min) as fraud,
       com.espertech.esper.example.atm.WithdrawalEvent.win:time(30 sec) as withdraw
 where fraud.accountNumber = withdraw.accountNumber
```

Joins can also include one or more pattern statements as the next example shows:

```
select * from FraudWarningEvent.win:time(30 min) as fraud,
       pattern [every w=WithdrawalEvent -> PINChangeEvent(acct=w.acct)] as withdraw
 where fraud.accountNumber = withdraw.w.accountNumber
```

The statement above joins the last 30 minutes of fraud warnings with a pattern. The pattern consists of every withdrawal event that is followed by a PIN change event for the same account number. It joins the two event streams on account number.

In a join and outer join, if your statement does not declare a data window view or other view onto a stream, then the engine retains all events for the stream equivalent to the keep-all data window.

The next example joins all FraudWarningEvent events that arrived since the statement was started, with the last 20 seconds of PINChangeEvent events:

```
select * from FraudWarningEvent as fraud, PINChangeEvent.win:time(20 sec) as pin
 where fraud.accountNumber = pin.accountNumber
```

4.11. Outer Joins

Esper supports left outer joins, right outer joins and full outer joins between an unlimited number of event streams. Outer joins can also join reference and historical data as explained in Section 4.14, “Joining Relational Data via SQL”, as well as join data returned by a method invocation as outlined in Section 4.15, “Joining Non-Relational Data via Method Invocation”.

The keywords `left, right` and `full` control the type of the join between two streams. The `on` clause specifies one or more properties that join each stream. The synopsis is as follows:

```
...from stream_def [as name]
  left|right|full outer join stream_def
  on property = property [and property = property ...]
  [left|right|full outer join stream_def on ...]...
```

If the outer join is a left outer join, there will be an output event for each event of the stream on the left-hand side of the clause. For example, in the left outer join shown below we will get output for each event in the stream RfidEvent, even if the event does not match any event in the event stream OrderList.

```
select * from RfidEvent.win:time(30 sec) as rfid
       left outer join
       OrderList.win:length(10000) as orderlist
     on rfid.itemId = orderList.itemId
```

Similarly, if the join is a Right Outer Join, then there will be an output event for each event of the stream on the right-hand side of the clause. For example, in the right outer join shown below we will get output for each event in the stream OrderList, even if the event does not match any event in the event stream RfidEvent.

```
select * from RfidEvent.win:time(30 sec) as rfid
       right outer join
       OrderList.win:length(10000) as orderlist
       on rfid.itemId = orderList.itemId
```

For all types of outer joins, if the join condition is not met, the select list is computed with the event properties of the arrived event while all other event properties are considered to be null.

The last type of outer join is a full outer join. In a full outer join, each point in time that an event arrives to one of the event streams, one or more output events are produced. In the example below, when either an RfidEvent or an OrderList event arrive, one or more output event is produced. The next example shows a full outer join that joins on multiple properties:

```
select * from RfidEvent.win:time(30 sec) as rfid
       full outer join
       OrderList.win:length(10000) as orderlist
       on rfid.itemId = orderList.itemId and rfid.assetId = orderList.assetId
```

Finally, this example outer joins multiple streams. Here the RfidEvent stream is outer joined to both ProductName and LocationDescription via left outer join:

```
select * from RfidEvent.win:time(30 sec) as rfid
      left outer join ProductName.win:keepall() as refprod
        on rfid.productId = refprod.prodId
      left outer join LocationDescription.win:keepall() as refdesc
        on rfid.location = refdesc.locId
```

4.12. Unidirectional Joins and Outer Joins

In a join or outer join your statement lists multiple event streams, views and/or patterns in the `from` clause. As events arrive into the engine, each of the streams (views, patterns) provides insert and remove stream events. The engine evaluates each insert and remove stream event provided by each stream, and joins or outer joins each event against data window contents of each stream, and thus generates insert and remove stream join results.

The direction of the join execution depends on which stream or streams are currently providing an insert or remove stream event for executing the join. A join is thus multidirectional, or bidirectional when only two streams are joined. A join can be made unidirectional if your application does not want new results when events arrive on a given stream or streams.

The `unidirectional` keyword can be used in the `from` clause to identify a single stream that provides the events to execute the join. If the keyword is present for a stream, all other streams in the `from` clause become passive streams. When events arrive or leave a data window of a passive stream then the join does not generate join results.

For example, consider a use case that requires us to join stock tick events (TickEvent) and news events (NewsEvent). The `unidirectional` keyword allows to generate results only when TickEvent events arrive, and not when NewsEvent arrive or leave the 10-second time window:

```
select * from TickEvent unidirectional, NewsEvent.win:time(10 sec)
where tick.symbol = news.symbol
```

The following restrictions apply to unidirectional joins:

1. The `unidirectional` keyword can only be specified for a single stream in the `from` clause.
2. Receiving data from a unidirectional join via the pull API (`iterator` method) is not allowed. This is because the engine holds no state for the single stream that provides the events to execute the join.
3. The stream that declares the `unidirectional` keyword cannot declare a data window view or other view for that stream, since remove stream events are not processed for the single stream.

4.13. Subqueries

A subquery is a `select` within another statement. Esper supports subqueries in the `select` clause and in the `where` clause of EPL statements. Subqueries provide an alternative way to perform operations that would otherwise require complex joins. Subqueries can also make statements more readable then complex joins.

Esper supports both simple subqueries as well as correlated subqueries. In a simple subquery, the inner query is not correlated to the outer query. Here is an example simple subquery within a `select` clause:

```
select assetId, (select zone from ZoneClosed.std:lastevent) as lastClosed from RFIDEvent
```

If the inner query is dependent on the outer query, we will have a correlated subquery. An example of a correlated subquery is shown below. Notice the `where` clause in the inner query, where the condition involves a stream from the outer query:

```
select * from RfidEvent as RFID where 'Dock 1' =
  (select name from Zones.std:unique(zoneId) where zoneId = RFID.zoneId)
```

The example above shows a subquery in the `where` clause. The statement selects RFID events in which the zone name matches a string constant based on zone id. The statement uses the view `std:unique` to guarantee that only the last event per zone id is held from processing by the subquery.

The next example is a correlated subquery within a `select` clause. In this statement the `select` clause retrieves the zone name by means of a subquery against the Zones set of events correlated by zone id:

```
select zoneId, (select name from Zones.std:unique(zoneId)
  where zoneId = RFID.zoneId) as name from RFIDEvent
```

Note that when a simple or correlated subquery returns multiple rows, the engine returns a `null` value as the subquery result. To limit the number of events returned by a subquery consider using one of the views `std:lastevent`, `std:unique` and `std:groupby`.

The `select` clause of a subquery also allows wildcard selects, which return as an event property the underlying event object of the event type as defined in the `from` clause. An example:

```
select (select * from MarketData.std:lastevent()) as md
  from pattern [every timer:interval(10 sec)]
```

The output events to the statement above contain the underlying MarketData event in a property named "md". The statement populates the last MarketData event into a property named "md" every 10 seconds following the pattern definition, or populates a `null` value if no MarketData event has been encountered so far.

Aggregation functions may be used in the `select` clause of the subselect as this example outlines:

```
select * from MarketData
where price > (select max(price) from MarketData(symbol='GOOG').std:lastevent())
```

As the sub-select expression is evaluated first, the query above actually never fires for the GOOG symbol, only for other symbols that have a price higher then the current maximum for GOOG. As a sidenote, the `insert into` clause can also be handy to compute aggregation results for use in multiple subqueries.

The following restrictions apply to subqueries:

1. The subquery stream definition must define a data window or other view to limit subquery results, reducing the number of events held for subquery execution
2. Subqueries can only consist of a `select` clause, a `from` clause and a `where` clause. The `group by` and `having` clauses, as well as joins, outer-joins and output rate limiting are not permitted within subqueries.
3. If using aggregation functions in a subquery, note these limitations:

    a. None of the properties of the correlated stream(s) can be used within aggregation functions.
    b. The properties of the subselect stream must all be within aggregation functions.
    c. The `where` clause cannot be used to correlate between the subselect stream and the enclosing stream, since the engine would otherwise be forced to re-evaluate the aggregation considering all events in the subselect-stream data window, which would likely be a very expensive operation.

Performance of your statement containing one or more subqueries principally depends on two parameters. First, if your subquery correlates one or more columns in the subquery stream with the enclosing statement's streams via equals '=', the engine automatically builds the appropriate indexes for fast row retrieval based on the key values correlated (joined). The second parameter is the number of rows found in the subquery stream and the complexity of the filter criteria (`where` clause), as each row in the subquery stream must evaluate against the `where` clause filter.

4.13.1. The '`exists`' keyword

The `exists` condition is considered "to be met" if the subquery returns at least one row. The `not exists` condition is considered true if the subquery returns no rows.

Let's take a look at a simple example. The following is an EPL statement that uses the `exists` condition:

```
select assetId from RFIDEvent as RFID
  where exists (select * from Asset.std:unique(assetId) where assetId = RFID.assetId)
```

This select statement will return all RFID events where there is at least one event in Assets unique by asset id with the same asset id.

4.13.2. The '`in`' keyword

The `in` subquery condition is true if the value of an expression matches one or more of the values returned by the subquery. Consequently, the `not in` condition is true if the value of an expression matches none of the values returned by the subquery.

The next statement demonstrates the use of the `in` subquery condition:

```
select assetId from RFIDEvent as RFID
  where zone in (select zone from ZoneUpdate.win:time(10 min) where status = 'closed' )
```

The above statement demonstrated the `in` subquery to select RFID events for which the zone status is in a closed state.

4.14. Joining Relational Data via SQL

This chapter outlines how reference data and historical data that are stored in a relational database can be queried via SQL within EPL statements.

Esper can join and outer join all types of event streams to stored data. In order for such data sources to become accessible to Esper, some configuration is required. The Section 10.4.7, “Relational Database Access” explains the required configuration for database access in greater detail, and includes information on configuring a query result cache.

Esper does not parse of otherwise inspect your SQL query. Therefore your SQL can make use of any database-specific SQL language extensions or features that your database provides.

If you have enabled query result caching in your Esper database configuration, Esper retains SQL query results in cache following the configured cache eviction policy.

Also if you have enabled query result caching in your Esper database configuration and provide EPL `where` clause and/or `on` clause (outer join) expressions, then Esper builds indexes on the SQL query results to enable fast lookup. This is especially useful if your queries return a large number of rows. For building the proper indexes, Esper inspects the expression found in your EPL query `where` clause, if present. For outer joins, Esper also inspects your EPL query `on` clause. Esper analyzes the EPL `on` clause and `where` clause expressions, if present, looking for property comparison with or without logical AND-relationships between properties. When a SQL query returns rows for caching, Esper builds the appropriate index and lookup strategies for fast row matching against indexes.

The following restrictions currently apply:

- Only one event stream and one SQL query can be joined; Joins of two or more event streams with an SQL query are not yet supported.
- Sub-views on an SQL query are not allowed; That is, one cannot create a time or length window on an SQL query. However one can use the `insert into` syntax to make join results available to a further statement.
- Your database software must support JDBC prepared statements that provide statement meta data at compilation time. Most major databases provide this function. A workaround is available for databases that do not provide this function.
- JDBC drivers must support the getMetadata feature. A workaround is available as below for JDBC drivers that don't support getting metadata.

The next sections assume basic knowledge of SQL (Structured Query Language).

4.14.1. Joining SQL Query Results

To join an event stream against stored data, specify the `sql` keyword followed by the name of the database and a parameterized SQL query. The syntax to use in the `from` clause of an EPL statement is:

```
sql:database_name [" parameterized_sql_query "]
```

The engine uses the *database_name* identifier to obtain configuration information in order to establish a database connection, as well as settings that control connection creation and removal. Please see Section 10.4.7, “Relational Database Access” to configure an engine for database access.

Following the database name is the SQL query to execute. The SQL query can contain one or more substitution parameters. The SQL query string is placed in single brackets `[` and `]`. The SQL query can be placed in either single quotes (') or double quotes ("). The SQL query grammer is passed to your database software unchanged, allowing you to write any SQL query syntax that your database understands, including stored procedure calls.

Substitution parameters in the SQL query string take the form `${`*event_property_name*`}`. The engine resolves *event_property_name* at statement execution time to the actual event property value supplied by the events in the joined event stream.

The engine determines the type of the SQL query output columns by means of the result set metadata that your database software returns for the statement. The actual query results are obtained via the `getObject` on `java.sql.ResultSet`.

The sample EPL statement below joins an event stream consisting of `CustomerCallEvent` events with the results of an SQL query against the database named `MyCustomerDB` and table `Customer`:

```
select custId, cust_name from CustomerCallEvent,
  sql:MyCustomerDB [' select cust_name from Customer where cust_id = ${custId} ']
```

The example above assumes that `CustomerCallEvent` supplies an event property named `custId`. The SQL query selects the customer name from the Customer table. The `where` clause in the SQL matches the Customer table column `cust_id` with the value of `custId` in each `CustomerCallEvent` event. The engine executes the SQL query for each new `CustomerCallEvent` encountered.

If the SQL query returns no rows for a given customer id, the engine generates no output event. Else the engine generates one output event for each row returned by the SQL query. An outer join as described in the next section can be used to control whether the engine should generate output events even when the SQL query returns no rows.

The next example adds a time window of 30 seconds to the event stream `CustomerCallEvent`. It also renames the selected properties to customerName and customerId to demonstrate how the naming of columns in an SQL query can be used in the `select` clause in the EPL query. And the example uses explicit stream names via the `as` keyword.

```
select customerId, customerName from
  CustomerCallEvent.win:time(30 sec) as cce,
  sql:MyCustomerDB ["select cust_id as customerId, cust_name as customerName from Customer
                  where cust_id = ${cce.custId}"] as cq
```

Any window, such as the time window, generates insert stream (istream) events as events enter the window, and remove stream (rstream) events as events leave the window. The engine executes the given SQL query for each `CustomerCallEvent` in both the insert stream and the remove stream. As a performance optimization, the `istream` or `rstream` keywords in the `select` clause can be used to instruct the engine to only join insert stream or remove stream events, reducing the number of SQL query executions.

4.14.2. SQL Query and the EPL `where` Clause

Consider using the EPL `where` clause to join the SQL query result to your event stream. Similar to EPL joins and outer-joins that join event streams or patterns, the EPL `where` clause provides join criteria between the SQL query results and the event stream (as a side note, an SQL `where` clause is a filter of rows executed by your database on your database server before returning SQL query results).

Esper analyzes the expression in the EPL `where` clause, if present, and builds the appropriate indexes from that information at runtime, to ensure fast matching of event stream events to SQL query results, even if your SQL query returns a large number of rows. Your applications must ensure to configure a cache for your database using Esper configuration, as such indexes are held with regular data in a cache. If you application does not enable caching of SQL query results, the engine does not build indexes on cached data.

The sample EPL statement below joins an event stream consisting of `OrderEvent` events with the results of an SQL query against the database named `MyRefDB` and table `SymbolReference`:

```
select symbol, symbolDesc from OrderEvent as orders,
  sql:MyRefDB ['select symbolDesc from SymbolReference'] as reference
  where reference.symbol = orders.symbol
```

Notice how the EPL `where` clause joins the `OrderEvent` stream to the `SymbolReference` table. In this example, the SQL query itself does not have a SQL `where` clause and therefore returns all rows from table `SymbolReference`.

If your application enables caching, the SQL query fires only at the arrival of the first `OrderEvent` event. When the second `OrderEvent` arrives, the join execution uses the cached query result. If the caching policy that you specified in the Esper database configuration evicts the SQL query result from cache, then the engine fires the SQL query again to obtain a new result and places the result in cache.

If SQL result caching is enabled and your EPL `where` clause, as show in the above example, provides the properties to join, then the engine indexes the SQL query results in cache and retains the index together with the query result in cache. Thus your application can benefit from high performance index-based lookups as long as the SQL query results are found in cache.

4.14.3. Outer Joins With SQL Queries

You can use outer joins to join data obtained from an SQL query and control when an event is produced. Use a left outer join, such as in the next statement, if you need an output event for each event regardless of whether or not the SQL query returns rows. If the SQL query returns no rows, the join result populates null values into the selected properties.

```
select custId, custName from
  CustomerCallEvent as cce
  left outer join
  sql:MyCustomerDB ["select cust_id, cust_name as custName
                      from Customer where cust_id = ${cce.custId}"] as cq
  on cce.custId = cq.cust_id
```

The statement above always generates at least one output event for each `CustomerCallEvent`, containing all columns selected by the SQL query, even if the SQL query does not return any rows. Note the `on` expression that is required for outer joins. The `on` acts as an additional filter to rows returned by the SQL query.

4.14.4. Using Patterns to Request (Poll) Data

Pattern statements and SQL queries can also be applied together in useful ways. One such use is to poll or request data from a database at regular intervals. The next statement is an example that shows a pattern that fires every 5 seconds to query the NewOrder table for new orders:

```
insert into NewOrders
select orderId, orderAmount from
  pattern [every timer:interval(5 sec)],
```

```
  sql:MyCustomerDB ['select orderId, orderAmount from NewOrders']
```

4.14.5. JDBC Implementation Overview

The engine translates SQL queries into JDBC `java.sql.PreparedStatement` statements by replacing ${name} parameters with '?' placeholders. It obtains name and type of result columns from the compiled `PreparedStatement` meta data when the EPL statement is created.

The engine supplies parameters to the compiled statement via the `setObject` method on `PreparedStatement`. The engine uses the `getObject` method on the compiled statement `PreparedStatement` to obtain column values.

4.14.6. Oracle Drivers and No-Metadata Workaround

Certain JDBC database drivers are known to not return metadata for precompiled prepared SQL statements. This can be a problem as metadata is required by Esper. Esper obtains SQL result set metadata to validate an EPL statement and to provide column types for output events. JDBC drivers that do not provide metadata for precompiled SQL statements require a workaround. Such drivers do generally provide metadata for executed SQL statements, however do not provide the metadata for precompiled SQL statements.

Please consult the Chapter 10, *Configuration* for the configuration options available in relation to metadata retrieval.

To obtain metadata for an SQL statement, Esper can alternatively fire a SQL statement which returns the same column names and types as the actual SQL statement but without returning any rows. This kind of SQL statement is referred to as a *sample* statement in below workaround description. The engine can then use the sample SQL statement to retrieve metadata for the column names and types returned by the actual SQL statement.

Applications can provide a sample SQL statement to retrieve metadata via the `metadatasql` keyword:

```
sql:database_name ["parameterized_sql_query" metadatasql "sql_meta_query"]
```

The *sql_meta_query* must be an SQL statement that returns the same number of columns, the same type of columns and the same column names as the *parameterized_sql_query*, and does not return any rows.

Alternatively, applications can choose not to provide an explicit sample SQL statement. If the EPL statement does not use the `metadatasql` syntax, the engine applies lexical analysis to the SQL statement. From the lexical analysis Esper generates a sample SQL statement adding a restrictive clause "where 1=0" to the SQL statement.

Alternatively, applications can add the following tag to the SQL statement: `${$ESPER-SAMPLE-WHERE}`. If the tag exists in the SQL statement, the engine does not perform lexical analysis and simply replaces the tag with the SQL `where` clause "where 1=0". Therefore this workaround is applicable to SQL statements that cannot be correctly lexically analyzed. The SQL text after the placeholder is not part of the sample query. For example:

```
select mycol from sql:myDB [
  'select mycol from mytesttable ${$ESPER-SAMPLE-WHERE} where ....'], ...
```

4.15. Joining Non-Relational Data via Method Invocation

Your application may need to join data that originates from a web service, a distributed cache, an object-oriented database or simply data held in memory by your application. Esper accommodates this need by allowing a method invocation (or procedure call or function) in the `from` clause of a statement.

Esper can join and outer join all types of event streams to the data returned by your method invocation. In addition, Esper can be configured to cache the data returned by your method invocations.

The following restrictions currently apply:

- Only one event stream and one method invocation can be joined; That is, in a join with a method invocation only one other event stream is allowed.
- Sub-views on a method invocations are not allowed; That is, one cannot create a time or length window on a method invocation. However one can use the `insert into` syntax to make join results available to a further statement.

4.15.1. Joining Method Invocation Results

The syntax for a method invocation in the `from` clause of an EPL statement is:

```
method:class_name.method_name[(parameter_expressions)]
```

The `method` keyword denotes a method invocation. It is followed by a class name and a method name separated by a dot (.) character. If you have parameters to your method invocation, these are placed in round brackets after the method name. Any expression is allowed as a parameter, and individual parameter expressions are separated by a comma. Expressions may also use event properties of the joined stream.

In the sample join statement shown next, the method 'lookupAsset' provided by class 'MyLookupLib' returns one or more rows based on the asset id (a property of the AssetMoveEvent) that is passed to the method:

```
select * from AssetMoveEvent, method:MyLookupLib.lookupAsset(assetId)
```

The following statement demonstrates the use of the `where` clause to join events to the rows returned by a method invocation, which in this example does not take parameters:

```
select assetId, assetDesc from AssetMoveEvent as asset,
       method:MyLookupLib.getAssetDescriptions() as desc
where asset.assetid = desc.assetid
```

Your method invocation may return zero, one or many rows for each method invocation. If you have caching enabled through configuration, then Esper can avoid the method invocation and instead use cached results. Similar to SQL joins, Esper also indexes cached result rows such that join operations based on the `where` clause can be very efficient, especially if your method invocation returns a large number of rows.

If the time taken by method invocations is critical to your application, you may configure local caches as Section 10.4.5, “Cache Settings for Method Invocations” describes.

4.15.2. Providing the Method

Your application must provide a Java class that exposes a public static method. The method must accept the same number and type of parameters as listed in the parameter expression list.

If your method invocation returns either no row or only one row, then the return type of the method can be a Java class or a `java.util.Map`. If your method invocation can return more then one row, then the return type of the method must be an array of Java class or an array of `Map`.

If you are using a Java class or an array of Java class as the return type, then the class must adhere to JavaBean conventions: it must expose properties through getter methods.

If you are using `java.util.Map` as the return type or an array of `Map`, then the map should have `String`-type keys and object values (`Map<String, Object>`). When using `Map` as the return type, your application must provide a second method that returns property metadata, as the next section outlines.

Your application method must return either of the following:

1. A `null` value or an empty array to indicate an empty result (no rows).
2. A Java object or `Map` to indicate a one-row result, or an array that consists of a single Java object or `Map`.
3. An array of Java objects or `Map` instances to return multiple result rows.

As an example, consider the method 'getAssetDescriptions' provided by class 'MyLookupLib' as discussed earlier:

```
select assetId, assetDesc from AssetMoveEvent as asset,
       method:com.mypackage.MyLookupLib.getAssetDescriptions() as desc
  where asset.assetid = desc.assetid
```

The 'getAssetDescriptions' method may return multiple rows and is therefore declared to return an array of the class 'AssetDesc'. The class AssetDesc is a POJO class (not shown here):

```
public class MyLookupLib {
  ...
  public static AssetDesc[] getAssetDescriptions() {
    ...
    return new AssetDesc[] {...};
  }
```

The example above specifies the full Java class name of the class 'MyLookupLib' class in the EPL statement. The package name does not need to be part of the EPL if your application imports the package using the auto-import configuration through the API or XML, as outlined in Section 10.4.4, “Class and package imports”.

4.15.3. Using a `Map` Return Type

Your application may return `java.util.Map` or an array of `Map` from method invocations. If doing so, your application must provide metadata about each row: it must declare the property name and property type of each `Map` entry of a row. This information allows the engine to perform type checking of expressions used within the statement.

You declare the property names and types of each row by providing a method that returns property metadata. The metadata method must follow these conventions:

1. The method name providing the property metadata must have same method name appended by the literal `Metadata`.
2. The method must have an empty parameter list and must be declared public and static.
3. The method providing the metadata must return a `Map` of `String` property name keys and `java.lang.Class` property name types (`Map<String, Class>`).

In the following example, a class 'MyLookupLib' provides a method to return historical data based on asset id and asset code:

```
select assetId, location, x_coord, y_coord from AssetMoveEvent as asset,
       method:com.mypackage.MyLookupLib.getAssetHistory(assetId, assetCode) as history
```

A sample implementation of the class 'MyLookupLib' is shown below.

```
public class MyLookupLib {
  ...
  // For each column in a row, provide the property name and type
  //
  public static Map<String, Class> getAssetHistoryMetadata() {
    Map<String, Class> propertyNames = new HashMap<String, Class>();
    propertyNames.put("location", String.class);
    propertyNames.put("x_coord", Integer.class);
    propertyNames.put("y_coord", Integer.class);
    return propertyNames;
  }
...
  // Lookup rows based on assetId and assetCode
  //
  public static Map<String, Object>[] getAssetHistory(String assetId, String assetCode) {
    Map rows = new Map[2]; // this sample returns 2 rows
    for (int i = 0; i < 2; i++) {
      rows[i] = new HashMap();
      rows[i].put("location", "somevalue");
      rows[i].put("x_coord", 100);
      // ... set more values for each row
    }
    return rows;
  }
```

In the example above, the 'getAssetHistoryMetadata' method provides the property metadata: the names and types of properties in each row. The engine calls this method once per statement to determine event typing information.

The 'getAssetHistory' method returns an array of `Map` objects that are two rows. The implementation shown above is a simple example. The parameters to the method are the assetId and assetCode properties of the AssetMoveEvent joined to the method. The engine calls this method for each insert and remove stream event in AssetMoveEvent.

To indicate that no rows are found in a join, your application method may return either a `null` value or an array of size zero.

4.16. Creating and Using Named Windows

A *named window* is a global data window that can take part in many statement queries, and that can be inserted-into and deleted-from by multiple statements.

The `create window` clause declares a new named window. The named window starts up empty. Events must be inserted into the named window using the `insert into` clause. Events can also be deleted from a named window via the `on delete` clause.

Events enter the named window by means of `insert into` clause of a `select` statement. Events leave a named window either because the expiry policy of the declared data window removes events from the named window, or through statements that use the `on delete` clause to explicitly delete from a named window.

To query a named window, simply use the window name in the `from` clause of your statement, including statements that contain subqueries, joins and outer-joins.

4.16.1. Creating Named Windows: the `Create Window` clause

The `create window` statement creates a named window by specifying a window name and one or more data window views, as well as the type of event to hold in the named window.

The syntax for creating a named window is as follows:

```
create window window_name.view_specifications as [select list_of_properties from] event_type
```

The *window_name* you assign to the named window can be any identifier. The name should not already be in use as an event type or stream name.

The *view_specifications* are one or more data window views that define the expiry policy for removing events from the data window. Named windows must explicitly declare a data window view. This is required to ensure that the policy for retaining events in the data window is well defined. To keep all events, use the keep-all view: It indicates that the named window should keep all events and only remove events from the named window that are deleted via the `on delete` clause. The view specification can only list data window views, derived-value views are not allowed since these don't represent an expiry policy.

The `select` clause and *list_of_properties* are optional. If present, they specify the column names and, implicitly by definition of the event type, the column types of events held by the named window. Expressions other then column names are not allowed in the `select` list of properties. Wildcards (*) and wildcards with additional properties can also be used.

Finally, the *event_type* is required and provides the name of the event type of events held in the data window, unless column names and types have been explicitly selected via `select`.

The next statement creates a named window 'AllOrdersNamedWindow' for which the expiry policy is simply to keep all events. Assume that the event type 'OrderMapEventType' has been configured. The named window is to hold events of type 'OrderMapEventType':

```
create window AllOrdersNamedWindow.win:keepall() as OrderMapEventType
```

The below sample statement demonstrates the `select` syntax. It defines a named window in which each row has the three properties 'symbol', 'volume' and 'price'. This named window actively removes events from the window that are older then 30 seconds.

```
create window OrdersTimeWindow.win:time(30 sec) as
  select symbol, volume, price from OrderEvent
```

In an alternate form, the `as` keyword can be used to rename columns:

```
create window OrdersTimeWindow.win:time(30 sec) as
  select symbol as sym, volume as vol, price from OrderEvent
```

A new named window starts up empty. It must be explicitly inserted into by one or more statements, as discussed below.

If your application stops or destroys the statement that creates the named window, any consuming statements no longer receive insert or remove stream events. The named window can also not be deleted from after it was stopped or destroyed.

The `create window` statement posts to listeners any events that are inserted into the named window as new data. The statement posts all deleted events or events that expire out of the data window to listeners as the remove stream (old data). The named window contents can also be iterated on via the pull API to obtain the current contents of a named window.

4.16.2. Deleting From Named Windows: the `On Delete` clause

An `on delete` clause removes events from a named window. The clause can be used to remove all events, or only events that match certain criteria, or events that correlate with an arriving event or a pattern of arriving events.

The syntax for the `on delete` clause is as follows:

```
on event_type[(filter_criteria)] [as alias_name]
delete from window_name [as alias_name]
[where criteria_expression]
```

The *event_type* is the name or alias of the type of events that trigger removal from the named window. It is optionally followed by *filter_criteria* which are filter expressions to apply to arriving events. The optional `as` keyword can be used to assign an alias for use in the `where` clause. Patterns can also be specified in the `on` clause as described in the next section.

The *window_name* is the name of the named window to delete events from. The `as` keyword is also available to assign an alias to the named window.

The optional `where` clause contains a *criteria_expression* that correlates the arriving (triggering) event to the events to be removed from the named window. The *criteria_expression* may also simply filter for events in the named window to be removed from the named window.

The `iterator` of the `EPStatement` object representing the `on delete` clause can also be helpful: It returns the last batch of deleted events in response to the last triggering event, in any order, or null if the last triggering event did not remove any rows.

Let's look at a couple of examples. In the simplest form, this statement deletes all events from the named window 'AllOrdersNamedWindow' when any 'FlushOrderEvent' arrives:

```
on FlushOrderEvent delete from AllOrdersNamedWindow
```

This example adds a `where` clause to the example above. Upon arrival of a triggering 'ZeroVolumeEvent', the statement removes from the named window any orders that have a volume of zero or less:

```
on ZeroVolumeEvent delete from AllOrdersNamedWindow where volume <= 0
```

The next example shows a more complete use of the syntax, and correlates the triggering event with events held by the named window:

```
on NewOrderEvent(volume>0) as myNewOrders
delete from AllOrdersNamedWindow as myNamedWindow
where myNamedWindow.symbol = myNewOrders.symbol
```

In the above sample statement, only if a 'NewOrderEvent' event with a volume greater then zero arrives does the statement trigger. Upon triggering, all events in the named window that have the same value for the symbol property as the triggering 'NewOrderEvent' event are then removed from the named window. The statement also showcases the `as` keyword to assign alias names for use in the `where` expression.

For correlated queries (as above) that correlate triggering events with events held by a named window, Esper internally creates efficient indexes to enable high performance removal of events especially from named windows that hold large numbers of events.

Your application can subscribe a listener to your `on delete` statements to determine removed events. The statement post any events that are deleted from a named window to all listeners attached to the statement as new data. Upon iteration, the statement provides the last deleted event, if any.

Using Patterns in the `On Delete` Clause

By means of patterns the `on delete` clause and `on select` clause (described below) can look for more complex conditions to occur, possibly involving multiple events or the passing of time. The syntax for `on delete` with a pattern expression is show next:

```
on pattern [pattern_expression] [as alias_name]
delete from window_name [as alias_name]
[where criteria_expression]
```

The *pattern_expression* is any pattern that matches zero or more arriving events. Tags can be used to name events in the pattern and can occur in the optional `where` clause to correlate to events to be removed from a named window.

In the next example the triggering pattern fires every 10 seconds. The effect is that every 10 seconds the statement removes from 'MyNamedWindow' all rows:

```
on pattern [every timer:interval(10 sec)] delete from MyNamedWindow
```

The following example shows the use of tags in a pattern:

```
on pattern [every ord=OrderEvent(volume>0) or every flush=FlushOrderEvent]
delete from OrderWindow as win
where ord.id = win.id or flush.id = win.id
```

The pattern above looks for OrderEvent events with a volume value greater then zero and tags such events as 'ord'. The pattern also looks for FlushOrderEvent events and tags such events as 'flush'. The `where` clause deletes from the 'OrderWindow' named window any events that match in the value of the 'id' property either of the arriving events.

4.16.3. Inserting Into Named Windows

The `insert into` clause inserts events into named windows. Your application must ensure that the column names and types match the declared column names and types of the named window to be inserted into.

In this example we first create a named window using some of the columns of an OrderEvent event type:

```
create window OrdersWindow.win:keepall() as select symbol, volume, price from OrderEvent
```

The insert into the named window selects individual columns to be inserted:

```
insert into OrdersWindow(symbol, volume, price) select name, count, price from FXOrderEvent
// .. alternative form...
insert into OrdersWindow select name as symbol, vol as volume, price from FXOrderEvent
```

Following above statement, the engine enters every FXOrderEvent arriving into the engine into the named window 'OrdersWindow'.

The following EPL creates a named window for an event type backed by a Java class, and inserts into the window any 'OrderEvent' where the symbol value is IBM:

```
create window OrdersWindow as com.mycompany.OrderEvent
insert into OrdersWindow select * from com.mycompany.OrderEvent(symbol='IBM')
```

The last example adds one column named 'derivedPrice' to the 'OrderEvent' type by specifying a wildcard, and uses a user-defined function to populate the column:

```
create window OrdersWindow as select *, price as derivedPrice from OrderEvent
insert into OrdersWindow select *, MyFunc.func(price, percent) as derivedPrice from OrderEvent
```

Event representations based on Java base classes or interfaces, and subclasses or implementing classes, are compatible as these statements show:

```
// create a named window for the base class
create window OrdersWindow as select * from ProductBaseEvent

// The ServiceProductEvent class subclasses the ProductBaseEvent class
insert into OrdersWindow select * from ServiceProductEvent

// The MerchandiseProductEvent class subclasses the ProductBaseEvent class
insert into OrdersWindow select * from MerchandiseProductEvent
```

To avoid duplicate events stored in a named window, use a subquery to test whether an event already exists in the named window:

```
insert into OrdersWindow
select * from ServiceProductEvent as spe
where not exists (select * from OrdersWindow as win where win.id = spe.id)
```

A statement that removes events from a named window via the `on delete` clause and a statement that inserts events into a named window via the `insert into` can be combined to replace events in the named window, by creating the two statements in the order as indicated by the sample:

```
// create in this order
on ServiceProductEvent as spe delete from OrdersWindow as win where win.id = spe.id
insert into OrdersWindow select * from ServiceProductEvent
```

4.16.4. Selecting From Named Windows

A named window can be referred to by any statement in the `from` clause of the statement. Filter criteria can also be specified. Additional views may be used onto named windows however such views cannot include data window views.

A statement selecting all events from a named window 'AllOrdersNamedWindow' is shown next. The named window must first be created via the `create window` clause before use.

```
select * from AllOrdersNamedWindow
```

The statement as above simply receives the unfiltered insert and remove stream of the named window and reports that stream to its listeners. An iterator on such statement returns all events in the named window, if any.

The next statement derives an average price per symbol from all events posted by a named window:

```
select symbol, avg(price) from AllOrdersNamedWindow group by symbol
```

Your application may create a consuming statement such as above on an empty named window, or your application may create the above statement on an already filled named window. The engine provides correct results in either case: At the time of statement creation the Esper engine internally initializes the consuming statement from the current named window, also taking your declared filters into consideration. Thus, your statement deriving data from a named window does not start empty if the named window already holds one or more events.

If you require a subset of the data in the named window, you can specify one or more filter expressions onto the named window as shown here:

```
select symbol, avg(price) from AllOrdersNamedWindow(sector='energy') group by symbol
```

By adding a filter to the named window, the aggregation and grouping as well as any views that may be declared onto to the named window receive a filtered insert and remove stream. The above statement thus outputs, continuously, the average price per symbol for all orders in the named window that belong to a certain sector.

A side note on variables in filters filtering events from named windows: The engine initializes consuming statements at statement creation time and changes aggregation state continuously as events arrive. If the filter criteria contain variables and variable values changes, then the engine does not re-evaluate or re-build aggregation state. In such a case you may want to place variables in the `having` clause which evaluates on already-built aggregation state.

The following example further declares a view into the named window. Such a view can be a plug-in view or one of the built-in views, but cannot be a data window view (with the exception of the group-by view which is allowed).

```
select * from AllOrdersNamedWindow(volume>0, price>0).mycompany:mypluginview()
```

Data window views cannot be used onto named windows since named windows post insert and remove streams for the events entering and leaving the named window, thus the expiry policy and batch behavior are well defined by the data window declared for the named window. For example, the following is not allowed and fails at time of statement creation:

```
// not a valid statement
select * from AllOrdersNamedWindow.win:time(30 sec)
```

4.16.5. Triggered Select on Named Windows: the `On Select` clause

The `on select` clause performs a one-time, non-continuous query on a named window every time a triggering event arrives or a triggering pattern matches. The query can consider all events in the named window, or only events that match certain criteria, or events that correlate with an arriving event or a pattern of arriving events.

The syntax for the `on select` clause is as follows:

```
on event_type[(filter_criteria)] [as alias_name]
[insert into insert_into_def]
select select_list
from window_name [as alias_name]
[where criteria_expression]
[group by grouping_expression_list]
[having grouping_search_conditions]
[order by order_by_expression_list]
```

The *event_type* is the name or alias of the type of events that trigger the query against the named window. It is optionally followed by *filter_criteria* which are filter expressions to apply to arriving events. The optional `as` keyword can be used to assign an alias. Patterns can also be specified in the `on` clause, see the samples in Section 4.16.2.1, “Using Patterns in the On Delete Clause”.

The *insert into* clause works as described in Section 4.9, “Merging Streams and Continuous Insertion: the Insert Into Clause”. The *select* clause is described in Section 4.3, “Choosing Event Properties And Events: the Select Clause”. For all clauses the semantics are equivalent to a join operation: The properties of the triggering event or events are available in the `select` clause and all other clauses.

The *window_name* in the `from` clause is the name of the named window to select events from. The `as` keyword is also available to assign an alias to the named window. The `as` keyword is helpful in conjunction with wildcard in the `select` clause to select named window events via the syntax `select alias.*` .

The optional `where` clause contains a *criteria_expression* that correlates the arriving (triggering) event to the events to be considered from the named window. The *criteria_expression* may also simply filter for events in the named window to be considered by the query.

The `group by` clause, the `having` clause and the `order by` clause are all optional and work as described in earlier chapters.

The similarities and differences between an `on select` clause and a regular or outer join are as follows:

1. A join is evaluated when any of the streams participating in the join have new events (insert stream) or events leaving data windows (remove stream). A join is therefore bi-directional or multi-directional. However, the `on select` statement has one triggering event or pattern that causes the query to be evaluated and is thus uni-directional.

2. The query within the `on select` statement is not continuous: It executes only when a triggering event or pattern occurs. Aggregation and groups are computed anew considering the contents of the named window at the time the triggering event arrives.

The `iterator` of the `EPStatement` object representing the `on select` clause returns the last batch of selected events in response to the last triggering event, or null if the last triggering event did not select any rows.

For correlated queries that correlate triggering events with events held by a named window, Esper internally creates efficient indexes to enable high performance querying of events. It analyzes the `where` clause to build one or more indexes for fast lookup in the named window based on the properties of the triggering event.

The next statement demonstrates the concept. Upon arrival of a QueryEvent event the statement selects all events in the 'OrdersNamedWindow' named window:

```
on QueryEvent select win.* from OrdersNamedWindow as win
```

The engine executes the query on arrival of a triggering event, in this case a QueryEvent. It posts the query results to any listeners to the statement, in a single invocation, as the new data array. By prefixing the wildcard (*) selector with the stream name, the `select` clause returns only events of the named window and does not also return triggering events.

The `where` clause filters and correlates events in the named window with the triggering event, as shown next:

```
on QueryEvent(volume>0) as query
select query.symbol, query.volume, win.symbol  from OrdersNamedWindow as win
where win.symbol = query.symbol
```

Upon arrival of a QueryEvent, if that event has a value for the volume property that is greater then zero, the engine executes the query. The query considers all events currently held by the 'OrdersNamedWindow' that match the symbol property value of the triggering QueryEvent event. The engine then posts query results to the statement's listeners.

Aggregation, grouping and ordering of results are possible as this example shows:

```
on QueryEvent as queryEvent
select symbol, sum(volume) from OrdersNamedWindow as win
group by symbol
having volume > 0
order by symbol
```

The above statement outputs the total volume per symbol for those groups where the sum of the volume is greater then zero, ordered by symbol ascending. The engine computes and posts the output based on the current contents of the 'OrdersNamedWindow' named window considering all events in the named window, since the query does not have a `where` clause.

4.16.6. Triggered Playback from Named Windows: the `On Insert` clause

The `on insert` clause is an `on select` clause as described in the prior chapter with the addition of an `insert into` clause.

Similar to the `on select` clause, the engine executes the query when a triggering event arrives. It then provides the query results as an event stream to further statements. It populates the event stream that is named in the `insert into` clause.

The statement below provides the query results to any consumers of the MyOrderStream, upon arrival of a QueryEvent event:

```
on QueryEvent as query
insert into MyOrderStream
select win.* from OrdersNamedWindow as win
```

Here is a sample consuming statement of the MyOrderStream. The statement further filters the events provided by the `on insert` statement by user id and reports a total of volume per symbol:

```
select symbol, sum(volume) from MyOrderStream(userId='user1') group by symbol
```

4.17. Variables

A *variable* is a scalar value that is available for use in all statements including patterns. Variables can be used in an expression anywhere in a statement as well as in the `output` clause for output rate limiting.

Variables must first be declared or configured before use, by defining each variable's type and name. Variables can be created via the `create variable` syntax or declared by configuration. Variables can be assigned new values by using the `on set` syntax or via the `setVariableValue` methods on `EPRuntime`. The `EPRuntime` also provides method to read variable values.

The engine guarantees consistency and atomicity of variable reads and writes on a statement-level (this is a soft guarantee, see below). Variables are optimized for fast read access and are also multithread-safe.

4.17.1. Creating Variables: the `Create Variable` clause

The `create variable` syntax creates a new variable by defining the variable type and name. In alternative to the syntax, variables can also be declared in the runtime and engine configuration options.

The synopsis for creating a variable is as follows:

```
create variable variable_type variable_name [ = assignment_expression ]
```

The *variable_type* can be any of the following:

```
variable_type
        :  string
        |  char
        |  character
        |  bool
        |  boolean
        |  byte
        |  short
        |  int
        |  integer
        |  long
        |  double
        |  float
```

The *variable_name* is an identifier that names the variable. The variable name should not already be in use by another variable.

The `assignment_expression` is optional. Without an assignment expression the initial value for the variable is `null`. If present, it supplies the initial value for the variable.

The `EPStatement` object of the `create variable` statement provides access to variable values. The pull API methods `iterator` and `safeIterator` return the current variable value. Listeners to the `create variable` statement subscribe to changes in variable value: the engine posts new and old value of the variable to all listeners when the variable value is updated by an `on set` statement.

The example below creates a variable that provides a threshold value. The name of the variable is `var_threshold` and its type is `long`. The variable's initial value is `null` as no other value has been assigned:

```
create variable long var_threshold
```

This statement creates an integer-type variable named `var_output_rate` and initializes it to the value ten (10):

```
create variable integer var_output_rate = 10
```

In addition to creating a variable via the `create variable` syntax, the runtime and engine configuration API also allows adding variables. The next code snippet illustrates the use of the runtime configuration API to create a string-typed variable:

```
epService.getEPAdministrator().getConfiguration()
  .addVariable("myVar", String.class, "init value");
```

4.17.2. Setting Variable Values: the `On Set` clause

The `on set` statement assigns a new value to one or more variables when a triggering event arrives or a triggering pattern occurs. Use the `setVariableValue` methods on `EPRuntime` to assign variable values programmatically.

The synopsis for setting variable values is:

```
on event_type[(filter_criteria)] [as alias_name]
  set variable_name = expression [, variable_name = expression [,...]]
```

The *event_type* is the name or alias of the type of events that trigger the variable assignments. It is optionally followed by *filter_criteria* which are filter expressions to apply to arriving events. The optional `as` keyword can be used to assign an alias. Patterns can also be specified in the `on` clause.

The comma-separated list of variable names and expressions set the value of one or more variables. All new variable values are applied atomically: the changes to variable values by the `on set` statement become visible to other statements all at the same time. No changes are visible to other processing threads until the `on set` statement completed processing, and at that time all changes become visible at once.

The `EPStatement` object provides access to variable values. The pull API methods `iterator` and `safeIterator` return the current variable values for each of the variables set by the statement. Listeners to the statement subscribe to changes in variable values: the engine posts new variable values of all variables to any listeners.

In the following example, a variable by name `var_output_rate` has been declared previously. When a NewOutputRateEvent event arrives, the variable is updated to a new value supplied by the event property 'rate':

```
on NewOutputRateEvent set var_output_rate = rate
```

The next example shows two variables that are updated when a ThresholdUpdateEvent arrives:

```
on ThresholdUpdateEvent as t
  set var_threshold_lower = t.lower,
      var_threshold_higher = t.higher
```

The sample statement shown next counts the number of pattern matches using a variable. The pattern looks for OrderEvent events that are followed by CancelEvent events for the same order id within 10 seconds of the OrderEvent:

```
on pattern[every a=OrderEvent -> (CancelEvent(orderId=a.orderId) where timer:within(10 sec))]
  set var_counter = var_counter + 1
```

4.17.3. Using Variables

A variable name can be used in any expression and can also occur in an output rate limiting clause. This section presents examples and discusses performance, consistency and atomicity attributes of variables.

The next statement assumes that a variable named 'var_threshold' was created to hold a total price threshold value. The statement outputs an event when the total price for a symbol is greater then the current threshold value:

```
select symbol, sum(price) from TickEvent
group by symbol
having sum(price) > var_threshold
```

In this example we use a variable to dynamically change the output rate on-the-fly. The variable 'var_output_rate' holds the current rate at which the statement posts a current count to listeners:

```
select count(*) from TickEvent output every var_output_rate seconds
```

Variables are optimized towards high read frequency and lower write frequency. Variable reads do not incur locking overhead (99% of the time) while variable writes do incur locking overhead.

The engine softly guarantees consistency and atomicity of variables when your statement executes in response to an event or timer invocation. Variables acquire a stable value (implemented by versioning) when your statement starts executing in response to an event or timer invocation, and variables do not change value during execution. When one or more variable values are updated via `on set` statements, the changes to all updated variables become visible to statements as one unit and only when the `on set` statement completes successfully.

The atomicity and consistency guarantee is a soft guarantee. If any of your application statements, in response to an event or timer invocation, execute for a time interval longer then 15 seconds (default interval length), then the engine may use current variable values after 15 seconds passed, rather then then-current variable values at the time the statement started executing in response to an event or timer invocation.

The length of the time interval that variable values are held stable for the duration of execution of a given statement is by default 15 seconds, but can be configured via engine default settings.

Chapter 5. EPL Reference: Patterns

5.1. Event Pattern Overview

Event patterns match when an event or multiple events occur that match the pattern's definition. Patterns can also be time-based.

Pattern expressions can consist of filter expressions combined with pattern operators. Expressions can contain further nested pattern expressions by including the nested expression(s) in `()` round brackets.

There are 5 types of operators:

1. Operators that control pattern subexpression repetition: `every`
2. Logical operators: `and`, `or`, `not`
3. Temporal operators that operate on event order: `->` (followed-by)
4. Guards are where-conditions that control the lifecycle of subexpressions. Examples are `timer:within`.
5. Observers observe time events as well as other events. Examples are `timer:interval` and `timer:at`.

5.2. How to use Patterns

5.2.1. Pattern Syntax

This is an example pattern expression that matches on every `ServiceMeasurement` events in which the value of the `latency` event property is over 20 seconds, and on every `ServiceMeasurement` event in which the `success` property is false. Either one or the other condition must be true for this pattern to match.

```
every (spike=ServiceMeasurement(latency>20000) or error=ServiceMeasurement(success=false))
```

In the example above, the pattern expression starts with an `every` operator to indicate that the pattern should fire for every matching events and not just the first matching event. Within the `every` operator in round brackets is a nested pattern expression using the `or` operator. The left hand of the `or` operator is a filter expression that filters for events with a high latency value. The right hand of the operator contains a filter expression that filters for events with error status. Filter expressions are explained in Section 5.4, “Filter Expressions In Patterns”.

The example above assigned the tags `spike` and `error` to the events in the pattern. The tags are important since the engine only places tagged events into the output event(s) that a pattern generates, and that the engine supplies to listeners of the pattern statement. The tags can further be selected in the select-clause of an EPL statement as discussed in Section 4.4.2, “Pattern-based Event Streams”.

Patterns can also contain comments within the pattern as outlined in Section 4.2.2, “Using Comments”.

Pattern statements are created via the `EPAdministrator` interface. The `EPAdministrator` interface allows to create pattern statements in two ways: Pattern statements that want to make use of the EPL `select` clause or any other EPL constructs use the `createEPL` method to create a statement that specifies one or more pattern expressions. EPL statements that use patterns are described in more detail in Section 4.4.2, “Pattern-based Event Streams”. Use the syntax as shown in below example.

```
EPAdministrator admin = EPServiceProviderManager.getDefaultProvider().getEPAdministrator();

String eventName = ServiceMeasurement.class.getName();
```

```
EPStatement myTrigger = admin.createEPL("select * from pattern [" +
  "every (spike=" + eventName + "(latency>20000) or error=" + eventName + "(success=false))]");
```

Pattern statements that do not need to make use of the EPL `select` clause or any other EPL constructs can use the `createPattern` method, as in below example.

```
EPStatement myTrigger = admin.createPattern(
  "every (spike=" + eventName + "(latency>20000) or error=" + eventName + "(success=false))");
```

5.2.2. Subscribing to Pattern Events

When a pattern fires it publishes one or more events to any listeners to the pattern statement. The listener interface is the `com.espertech.esper.client.UpdateListener` interface.

The example below shows an anonymous implementation of the `com.espertech.esper.client.UpdateListener` interface. We add the anonymous listener implementation to the `myPattern` statement created earlier. The listener code simply extracts the underlying event class.

```
myPattern.addListener(new UpdateListener()
{
  public void update(EventBean[] newEvents, EventBean[] oldEvents)
  {
    ServiceMeasurement spike = (ServiceMeasurement) newEvents[0].get("spike");
    ServiceMeasurement error = (ServiceMeasurement) newEvents[0].get("error");
    ... // either spike or error can be null, depending on which occurred
    ... // add more logic here
  }
});
```

Listeners receive an array of `EventBean` instances in the `newEvents` parameter. There is one `EventBean` instance passed to the listener for each combination of events that matches the pattern expression. At least one `EventBean` instance is always passed to the listener.

The properties of each `EventBean` instance contain the underlying events that caused the pattern to fire, if events have been named in the filter expression via the `name=eventType` syntax. The property name is thus the name supplied in the pattern expression, while the property type is the type of the underlying class, in this example `ServiceMeasurement`.

5.2.3. Pulling Data from Patterns

Data can also be obtained from pattern statements via the `safeIterator()` and `iterator()` methods on `EPStatement` (the pull API). If the pattern had fired at least once, then the iterator returns the last event for which it fired. The `hasNext()` method can be used to determine if the pattern had fired.

```
if (myPattern.iterator().hasNext())
{
        ServiceMeasurement event = (ServiceMeasurement) view.iterator().next().get("alert");
    ... // some more code here to process the event
}
else
{
    ... // no matching events at this time
}
```

5.3. Operator Precedence

The operators at the top of this table take precedence over operators lower on the table.

Table 5.1. Pattern Operator Precedence

Precedence	Operator	Description	Example
1	unary	`every, not`	`every MyEvent` `timer:interval(5 min) and not MyEvent`
2	Guard postfix	`where timer:within` (or plug-in pattern guard)	`MyEvent where timer:within(1 sec)`
3	and	`and`	`every (MyEvent and MyOtherEvent)`
4	or	`or`	`every (MyEvent or MyOtherEvent)`
5	followed-by	`->`	`every (MyEvent -> MyOtherEvent)`

If you are not sure about the precedence, please consider placing parenthesis `()` around your subexpressions. Parenthesis can also help make expressions easier to read and understand.

Note that we are also providing the EPL grammar as a HTML file as part of the documentation set on the project website.

The following table outlines sample equivalent expressions, with and without the use of parenthesis for subexpressions.

Table 5.2. Equivalent Pattern Expressions

Expression	Equivalent	Reason
every A or B	(every A) or B	The `every` operator has higher precedence then the `or` operator
every A -> B or C	(every A) -> (B or C)	The `or` operator has higher precedence then the `followed-by` operator
A and B or C	(A and B) or C	The `and` operator has higher precedence then the `or` operator
every A where timer:within(5)	every (A where timer:within(5))	The `every` operator has higher precedence then the `timer:within` guard postfix

5.4. Filter Expressions In Patterns

The simplest form of filter is a filter for events of a given type without any conditions on the event property values. This filter matches any event of that type regardless of the event's properties. The example below is such a filter. Note that this event pattern would stop firing as soon as the first RfidEvent is encountered.

```
com.mypackage.myevents.RfidEvent
```

To make the event pattern fire for every RfidEvent and not just the first event, use the `every` keyword.

```
every com.mypackage.myevents.RfidEvent
```

The example above specifies the fully-qualified Java class name as the event type. Via configuration, the event pattern above can be simplified by using the alias that has been defined for the event type.

```
every RfidEvent
```

Interfaces and superclasses are also supported as event types. In the below example `IRfidReadable` is an interface class, and the statement matches any event that implements this interface:

```
every org.myorg.rfid.IRfidReadable
```

The filtering criteria to filter for events with certain event property values are placed within parenthesis after the event type name:

```
RfidEvent(category="Perishable")
```

All expressions can be used in filters, including static method invocations that return a boolean value:

```
RfidEvent(com.mycompany.MyRFIDLib.isInRange(x, y) or (x<0 and y < 0))
```

Filter expressions can be separated via a single comma '`,`'. The comma represents a logical AND between expressions:

```
RfidEvent(zone=1, category=10)
...is equivalent to...
RfidEvent(zone=1 and category=10)
```

The following set of operators are highly optimized through indexing and are the preferred means of filtering high-volume event streams:

- equals `=`
- not equals `!=`
- comparison operators `< , > , >=, <=`
- ranges
  - use the `between` keyword for a closed range where both endpoints are included
  - use the `in` keyword and round `()` or square brackets `[]` to control how endpoints are included
  - for inverted ranges use the `not` keyword and the `between` or `in` keywords
- list-of-values checks using the `in` keyword or the `not in` keywords followed by a comma-separated list of values

At compile time as well as at run time, the engine scans new filter expressions for subexpressions that can be indexed. Indexing filter values to match event properties of incoming events enables the engine to match incoming events faster. The above list of operators represents the set of operators that the engine can best convert into indexes. The use of comma or logical `and` in filter expressions does not impact optimizations by the engine.

For more information on filters please see Section 4.4.1, “Filter-based Event Streams”.

Filter criteria can also refer to events matching prior named events in the same expression. Below pattern is an example in which the pattern matches once for every RfidEvent that is preceded by an RfidEvent with the same asset id.

```
every A=RfidEvent -> B=RfidEvent(assetId=A.assetId)
```

The syntax shown above allows filter criteria to reference prior results by specifying the event name tag of the prior event, and the event property name. This syntax can be used in all filter operators or expressions including ranges and the `in` set-of-values check:

```
every A=RfidEvent ->
  B=RfidEvent(MyLib.isInRadius(A.x, A.y, x, y) and zone in (1, A.zone))
```

5.5. Pattern Operators

5.5.1. Every

The `every` operator indicates that the pattern subexpression should restart when the subexpression qualified by the `every` keyword evaluates to true or false. Without the `every` operator the pattern subexpression stops when the pattern subexpression evaluates to true or false.

Thus the `every` operator works like a factory for the pattern subexpression contained within. When the pattern subexpression within it fires and thus quits checking for events, the `every` causes the start of a new pattern subexpression listening for more occurances of the same event or set of events.

Every time a pattern subexpression within an `every` operator turns true the engine starts a new active subexpression looking for more event(s) or timing conditions that match the pattern subexpression. If the `every` operator is not specified for a subexpression, the subexpression stops after the first match was found.

This pattern fires when encountering event A and then stops looking.

```
A
```

This pattern keeps firing when encountering event A, and doesn't stop looking.

```
every A
```

Let's consider an example event sequence as follows.

A_1 B_1 C_1 B_2 A_2 D_1 A_3 B_3 E_1 A_4 F_1 B_4

Table 5.3. 'Every' operator examples

Example	Description
`every ( A -> B )`	Detect event A followed by event B. At the time when B occurs the pattern matches, then the pattern matcher restarts and looks for event A again. 1. Matches on B_1 for combination $\{A_1, B_1\}$ 2. Matches on B_3 for combination $\{A_2, B_3\}$ 3. Matches on B_4 for combination $\{A_4, B_4\}$

Example	Description
`every A -> B`	The pattern fires for every event A followed by an event B. 1. Matches on B_1 for combination {A_1, B_1} 2. Matches on B_3 for combination {A_2, B_3} and {A_3, B_3} 3. Matches on B_4 for combination {A_4, B_4}
`A -> every B`	The pattern fires for an event A followed by every event B. 1. Matches on B_1 for combination {A_1, B_1}. 2. Matches on B_2 for combination {A_1, B_2}. 3. Matches on B_3 for combination {A_1, B_3} 4. Matches on B_4 for combination {A_1, B_4}
`every A -> every B`	The pattern fires for every event A followed by every event B. 1. Matches on B_1 for combination {A_1, B_1}. 2. Matches on B_2 for combination {A_1, B_2}. 3. Matches on B_3 for combination {A_1, B_3} and {A_2, B_3} and {A_3, B_3} 4. Matches on B_4 for combination {A_1, B_4} and {A_2, B_4} and {A_3, B_4} and {A_4, B_4}

The examples show that it is possible that a pattern fires for multiple combinations of events that match a pattern expression. Each combination is posted as an `EventBean` instance to the `update` method in the `UpdateListener` implementation.

Let's consider the `every` operator in conjunction with a subexpression that matches 3 events that follow each other:

```
every (A -> B -> C)
```

The pattern first looks for event A. When event A arrives, it looks for event B. After event B arrives, the pattern looks for event C. Finally when event C arrives the pattern fires. The engine then starts looking for event A again.

Assume that between event B and event C a second event A_2 arrives. The pattern would ignore the A_2 entirely since it's then looking for event C. As observed in the prior example, the `every` operator restarts the subexpression `A -> B -> C` only when the subexpression fires.

In the next statement the `every` operator applies only to the A event, not the whole subexpression:

```
every A -> B -> C
```

This pattern now matches for any event A that is followed by an event B and then event C, regardless of when the event A arrives. Oftentimes this can be practical in combination with the `and not` syntax and the `timer:within` syntax as the next example shows.

`Every` Operator Example

In this example we consider a generic pattern in which the pattern must match for each A event followed by a B and followed by a C event, in which both B and C must arrive within 1 hour of the A event. The first approach to the pattern is as follows:

```
every A  -> (B -> C) where timer:within(1 hour)
```

Consider the following sequence of events arriving:

A_1 A_2 B_1 C_1 B_2 C_2

First, the pattern as above never stops looking for A events since the `every` operator instructs the pattern to keep looking for A events.

When A_1 arrives, the pattern starts a new subexpression that keeps A_1 in memory and looks for any B event. At the same time, it also keeps looking for more A events.

When A_2 arrives, the pattern starts a new subexpression that keeps A_2 in memory and looks for any B event. At the same time, it also keeps looking for more A events.

After the arrival of A_2, there are 3 subexpressions active:

1. The first active subexpression with A_1 in memory, looking for any B event
2. The second active subexpression with A_2 in memory, looking for any B event
3. A third active subexpression, looking for the next A event

In the pattern above, we have specified a 1-hour lifetime for subexpressions looking for B and C events. Thus, if no B and no C event arrive within 1 hour after A_1, the first subexpression goes away. If no B and no C event arrive within 1 hour after A_2, the second subexpression goes away. The third subexpression however stays around looking for more A events.

The pattern as shown above thus matches on arrival of C_1 for combination $\{A_1, B_1, C_1\}$ and for combination $\{A_2, B_1, C_1\}$, provided that B_1 and C_1 arrive within an hour of A_1 and A_2.

You may now ask how to match on $\{A_1, B_1, C_1\}$ and $\{A_2, B_2, C_2\}$ instead, since you may need to correlate on a given property.

The pattern as discussed above matches every A event followed by the first B event followed by the next C event, and doesn't specifically qualify the B or C events to look for based on the A event. To look for specific B and C events in relation to a given A event, the correlation must use one or more of the properties of the A event, such as the "id" property:

```
every a=A -> (B(id=a.id -> C(id=a.id)) where timer:within(1 hour)
```

The pattern as shown above thus matches on arrival of C_1 for combination $\{A_1, B_1, C_1\}$ and on arrival of C_2 for combination $\{A_2, B_2, C_2\}$.

Sensor Example

This example looks at temperature sensor events named Sample. The pattern detects when 3 sensor events indicate a temperature of more then 50 degrees uninterrupted within 90 seconds of the first event, considering events for the same sensor only.

```
every sample=Sample(temp > 50) ->
( (Sample(sensor=sample.sensor, temp > 50) and not Sample(sensor=sample.sensor, temp <= 50))
  ->
  (Sample(sensor=sample.sensor, temp > 50) and not Sample(sensor=sample.sensor, temp <= 50))
 ) where timer:within(90 seconds))
```

The pattern starts a new subexpression in the round braces after the first followed-by operator for each time a sensor indicated more then 50 degrees. Each subexpression then lives a maximum of 90 seconds. Each subexpression ends if a temperature of 50 degress or less is encountered for the same sensor. Only if 3 temperature events in a row indicate more then 50 degrees, and within 90 seconds of the first event, and for the same sensor, does this pattern fire.

5.5.2. And

Similar to the Java && operator the `and` operator requires both nested pattern expressions to turn true before the whole expression turns true (a join pattern).

Pattern matches when both event A and event B are found.

```
A and B
```

Pattern matches on any sequence A followed by B and C followed by D, or C followed by D and A followed by B

```
(A -> B) and (C -> D)
```

Note that in an `and` pattern expression it is not possible to correlate events based on event property values. For example, this is an invalid pattern:

```
// This is NOT valid
a=A and B(id = a.id)
```

The above expression is invalid as it relies on the order of arrival of events, however in an `and` expression the order of events is not specified and events fulfill an `and` condition in any order. The above expression can be changed to use the followed-by operator:

```
// This is valid
a=A -> B(id = a.id)
// another example using 'and'...
a=A -> (B(id = a.id) and C(id = a.id))
```

5.5.3. Or

Similar to the Java "||" operator the `or` operator requires either one of the expressions to turn true before the whole expression turns true.

Look for either event A or event B. As always, A and B can itself be nested expressions as well.

```
A or B
```

Detect all stock ticks that are either above or below a threshold.

```
every (StockTick(symbol='IBM', price < 100) or StockTick(symbol='IBM', price > 105)
```

5.5.4. Not

The `not` operator negates the truth value of an expression. Pattern expressions prefixed with `not` are automatically defaulted to true upon start, and turn permanently false when the expression within turns true.

The `not` operator is generally used in conjuntion with the `and` operator or subexpressions as the below examples show.

This pattern matches only when an event A is encountered followed by event B but only if no event C was encountered before either event A and B, counting from the time the pattern is started:

```
(A -> B) and not C
```

Assume we'd like to detect when an event A is followed by event D, without any B or C events between the A and D events:

```
A -> (D and not (B or C))
```

It may help your understanding to discuss a pattern that uses the `or` operator and the `not` operator together:

```
a=A -> (b=B or not C)
```

In the pattern above, when an A event arrives then the engine starts the subexpression `B or not C`. As soon as the subexpression starts, the `not` operator turns to true. The `or` expression turns true and thus your listener receives an invocation providing the A event in the property 'a'. The subexpression does not end and continues listening for B and C events. Upon arrival of a B event your listener receives a second invocation. If instead a C event arrives, the `not` turns permanently false however that does not affect the `or` operator (but would end an `and` operator).

5.5.5. Followed-by

The followed by `->` operator specifies that first the left hand expression must turn true and only then is the right hand expression evaluated for matching events.

Look for event A and if encountered, look for event B. As always, A and B can itself be nested event pattern expressions.

```
A -> B
```

This is a pattern that fires when 2 status events indicating an error occur one after the other.

```
StatusEvent(status='ERROR') -> StatusEvent(status='ERROR')
```

5.6. Pattern Guards

Guards are where-conditions that control the lifecycle of subexpressions. Custom guard functions can also be used. The section Chapter 11, *Extension and Plug-in* outlines guard plug-in development in greater detail.

Take as an example the following pattern expression:

```
MyEvent where timer.within(10 sec)
```

In this pattern the `timer:within` guard controls the subexpression that is looking for MyEvent events. The guard terminates the subexpression looking for MyEvent events after 10 seconds after start of the pattern. Thus the pattern alerts only once when the first MyEvent event arrives within 10 seconds after start of the pattern.

The `every` keyword requires additional discussion since it also controls subexpression lifecycle. Let's add the `every` keyword to the example pattern:

```
every MyEvent where timer.within(10 sec)
```

The difference to the pattern without `every` is that each MyEvent event that arrives now starts a new subexpression, including a new guard, looking for a further MyEvent event. The result is that, when a MyEvent arrives within 10 seconds after pattern start, the pattern execution will look for the next MyEvent event to arrive within 10 seconds after the previous one.

By placing parentheses around the `every` keyword and its subexpression, we can have the `every` under the control of the guard:

```
(every MyEvent) where timer.within(10 sec)
```

In the pattern above, the guard terminates the subexpression looking for all MyEvent events after 10 seconds after start of the pattern. This pattern alerts for all MyEvent events arriving within 10 seconds after pattern start, and then stops.

Guards do not change the truth value of the subexpression of which the guard controls the lifecycle, and therefore do not cause a restart of the subexpression when used with the `every` operator. For example, the next pattern stops returning matches after 10 seconds unless a match occured within 10 seconds after pattern start:

```
every ( (A and B) where timer.within(10 sec) )
```

5.6.1. timer:within

The `timer:within` guard acts like a stopwatch. If the associated pattern expression does not turn true within the specified time period it is stopped and permanently false. The `timer:within` guard takes a time period (see Section 4.2.1, “Specifying Time Periods”) or a number of seconds as a parameter.

This pattern fires if an A event arrives within 5 seconds after statement creation.

```
A where timer:within (5 seconds)
```

This pattern fires for all A events that arrive within 5 seconds. After 5 seconds, this pattern stops matching even if more A events arrive.

```
(every A) where timer:within (5 seconds)
```

This pattern is similar to the first pattern but here every time A arrives within 5 seconds, the pattern begins looking for A for another 5 seconds. As long as A events arrive within 5 seconds after the last A, the pattern does not stop matching.

```
every (A where timer:within (5 sec))
```

This pattern matches for any one A or B event in the next 5 seconds.

```
( A or B ) where timer:within (5 sec)
```

This pattern matches for any 2 errors that happen 10 seconds within each other.

```
every (StatusEvent(status='ERROR') -> StatusEvent(status='ERROR') where timer:within (10 sec))
```

The following guards are equivalent:

```
timer:within(2 minutes 5 seconds)
timer:within(125 sec)
timer:within(125)
```

5.7. Pattern Observers

Observers observe time-based events for which the thread-of-control originates by the engine timer thread. Custom observers can also be developed that observe timer events or other engine-external events. The section Chapter 11, *Extension and Plug-in* outlines observer plug-in development in greater detail.

5.7.1. timer:interval

The `timer:interval` observer waits for the defined time before the truth value of the observer turns true. The observer takes a time period (see Section 4.2.1, “Specifying Time Periods”) or a number of seconds as a parameter.

After event A arrived wait 10 seconds then indicate that the pattern matches.

```
A -> timer:interval(10 seconds)
```

The pattern below fires every 20 seconds.

```
every timer:interval(20 sec)
```

The next example pattern fires for every event A that is not followed by an event B within 60 seconds after event A arrived. B must have the same "id" property value as A.

```
every a=A -> (timer:interval(60 sec) and not B(id=a.id))
```

5.7.2. timer:at

The `timer:at` observer is similar in function to the Unix “crontab” command. At a specified time the expression turns true. The `at` operator can also be made to pattern match at regular intervals by using an `every` operator in front of the `timer:at` operator.

The syntax is: `timer:at (minutes, hours, days of month, months, days of week [, seconds]).`

The value for seconds is optional. Each element allows wildcard `*` values. Ranges can be specified by means of lower bounds then a colon ‘:’ then the upper bound. The division operator `*/x` can be used to specify that every x_{th} value is valid. Combinations of these operators can be used by placing these into square brackets([]).

This expression pattern matches every 5 minutes past the hour.

```
every timer:at(5, *, *, *, *)
```

The below `timer:at` pattern matches every 15 minutes from 8am to 5pm on even numbered days of the month as well as on the first day of the month.

```
timer:at (*/15, 8:17, [*/2, 1], *, *)
```

The below table outlines the fields, valid values and keywords available for each field:

Table 5.4. Properties offered by sample statement aggregating price

Field Name	Mandatory?	Allowed Values	Additional Keywords
Minutes	yes	0 - 59	
Hours	yes	0 - 23	
Days Of Month	yes	1 - 31	last, weekday, lastweekday
Months	yes	1 - 12	
Days Of Week	yes	0 (Sunday) - 6 (Saturday)	last
Seconds	no	0 - 59	

The keyword `last` used in the days-of-month field means the last day of the month (current month). To specify the last day of another month, a value for the month field has to be provided. For example: `timer:at(*, *, last,2,*)` is the last day of February.

The `last` keyword in the day-of-week field by itself simply means Saturday. If used in the day-of-week field after another value, it means "the last xxx day of the month" - for example "5 last" means "the last friday of the month". So the last Friday of the current month will be: `timer:at(*, *, *, *, 5 last)`. And the last Friday of June: `timer:at(*, *, *, 6, 5 last)`.

The keyword `weekday` is used to specify the weekday (Monday-Friday) nearest the given day. Variant could include month like in: `timer:at(*, *, 30 weekday, 9, *)` which is Friday September 28th (no jump over month).

The keyword `lastweekday` is a combination of two parameters, the `last` and the `weekday` keywords. A typical example could be: `timer:at(*, *, *, lastweekday, 9, *)` which will define Friday September 28th (example year is 2007).

Chapter 6. EPL Reference: Operators

Esper arithmetic and logical operator precedence follows Java standard arithmetic and logical operator precedence.

6.1. Arithmetic Operators

The below table outlines the arithmetic operators available.

Table 6.1. Syntax and results of arithmetic operators

Operator	Description
+, -	As unary operators they denote a positive or negative expression. As binary operators they add or subtract.
*, /	Multiplication and division are binary operators.
%	Modulo binary operator.

6.2. Logical And Comparsion Operators

The below table outlines the logical and comparison operators available.

Table 6.2. Syntax and results of logical and comparison operators

Operator	Description
NOT	Returns true if the following condition is false, returns false if it is true.
OR	Returns true if either component condition is true, returns false if both are false.
AND	Returns true if both component conditions are true, returns false if either is false.
=, !=, <, > <=, >=,	Comparison.

6.3. Concatenation Operators

The below table outlines the concatenation operators available.

Table 6.3. Syntax and results of concatenation operators

Operator	Description
||	Concatenates character strings

6.4. Binary Operators

The below table outlines the binary operators available.

Table 6.4. Syntax and results of binary operators

Operator	Description
&	Bitwise AND if both operands are numbers; conditional AND if both operands are boolean
|	Bitwise OR if both operands are numbers; conditional OR if both operands are boolean
^	Bitwise exclusive OR (XOR)

6.5. Array Definition Operator

The `{` and `}` curly braces are array definition operators following the Java array initialization syntax. Arrays can be useful to pass to user-defined functions or to select array data in a select clause.

Array definitions consist of zero or more expressions within curly braces. Any type of expression is allowed within array definitions including constants, arithmetic expressions or event properties. This is the syntax of an array definition:

```
{ [expression [,expression...]] }
```

Consider the next statement that returns an event property named `actions`. The engine populates the `actions` property as an array of `java.lang.String` values with a length of 2 elements. The first element of the array contains the `observation` property value and the second element the `command` property value of `RFIDEvent` events.

```
select {observation, command} as actions from RFIDEvent
```

The engine determines the array type based on the types returned by the expressions in the array definiton. For example, if all expressions in the array definition return integer values then the type of the array is `java.lang.Integer[]`. If the types returned by all expressions are compatible number types, such as integer and double values, the engine coerces the array element values and returns a suitable type, `java.lang.Double[]` in this example. The type of the array returned is `Object[]` if the types of expressions cannot be coerced or return object values. Null values can also be used in an array definition.

Arrays can come in handy for use as parameters to user-defined functions:

```
select * from RFIDEvent where Filter.myFilter(zone, {1,2,3})
```

6.6. The 'in' Keyword

The `in` keyword determines if a given value matches any value in a list. The syntax of the keyword is:

```
test_expression [not] in (expression [,expression...] )
```

The *test_expression* is any valid expression. The keyword is followed by a list of expressions to test for a match. The optional `not` keyword specifies that the result of the predicate be negated.

The result of an `in` expression is of type `Boolean`. If the value of *test_expression* is equal to any expression from the comma-separated list, the result value is `true`. Otherwise, the result value is `false`. All expressions must be of the same type as or a compatible type to *test_expression*.

The next example shows how the `in` keyword can be applied to select certain command types of RFID events:

```
select * from RFIDEvent where command in ('OBSERVATION', 'SIGNAL')
```

The statement is equivalent to:

```
select * from RFIDEvent where command = 'OBSERVATION' or command = 'SIGNAL'
```

6.7. The 'between' Keyword

The `between` keyword specifies a range to test. The syntax of the keyword is:

```
test_expression [not] between begin_expression and end_expression
```

The *test_expression* is any valid expression and is the expression to test for in the range defined by *begin_expression* and *end_expression*. The `not` keyword specifies that the result of the predicate be negated.

The result of a `between` expression is of type `Boolean`. If the value of *test_expression* is greater then or equal to the value of *begin_expression* and less than or equal to the value of *end_expression*, the result is `true`.

The next example shows how the `between` keyword can be used to select events with a price between 55 and 60 (inclusive).

```
select * from StockTickEvent where price between 55 and 60
```

The equivalent expression without `between` is:

```
select * from StockTickEvent where price >= 55 and price <= 60
```

And also equivalent to:

```
select * from StockTickEvent where price between 60 and 55
```

6.8. The 'like' Keyword

The like keyword provides standard SQL pattern matching. SQL pattern matching allows you to use '_' to match any single character and '%' to match an arbitrary number of characters (including zero characters). In Esper, SQL patterns are case-sensitive by default. The syntax of like is:

```
test_expression [not] like pattern_expression [escape string_literal]
```

The *test_expression* is any valid expression yielding a String-type or a numeric result. The optional not keyword specifies that the result of the predicate be negated. The like keyword is followed by any valid standard SQL *pattern_expression* yielding a String-typed result. The optional escape keyword signals the escape character to escape '_' and '%' values in the pattern.

The result of a like expression is of type Boolean. If the value of *test_expression* matches the *pattern_expression*, the result value is true. Otherwise, the result value is false.

An example for the like keyword is below.

```
select * from PersonLocationEvent where name like '%Jack%'
```

The escape character can be defined as follows. In this example the where-clause matches events where the suffix property is a single '_' character.

```
select * from PersonLocationEvent where suffix like '!_' escape '!'
```

6.9. The 'regexp' Keyword

The regexp keyword is a form of pattern matching based on regular expressions implemented through the Java java.util.regex package. The syntax of regexp is:

```
test_expression [not] regexp pattern_expression
```

The *test_expression* is any valid expression yielding a String-type or a numeric result. The optional not keyword specifies that the result of the predicate be negated. The regexp keyword is followed by any valid regular expression *pattern_expression* yielding a String-typed result.

The result of a regexp expression is of type Boolean. If the value of *test_expression* matches the regular expression *pattern_expression*, the result value is true. Otherwise, the result value is false.

An example for the regexp keyword is below.

```
select * from PersonLocationEvent where name regexp '*Jack*'
```

Chapter 7. EPL Reference: Functions

7.1. Single-row Function Reference

Single-row functions return a single value for every single result row generated by your statement. These functions can appear anywhere where expressions are allowed.

Esper allows static Java library methods as single-row functions, and also features built-in single-row functions. In addition, Esper allows instance method invocations on named streams.

Esper auto-imports the following Java library packages:

- java.lang.*
- java.math.*
- java.text.*
- java.util.*

Thus Java static library methods can be used in all expressions as shown in below example:

```
select symbol, Math.round(volume/1000)
from StockTickEvent.win:time(30 sec)
```

In general, arbitrary Java class names have to be fully qualified (e.g. java.lang.Math) but Esper provides a mechanism for user-controlled imports of classes and packages as outlined in Section 10.4.4, “Class and package imports”.

The below table outlines the built-in single-row functions available.

Table 7.1. Syntax and results of single-row functions

Single-row Function	Result
`case value when compare_value then result [when compare_value then result ...] [else result] end`	Returns `result` where the first `value` equals `compare_value`.
`case when condition then result [when condition then result ...] [else result] end`	Returns the `result` for the first condition that is true.
`cast(expression, type_name)`	Casts the result of an expression to the given type.
`coalesce(expression, expression [, expression ...])`	Returns the first non-`null` value in the list, or `null` if there are no non-`null` values.
`current_timestamp[()]`	Returns the current engine time as a `long` mil-

Single-row Function	Result
	lisecond value. Reserved keyword with optional parenthesis.
`exists(dynamic_property_name)`	Returns true if the dynamic property exists for the event, or false if the property does not exist.
`instanceof(expression, type_name [, type_name ...])`	Returns true if the expression returns an object whose type is one of the types listed.
`max(expression, expression [, expression ...])`	Returns the highest numeric value among the 2 or more comma-separated expressions.
`min(expression, expression [, expression ...])`	Returns the lowest numeric value among the 2 or more comma-separated expressions.
`prev(expression, event_property)`	Returns a property value of a previous event, relative to the event order within a data window
`prior(integer, event_property)`	Returns a property value of a prior event, relative to the natural order of arrival of events

7.1.1. The `case` Control Flow Function

The `case` control flow function has two versions. The first version takes a value and a list of compare values to compare against, and returns the result where the first value equals the compare value. The second version takes a list of conditions and returns the result for the first condition that is true.

The return type of a `case` expression is the compatible aggregated type of all return values.

The example below shows the first version of a `case` statement. It has a `String` return type and returns the value 'one'.

```
select case 1 when 1 then 'one' when 2 then 'two' else 'more' end from ...
```

The second version of the `case` function takes a list of conditions. The next example has a `Boolean` return type and returns the boolean value true.

```
select case when 1>0 then true else false end from ...
```

7.1.2. The `cast` Function

The `cast` function casts the return type of an expression to a designated type. The function accepts two parameters: The first parameter is the property name or expression that returns the value to be casted. The second parameter is the type to cast to.

Valid parameters for the second (type) parameter are:

- Any of the Java built-in types: `int, long, byte, short, char, double, float, string`, where `string` is a short notation for `java.lang.String`. The type name is not case-sensitive. For example:

```
cast(price, double)
```

- The fully-qualified class name of the class to cast to, for example:

```
cast(product, org.myproducer.Product)
```

The `cast` function is often used to provide a type for dynamic (unchecked) properties. Dynamic properties are properties whose type is not known at compile type. These properties are always of type `java.lang.Object`.

The `cast` function as shown in the next statement casts the dynamic "price" property of an "item" in the OrderEvent to a double value.

```
select cast(item.price?, double) from OrderEvent
```

The `cast` function returns a `null` value if the expression result cannot be casted to the desired type, or if the expression result itself is `null`.

The `cast` function adheres to the following type conversion rules:

- For all numeric types, the `cast` function utilitzes `java.lang.Number` to convert numeric types, if required.
- For casts to `string` or `java.lang.String`, the function calls `toString` on the expression result.
- For casts to other objects including application objects, the `cast` function considers a Java class's superclasses as well as all directly or indirectly-implemented interfaces by superclasses .

7.1.3. The `Coalesce` Function

The result of the `coalesce` function is the first expression in a list of expressions that returns a non-null value. The return type is the compatible aggregated type of all return values.

This example returns a String-typed result of value 'foo':

```
select coalesce(null, 'foo') from ...
```

7.1.4. The `Current_Timestamp` Function

The `current_timestamp` function is a reserved keyword and requires no parameters. The result of the `current_timestamp` function is the `long`-type millisecond value of the current engine system time.

The function returns the current engine timestamp at the time of expression evaluation. When using external-timer events, the function provides the last value of the externally-supplied time at the time of expression evaluation.

This example selects the current engine time:

```
select current_timestamp from MyEvent
// equivalent to
```

```
select current_timestamp() from MyEvent
```

7.1.5. The `Exists` Function

The `exists` function returns a boolean value indicating whether the dynamic property, provided as a parameter to the function, exists on the event. The `exists` function accepts a single dynamic property name as it's only parameter.

The `exists` function is for use with dynamic (unchecked) properties. Dynamic properties are properties whose type is not known at compile type. Dynamic properties return a null value if the dynamic property does not exists on an event, or if the dynamic property exists but the value of the dynamic property is null.

The `exists` function as shown next returns true if the "item" property contains an object that has a "serviceName" property. It returns false if the "item" property is null, or if the "item" property does not contain an object that has a property named "serviceName" :

```
select exists(item.serviceName?) from OrderEvent
```

7.1.6. The `Instance-Of` Function

The `instanceof` function returns a boolean value indicating whether the type of value returned by the expression is one of the given types. The first parameter to the `instanceof` function is an expression to evaluate. The second and subsequent parameters are Java type names.

The function determines the return type of the expression at runtime by evaluating the expression, and compares the type of object returned by the expression to the defined types. If the type of object returned by the expression matches any of the given types, the function returns `true`. If the expression returned `null` or a type that does not match any of the given types, the function returns `false`.

The `instanceof` function is often used in conjunction with dynamic (unchecked) properties. Dynamic properties are properties whose type is not known at compile type.

This example uses the `instanceof` function to select different properties based on the type:

```
select case when instanceof(item, com.mycompany.Service) then serviceName?
  when instanceof(item, com.mycompany.Product) then productName? end
  from OrderEvent
```

The `instanceof` function returns `false` if the expression tested by instanceof returned null.

Valid parameters for the type parameter list are:

- Any of the Java built-in types: `int, long, byte, short, char, double, float, string`, where `string` is a short notation for `java.lang.String`. The type name is not case-sensitive. For example, the next function tests if the dynamic "price" property is either of type float or type double:

```
instanceof(price?, double, float)
```

- The fully-qualified class name of the class to cast to, for example:

```
instanceof(product, org.myproducer.Product)
```

The function considers an event class's superclasses as well as all the directly or indirectly-implemented interfaces by superclasses.

7.1.7. The `Min` and `Max` Functions

The `min` and `max` function take two or more parameters that itself can be expressions. The `min` function returns the lowest numeric value among the 2 or more comma-separated expressions, while the `max` function returns the highest numeric value. The return type is the compatible aggregated type of all return values.

The next example shows the `max` function that has a `Double` return type and returns the value 1.1.

```
select max(1, 1.1, 2 * 0.5) from ...
```

The `min` function returns the lowest value. The statement below uses the function to determine the smaller of two timestamp values.

```
select symbol, min(ticks.timestamp, news.timestamp) as minT
       from StockTickEvent.win:time(30 sec) as ticks, NewsEvent.win:time(30 sec) as news
       where ticks.symbol = news.symbol
```

7.1.8. The `Previous` Function

The `prev` function returns the property value of a previous event. The first parameter denotes the i-th previous event in the order established by the data window. The second parameter is a property name for which the function returns the value for the previous event.

This example selects the value of the `price` property of the 2nd-previous event from the current Trade event.

```
select prev(2, price) from Trade.win:length(10)
```

Since the `prev` function takes the order established by the data window into account, the function works well with sorted windows. In the following example the statement selects the symbol of the 3 Trade events that had the largest, second-largest and third-largest volume.

```
select prev(0, symbol), prev(1, symbol), prev(2, symbol)
  from Trade.ext:sort(volume, true, 10)
```

The i-th previous event parameter can also be an expression returning an Integer-type value. The next statement joins the Trade data window with an `RankSelectionEvent` event that provides a `rank` property used to look up a certain position in the sorted Trade data window:

```
select prev(rank, symbol) from Trade.ext:sort(volume, true, 10), RankSelectionEvent
```

And the expression `count(*) - 1` allows us to select the oldest event in the length window:

```
select prev(count(*) - 1, price) from Trade.win:length(100)
```

The `prev` function returns a `null` value if the data window does not currently hold the i-th previous event. The example below illustrates this using a time batch window. Here the `prev` function returns a null value for any events in which the previous event is not in the same batch of events. Note that the `prior` function as discussed below can be used if a null value is not the desired result.

```
select prev(1, symbol) from Trade.win:time_batch(1 min)
```

Previous Event per Group

The combination of `prev` function and group-by view returns the property value for a previous event in the given group.

Let's look at an example. Assume we want to obtain the price of the previous event of the same symbol as the current event.

The statement that follows solves this problem. It declares a group-by view grouping on the symbol property and a time window of 1 minute. As a result, when the engine encounters a new symbol value that it hasn't seen before, it creates a new time window specifically to hold events for that symbol. Consequently, the previous function returns the previous event within the respective time window for that event's symbol value.

```
select prev(1, price) as prevPrice from Trade.std:groupby(symbol).win:time(1 min)
```

In a second example, assume we need to return, for each event, the current top price per symbol. We can use the `prev` to obtain the highest price from a sorted data window, and use the group-by view to group by symbol:

```
select prev(0, price) as topPricePerSymbol
from Trade.std:groupby(symbol).ext:sort(price, false, 1)
```

Restrictions

The following restrictions apply to the `prev` functions and its results:

- The function always returns a `null` value for remove stream (old data) events
- The function requires a data window view, or a group-by and data window view, without any additional sub-views. Data window views are: length window, time and time batch window and sorted window

Comparison to the `prior` Function

The `prev` function is similar to the `prior` function. The key differences between the two functions are as follows:

- The `prev` function returns previous events in the order provided by the data window, while the `prior` function returns prior events in the order of arrival as posted by a stream's declared views.
- The `prev` function requires a data window view while the `prior` function does not have any view requirements.
- The `prev` function returns the previous event grouped by a criteria by combining the `std:groupby` view and a data window. The `prior` function returns prior events posted by the last view regardless of data window grouping.
- The `prev` function returns a `null` value for remove stream events, i.e. for events leaving a data window. The `prior` function does not have this restriction.

7.1.9. The `Prior` Function

The `prior` function returns the property value of a prior event. The first parameter is an integer value that denotes the i-th prior event in the natural order of arrival. The second parameter is a property name for which the function returns the value for the prior event.

This example selects the value of the `price` property of the 2nd-prior event to the current Trade event.

```
select prior(2, price) from Trade
```

The `prior` function can be used on any event stream or view and does not have any specific view requirements. The function operates on the order of arrival of events by the event stream or view that provides the events.

The next statement uses a time batch window to compute an average volume for 1 minute of Trade events, posting results every minute. The select-clause employs the `prior` function to select the current average and the average before the current average:

```
select average, prior(1, average)
    from TradeAverages.win:time_batch(1 min).stat:uni(volume)
```

7.2. Aggregate Functions

The syntax of the aggregation functions and the results they produce are shown in below table.

Table 7.2. Syntax and results of aggregate functions

Aggregate Function	Result
sum([all|distinct] *expression*)	Totals the (distinct) values in the expression, returning a value of `long`, `double`, `float` or `integer` type depending on the expression
avg([all|distinct] *expression*)	Average of the (distinct) values in the expression, returning a value of `double` type
count([all|distinct] *expression*)	Number of the (distinct) non-null values in the expression, returning a value of `long` type
count(*)	Number of events, returning a value of `long` type
max([all|distinct] *expression*)	Highest (distinct) value in the expression, returning a value of the same type as the expression itself returns
min([all|distinct] *expression*)	Lowest (distinct) value in the expression, returning a value of the same type as the expression itself returns
median([all|distinct] *expression*)	Median (distinct) value in the expression, returning a value of `double` type
stddev([all|distinct] *expression*)	Standard deviation of the (distinct) values in the expression, returning a value of `double` type
avedev([all|distinct] *expression*)	Mean deviation of the (distinct) values in the expression, returning a value of `double` type

7.3. User-Defined Functions

A user-defined function (UDF) can be invoked anywhere as an expression itself or within an expresson. The function must simply be a public static method that the classloader can resolve at statement creation time. The engine resolves the function reference at statement creation time and verifies parameter types.

User-defined functions can be also be invoked on instances of an event: Please see Section 4.4.4, “Using the Stream Name” to invoke event instance methods on a named stream.

The example below assumes a class `MyClass` that exposes a public static method `myFunction` accepting 2 parameters, and returing a numeric type such as `double`.

```
select 3 * com.mycompany.MyClass.myFunction(price, volume) as myValue
from StockTick.win:time(30 sec)
```

User-defined functions also take array parameters as this example shows. The section on Section 6.5, “Array Definition Operator” outlines in more detail the types of arrays produced.

```
select * from RFIDEvent where com.mycompany.rfid.MyChecker.isInZone(zone, {10, 20, 30})
```

Java class names have to be fully qualified (e.g. java.lang.Math) but Esper provides a mechanism for user-controlled imports of classes and packages as outlined in Section 10.4.4, “Class and package imports”.

User-defined functions can return any value including `null`, Java objects or arrays. Therefore user-defined functions can serve to transform, convert or map events, or to extract information and assemble further events.

The following statement is a simple pattern that looks for events of type E1 that are followed by events of type E2. It assigns the tags "e1" and "e2" that the function can use to assemble a final event for output:

```
select MyLib.mapEvents(e1, e2) from pattern [every e1=E1 -> e2=E2]
```

A function that converts from one event type to another event type is shown in the next example. The first statement declares a stream that consists of MyEvent events. The second statement employs a conversion function to convert MyOtherEvent events to events of type MyEvent:

```
insert into MyStream select * from MyEvent
insert into MyStream select MyLib.convert(other) from MyOtherEvent as other
```

In the example above, assuming the event classes MyEvent and MyOtherEvent are Java classes and not aliases, the static method should have the following footprint:

```
public static MyEvent convert(MyOtherEvent otherEvent)
```

Chapter 8. EPL Reference: Views

This chapter outlines the views that are built into Esper. All views can be arbitrarily combined as many of the examples below show. The section on Chapter 3, *Processing Model* provides additional information on the relationship of views, filtering and aggregation.

Esper organizes built-in views in namespaces and names. Views that provide sliding or tumbling data windows are in the `win` namespace. Other most commonly used views are in the `std` namespace. The `ext` namespace are views that order events. The `stat` namespace is used for views that derive statistical data.

Esper distinguishes between data window views and derived-value views. Data windows, or data window views, are views that retain incoming events until an expiry policy indicates to release events.

The next table summarizes data window views:

Table 8.1. Built-in Data Window Views

View	Syntax	Description
Length window	win:length(*size*)	Sliding length window extending the specified number of elements into the past
Length batch window	win:length_batch(*size*)	Tumbling window that batches events and releases them when a given minimum number of events has been collected
Time window	win:time(*time period*)	Sliding time window extending the specified time interval into the past
Externally-timed window	win:ext_timed(*timestamp property*, *time period*)	Sliding time window, based on the millisecond time value supplied by an event property
Time batch window	win:time_batch(*time period*)	Tumbling window that batches events and releases them every specified time interval
Time-Length combination batch window	win:time_length_batch(*time period*, *size*)	Tumbling multi-policy time and length batch window with flow control options
Time-Accumulating window	win:time_accum(*time period*)	Sliding time window accumulates events until no more events arrive within a given time interval
Keep-All window	win:keepall()	The keep-all data window view simply retains all events
Sorted window	ext:sort(*property names*, *descending*, *size*)	Sorts by values of the specified properties and keeps only the top events up to the given size
Time-Order View	ext:time_order(*property name*, *time_period*)	Orders events that arrive out-of-order, using a timestamp-property supplied

View	Syntax	Description
		by each event to be ordered
Unique	std:unique(*property name(s)*)	Retains only the most recent among events having the same value for the specified property or properties. Acts as a length window of size 1 for each distinct value of the property
Group By	std:groupby(*property name(s)*)	Groups events into sub-views by the value of the specified field or fields, generally used to provide a separate data window per group
Last Event	std:lastevent()	Retains the last event, acts as a length window of size 1

The table below summarizes views that derive information from received events and present the derived information as an insert and remove stream:

Table 8.2. Built-in Derived-Value Views

View	Syntax	Description
Size	std:size()	Derives a count of the number of events in a data window, or in an insert stream if used without a data window.
Univariate statistics	stat:uni(*property name*)	Calculates univariate statistics on an event property
Regression	stat:linest(*property name, property name*)	Calculates regression on two event properties
Correlation	stat:correl(*property name, property name*)	Calculates the correlation value on two event properties
Weighted average	stat:weighted_avg(*property name, property name*)	Calculates weighted average given a weight field and a field to compute the average for
Multi-dimensional statistics	stat:cube(*values to derive, property datapoint, property column, row, page*)	Groups and calculates statistics by one or more event properties

A Note on View Parameters

View parameters follow the view namespace name and view name in parenthesis. Where the view parameters include one or more event property names, property names can be placed in single or double quotes, or appear unquoted:

```
select * from StockTickEvent.stat:groupby(feed).win:length(10)
... equivalent to ...
select * from StockTickEvent.stat:groupby('feed').win:length(10)
```

Expressions are not allowed as view parameters.

8.1. Window views

8.1.1. Length window (`win:length`)

This view is a moving (sliding) length window extending the specified number of elements into the past. The view takes a single numeric parameter that defines the window size:

```
win:length(size)
```

The below example calculates univariate statistics on price for the last 5 stock ticks for symbol IBM.

```
select * from StockTickEvent(symbol='IBM').win:length(5).stat:uni(price)
```

The next example keeps a length window of 10 events of stock trade events, with a separate window for each symbol. The statistics on price is calculated only for the last 10 events for each symbol.

```
select * from StockTickEvent.std:groupby(symbol).win:length(10).stat:uni(price)
```

8.1.2. Length batch window (`win:length_batch`)

This window view buffers events (tumbling window) and releases them when a given minimum number of events has been collected. The view takes the number of events to batch as a parameter:

```
win:length_batch(size)
```

The next statement buffers events until a minimum of 5 events have collected. Listeners to updates posted by this view receive updated information only when 5 or more events have collected.

```
select * from StockTickEvent.win:length_batch(5)
```

8.1.3. Time window (`win:time`)

This view is a moving (sliding) time window extending the specified time interval into the past based on the system time. This view takes a time period (see Section 4.2.1, “Specifying Time Periods”) or a number of seconds as a parameter:

```
win:time(time period)
```

```
win:time(number of seconds)
```

For the IBM stock tick events in the last 1 second, calculate statistics on price.

```
select * from StockTickEvent(symbol='IBM').win:time(1 sec).stat:uni(price)
```

The same statement rewritten to use a parameter supplying number-of-seconds is:

```
select * from StockTickEvent(symbol='IBM').win:time(1).stat:uni(price)
```

The following time windows are equivalent specifications:

```
win:time(2 minutes 5 seconds)
win:time(125 sec)
win:time(125)
```

8.1.4. Externally-timed window (`win:ext_timed`)

Similar to the time window, this view is a moving (sliding) time window extending the specified time interval into the past, but based on the millisecond time value supplied by an event property. The view takes two parameters: the name of the event property to return the long-typed timestamp value, and a time period or a number of seconds:

```
win:ext_timed(timestamp_property_name, time_period)
```

```
win:ext_timed(timestamp_property_name, number_of_seconds)
```

The key difference comparing the externally-timed window to the regular time window is that the window slides not based on the engine time, but strictly based on the timestamp property value of the events entering the window. The algorithm underlying the view compares the timestamp value of the oldest event with the timestamp value of the newest arriving event on event arrival. If the time interval between the oldest and newest event is larger then the timer period parameter, then the algorithm removes all oldest events tail-first until the difference between the oldest and newest event is within the time interval. The window therefore slides only when events arrive and only considers each event's timestamp property and not engine time.

This view holds stock tick events of the last 10 seconds based on the timestamp property in `StockTickEvent`.

```
select * from StockTickEvent.win:ext_timed(timestamp, 10 seconds)
```

The externally-timed data window expects to receive event in timestamp order. The view is not useful for ordering events in time order, please us the time-order view instead.

8.1.5. Time batch window (`win:time_batch`)

This window view buffers events (tumbling window) and releases them every specified time interval in one update. The view takes a time period or a number of seconds as a parameter.

```
win:time_batch(time_period)
```

```
win:time_batch(number_of_seconds)
```

The below example batches events into a 5 second window releasing new batches every 5 seconds. Listeners to updates posted by this view receive updated information only every 5 seconds.

```
select * from StockTickEvent.win:time_batch(5 sec)
```

8.1.6. Time-Length combination batch window (`win:time_length_batch`)

This data window view is a combination of time and length batch (tumbling) windows. Similar to the time and length batch windows, this view batches events and releases the batched events when either one of the following conditions occurs, whichever occurs first: the data window has collected a given number of events, or a given time interval has passed.

The view parameters take 2 forms. The first form accepts a time period or number of seconds, and a parameter for the number of events:

```
win:time_length_batch(time_period or number_of_seconds, number of events)
```

The next example shows a time-length combination batch window that batches up to 100 events or all events arriving within a 1-second time interval, whichever condition occurs first:

```
select * from MyEvent.win:time_length_batch(1 sec, 100)
```

In this example, if 100 events arrive into the window before a 1-second time interval passes, the view posts the batch of 100 events. If less then 100 events arrive within a 1-second interval, the view posts all events that arrived within the 1-second interval at the end of the interval.

By default, if there are no events arriving in the current interval (insert stream), and no events remain from the prior batch (remove stream), then the view does not post results to listeners. This view allows overriding this default behavior via flow control keywords.

The synopsis of the view with flow control parameters is:

```
win:time_length_batch(time_period or number_of_seconds, number of events,
    "flow control keyword [, keyword...]")
```

The `FORCE_UPDATE` flow control keyword instructs the view to post an empty result set to listeners if there is no data to post for an interval. The view begins posting no later then after one time interval passed after the first event arrives.

The `START_EAGER` flow control keyword instructs the view to post empty result sets even before the first event arrives, starting a time interval at statement creation time. As when using `FORCE_UPDATE`, the view also posts an empty result set to listeners if there is no data to post for an interval, however it starts doing so at time of statement creation rather then at the time of arrival of the first event.

Taking the two flow control keywords in one sample statement, this example presents a view that waits for 10 seconds or reacts when the 5th event arrives, whichever comes first. It posts empty result sets after one interval after the statement is created, and keeps posting an empty result set as no events arrive during intervals:

```
select * from MyEvent.win:time_length_batch(10 sec, 5, "FORCE_UPDATE, START_EAGER")
```

8.1.7. Time-Accumulating window (`win:time_accum`)

This data window view is a specialized moving (sliding) time window that differs from the regular time window in that it accumulates events until no more events arrive within a given time interval, and only then releases the accumulated events as a remove stream.

The view accepts a single parameter: the time period or number of seconds specifying the length of the time interval during which no events must arrive until the view releases accumulated events. The synopsis is as follows:

```
win:time_accum(time_period or number_of_seconds)
```

The next example shows a time-accumulating window that accumulates events, and then releases events if within the time interval no more events arrive:

```
select * from MyEvent.win:time_accum(10 sec)
```

This example accumulates events, until when for a period of 10 seconds no more MyEvent events arrive, at which time it posts all accumulated MyEvent events.

Your application may only be interested in the batches of events as events leave the data window. This can be done simply by selecting the remove stream of this data window, populated by the engine as accumulated events leave the data window all-at-once when no events arrive during the time interval following the time the last event arrived:

```
select rstream * from MyEvent.win:time_accum(10 sec)
```

If there are no events arriving, then the view does not post results to listeners.

8.1.8. Keep-All window (`win:keepall`)

This keep-all data window view simply retains all events. The view does not remove events from the data window, unless used with a named window and the `on delete` clause.

The view accepts no parameters. The synopsis is as follows:

```
win:keepall()
```

The next example shows a keep-all window that accumulates all events received into the window:

```
select * from MyEvent.win:keepall()
```

Note that since the view does not release events, care must be taken to prevent retained events from using all available memory.

8.2. Standard view set

8.2.1. Unique (`std:unique`)

The `unique` view is a view that includes only the most recent among events having the same value for the specified field or list of fields.

The synopsis is:

```
std:unique(property_name [, property_name ...])
```

The view acts as a length window of size 1 for each distinct value of the event property, or combination of event property values. It thus posts as old events the prior event of the same property value(s), if any.

The below example creates a view that retains only the last event per symbol.

```
select * from StockTickEvent.std:unique(symbol)
```

The next example creates a view that retains the last event per symbol and feed.

```
select * from StockTickEvent.std:unique(symbol, feed)
```

8.2.2. Group-By (`std:groupby`)

This view groups events into sub-views by the value of the specified field or combination of fields. The view takes a single property name to supply the group-by values, or a list of property names as the synopsis shows:

```
std:groupby(property_name [, property_name ...])
```

This example calculates statistics on price separately for each symbol.

```
select * from StockTickEvent.std:groupby(symbol).stat:uni(price)
```

The group-by view can also take multiple fields to group by. This example calculates statistics on price for each symbol and feed:

```
select * from StockTickEvent.std:groupby(symbol, feed).stat:uni(price)
```

The order in which the group-by view appears within sub-views of a stream controls the data the engine derives from events for each group. The next 2 statements demonstrate this using a length window.

This example keeps a length window of 10 events of stock trade events, with a separate length window for each symbol. The engine calculates statistics on price for the last 10 events for each symbol. During runtime, the engine actually allocates a separate length window for each new symbol arriving.

```
select * from StockTickEvent.std:groupby(symbol).win:length(10).stat:uni(price)
```

By putting the group-by view in position after the length window, we can change the semantics of the query. The query now returns the statistics on price per symbol for only the last 10 events across all symbols. Here the engine allocates only one length window for all events.

```
select * from StockTickEvent.win:length(10).std:groupby(symbol).stat:uni(price)
```

We have learned that by placing the group-by view before other views, these other views become part of the grouped set of views. The engine dynamically allocates a new view instance for each subview, every time it encounters a new group key such as a new value for symbol. Therefore, in `std:groupby(symbol).win:length(10)` the engine allocates a new length window for each distinct symbol. However in `win:length(10).std:groupby(symbol)` the engine maintains a single length window.

Multiple group-by views can also be used in the same statement. The statement below groups by symbol and feed. As the statement declares the time window after the group-by view for symbols, the engine allocates a new time window per symbol however reports statistics on price per symbol and feed. The query results are statistics on price per symbol and feed for the last 1 minute of events per symbol (and not per feed).

```
select * from StockTickEvent.std:groupby(symbol).win:time(1 minute)
    .std:groupby(feed).stat:uni(price)
```

Last, we consider the permutation where the time window is declared after the group-by. Here, the query results are statistics on price per symbol and feed for the last 1 minute of events per symbol and feed.

```
select * from StockTickEvent.std:groupby(symbol, feed)
    .win:time(1 minute).stat:uni(price)
```

8.2.3. Size (`std:size`)

This view simply posts the number of events received from a stream or view. The synopsis is simply:

```
std:size()
```

The view posts a single long-typed property named `size`. The view posts the prior size as old data, and the current size as new data to update listeners of the view. Via the `iterator` method of the statement the size value can also be polled (read).

When combined with a data window view, the size view reports the current and prior number of events in the data window. This example reports the number of tick events within the last 1 minute:

```
select size from StockTickEvent.win:time(1 min).std:size()
```

The size view is also useful in conjunction with a group-by view to count the number of events per group. The EPL below returns the number of events per symbol.

```
select size from StockTickEvent.std:groupby(symbol).std:size()
```

When used without a data window, the view simply counts the number of events:

```
select size from StockTickEvent.std:size()
```

All views can be used with pattern statements as well. The next EPL snippet shows a pattern where we look for tick events followed by trade events for the same symbol. The size view counts the number of occurances of the pattern.

```
select size from pattern[every s=StockTickEvent -> TradeEvent(symbol=s.symbol)].std:size()
```

8.2.4. Last Event (`std:lastevent`)

This view exposes the last element of its parent view:

```
std:lastevent()
```

The view acts as a length window of size 1. It thus posts as old events the prior event in the stream, if any.

This example statement retains statistics calculated on stock tick price for the symbol IBM.

```
select * from StockTickEvent(symbol='IBM').stat:uni(price).std:lastevent()
```

8.3. Statistics views

8.3.1. Univariate statistics (`stat:uni`)

This view calculates univariate statistics on an event property. The view takes a single event property name as a parameter. The event property must be of numeric type:

```
stat:uni(event_property_name)
```

Table 8.3. Univariate statistics derived properties

Property Name	Description
`datapoints`	Number of values, equivalent to `count(*)` for the stream
`total`	Sum of values
`average`	Average of values
`variance`	Variance
`stdev`	Sample standard deviation (square root of variance)
`stdevpa`	Population standard deviation

The below example selects the standard deviation on price for stock tick events for the last 10 events.

```
select stdev from StockTickEvent.win:length(10).stat:uni(price)
```

8.3.2. Regression (`stat:linest`)

This view calculates regression on two event properties. The view takes two event property names as parameters. The event properties must be of numeric type:

```
stat:linest(event_property_name_1, event_property_name_2)
```

Table 8.4. Regression derived properties

Property Name	Description
`slope`	Slope
`YIntercept`	Y Intercept

Calculate slope and y-intercept on price and offer for all events in the last 10 seconds.

```
select slope, YIntercept from StockTickEvent.win:time(10 seconds).stat:linest(price, offer)
```

8.3.3. Correlation (`stat:correl`)

This view calculates the correlation value on two event properties. The view takes two event property names as parameters. The event properties must be of numeric type:

```
stat:correl(event_property_name_1, event_property_name_2)
```

Table 8.5. Correlation derived properties

Property Name	Description
`correlation`	Correlation between two event properties

Calculate correlation on price and offer over all stock tick events for IBM.

```
select correlation from StockTickEvent(symbol='IBM').stat:correl(price, offer)
```

8.3.4. Weighted average (`stat:weighted_avg`)

This view returns the weighted average given a weight field and a field to compute the average for. The view takes two event property names as parameters. The event properties must be of numeric type:

```
stat:weighted_avg(event_property_name_field, event_property_name_weight)
```

Table 8.6. Weighted average derived properties

Property Name	Description
`average`	Weighted average

A statement that derives the volume-weighted average price for the last 3 seconds:

```
select average
from StockTickEvent(symbol='IBM').win:time(3 seconds).stat:weighted_avg(price, volume)
```

8.3.5. Multi-dimensional statistics (`stat:cube`)

This view works similar to the `std:groupby` views in that it groups information by one or more event properties. The view accepts 3 or more parameters: The first parameter to the view defines the univariate statistics values to derive. The second parameter is the property name to derive data from. The remaining parameters supply the event property names to use to derive dimensions.

```
stat:cube(values_to_derive, property_name_datapoint, property_name_column)
```

```
stat:cube(values_to_derive, property_name_datapoint, property_name_column,
    property_name_row)
```

```
stat:cube(values_to_derive, property_name_datapoint, property_name_column,
    property_name_row, property_name_page)
```

Table 8.7. Multi-dim derived properties

Property Name	Description
`cube`	The cube following the `com.espertech.esper.view.stat.olap.Cube` interface

The example below derives the count, average and standard deviation latency of service measurement events per customer.

```
select cube from ServiceMeasurement.stat:cube({‘datapoints’, ‘average’, ‘stdev’},
    latency, customer)
```

This example derives the average latency of service measurement events per customer, service and error status for events in the last 30 seconds.

```
select * from ServiceMeasurement.win:length(30000).stat:cube({‘average’},
        latency, customer, service, status)
```

8.4. Extension View Set

8.4.1. Sorted Window View (ext:sort)

This view sorts by values of the specified event properties and keeps only the top events up to the given size.

The syntax to sort on a single event property is as follows.

```
ext:sort(property_name, is_descending, size)
```

To sort on a multiple event properties the syntax is as follows.

```
sort( { property_name, is_descending [ , property_name, is_descending ...] }, size)
```

The view below sorts on price descending keeping the lowest 10 prices and reporting statistics on price.

```
select * from StockTickEvent.ext:sort(price, false, 10).stat:uni(price)
```

The following example sorts events first by price in descending order, and then by symbol name in ascending (alphabetical) order, keeping only the 10 events with the highest price (with ties resolved by alphabetical order of symbol).

```
select * from StockTickEvent.ext:sort({'price', true, 'symbol', false}, 10)
```

8.4.2. Time-Order View (ext:time_order)

This view orders events that arrive out-of-order, using a timestamp-property supplied by each event to be ordered, and by comparing the event timestamp to engine system time.

The syntax for this view is as follows.

```
ext:time_order(timestamp_property_name, time_period or number_of_seconds)
```

The first parameter to the view is the name of the property on the event that supplies the timestamp value. The timestamp is expected to be a long-typed millisecond value that denotes an event's time of consideration by the view. This is typically the time of arrival. The second parameter is a number of seconds or the time period specifying the time interval that an arriving event should maximally be held, in order to consider older events arriving at a later time.

Since the view compares an event's timestamp property to engine time, the view requires that the event timestamp values and current engine time are both following the same clock. Therefore, to the extend that the clocks that originated both timestamps differ, the view may produce inaccurate results.

As an example, the next statement uses the `arrival_time` property of `MyTimestampedEvent` events to order and release events by arrival time:

```
insert into ArrivalTimeOrderedStream
select rstream * from MyTimestampedEvent.ext:time_order(arrival_time, 10 sec)
```

In the example above, the `arrival_time` property holds a long-typed timestamp value in milliseconds. On arrival of an event, the engine compares the timestamp value of each event to the tail-time of the window. The tail-time of the window is, in this example, 10 seconds before engine time (continuously sliding). If the timestamp value indicates that the event is older then the tail-time of the time window, the event is released immediately in the remove stream. If the timestamp value indicates that the event is newer then the tail-time of the window, the view retains the event until engine time moves such that the event timestamp is older then tail-time.

The examples thus holds each arriving event in memory anywhere from zero seconds to 10 seconds, to allow for older events (considering arrival time timestamp) to arrive. In other words, the view holds an event with an arrival time equal to engine time for 10 seconds. The view holds an event with an arrival time that is 2 seconds older then engine time for 8 seconds. The view holds an event with an arrival time that is 10 or more seconds older then engine time for zero seconds, and releases such (old) events immediately into the remove stream.

The insert stream of this sliding window consists of all arriving events. The remove stream of the view is ordered by timestamp value: The event that has the oldest timestamp value is released first, followed by the next newer events. Note the statement above uses the `rstream` keyword to select ordered events only and uses the `insert into` clause to makes such ordered stream available for subsequent statements to use.

It is up to your application to populate the timestamp property into your events for consideration by the view. The view also works well if you use externally-provided time via timer events.

Chapter 9. API Reference

9.1. API Overview

Esper has 2 primary interfaces that this section outlines: The administrative interface and the runtime interface.

Use Esper's administrative interface to create and manage EPL and pattern statements, and set runtime configurations, as discussed in Section 4.1, “EPL Introduction” and Section 5.1, “Event Pattern Overview”.

Use Esper's runtime interface to send events into the engine, emit events and get statistics for an engine instance.

The JavaDoc documentation is also a great source for API information.

9.2. Engine Instances

Each instance of an Esper engine is completely independent of other engine instances and has its own administrative and runtime interface.

An instance of the Esper engine is obtained via static methods on the `EPServiceProviderManager` class. The `getDefaultProvider` method and the `getProvider(String URI)` methods return an instance of the Esper engine. The latter can be used to obtain multiple instances of the engine for different URI values. The `EPServiceProviderManager` determines if the URI matches all prior URI values and returns the same engine instance for the same URI value. If the URI has not been seen before, it creates a new engine instance.

The code snipped below gets the default instance Esper engine. Subsequent calls to get the default engine instance return the same instance.

```
EPServiceProvider epService = EPServiceProviderManager.getDefaultProvider();
```

This code snippet gets an Esper engine for URI `RFIDProcessor1`. Subsequent calls to get an engine with the same URI return the same instance.

```
EPServiceProvider epService = EPServiceProviderManager.getProvider("RFIDProcessor1");
```

An existing Esper engine instance can be reset via the `initialize` method on the `EPServiceProvider` instance. This stops and removes all statements in the Engine.

9.3. The Administrative Interface

9.3.1. Creating Statements

Create event pattern expression and EPL statements via the administrative interface `EPAdministrator`.

This code snippet gets an Esper engine then creates an event pattern and an EPL statement.

```
EPServiceProvider epService = EPServiceProviderManager.getDefaultProvider();
EPAdministrator admin = epService.getEPAdministrator();

EPStatement 10secRecurTrigger = admin.createPattern(
```

```
  "every timer:at(*, *, *, *, *, */10)");

EPStatement countStmt = admin.createEPL(
  "select count(*) from MarketDataBean.win:time(60 sec)");
```

Note that event pattern expressions can also occur within EPL statements. This is outlined in more detail in Section 4.4.2, “Pattern-based Event Streams”.

The `create` methods on `EPAdministrator` are overloaded and allow an optional statement name to be passed to the engine. A statement name can be useful for retrieving a statement by name from the engine at a later time. The engine assigns a statement name if no statement name is supplied on statement creation.

The `createPattern` and `createEPL` methods return `EPStatement` instances. Statements are automatically started and active when created. A statement can also be stopped and started again via the `stop` and `start` methods shown in the code snippet below.

```
countStmt.stop();
countStmt.start();
```

9.3.2. Receiving Statement Results

Esper provides three choices for your application to receive statement results. Your application can use all three mechanisms alone or in any combination for each statement. The choices are:

Table 9.1. Choices For Receiving Statement Results

Name	Methods on `EPStatement`	Description
Listener Callbacks	`addListener` and `removeListener`	Your application provides implementations of the `UpdateListener` or the `StatementAwareUpdateListener` interface to the statement. Listeners receive `EventBean` instances containing statement results. The engine continuously indicates results to all listeners as soon they occur, and following output rate limiting clauses if specified. Since Esper 2.0 it is advised to use the Subscriber Object when possible.
Subscriber Object	`setSubscriber`	Your application provides a POJO (plain Java object) that exposes methods to receive statement results. The engine continuously indicates results to the single subscriber as soon they occur, and following output rate limiting clauses if specified. This is the fastest method to receive statement results, as the engine delivers strongly-typed results directly to your application objects without the need for building an `EventBean` result set as in the Listener Callback choice. There can be at most 1 Subscriber Object registered per

Name	Methods on `EPStatement`	Description
		statement. If you require more than one listener, use the Listener Callback instead (or in addition). The Subscriber Object is bound to the statement with a strongly typed support which ensure direct delivery of new events without type conversion. This optimization is made possible because there can only be 0 or 1 Subscriber Object per statement.
Pull API	`safeIterator` and `iterator`	Your application asks the statement for results and receives a set of events via `java.util.Iterator<EventBean>`. This is useful if your application does not need continuous indication of new results in real-time.

Your application may attach one or more listeners, zero or one single subscriber and in addition use the Pull API on the same statement. There are no limitations to the use of iterator, subscriber or listener alone or in combination to receive statement results.

The best delivery performance can generally be achieved by attaching a subscriber and by not attaching listeners. The engine is aware of the listeners and subscriber attached to a statement. The engine uses this information internally to reduce statement overhead. For example, if your statement does not have listeners or a subscriber attached, the engine does not need to continuously generate results for delivery.

9.3.3. Setting a Subscriber Object

A subscriber object is a direct binding of query results to a Java object. The object, a POJO, receives statement results via method invocation. The subscriber class need not implement an interface or extend a superclass.

Subscriber objects have several advantages over listeners. First, they offer a substantial performance benefit: Query results are delivered directly to your method(s) through Java virtual machine method calls, and there is no intermediate representation (`EventBean`). Second, as subscribers receive strongly-typed parameters, the subscriber code tends to be simpler.

This chapter describes the requirements towards the methods provided by your subscriber class.

The engine can deliver results to your subscriber in two ways:

1. Each evert in the insert stream results in a method invocation, and each event in the remove stream results in further method invocations. This is termed *row-by-row delivery*.
2. A single method invocation that delivers all rows of the insert and remove stream. This is termed *multi-row* delivery.

Row-By-Row Delivery

Your subscriber class must provide a method by name `update` to receive insert stream events row-by-row. The number and types of parameters declared by the `update` method must match the number and types of columns as specified in the `select` clause, in the same order as in the `select` clause.

For example, if your statement is:

```
select orderId, price, count(*) from OrderEvent
```

Then your subscriber `update` method looks as follows:

```
public class MySubscriber {
  ...
  public void update(String orderId, double price, long count) {...}
  ...
}
```

Each method parameter declared by the `update` method must be assignable from the respective column type as listed in the `select`-clause, in the order selected. The assignability rules are:

- Widening of types follows Java standards. For example, if your `select` clause selects an integer value, the method parameter for the same column can be typed int, long, float or double (or any equivalent boxed type).
- Auto-boxing and unboxing follows Java standards. For example, if your `select` clause selects an `java.lang.Integer` value, the method parameter for the same column can be typed `int`. Note that if your `select` clause column may generate `null` values, an exception may occur at runtime unboxing the `null` value.
- Interfaces and super-classes are honored in the test for assignability. Therefore `java.lang.Object` can be used to accept any `select` clause column type

Wildcards

If your `select` clause contains one or more wildcards (*), then the equivalent parameter type is the underlying event type of the stream selected from.

For example, your statement may be:

```
select *, count(*) from OrderEvent
```

Then your subscriber `update` method looks as follows:

```
public void update(OrderEvent orderEvent, long count) {...}
```

In a join, the wildcard expands to the underlying event type of each stream in the join in the order the streams occur in the `from` clause. An example statement for a join is:

```
select *, count(*) from OrderEvent order, OrderHistory hist
```

Then your subscriber `update` method should be:

```
public void update(OrderEvent orderEvent, OrderHistory orderHistory, long count) {...}
```

The stream wildcard syntax and the stream name itself can also be used:

```
select hist.*, order from OrderEvent order, OrderHistory hist
```

The matching `update` method is:

```
public void update(OrderHistory orderHistory, OrderEvent orderEvent) {...}
```

Row Delivery as Map and Object Array

Alternatively, your `update` method may simply choose to accept `java.util.Map` as a representation for each row. Each column in the `select` clause is then made an entry in the resulting `Map`. The `Map` keys are the column alias name if supplied, or the expression string itself for columns without an alias.

The `update` method for `Map` delivery is:

```
public void update(Map row) {...}
```

The engine also supports delivery of `select` clause columns as an object array. Each item in the object array represents a column in the `select` clause. The `update` method then looks as follows:

```
public void update(Object[] row) {...}
```

Delivery of Remove Stream Events

Your subscriber receives remove stream events if it provides a method named `updateRStream`. The method must accept the same number and types of parameters as the `update` method.

An example statement:

```
select orderId, count(*) from OrderEvent.win:time(20 sec) group by orderId
```

Then your subscriber `update` and `updateRStream` methods should be:

```
public void update(String, long count) {...}
public void updateRStream(String orderId, long count) {...}
```

Delivery of Begin and End Indications

If your subscriber requires a notification for begin and end of event delivery, it can expose methods by name `start` and `end`.

The `start` method must take two integer parameters that indicate the number of events of the insert stream and remove stream to be delivered. The engine invokes the `start` method immediately prior to delivering events to the `update` and `updateRStream` methods.

The `end` method must take no parameters. The engine invokes the `end` method immediately after delivering events to the `update` and `updateRStream` methods.

An example set of delivery methods:

```
// Called by the engine before delivering events to update methods
public void start(int insertStreamLength, int removeStreamLength)

// To deliver insert stream events
public void update(String orderId, long count) {...}

// To deliver remove stream events
public void updateRStream(String orderId, long count) {...}

// Called by the engine after delivering events
public void end() {...}
```

Multi-Row Delivery

In place of row-by-row delivery, your subscriber can receive all events in the insert and remove stream via a single method invocation.

The event delivery follow the scheme as described earlier in Section 9.3.3.1.2, “Row Delivery as Map and Object Array ”. The subscriber class must provide one of the following methods:

Table 9.2. Update Method for Multi-Row Delivery of Underlying Events

Method	Description
`update(Object[][] insertStream, Object[][] removeStream)`	The first dimension of each Object array is the event row, and the second dimension is the column matching the column order of the statement `select` clause
`update(Map[] insertStream, Map[] removeStream)`	Each map represents one event, and Map entries represent columns of the statement `select` clause

Wildcards

If your `select` clause contains a single wildcard (*) or wildcard stream selector, the subscriber object may also directly receive arrays of the underlying events. In this case, the subscriber class should provide a method `update(`*Underlying*`[] insertStream,` *Underlying*[] *removeStream*`)` , such that *Underlying* represents the class of the underlying event.

For example, your statement may be:

```
select * from OrderEvent.win:time(30 sec)
```

Your subscriber class exposes the method:

```
public void update(OrderEvent[] insertStream, OrderEvent[] removeStream) {...}
```

9.3.4. Adding Listeners

Your application can subscribe to updates posted by a statement via the `addListener` and `removeListener` methods on `EPStatement` . Your application must to provide an implementation of the `UpdateListener` or the `StatementAwareUpdateListener` interface to the statement:

```
UpdateListener myListener = new MyUpdateListener();
countStmt.addListener(myListener);
```

EPL statements and event patterns publish old data and new data to registered `UpdateListener` listeners. New data published by statements is the events representing the new values of derived data held by the statement. Old data published by statements constists of the events representing the prior values of derived data held by the statement.

It is important to understand that `UpdateListener` listeners receive multiple result rows in one invocation by the engine: the new data and old data parameters to your listener are array parameters. For example, if your application uses one of the batch data windows, or your application creates a pattern that matches multiple times when a single event arrives, then the engine indicates such multiple result rows in one invocation and your new data array carries two or more rows.

A second listener interface is the `StatementAwareUpdateListener` interface. A `StatementAwareUpdateListener` is especially useful for registering the same listener object with multiple statements, as the listener receives the statement instance and engine instance in addition to new and old data when the engine indicates new results to a listener.

```
StatementAwareUpdateListener myListener = new MyStmtAwareUpdateListener();
statement.addListener(myListener);
```

To indicate results the engine invokes this method on `StatementAwareUpdateListener` listeners: `update(EventBean[] newEvents, EventBean[] oldEvents, EPStatement statement, EPServiceProvider epServiceProvider)`

9.3.5. Using Iterators

Subscribing to events posted by a statement is following a push model. The engine pushes data to listeners when events are received that cause data to change or patterns to match. Alternatively, you need to know that statements serve up data that your application can obtain via the `safeIterator` and `iterator` methods on `EPStatement`. This is called the pull API and can come in handy if your application is not interested in all new updates, and only needs to perform a frequent or infrequent poll for the latest data.

The `safeIterator` method on `EPStatement` returns a concurrency-safe iterator returning current statement results, even while concurrent threads may send events into the engine for processing. The safe iterator guarantees correct results even as events are being processed by other threads. The cost is that the iterator obtains and holds a statement lock that must be released via the `close` method on the `SafeIterator` instance.

The `iterator` method on `EPStatement` returns a concurrency-unsafe iterator. This iterator is only useful for applications that are single-threaded, or applications that themselves perform coordination between the iterating thread and the threads that send events into the engine for processing. The advantage to this iterator is that it does not hold a lock.

The next code snippet shows a short example of use of safe iterators:

```
EPStatement statement = epAdmin.createEPL("select avg(price) as avgPrice from MyTick");
// .. send events into the engine
// then use the pull API...
SafeIterator<EventBean> safeIter = statement.safeIterator();
try {
  for (;safeIter.hasNext();) {
     // .. process event ..
     EventBean event = safeIter.next();
     System.out.println("avg:" + event.get("avgPrice");
  }
}
finally {
  safeIter.close(); // Note: safe iterators must be closed
}
```

This is a short example of use of the regular iterator that is not safe for concurrent event processing:

```
double averagePrice = (Double) eplStatement.iterator().next().get("average");
```

The `safeIterator` and `iterator` methods can be used to pull results out of all statements, including statements that join streams, contain aggregation functions, pattern statements, and statements that contain a `where` clause, `group by` clause, `having` clause or `order by` clause.

For statements without an `order by` clause, the `iterator` method returns events in the order maintained by the data window. For statements that contain an `order by` clause, the `iterator` method returns events in the order indicated by the `order by` clause.

Esper places the following restrictions on the pull API and usage of the `safeIterator` and `iterator` methods:

1. In multithreaded applications, use the `safeIterator` method. Note: make sure your application closes the iterator via the `close` method when done, otherwise the iterated statement stays locked and event processing for that statement does not resume.
2. In multithreaded applications, the `iterator` method does not hold any locks. The iterator returned by this method does not make any guarantees towards correctness of results and fail-behavior, if your application processes events into the engine instance by multiple threads. Use the `safeIterator` method for concurrency-safe iteration instead.
3. Since the `safeIterator` and `iterator` methods return events to the application immediately, the iterator does not honor an output rate limiting clause, if present. That is, the iterator returns results as if there is no output-rate clause for the statement. Use a separate statement and the `insert into` clause to control the output rate for iteration.

9.3.6. Managing Statements

The `EPAdministrator` interface provides the facilities for managing statements:

- Use the `getStatement` method to obtain an existing started or stopped statement by name
- Use the `getStatementNames` methods to obtain a list of started and stopped statement names
- Use the `startAllStatements`, `stopAllStatements` and `destroyAllStatements` methods to manage all statements in one operation

9.3.7. Runtime Engine Configuration

Certain configuration changes are available to perform on an engine instance while in operation. Such configuration operations are available via the `getConfiguration` method on `EPAdministrator`, which returns an `ConfigurationOperations` object.

The configuration operations available on a running engine instance are as follows. Please see Chapter 10, *Configuration* for more information.

- Add an new event type for a JavaBean class, legacy Java class or custom Java class
- Add an new DOM XML event type
- Add an new Map-based event type

9.4. The Runtime Interface

The `EPRuntime` interface is used to send events for processing into an Esper engine, and to emit Events from an engine instance to the outside world.

The below code snippet shows how to send a Java object event to the engine. Note that the `sendEvent` method is overloaded. As events can take on different representation classes in Java, the `sendEvent` takes parameters to reflect the different types of events that can be send into the engine. The Chapter 2, *Event Representations* section explains the types of events accepted.

```
EPServiceProvider epService = EPServiceProviderManager.getDefaultProvider();
EPRuntime runtime = epService.getEPRuntime();

// Send an example event containing stock market data
runtime.sendEvent(new MarketDataBean('IBM', 75.0));
```

```

```

Events, in theoretical terms, are observations of a state change that occured in the past. Since one cannot change an event that happened in the past, events are best modelled as immutable objects.

Note that the Esper engine relies on events that are sent into an engine to not change their state. Typically, applications create a new event object for every new event, to represent that new event. Application should not modify an existing event that was sent into the engine.

Another important method in the runtime interface is the `route` method. This method is designed for use by `UpdateListener` implementations that need to send events into an engine instance.

9.4.1. Receiving Unmatched Events

Your application can register an implementation of the `UnmatchedListener` interface with the `EPRuntime` runtime via the `setUnmatchedListener` method to receive events that were not matched by any statement.

Events that can be unmatched are all events that your application sends into the runtime via one of the `sendEvent` or `route` methods, or that have been generated via an `insert into` clause.

For an event to become unmatched by any statement, the event must not match any statement's event stream filter criteria. Note that the EPL `where` clause or `having` clause are not considered part of the filter criteria for a stream, as explained by example below.

In the next statement all MyEvent events match the statement's event stream filter criteria, regardless of the value of the 'quantity' property. As long as the below statement remains started, the engine would not deliver MyEvent events to your registered `UnmatchedListener` instance:

```
select * from MyEvent where quantity > 5
```

In the following statement a MyEvent event with a 'quantity' property value of 5 or less does not match this statement's event stream filter criteria. The engine delivers such an event to the registered `UnmatchedListener` instance provided no other statement matches on the event:

```
select * from MyEvent(quantity > 5)
```

For patterns, if no pattern sub-expression is active for an event type, an event of that type also counts as unmatched in regards to the pattern statement.

9.4.2. Emit Facility for Publish-Subscribe

The `emit` and `addEmittedListener` methods can be used to emit events from a runtime to a registered set of one or more emitted event listeners. This mechanism is available as a service to enable channel-based publish-subscribe of events emitted from an engine instance via the `emit` method. Emitting events is not integrated with EPL and is available only via the `EPRuntime` interface.

Events are emitted on an event channel identified by a name. Listeners are implementations of the `EmittedListener` interface. Via the `addEmittedListener` method a listener can be added to the specified event channel. The lister receives only those events posted to that channel. The channel parameter to `addEmittedListener` also allows null values. If a null channel value is specified, the listeners receives emitted events posted on any channel.

9.5. Events Received from the Engine

The Esper engine posts events to registered `UpdateListener` instances ('push' method for receiving events). For all statements events can also be pulled from statements via the `safeIterator` and `iterator` methods. Both pull and push supply `EventBean` instances representing the events generated by the engine or events supplied to the engine. Each `EventBean` instance represents an event, with each event being either an artificial event, composite event or an event supplied to the engine via its runtime interface.

The `getEventType` method supplies an event's event type information represented by an `EventType` instance. The `EventType` supplies event property names and types as well as information about the underlying object to the event.

The engine may generate artificial events that contain information derived from event streams. A typical example for artificial events is the events posted for a statement to calculate univariate statistics on an event property. The below example shows such a statement and queries the generated events for an average value.

```
// Derive univariate statistics on price for the last 100 market data events
String stmt = "select * from MarketDataBean(symbol='IBM').win:length(100).stat:uni(price)";
EPStatement priceStatsView = epService.getEPAdministrator().createEPL(stmt);
priceStatsView.addListener(testListener);
```

```
// Example listener code
public class MyUpdateListener implements UpdateListener
{
    public void update(EventBean[] newData, EventBean[] oldData)
    {
        // Interrogate events
        System.out.println("new average price=" + newData[0].get("average");
        }
}
```

Composite events are events that aggregate one or more other events. Composite events are typically created by the engine for statements that join two event streams, and for event patterns in which the causal events are retained and reported in a composite event. The example below shows such an event pattern.

```
// Look for a pattern where BEvent follows AEvent
String pattern = "a=AEvent -> b=BEvent";
EPStatement stmt = epService.getEPAdministrator().createPattern(pattern);
stmt.addListener(testListener);
```

```
// Example listener code
public class MyUpdateListener implements UpdateListener
{
    public void update(EventBean[] newData, EventBean[] oldData)
    {
        System.out.println("a event=" + newData[0].get("a").getUnderlying());
        System.out.println("b event=" + newData[0].get("b").getUnderlying());
        }
}
```

Note that the `update` method can receive multiple events at once as it accepts an array of `EventBean` instances. For example, a time batch window may post multiple events to listeners representing a batch of events received during a given time period.

Pattern statements can also produce multiple events delivered to update listeners in one invocation. The pattern statement below, for instance, delivers an event for each A event that was not followed by a B event with the same `id` property within 60 seconds of the A event. The engine may deliver all matching A events as an array of events in a single invocation of the `update` method of each listener to the statement:

```
every a=A -> (timer:interval(60 sec) and not B(id=a.id))
```

9.6. Engine Threading and Concurrency

Esper is designed from the ground up to operate as a component to multi-threaded, highly-concurrent applications that require efficient use of Java VM resources. In addition, multi-threaded execution requires guarantees in predictability of results and deterministic processing. This section discusses these concerns in detail.

In Esper, an engine instance is a unit of separation. Applications can obtain and discard (initialize) one or more engine instances within the same Java VM and can provide the same or different engine configurations to each instance. An engine instance efficiently shares resources between statements. For example, consider two statements that declare the same data window. The engine matches up view declarations provided by each statement and can thus provide a single data window representation shared between the two statements.

Applications can use Esper APIs to concurrently, by multiple threads of execution, perform such functions as creating and managing statements, or sending events into an engine instance for processing. Applications can use one or more thread pools or any set of same or different threads of execution with any of the public Esper APIs. There are no restrictions towards threading other then those noted in specific sections of this document.

Applications using Esper retain full control over threading, allowing an engine to be easily embedded and used as a component or library in your favorite Java container or process. It is up to the application code to use multiple threads for processing events by the engine, if so desired. All event processing takes places within your application thread call stack. The exception is timer-based processing if your engine instance relies on the internal timer (default).

The fact that event processing takes places within an application thread call stack makes developing applications with Esper easier: Any common Java integrated development environment (IDE) can host an Esper engine instance. This allows developers to easily set up test cases, debug through listener code and inspect input or output events, or trace their call stack.

To send events into an engine concurrently by multiple execution threads, typically applications use the Java `java.lang.Thread` or `java.lang.Runnable` classes or Java 5 concurrent utilities that include abstractions for thread pools and blocking in-memory queues.

Each engine instance maintains a single timer thread (internal timer) providing for time or schedule-based processing within the engine. The default resolution at which the internal timer operates is 100 milliseconds. The internal timer thread can be disabled and applications can instead send external time events to an engine instance to perform timer or scheduled processing at the resolution required by an application.

Each engine instance performs minimal locking to enable high levels of concurrency. An engine instance locks on a statement level to protect statement resources.

For an engine instance to produce predictable results from the viewpoint of listeners to statements, an engine instance by default ensures that it dispatches statement result events to listeners in the order in which a statement produced result events. Applications that require the highest possible concurrency and do not require predictable order of delivery of events to listeners, this feature can be turned off via configuration.

In multithreaded environments, when one or more statements make result events available via the `insert into` clause to further statements, the engine preserves the order of events inserted into the generated insert-into stream, allowing statements that consume other statement's events to behave deterministic. This feature can also be turned off via configuration.

We generally recommended that listener implementations do not block. By implementing listener code as nonblocking code execution threads can often achieve higher levels of concurrency.

9.7. Time-Keeping Events

Special events are provided that can be used to control the time-keeping of an engine instance. There are two models for an engine to keep track of time. Internal clocking is when the engine instance relies on the `java.util.concurrent.ScheduledThreadPoolExecutor` class for time tick events. External clocking can be used to supply time ticks to the engine. The latter is useful for testing time-based event sequences or for synchronizing the engine with an external time source.

By default, the Esper engine uses internal time ticks. This behavior can be changed by sending a timer control event to the engine as shown below.

```
EPServiceProvider epService = EPServiceProviderManager.getDefaultProvider();
EPRuntime runtime = epService.getEPRuntime();
// switch to external clocking
runtime.sendEvent(new TimerControlEvent(TimerControlEvent.ClockType.CLOCK_EXTERNAL));

// send a time tick
long timeInMillis = System.currentTimeMillis(); // Or get the time somewhere else
runtime.sendEvent(new CurrentTimeEvent(timeInMillis));
```

We recommend that when disabling the internal timer, applications send an external timer event setting the start time before creating statements, such that statement start time is well-defined. Also, note that the engine outputs a warning is duplicate external time events are received.

9.8. Time Resolution

The minimum resolution that all data windows, patterns and output rate limiting operate at is the millisecond. Parameters to time window views, pattern operators or the `output` clause that are less then 1 millisecond are not allowed. As stated earlier, the default frequency at which the internal timer operates is 100 milliseconds.

The internal timer thread uses the call `System.currentTimeMillis()` to obtain system time. Please see the JIRA issue ESPER-191 Support nano/microsecond resolution for more information on Java system time-call performance, accuracy and drift. The internal timer is based on `java.util.concurrent.ScheduledThreadPoolExecutor` rather then `java.util.Timer` as the former supports high accuracy VM time.

Your application can achieve a higher tick rate then 1 tick per millisecond by sending external timer events that carry a long-value which is not based on milliseconds since January 1, 1970, 00:00:00 GMT. In this case, your time interval parameters need to take consideration of the changed use of engine time.

Thus, if your external timer events send long values that represents microseconds (1E-6 sec), then your time window interval must be 1000-times larger, i.e. "win:time(1000)" becomes a 1-second time window.

And therefore, if your external timer events send long values that represents nanoseconds (1E-9 sec), then your time window interval must be 1000000-times larger, i.e. "win:time(1000000)" becomes a 1-second time window.

9.9. Statement Object Model

The statement object model is a set of classes that provide an object-oriented representation of an EPL or pattern statement. The object model classes are found in package `com.espertech.esper.client.soda`. An instance of `EPStatementObjectModel` represents a statement's object model.

The statement object model classes are a full and complete specification of a statement. All EPL and pattern constructs including expressions and sub-queries are available via the statement object model.

In conjunction with the administrative API, the statement object model provides the means to build, change or interrogate statements beyond the EPL or pattern syntax string representation. The object graph of the statement object model is fully navigable for easy querying by code, and is also serializable allowing applications to persist or transport statements in object form, when required.

The statement object model supports full round-trip from object model to EPL statement string and back to object model: A statement object model can be rendered into an EPL string representation via the `toEPL` method on `EPStatementObjectModel`. Further, the administrative API allows to compile a statement string into an object model representation via the `compileEPL` method on `EPAdministrator`.

The `create` method on `EPAdministrator` creates and starts a statement as represented by an object model. In order to obtain an object model from an existing statement, obtain the statement expression text of the statement via the `getText` method on `EPStatement` and use the `compileEPL` method to obtain the object model.

The following limitations apply:

- Statement object model classes are not safe for sharing between threads other then for read access.
- Between versions of Esper, the serialized form of the object model is subject to change. Esper makes no guarantees that the serialized object model of one version will be fully compatible with the serialized object model generated by another version of Esper. Please consider this issue when storing Esper object models in persistent store.

9.9.1. Building an Object Model

A `EPStatementObjectModel` consists of an object graph representing all possible clauses that can be part of an EPL statement.

Among all clauses, the `SelectClause` and `FromClause` objects are required clauses that must be present, in order to define what to select and where to select from.

Table 9.3. Required Statement Object Model Instances

Class	Description
EPStatementObjectModel	All statement clauses for a statement, such as the select-clause and the from-clause, are specified within the object graph of an instance of this class
SelectClause	A list of the selection properties or expressions, or a wildcard
FromClause	A list of one or more streams; A stream can be a filter-based, a pattern-based or a SQL-based stream; Views are added to streams to provide data window or other projections

Part of the statement object model package are convenient builder classes that make it easy to build a new object model or change an existing object model. The `SelectClause` and `FromClause` are such builder classes and provide convenient `create` methods.

Within the from-clause we have a choice of different streams to select on. The `FilterStream` class represents a stream that is filled by events of a certain type and that pass an optional filter expression.

We can use the classes introduced above to create a simple statement object model:

```
EPStatementObjectModel model = new EPStatementObjectModel();
model.setSelectClause(SelectClause.createWildcard());
model.setFromClause(FromClause.create(FilterStream.create("com.chipmaker.ReadyEvent")));
```

The model as above is equivalent to the EPL :

```
select * from com.chipmaker.ReadyEvent
```

Last, the code snippet below creates a statement from the object model:

```
EPStatement stmt = epService.getEPAdministrator().create(model);
```

Notes on usage:

- Variable names can simply be treated as property names
- When selecting from named windows, the name of the named window is the event type alias for us in `FilterStream` instances or patterns
- To compile an arbitrary sub-expression text into an `Expression` object representation, simply add the expression text to a `where` clause, compile the EPL string into an object model via the `compileEPL` on `EPAdministrator`, and obtain the compiled `where` from the `EPStatementObjectModel` via the `getWhereClause` method.

Notes on usage:

- Variable names can simply be treated as property names
- When selecting from named windows, the name of the named window is the event type alias for us in `FilterStream` instances or patterns
- To compile an arbitrary sub-expression text into an `Expression` object representation, simply add the expression text to a `where` clause, compile the EPL string into an object model via the `compileEPL` method on `EPAdministrator`, and obtain the compiled `where` clause expression object from the `EPStatementObjectModel` via the `getWhereClause` method.

9.9.2. Building Expressions

The `EPStatementObjectModel` includes an optional where-clause. The where-clause is a filter expression that the engine applies to events in one or more streams. The key interface for all expressions is the `Expression` interface.

The `Expressions` class provides a convenient way of obtaining `Expression` instances for all possible expressions. Please consult the JavaDoc for detailed method information. The next example discusses sample where-clause expressions.

Use the `Expressions` class as a service for creating expression instances, and add additional expressions via the `add` method that most expressions provide.

In the next example we add a simple where-clause to the EPL as shown earlier:

```
select * from com.chipmaker.ReadyEvent where line=8
```

And the code to add a where-clause to the object model is below.

```
model.setWhereClause(Expressions.eq("line", 8));
```

The following example considers a more complex where-clause. Assume we need to build an expression using logical-and and logical-or:

```
select * from com.chipmaker.ReadyEvent
where (line=8) or (line=10 and age<5)
```

The code for building such a where-clause by means of the object model classes is:

```
model.setWhereClause(Expressions.or()
  .add(Expressions.eq("line", 8))
  .add(Expressions.and()
      .add(Expressions.eq("line", 10))
      .add(Expressions.lt("age", 5))
  ));
```

9.9.3. Building a Pattern Statement

The `Patterns` class is a factory for building pattern expressions. It provides convenient methods to create all pattern expressions of the pattern language.

Patterns in EPL are seen as a stream of events that consist of patterns matches. The `PatternStream` class represents a stream of pattern matches and contains a pattern expression within.

For instance, consider the following pattern statement.

```
select * from pattern [every a=MyAEvent and not b=MyBEvent]
```

The next code snippet outlines how to use the statement object model and specifically the `Patterns` class to create a statement object model that is equivalent to the pattern statement above.

```
EPStatementObjectModel model = new EPStatementObjectModel();
model.setSelectClause(SelectClause.createWildcard());
PatternExpr pattern = Patterns.and()
  .add(Patterns.everyFilter("MyAEvent", "a"))
  .add(Patterns.notFilter("MyBEvent", "b"));
model.setFromClause(FromClause.create(PatternStream.create(pattern)));
```

9.9.4. Building a Select Statement

In this section we build a complete example statement and include all optional clauses in one EPL statement, to demonstrate the object model API.

A sample statement:

```
insert into ReadyStreamAvg(line, avgAge)
select line, avg(age) as avgAge
from com.chipmaker.ReadyEvent(line in (1, 8, 10)).win:time(10) as RE
where RE.waverId != null
group by line
having avg(age) < 0
output every 10.0 seconds
order by line
```

Finally, this code snippet builds the above statement from scratch:

```
EPStatementObjectModel model = new EPStatementObjectModel();
model.setInsertInto(InsertIntoClause.create("ReadyStreamAvg", "line", "avgAge"));
model.setSelectClause(SelectClause.create()
    .add("line")
    .add(Expressions.avg("age"), "avgAge"));
Filter filter = Filter.create("com.chipmaker.ReadyEvent", Expressions.in("line", 1, 8, 10));
model.setFromClause(FromClause.create(
    FilterStream.create(filter, "RE").addView("win", "time", 10)));
model.setWhereClause(Expressions.isNotNull("RE.waverId"));
model.setGroupByClause(GroupByClause.create("line"));
model.setHavingClause(Expressions.lt(Expressions.avg("age"), Expressions.constant(0)));
model.setOutputLimitClause(OutputLimitClause.create(10, OutputLimitUnit.SECONDS));
model.setOrderByClause(OrderByClause.create("line"));
```

9.9.5. Building a Create-Variable and On-Set Statement

This sample statement creates a variable:

```
create variable integer var_output_rate = 10
```

The code to build the above statement using the object model:

```
EPStatementObjectModel model = new EPStatementObjectModel();
model.setCreateVariable(CreateVariableClause.create("integer", "var_output_rate", 10));
epService.getEPAdministrator().create(model);
```

A second statement sets the variable to a new value:

```
on NewValueEvent set var_output_rate = new_rate
```

The code to build the above statement using the object model:

```
EPStatementObjectModel model = new EPStatementObjectModel();
model.setOnExpr(OnClause.createOnSet("var_output_rate", Expressions.property("new_rate")));
model.setFromClause(FromClause.create(FilterStream.create("NewValueEvent")));
EPStatement stmtSet = epService.getEPAdministrator().create(model);
```

9.9.6. Building Create-Window, On-Delete and On-Select Statements

This sample statement creates a named window:

```
create window OrdersTimeWindow.win:time(30 sec) as select symbol as sym, volume as vol, price from Or
```

The is the code that builds the create-window statement as above:

```
EPStatementObjectModel model = new EPStatementObjectModel();
model.setCreateWindow(CreateWindowClause.create("OrdersTimeWindow").addView("win", "time", 30));
model.setSelectClause(SelectClause.create()
                .addWithAlias("symbol", "sym")
                .addWithAlias("volume", "vol")
                .add("price"));
model.setFromClause(FromClause.create(FilterStream.create("OrderEvent)));
```

A second statement deletes from the named window:

```
on NewOrderEvent as myNewOrders
delete from AllOrdersNamedWindow as myNamedWindow
```

```
where myNamedWindow.symbol = myNewOrders.symbol
```

The object model is built by:

```
EPStatementObjectModel model = new EPStatementObjectModel();
model.setOnExpr(OnClause.createOnDelete("AllOrdersNamedWindow", "myNamedWindow"));
model.setFromClause(FromClause.create(FilterStream.create("NewOrderEvent", "myNewOrders")));
model.setWhereClause(Expressions.eqProperty("myNamedWindow.symbol", "myNewOrders.symbol"));
EPStatement stmtOnDelete = epService.getEPAdministrator().create(model);
```

A third statement selects from the named window using the non-continuous on-demand selection via on-select:

```
on QueryEvent(volume>0) as query
select count(*) from OrdersNamedWindow as win
where win.symbol = query.symbol
```

The on-select statement is built from scratch via the object model as follows:

```
EPStatementObjectModel model = new EPStatementObjectModel();
model.setOnExpr(OnClause.createOnSelect("OrdersNamedWindow", "win"));
model.setWhereClause(Expressions.eqProperty("win.symbol", "query.symbol"));
model.setFromClause(FromClause.create(FilterStream.create("QueryEvent", "query",
  Expressions.gt("volume", 0))));
model.setSelectClause(SelectClause.create().add(Expressions.countStar()));
EPStatement stmtOnSelect = epService.getEPAdministrator().create(model);
```

9.10. Prepared Statement and Substitution Parameters

The `prepare` method that is part of the administrative API pre-compiles an EPL statement and stores the pre-compiled statement in an `EPPreparedStatement` object. This object can then be used to efficiently start the parameterized statement multiple times.

Substitution parameters are inserted into an EPL statement as a single question mark character `'?'`. The engine assigns the first substitution parameter an index of 1 and subsequent parameters increment the index by one.

Substitution parameters can be inserted into any EPL construct that takes an expression. They are therefore valid in any clauses such as the select-clause, from-clause filters, where-clause, group-by-clause, having-clause or order-by-clause. Substitution parameters cannot be used as parameters to views, pattern observers and guards. They also cannot be used where a numeric constant is required rather then an expression.

All substitution parameters must be replaced by actual values before a statement with substitution parameters can be started. Substitution parameters can be replaced with an actual value using the `setObject` method for each index. Substitution parameters can be set to new values and new statements can be created from the same `EPPreparedStatement` object more then once.

While the `setObject` method allows substitution parameters to assume any actual value including application Java objects or enumeration values, the application must provide the correct type of substitution parameter that matches the requirements of the expression the parameter resides in.

In the following example of setting parameters on a prepared statement and starting the prepared statement, `epService` represents an engine instance:

```
String stmt = "select * from com.chipmaker.ReadyEvent(line=?)";
EPPreparedStatement prepared = epService.getEPAdministrator().prepareEPL(stmt);
prepared.setObject(1, 8);
EPStatement statement = epService.getEPAdministrator().create(prepared);
```

Chapter 10. Configuration

Esper engine configuration is entirely optional. Esper has a very small number of configuration parameters that can be used to simplify event pattern and EPL statements, and to tune the engine behavior to specific requirements. The Esper engine works out-of-the-box without configuration.

An application can supply configuration at the time of engine allocation using the `Configuration` class, and can also use XML files to hold configuration. Configuration can be changed at runtime via the `ConfigurationOperations` interface available from `EPAdministrator` via the `getConfiguration` method.

10.1. Programmatic Configuration

An instance of `com.espertech.esper.client.Configuration` represents all configuration parameters. The `Configuration` is used to build an `EPServiceProvider`, which provides the administrative and runtime interfaces for an Esper engine instance.

You may obtain a `Configuration` instance by instantiating it directly and adding or setting values on it. The `Configuration` instance is then passed to `EPServiceProviderManager` to obtain a configured Esper engine.

```
Configuration configuration = new Configuration();
configuration.addEventTypeAlias("PriceLimit", PriceLimit.class.getName());
configuration.addEventTypeAlias("StockTick", StockTick.class.getName());
configuration.addImport("org.mycompany.mypackage.MyUtility");
configuration.addImport("org.mycompany.util.*");

EPServiceProvider epService = EPServiceProviderManager.getProvider("sample", configuration);
```

Note that `Configuration` is meant only as an initialization-time object. The Esper engine represented by an `EPServiceProvider` does not retain any association back to the `Configuration`.

The `ConfigurationOperations` interface provides runtime configuration options. Through this interface applications can, for example, add new event types or aliases at runtime and then create new statements that rely on the additional configuration. The `getConfiguration` method on `EPAdministrator` allows access to `ConfigurationOperations`.

10.2. Configuration via XML File

An alternative approach to configuration is to specify a configuration in a XML file.

The default name for the XML configuration file is `esper.cfg.xml`. Esper reads this file from the root of the `CLASSPATH` as an application resource via the `configure` method.

```
Configuration configuration = new Configuration();
configuration.configure();
```

The `Configuration` class can read the XML configuration file from other sources as well. The `configure` method accepts `URL, File and String` filename parameters.

```
Configuration configuration = new Configuration();
configuration.configure("myengine.esper.cfg.xml");
```

10.3. XML Configuration File

Here is an example configuration file. The schema for the configuration file can be found in the `etc` folder and is named `esper-configuration-2-0.xsd`. It is also available online at `http://www.espertech.com/schema/esper/esper-configuration-2.0.xsd` so that IDE can fetch it automatically. The namespace used is `http://www.espertech.com/schema/esper`.

```
<?xml version="1.0" encoding="UTF-8"?>
<esper-configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.espertech.com/schema/esper"
    xsi:schemaLocation="
http://www.espertech.com/schema/esper http://www.espertech.com/schema/esper/esper-configuration-2.0.xs
  <event-type alias="StockTick" class="com.espertech.esper.example.stockticker.event.StockTick"/>
  <event-type alias="PriceLimit" class="com.espertech.esper.example.stockticker.event.PriceLimit"/>
  <auto-import import-name="org.mycompany.mypackage.MyUtility"/>
  <auto-import import-name="org.mycompany.util.*"/>
</esper-configuration>
```

The example above is only a subset of the configuration items available. The next chapters outline the available configuration in greater detail.

10.4. Configuration Items

10.4.1. Events represented by Java Classes

Package of Java Event Classes

Via this configuration an application can make the Java package or packages that contain an application's Java event classes known to an engine. Thereby an application can simply refer to event types in statements by using the simple class name of each Java class representing an event type.

For example, consider an order-taking application that places all event classes in package `com.mycompany.order.event`. One Java class representing an event is the class `OrderEvent`. The application can simply issue a statement as follows to select `OrderEvent` events:

```
select * from OrderEvent
```

The XML configuration for defining the Java packages that contain Java event classes is:

```
<event-type-auto-alias package-name="com.mycompany.order.event"/>
```

The same configuration but using the `Configuration` class:

```
Configuration config = new Configuration();
config.addEventTypeAutoAlias("com.mycompany.order.event");
// ... or ...
config.addEventTypeAutoAlias(MyEvent.getPackage().getName());
```

Event type alias to Java class mapping

This configuration item can be used to allow event pattern statements and EPL statements to use an event type alias rather then the fully qualified Java class name. Note that Java Interface classes and abstract classes are also supported as event types via the fully qualified Java class name, and an event type alias can also be defined for such classes.

The example pattern statement below first shows a pattern that uses the alias `StockTick`. The second pattern statement is equivalent but specifies the fully-qualified Java class name.

```
every StockTick(symbol='IBM')"
```

```
every com.espertech.esper.example.stockticker.event.StockTick(symbol='IBM')
```

The event type alias can be listed in the XML configuration file as shown below. The `Configuration` API can also be used to programatically specify an event type alias, as shown in an earlier code snippet.

```
<event-type alias="StockTick" class="com.espertech.esper.example.stockticker.event.StockTick"/>
```

Non-JavaBean and Legacy Java Event Classes

Esper can process Java classes that provide event properties through other means then through JavaBean-style getter methods. It is not necessary that the method and member variable names in your Java class adhere to the JavaBean convention - any public methods and public member variables can be exposed as event properties via the below configuration.

A Java class can optionally be configured with an accessor style attribute. This attribute instructs the engine how it should expose methods and fields for use as event properties in statements.

Table 10.1. Accessor Styles

Style Name	Description
`javabean`	As the default setting, the engine exposes an event property for each public method following the JavaBean getter-method conventions
`public`	The engine exposes an event property for each public method and public member variable of the given class
`explicit`	The engine exposes an event property only for the explicitly configured public methods and public member variables

Using the `public` setting for the `accessor-style` attribute instructs the engine to expose an event property for each public method and public member variable of a Java class. The engine assigns event property names of the same name as the name of the method or member variable in the Java class.

For example, assuming the class `MyLegacyEvent` exposes a method named `readValue` and a member variable named `myField`, we can then use properties as shown.

```
select readValue, myField from MyLegacyEvent
```

Using the `explicit` setting for the `accessor-style` attribute requires that event properties are declared via configuration. This is outlined in the next chapter.

When configuring an engine instance from a XML configuration file, the XML snippet below demonstrates the use of the `legacy-type` element and the `accessor-style` attribute.

```
<event-type alias="MyLegacyEvent" class="com.mycompany.mypackage.MyLegacyEventClass">
```

```
  <legacy-type accessor-style="public"/>
</event-type>
```

When configuring an engine instance via Configuration API, the sample code below shows how to set the accessor style.

```
Configuration configuration = new Configuration();
ConfigurationEventTypeLegacy legacyDef = new ConfigurationEventTypeLegacy();
legacyDef.setAccessorStyle(ConfigurationEventTypeLegacy.AccessorStyle.PUBLIC);
config.addEventTypeAlias("MyLegacyEvent", MyLegacyEventClass.class.getName(), legacyDef);

EPServiceProvider epService = EPServiceProviderManager.getProvider("sample", configuration);
```

Specifying Event Properties for Java Classes

Sometimes it may be convenient to use event property names in pattern and EPL statements that are backed up by a given public method or member variable (field) in a Java class. And it can be useful to declare multiple event properties that each map to the same method or member variable.

We can configure properties of events via `method-property` and `field-property` elements, as the next example shows.

```
<event-type alias="StockTick" class="com.espertech.esper.example.stockticker.event.StockTickEvent">
        <legacy-type accessor-style="javabean" code-generation="enabled">
                <method-property name="price" accessor-method="getCurrentPrice" />
                <field-property name="volume" accessor-field="volumeField" />
        </legacy-type>
</event-type>
```

The XML configuration snippet above declared an event property named `price` backed by a getter-method named `getCurrentPrice`, and a second event property named `volume` that is backed by a public member variable named `volumeField`. Thus the price and volume properties can be used in a statement:

```
select avg(price * volume) from StockTick
```

As with all configuration options, the API can also be used:

```
Configuration configuration = new Configuration();
ConfigurationEventTypeLegacy legacyDef = new ConfigurationEventTypeLegacy();
legacyDef.addMethodProperty("price", "getCurrentPrice");
legacyDef.addFieldProperty("volume", "volumeField");
config.addEventTypeAlias("StockTick", StockTickEvent.class.getName(), legacyDef);
```

Turning off Code Generation

Esper employes the `CGLIB` library for very fast read access to event property values. For certain legacy Java classes it may be desirable to disable the use of this library and instead use Java reflection to obtain event property values from event objects.

In the XML configuration, the optional `code-generation` attribute in the `legacy-type` section can be set to `disabled` as shown next.

```
<event-type alias="MyLegacyEvent" class="com.mycompany.package.MyLegacyEventClass">
        <legacy-type accessor-style="javabean" code-generation="disabled" />
</event-type>
```

The sample below shows how to configure this option via the API.

```
Configuration configuration = new Configuration();
ConfigurationEventTypeLegacy legacyDef = new ConfigurationEventTypeLegacy();
legacyDef.setCodeGeneration(ConfigurationEventTypeLegacy.CodeGeneration.DISABLED);
config.addEventTypeAlias("MyLegacyEvent", MyLegacyEventClass.class.getName(), legacyDef);
```

Case Sensitivity and Property Names

By default the engine resolves Java event properties case sensitive. That is, property names in statements must match JavaBean-convention property names in name and case. This option controls case sensitivity per Java class.

In the configuration XML, the optional `property-resolution-style` attribute in the `legacy-type` element can be set to any of these values:

Table 10.2. Property Resolution Case Sensitivity Styles

Style Name	Description
`case_sensitive (default)`	As the default setting, the engine matches property names for the exact name and case only.
`case_insensitive`	Properties are matched if the names are identical. A case insensitive search is used and will choose the first property that matches the name exactly or the first property that matches case insensitively should no match be found.
`distinct_case_insensitive`	Properties are matched if the names are identical. A case insensitive search is used and will choose the first property that matches the name exactly case insensitively. If more than one 'name' can be mapped to the property an exception is thrown.

The sample below shows this option in XML configuration, however the setting can also be changed via API:

```
<event-type alias="MyLegacyEvent" class="com.mycompany.package.MyLegacyEventClass">
  <legacy-type property-resolution-style="case_insensitive"/>
</event-type>
```

10.4.2. Events represented by `java.util.Map`

The engine can process `java.util.Map` events via the `sendEvent(Map map, String eventTypeAlias)` method on the `EPRuntime` interface. Entries in the Map represent event properties. Keys must be of type `java.util.String` for the engine to be able to look up event property names in pattern or EPL statements. Values can be of any type. JavaBean-style objects as values in a `Map` can be processed by the engine, and strongly-typed nested maps are also supported. Please see the Chapter 2, *Event Representations* section for details on how to use `Map` events with the engine.

Via configuration we provide an event type alias name for `Map` events for use in statements, and the event property names and types enabling the engine to validate properties in statements.

The below snippet of XML configuration configures an event named `MyMapEvent`.

```
<event-type alias="MyMapEvent">
  <java-util-map>
    <map-property name="carId" class="int"/>
    <map-property name="carType" class="string"/>
```

```
    <map-property name="assembly" class="com.mycompany.Assembly"/>
  </java-util-map>
</event-type>
```

This configuration defines the `carId` property of `MyMapEvent` events to be of type `int`, and the `carType` property to be of type `java.util.String`. The `assembly` property of the Map event will contain instances of `com.mycompany.Assembly` for the engine to query.

The valid list of values for the type definition via the `class` attribute is:

- `string` or `java.lang.String`
- `char` or `java.lang.Character`
- `byte` or `java.lang.Byte`
- `short` or `java.lang.Short`
- `int` or `java.lang.Integer`
- `long` or `java.lang.Long`
- `float` or `java.lang.Float`
- `double` or `java.lang.Double`
- `boolean` or `java.lang.Boolean`
- Any fully-qualified Java class name that can be resolved by the engine via `Class.forName`

You can also use the configuration API to configure `Map` event types, as the short code snippet below demonstrates:

```
Properties properties = new Properties();
properties.put("carId", "int");
properties.put("carType", "string");
properties.put("assembly", Assembly.class.getName());

Configuration configuration = new Configuration();
configuration.addEventTypeAlias("MyMapEvent", properties);
```

For strongly-typed nested maps (maps-within-maps), the configuration API method `addNestableEventTypeAlias` must be used to define the nested types. The XML configuration does not provide the capability to configure nested maps.

Finally, here is a sample EPL statement that uses the configured `MyMapEvent` map event. This statement uses the `chassisTag` and `numParts` properties of `Assembly` objects in each map.

```
select carType, assembly.chassisTag, count(assembly.numParts) from MyMapEvent.win:time(60 sec)
```

10.4.3. Events represented by `org.w3c.dom.Node`

Via this configuration item the Esper engine can natively process `org.w3c.dom.Node` instances, i.e. XML document object model (DOM) nodes. Please see the Chapter 2, *Event Representations* section for details on how to use `Node` events with the engine.

Esper allows configuring XPath expressions as event properties. You can specify arbitrary XPath functions or expressions and provide a property name by which their result values will be available for use in expressions.

For XML documents that follow a XML schema, Esper can load and interrogate your schema and validate event property names and types against the schema information.

Nested, mapped and indexed event properties are also supported in expressions against `org.w3c.dom.Node` events. Thus XML trees can conveniently be interrogated using the existing event property syntax for querying

JavaBean objects, JavaBean object graphs or `java.util.Map` events.

In the simplest form, the Esper engine only requires a configuration entry containing the root element name and the event type alias in order to process `org.w3c.dom.Node` events:

```
<event-type alias="MyXMLNodeEvent">
  <xml-dom root-element-name="myevent" />
</event-type>
```

You can also use the configuration API to configure XML event types, as the short example below demonstrates. In fact, all configuration options available through XML configuration can also be provided via setter methods on the `ConfigurationEventTypeXMLDOM` class.

```
Configuration configuration = new Configuration();
ConfigurationEventTypeXMLDOM desc = new ConfigurationEventTypeXMLDOM();
desc.setRootElementName("myevent");
desc.addXPathProperty("name1", "/element/@attribute", XPathConstants.STRING);
desc.addXPathProperty("name2", "/element/subelement", XPathConstants.NUMBER);
configuration.addEventTypeAlias("MyXMLNodeEvent", desc);
```

The next example presents configuration options in a sample configuration entry.

```
<event-type alias="AutoIdRFIDEvent">
  <xml-dom root-element-name="Sensor" schema-resource="data/AutoIdPmlCore.xsd"
       default-namespace="urn:autoid:specification:interchange:PMLCore:xml:schema:1">
    <namespace-prefix prefix="pmlcore"
       namespace="urn:autoid:specification:interchange:PMLCore:xml:schema:1"/>
    <xpath-property property-name="countTags"
       xpath="count(/pmlcore:Sensor/pmlcore:Observation/pmlcore:Tag)" type="number"/>
  </xml-dom>
</event-type>
```

This example configures an event property named `countTags` whose value is computed by an XPath expression. The namespace prefixes and default namespace are for use with XPath expressions and must also be made known to the engine in order for the engine to compile XPath expressions. Via the `schema-resource` attribute we instruct the engine to load a schema file.

Here is an example EPL statement using the configured event type named `AutoIdRFIDEvent`.

```
select ID, countTags from AutoIdRFIDEvent.win:time(30 sec)
```

Schema Resource

The `schema-resource` attribute takes a schema resource URL or classpath-relative filename. The engine attempts to resolve the schema resource as an URL. If the schema resource name is not a valid URL, the engine attempts to resolve the resource from classpath via the `ClassLoader.getResource` method using the thread context class loader. If the name could not be resolved, the engine uses the Configuration class classloader.

By configuring a schema file for the engine to load, the engine performs these additional services:

- Validates the event properties in a statement, ensuring the event property name matches an attribute or element in the XML
- Determines the type of the event property allowing event properties to be used in type-sensitive expressions such as expressions involving arithmetic (Note: XPath properties are also typed)
- Matches event property names to either element names or attributes

If no schema resource is specified, none of the event properties specified in statements are validated at statement creation time and their type defaults to `java.lang.String`. Also, attributes are not supported if no schema resource is specified and must thus be declared via XPath expression.

XPath Property

The `xpath-property` element adds event properties to the event type that are computed via an XPath expression. In order for the XPath expression to compile, be sure to specify the `default-namespace` attribute and use the `namespace-prefix` to declare namespace prefixes.

XPath expression properties are strongly typed. The `type` attribute allows the following values. These values correspond to those declared by `javax.xml.xpath.XPathConstants`.

- number (Note: resolves to a `double`)
- string
- boolean

Absolute or Deep Property Resolution

This setting indicates that when properties are compiled to XPath expressions that the compilation should generate an absolute XPath expression or a deep (find element) XPath expression.

For example, consider the following statement against an event type that is represented by a XML DOM document, assuming the event type GetQuote has been configured with the engine as a XML DOM event type:

```
select request, request.symbol from GetQuote
```

By default, the engine compiles the "request" property name to an XPath expression "/GetQuote/request". It compiles the nested property named "request.symbol" to an XPath expression "/GetQuote/request/symbol", wherein the root element node is "GetQuote".

By setting absolute property resolution to false, the engine compiles the "request" property name to an XPath expression "//request". It compiles the nested property named "request.symbol" to an XPath expression "//request/symbol". This enables these elements to be located anywhere in the XML document.

The setting is available in XML via the attribute `resolve-properties-absolute`.

The configuration API provides the above settings as shown here in a sample code:

```
ConfigurationEventTypeXMLDOM desc = new ConfigurationEventTypeXMLDOM();
desc.setRootElementName("GetQuote");
desc.setDefaultNamespace("http://services.samples/xsd");
desc.setRootElementNamespace("http://services.samples/xsd");
desc.addNamespacePrefix("m0", "http://services.samples/xsd");
desc.setResolvePropertiesAbsolute(false);
configuration.addEventTypeAlias("GetQuote", desc);
```

10.4.4. Class and package imports

Esper allows invocations of static Java library functions in expressions, as outlined in Section 7.1, “Single-row Function Reference”. This configuration item can be set to allow a partial rather than a fully qualified class name in such invocations. The imports work in the same way as in Java files, so both packages and classes can be imported.

```
select Math.max(priceOne, PriceTwo)
// via configuration equivalent to
select java.lang.Math.max(priceOne, priceTwo)
```

Esper auto-imports the following Java library packages if no other configuration is supplied. This list is replaced with any configuration specified in a configuration file or through the API.

- java.lang.*
- java.math.*
- java.text.*
- java.util.*

In a XML configuration file the auto-import configuration may look as below:

```
<auto-import import-name="com.mycompany.mypackage.*"/>
<auto-import import-name="com.mycompany.myapp.MyUtilityClass"/>
```

Here is an example of providing imports via the API:

```
Configuration config = new Configuration();
config.addImport("com.mycompany.mypackage.*"); // package import
config.addImport("com.mycompany.mypackage.MyLib");   // class import
```

10.4.5. Cache Settings for Method Invocations

Method invocations are allowed in the `from` clause in EPL, such that your application may join event streams to the data returned by a web service, or to data read from a distributed cache or object-oriented database, or obtain data by other means. A local cache may be placed in front of such method invocations through the configuration settings described herein.

The LRU cache is described in detail in Section 10.4.7.5.1, “LRU Cache”. The expiry-time cache documentation can be found in Section 10.4.7.5.2, “Expiry-time Cache”

The next XML snippet is a sample cache configuration that applies to methods provided by the classes 'MyFromClauseLookupLib' and 'MyFromClauseWebServiceLib'. The XML and API configuration understand both the fully-qualified Java class name, as well as the simple class name:

```
<method-reference class-name="com.mycompany.MyFromClauseLookupLib">
  <expiry-time-cache max-age-seconds="10" purge-interval-seconds="10" ref-type="weak"/>
</method-reference>
<method-reference class-name="MyFromClauseWebServiceLib">
  <lru-cache size="1000"/>
</method-reference>
```

10.4.6. Variables

Variables can be created dynamically in EPL via the `create variable` syntax but can also be configured at runtime and at configuration time.

A variable is declared by specifying a variable name, the variable type and an optional initialization value. The initialization value can be of the same or compatible type as the variable type, or can also be a String value that, when parsed, is compatible to the type declared for the variable.

In a XML configuration file the variable configuration may look as below. The `Configuration` API can also be used to configure variables.

```
<variable name="var_threshold" type="long" initialization-value="100"/>
<variable name="var_key" type="string"/>
```

10.4.7. Relational Database Access

Esper has the capability to join event streams against historical data sources, such as a relational database. This section describes the configuration entries that the engine requires to access data stored in your database. Please see Section 4.14, “Joining Relational Data via SQL” for information on the use of EPL queries that include historical data sources.

EPL queries that poll data from a relational database specify the name of the database as part of the EPL statement. The engine uses the configuration information described here to resolve the database name in the statement to database settings. The required and optional database settings are summarized below.

- Database connections can be obtained via JDBC `javax.xml.DataSource` or alternatively via `java.sql.DriverManager`. Either one of these methods to obtain new database connections is a required configuration.
- Optionally, JDBC connection-level settings such as auto-commit, transaction isolation level, read-only and the catalog name can be defined.
- Optionally, a connection lifecycle can be set to indicate to the engine whether the engine must retain connections or must obtain a new connection for each lookup.
- Optionally, define a cache policy to allow the engine to retrieve data from a query cache, reducing the number of query executions.

Some of the settings can have important performance implications that need to be carefully considered in relationship to your database software, JDBC driver and runtime environment. This section attempts to outline such implications where appropriate.

The sample XML configuration file in the "etc" folder can be used as a template for configuring database settings. All settings are also available by means of the configuration API through the classes `Configuration` and `ConfigurationDBRef`.

Connections obtained via DataSource

The snippet of XML below configures a database named `mydb1` to obtain connections via a `javax.sql.DataSource`. The `datasource-connection` element instructs the engine to obtain new connections to the database `mydb1` by performing a lookup via `javax.naming.InitialContext` for the given object lookup name. Optional environment properties for the `InitialContext` are also shown in the example.

```
<database-reference name="mydb1">
  <datasource-connection context-lookup-name="java:comp/env/jdbc/mydb">
    <env-property name="java.naming.factory.initial" value ="com.myclass.CtxFactory"/>
    <env-property name="java.naming.provider.url" value ="iiop://localhost:1050"/>
  </datasource-connection>
</database-reference>
```

To help you better understand how the engine uses this information to obtain connections, we have included the logic below.

```
if (envProperties.size() > 0) {
  initialContext = new InitialContext(envProperties);
}
else {
  initialContext = new InitialContext();
}
DataSource dataSource = (DataSource) initialContext.lookup(lookupName);
Connection connection = dataSource.getConnection();
```

Connections obtained via DriverManager

The next snippet of XML configures a database named `mydb2` to obtain connections via `java.sql.DriverManager`. The `drivermanager-connection` element instructs the engine to obtain new connections to the database `mydb2` by means of `Class.forName` and `DriverManager.getConnection` using the class name, URL and optional username, password and connection arguments.

```
<database-reference name="mydb2">
  <drivermanager-connection class-name="my.sql.Driver"
        url="jdbc:mysql://localhost/test?user=root&amp;password=mypassword"
        user="myuser" password="mypassword">
    <connection-arg name="user" value ="myuser"/>
    <connection-arg name="password" value ="mypassword"/>
    <connection-arg name="somearg" value ="someargvalue"/>
  </drivermanager-connection>
</database-reference>
```

The username and password are shown in multiple places in the XML only as an example. Please check with your database software on the required information in URL and connection arguments.

Connections-level settings

Additional connection-level settings can optionally be provided to the engine which the engine will apply to new connections. When the engine obtains a new connection, it applies only those settings to the connection that are explicitly configured. The engine leaves all other connection settings at default values.

The below XML is a sample of all available configuration settings. Please refer to the Java API JavaDocs for `java.sql.Connection` for more information to each option or check the documentation of your JDBC driver and database software.

```
<database-reference name="mydb2">
... configure data source or driver manager settings...
  <connection-settings auto-commit="true" catalog="mycatalog"
      read-only="true" transaction-isolation="1" />
</database-reference>
```

The `read-only` setting can be used to indicate to your database engine that SQL statements are read-only. The `transaction-isolation` and `auto-commit` help you database software perform the right level of locking and lock release. Consider setting these values to reduce transactional overhead in your database queries.

Connections lifecycle settings

By default the engine retains a separate database connection for each started EPL statement. However, it is possible to override this behavior and require the engine to obtain a new database connection for each lookup, and to close that database connection after the lookup is completed. This often makes sense when you have a large number of EPL statements and require pooling of connections via a connection pool. If your runtime environment includes an application server, the connection pool may be exposed as a `DataSource`.

The XML for this option is below. The connection lifecycle allows the following values: `pooled` and `retain`.

```
<database-reference name="mydb2">
... configure data source or driver manager settings...
    <connection-lifecycle value="pooled"/>
</database-reference>
```

Cache settings

Cache settings can dramatically reduce the number of database queries that the engine executes for EPL statements. If no cache setting is specified, the engine does not cache query results and executes a separate database query for every event.

Caches store the results of database queries and make these results available to subsequent queries using the exact same query parameters as the query for which the result was stored. If your query returns one or more rows, the cache keep the result rows of the query keyed to the parameters of the query. If your query returns no rows, the cache also keeps the empty result. Query results are held by a cache until the cache entry is evicted. The strategies available for evicting cached query results are listed next.

LRU Cache

The least-recently-used (LRU) cache is configured by a maximum size. The cache discards the least recently used query results first once the cache reaches the maximum size.

The XML configuration entry for a LRU cache is as below. This entry configures an LRU cache holding up to 1000 query results.

```
<database-reference name="mydb">
... configure data source or driver manager settings...
    <lru-cache size="1000"/>
</database-reference>
```

Expiry-time Cache

The expiry time cache is configured by a maximum age in seconds, a purge interval and an optional reference type. The cache discards (on the get operation) any query results that are older then the maximum age so that stale data is not used. If the cache is not empty, then every purge interval number of seconds the engine purges any expired entries from the cache.

The XML configuration entry for an expiry-time cache is as follows. The example configures an expiry time cache in which prior query results are valid for 60 seconds and which the engine inspects every 2 minutes to remove query results older then 60 seconds.

```
<database-reference name="mydb">
... configure data source or driver manager settings...
    <expiry-time-cache max-age-seconds="60" purge-interval-seconds="120" />
</database-reference>
```

By default, the expiry-time cache is backed by a `java.util.WeakHashMap` and thus relies on weak references. That means that cached SQL results can be freed during garbage collection.

Via XML or using the configuration API the type of reference can be configured to not allow entries to be garbage collected, by setting the `ref-type` property to `hard`:

```
<database-reference name="mydb">
... configure data source or driver manager settings...
    <expiry-time-cache max-age-seconds="60" purge-interval-seconds="120" ref-type="hard"/>
</database-reference>
```

The last setting for the cache reference type is `soft`: This strategy allows the garbage collection of cache entries only when all other weak references have been collected.

Column Change Case

This setting instructs the engine to convert to lower- or uppercase any output column names returned by your database system. When using Oracle relational database software, for example, column names can be changed to lowercase via this setting.

A sample XML configuration entry for this setting is:

```
<column-change-case value="lowercase"/>
```

SQL Types Mapping

By providing a mapping of SQL types (`java.sql.Types`) to Java built-in types your code can avoid using sometimes awkward default database types and can easily change the way Esper returns Java types for columns returned by a SQL query.

The mapping maps a constant as defined by `java.sql.Types` to a Java built-in type of any of the following Java type names: `String, BigDecimal, Boolean, Byte, Short, Int, Long, Float, Double, ByteArray, SqlDate, SqlTime, SqlTimestamp`. The Java type names are not case-sensitive.

A sample XML configuration entry for this setting is shown next. The sample maps `Types.NUMERIC` (of value `2`) to the Java `int` type.

```
<sql-types-mapping sql-type="2" java-type="int" />
```

Metadata Origin

This setting controls how the engine retrieves SQL statement metadata from JDBC prepared statements.

Table 10.3. Syntax and results of aggregate functions

Option	Description
default	By default, the engine detects the driver name and queries prepared statement metadata if the driver is not an Oracle database driver. For Oracle drivers, the engine uses lexical analysis of the SQL statement to construct a sample SQL statement and then fires that statement to retrieve statement metadata.
metadata	The engine always queries prepared statement metadata regardless of the database driver used.
sample	The engine always uses lexical analysis of the SQL statement to construct a sample SQL statement, and then fires that statement to retrieve statement metadata.

10.4.8. Engine Settings related to Concurrency and Threading

Preserving the order of events delivered to listeners

In multithreaded environments, this setting controls whether dispatches of statement result events to listeners preserve the ordering in which a statement processes events. By default the engine guarantees that it delivers a statement's result events to statement listeners in the order in which the result is generated. This behavior can be turned off via configuration as below.

The next code snippet shows how to control this feature:

```
Configuration config = new Configuration();
```

```
config.getEngineDefaults().getThreading().setListenerDispatchPreserveOrder(false);
engine = EPServiceProviderManager.getDefaultProvider(config);
```

And the XML configuration file can also control this feature by adding the following elements:

```
<engine-settings>
  <defaults>
    <threading>
      <listener-dispatch preserve-order="true" timeout-msec="1000" locking="spin"/>
    </threading>
  </defaults>
</engine-settings>
```

As discussed, by default the engine can temporarily block another processing thread when delivering result events to listeners in order to preserve the order in which results are delivered to a given statement. The maximum time the engine blocks a thread can also be configured, and by default is set to 1 second.

As such delivery locks are typically held for a very short amount of time, the default blocking technique employs a spin lock (There are two techniques for implementing blocking; having the operating system suspend the thread until it is awakened later or using spin locks). While spin locks are CPU-intensive and appear inefficient, a spin lock can be more efficient than suspending the thread and subsequently waking it up, especially if the lock in question is held for a very short time. That is because there is significant overhead to suspending and rescheduling a thread.

The locking technique can be changed to use a blocking strategy that suspends the thread, by means of setting the locking property to 'suspend'.

Preserving the order of events for insert-into streams

In multithreaded environments, this setting controls whether statements producing events for other statements via insert-into preserve the order of delivery within the producing and consuming statements, allowing statements that consume other statement's events to behave deterministic in multithreaded applications, if the consuming statement requires such determinism. By default, the engine makes this guarantee (the setting is on).

Take, for example, an application where a single statement (S1) inserts events into a stream that another statement (S2) further evaluates. A multithreaded application may have multiple threads processing events into statement S1. As statement S1 produces events for consumption by statement S2, such results may need to be delivered in the exact order produced as the consuming statement may rely on the order received. For example, if the first statement counts the number of events, the second statement may employ a pattern that inspects counts and thus expect the counts posted by statement S1 to continuously increase by 1 even though multiple threads process events.

The engine may need to block a thread such that order of delivery is maintained, and statements that require order (such as pattern detection, previous and prior functions) receive a deterministic order of events. The settings available control the blocking technique and parameters. As described in the section immediately prior, the default blocking technique employs spin locks per statement inserting events for consumption, as the locks in questions are typically held a very short time. The 'suspend' blocking technique can be configured and a timeout value can also defined.

The XML configuration file may change settings via the following elements:

```
<engine-settings>
  <defaults>
    <threading>
      <insert-into-dispatch preserve-order="true" timeout-msec="100" locking="spin"/>
    </threading>
  </defaults>
```

```
</engine-settings>
```

Internal Timer Settings

This option can be used to disable the internal timer thread and such have the application supply external time events, as well as to set a timer resolution.

The next code snippet shows how to disable the internal timer thread via the configuration API:

```
Configuration config = new Configuration();
  config.getEngineDefaults().getThreading().setInternalTimerEnabled(false);
```

This snippet of XML configuration leaves the internal timer enabled (the default) and sets a resolution of 200 milliseconds (the default is 100 milliseconds):

```
<engine-settings>
  <defaults>
    <threading>
      <internal-timer enabled="true" msec-resolution="200"/>
    </threading>
  </defaults>
</engine-settings>
```

We recommend that when disabling the internal timer, applications send an external timer event setting the start time before creating statements, such that statement start time is well-defined.

10.4.9. Engine Settings related to Event Metadata

Java Class Property Names and Case Sensitivity

As discussed in Section 10.4.1.6, “Case Sensitivity and Property Names” this setting controls case sensitivity for Java event class properties of all Java classes as a default, rather then at a class level.

The next code snippet shows how to control this feature via the API:

```
Configuration config = new Configuration();
config.getEngineDefaults().getEventMeta().setClassPropertyResolutionStyle(
    Configuration.PropertyResolutionStyle.CASE_INSENSITIVE);
```

10.4.10. Engine Settings related to View Resources

Sharing View Resources between Statements

The engine by default attempts to optimize resource usage and thus re-uses or shares views between statements that declare same views. However, in multi-threaded environments, this can lead to reduced concurrency as locking for shared view resources must take place. Via this setting this behavior can be turned off for higher concurrency in multi-threaded processing.

The next code snippet outlines the API to turn off view resource sharing between statements:

```
Configuration config = new Configuration();
config.getEngineDefaults().getViewResources().setShareViews(false);
```

10.4.11. Engine Settings related to Logging

Execution Path Debug Logging

By default, the engine does not produce debug output for the event processing execution paths even when Log4j or Logger configurations have been set to output debug level logs. To enable debug level logging, set this option in the configuration as well as in your Log4j configuration file.

The API to use to enable debug logging is shown here:

```
Configuration config = new Configuration();
config.getEngineDefaults().getLogging().setEnableExecutionDebug(true);
```

Note: this is a configuration option that applies to all engine instances of a given Java module or VM.

10.4.12. Engine Settings related to Variables

Variable Version Release Interval

This setting controls the length of time that the engine retains variable versions for use by statements that use variables and that execute, within the same statement for the same event, longer then the time interval. By default, the engine retains 15 seconds of variable versions.

For statements that use variables and that execute (in response to a single timer or other event) longer then the time period, the engine returns the current variable version at the time the statement executes, thereby softening the guarantee of consistency of variable values within the long-running statement. Please see Section 4.17.3, “Using Variables” for more information.

The XML configuration for this setting is shown below:

```
<engine-settings>
  <defaults>
    <variables>
      <msec-version-release value="15000"/>
    </variables>
  </defaults>
</engine-settings>
```

10.4.13. Engine Settings related to Stream Selection

Default Statement Stream Selection

Statements can produce both insert stream (new data) and remove stream (old data) results. Remember that insert stream refers to arriving events and new aggregation values, while remove stream refers to events leaving data windows and prior aggregation values. By default, the engine delivers only the insert stream to listeners and observers of a statement.

There are keywords in the `select` clause that instruct the engine to not generate insert stream and/or remove stream results if your application does not need either one of the streams. These keywords are the `istream`, `rstream` and the `irstream` keywords.

By default, the engine only generates insert stream results equivalent to using the optional `istream` keyword in the `select` clause. If you application requires insert and remove stream results for many statements, your application can add the `irstream` keyword to the `select` clause of each statement, or you can set a new default stream selector via this setting.

The XML configuration for this setting is shown below:

```
<engine-settings>
  <defaults>
    <stream-selection>
      <stream-selector value="irstream" />
    </stream-selection>
  </defaults>
</engine-settings>
```

The equivalent code snippet using the configuration API is here:

```
Configuration config = new Configuration();
config.getEngineDefaults().getStreamSelection()
    .setDefaultStreamSelector(StreamSelector.RSTREAM_ISTREAM_BOTH);
```

Chapter 11. Extension and Plug-in

11.1. Overview

Esper can currently be extended by these means:

- User-defined functions - these can be used anywhere where expressions are allowed, please see Section 7.3, “User-Defined Functions”
- Custom-developed Plug-in Views

11.2. Custom View Implementation

Views in Esper are used to derive information from an event stream, and to represent data windows onto an event stream. This chapter describes how to plug-in a new, custom view.

The following steps are required to develop and use a custom view with Esper.

1. Implement a view factory class. View factories are classes that accept and check view parameters and instantiate the appropriate view class.
2. Implement a view class. A view class commonly represents a data window or derives new information from a stream.
3. Configure the view factory class supplying a view namespace and name in the engine configuration file.

The example view factory and view class that are used in this chapter can be found in the test source folder in the package `com.espertech.esper.regression.client` by the name `MyTrendSpotterViewFactory` and `MyTrendSpotterView`.

Views can make use of the following engine services available via `StatementServiceContext`:

- The `SchedulingService` interface allows views to schedule timer callbacks to a view
- The `EventAdapterService` interface allows views to create new event types and event instances of a given type.
- The `StatementStopService` interface allows view to register a callback that the engine invokes to indicate that the view's statement has been stopped

Note that custom views may use engine services and APIs that can be subject to change between major releases. The engine services discussed above and view APIs are considered part of the engine internal public API and are stable. Any changes to such APIs are disclosed through the release change logs and history. Please also consider contributing your custom view to the Esper project team by submitting the view code through the mailing list or via a JIRA issue.

11.2.1. Implementing a View Factory

A view factory class is responsible for the following functions:

- Accept zero, one or more view parameters. Validate and parse the parameters as required.
- Validate that the parameterized view is compatible with its parent view. For example, validate that field names are valid in the event type of the parent view.
- Instantiate the actual view class.
- Provide information about the event type of events posted by the view.

View factory classes simply subclass `com.espertech.esper.view.ViewFactorySupport`:

```
public class MyTrendSpotterViewFactory extends ViewFactorySupport { ...
```

Your view factory class must implement the `setViewParameters` method to accept and parse view parameters. The next code snippet shows an implementation of this method. The code obtains a single field name parameter from the parameter list passed to the method:

```
public class MyTrendSpotterViewFactory extends ViewFactorySupport {
  private String fieldName;
  private EventType eventType;

  public void setViewParameters(ViewFactoryContext viewFactoryContext,
                          List<Object> viewParameters) throws ViewParameterException
  {
    String errorMessage = "'Trend spotter' view require a single field name as a parameter";
    if (viewParameters.size() != 1) {
      throw new ViewParameterException(errorMessage);
    }

    if (!(viewParameters.get(0) instanceof String)) {
      throw new ViewParameterException(errorMessage);
    }

    fieldName = (String) viewParameters.get(0);
  }
  ...
```

After the engine supplied view parameters to the factory, the engine will ask the view to attach to its parent view and validate any field name parameters against the parent view's event type. If the view will be generating events of a different type then the events generated by the parent view, then the view factory can create the new event type in this method:

```
public void attach(EventType parentEventType,
                StatementServiceContext statementServiceContext,
                ViewFactory optionalParentFactory,
                List<ViewFactory> parentViewFactories)
                    throws ViewAttachException {
  String result = PropertyCheckHelper.checkNumeric(parentEventType, fieldName);
  if (result != null) {
    throw new ViewAttachException(result);
  }

  // create new event type
  Map<String, Class> eventTypeMap = new HashMap<String, Class>();
  eventTypeMap.put(PROPERTY_NAME, Long.class);
  eventType = statementServiceContext.getEventAdapterService().
                         createAnonymousMapType(eventTypeMap);
}
```

Finally, the engine asks the view factory to create a view instance:

```
public View makeView(StatementServiceContext statementServiceContext) {
  return new MyTrendSpotterView(statementServiceContext, fieldName);
}
```

11.2.2. Implementing a View

A view class is responsible for:

- The `setParent` method informs the view of the parent view's event type

- The `update` method receives insert streams and remove stream events from its parent view
- The `iterator` method supplies an (optional) iterator to allow an application to pull or request results from an `EPStatement`
- The `cloneView` method must make a configured copy of the view to enable the view to work in a grouping context together with a `std:groupby` parent view

View classes simply subclass `com.espertech.esper.view.ViewSupport`:

```
public class MyTrendSpotterView extends ViewSupport { ...
```

The view class must implement the `setParent(Viewable parent)` method. This is an opportunity for the view to initialize and obtain a fast event property getter for later use to obtain event property values. The next code snippet shows an implementation of this method:

```
public void setParent(Viewable parent) {
  super.setParent(parent);
  if (parent != null)  {
    fieldGetter = parent.getEventType().getGetter(fieldName);
  }
}
```

Your `update` method will be processing incoming (insert stream) and outgoing (remove stream) events, as well as providing incoming and outgoing events to child views. The convention required of your update method implementation is that the view releases any insert stream events which the view generates as semantically-equal remove stream events at a later time. A sample `update` method implementation that computes a number of events in an upward trend is shown below:

```
public final void update(EventBean[] newData, EventBean[] oldData) {
  EventBean oldDataPost = populateMap(trendcount);

  // add data points
  if (newData != null) {
    for (int i = 0; i < newData.length; i++) {
      double dataPoint = ((Number) fieldGetter.get(newData[i])).doubleValue();

      if (lastDataPoint == null) {
        trendcount = 1L;
      }
      else if (lastDataPoint < dataPoint) {
        trendcount++;
      }
      else if (lastDataPoint > dataPoint) {
        trendcount = 0L;
      }
      lastDataPoint = dataPoint;
    }
  }

  if (this.hasViews()) {
    EventBean newDataPost = populateMap(trendcount);
    updateChildren(new EventBean[] {newDataPost}, new EventBean[] {oldDataPost});
  }
}
```

This `update` method must adhere to the following view conventions, to prevent memory leaks and to enable correct behavior within the engine:

- Views must post a remove stream in the form of old data to child views. The remove stream must consist of the same event reference(s) posted as insert stream (new data).

The engine can provide a callback to the view indicating when a statement using the view is stopped. The callback is available to the view via the `com.espertech.esper.view.StatementStopCallback` interface. Your view code must subscribe to the stop callback in order for the engine to invoke the callback:

```
statementContext.getStatementStopService().addSubscriber(this);
```

Please refer to the sample views for a code sample on how to implement `iterator` and `cloneView` methods.

11.2.3. Configuring View Namespace and Name

The view factory class name as well as the view namespace and name for the new view must be added to the engine configuration via the configuration API or using the XML configuration file. The configuration shown below is XML however the same options are available through the configuration API:

```
<esper-configuration
  <plugin-view namespace="custom" name="trendspotter"
      factory-class="com.espertech.esper.regression.view.MyTrendSpotterViewFactory" />
</esper-configuration>
```

The new view is now ready to use in a statement:

```
select * from StockTick.custom:trendspotter(price)
```

Note that the view must implement the `copyView` method to enable the view to work in a grouping context as shown in the next statement:

```
select * from StockTick.std:groupby(symbol).custom:trendspotter(price)
```

11.3. Custom Aggregation Functions

Aggregation functions aggregate event property values or expression results obtained from one or more streams. Examples for built-in aggregation functions are `count(*)`, `sum(price * volume)` or `avg(distinct volume)`.

The optional keyword `distinct` ensures that only distinct (unique) values are aggregated and duplicate values are ignored by the aggregation function. Custom plug-in aggregation functions do not need to implement the logic to handle `distinct` values. This is because when the engine encounters the `distinct` keyword, it eliminates any non-distinct values before passing the value for aggregation to the custom aggregation function.

The following steps are required to develop and use a custom aggregation function with Esper.

1. Implement an aggregation function class.
2. Register the aggregation function class with the engine by supplying a function name, via the engine configuration file or the configuration API.

The code for the example aggregation function as shown in this chapter can be found in the test source folder in the package `com.espertech.esper.regression.client` by the name `MyConcatAggregationFunction`. The sample function simply concatenates string-type values.

11.3.1. Implementing an Aggregation Function

An aggregation function class is responsible for the following functions:

- Implement a `validate` method that validates the value type of the data points that the function must process.

* Implement a `getValueType` method that returns the type of the aggregation value generated by the function. For example, the built-in `count` aggregation function returns `Long.class` as it generates `long` -typed values.
* Implement an `enter` method that the engine invokes to add a data point into the aggregation, when an event enters a data window
* Implement a `leave` method that the engine invokes to remove a data point from the aggregation, when an event leaves a data window
* Implement a `getValue` method that returns the current value of the aggregation.

Aggregation function classes simply subclass `com.espertech.esper.epl.agg.AggregationSupport`:

```
public class MyConcatAggregationFunction extends AggregationSupport { ...
```

The engine generally constructs one instance of the aggregation function class for each time the function is listed in a statement, however the engine may decide to reduce the number of aggregation class instances if it finds equivalent aggregations. The constructor initializes the aggregation function:

```
public class MyConcatAggregationFunction extends AggregationSupport {
  private final static char DELIMITER = ' ';
  private StringBuilder builder;
  private String delimiter;

  public MyConcatAggregationFunction()
  {
    super();
    builder = new StringBuilder();
    delimiter = "";
  }
  ...
```

An aggregation function must provide an implementation of the `validate` method that is passed the result type of the expression within the aggregation function. Since the example concatenation function requires string types, it implements a type check:

```
public void validate(Class childNodeType) {
  if (childNodeType != String.class) {
    throw new IllegalArgumentException("Concat aggregation requires a String parameter");
  }
}
```

The `enter` method adds a datapoint to the current aggregation value. The example `enter` method shown below adds a delimiter and the string value to a string buffer:

```
public void enter(Object value) {
  if (value != null) {
    builder.append(delimiter);
    builder.append(value.toString());
    delimiter = String.valueOf(DELIMITER);
  }
}
```

Conversly, the `leave` method removes a datapoint from the current aggregation value. The example `leave` method removes from the string buffer:

```
public void leave(Object value) {
  if (value != null) {
    builder.delete(0, value.toString().length() + 1);
  }
}
```

In order for the engine to validate the type returned by the aggregation function against the types expected by enclosing expressions, the `getValueType` must return the result type of any values produced by the aggregation function:

```
public Class getValueType() {
  return String.class;
}
```

Finally, the engine obtains the current aggregation value by means of the `getValue` method:

```
public Object getValue() {
  return builder.toString();
}
```

11.3.2. Configuring Aggregation Function Name

The aggregation function class name as well as the function name for the new aggregation function must be added to the engine configuration via the configuration API or using the XML configuration file. The configuration shown below is XML however the same options are available through the configuration API:

```
<esper-configuration
  <plugin-aggregation-function name="concat"
    function-class="com.espertech.esper.regression.client.MyConcatAggregationFunction" />
</esper-configuration>
```

The new aggregation function is now ready to use in a statement:

```
select concat(symbol) from StockTick.win:length(3)
```

11.4. Custom Pattern Guard

Pattern guards are pattern objects that control the lifecycle of the guarded sub-expression, and can filter the events fired by the subexpression.

The following steps are required to develop and use a custom guard object with Esper.

1. Implement a guard factory class, responsible for creating guard object instances.
2. Implement a guard class.
3. Register the guard factory class with the engine by supplying a namespace and name, via the engine configuration file or the configuration API.

The code for the example guard object as shown in this chapter can be found in the test source folder in the package `com.espertech.esper.regression.client` by the name `MyCountToPatternGuardFactory`. The sample guard discussed here counts the number of events occurring up to a maximum number of events, and end the sub-expression when that maximum is reached.

11.4.1. Implementing a Guard Factory

A guard factory class is responsible for the following functions:

- Implement a `setGuardParameters` method that validates guard parameters.
- Implement a `makeGuard` method that constructs a new guard instance.

Guard factory classes subclass `com.espertech.esper.pattern.guard.GuardFactorySupport`:

```
public class MyCountToPatternGuardFactory extends GuardFactorySupport { ...
```

The engine constructs one instance of the guard factory class for each time the guard is listed in a statement.

The guard factory class implements the `setGuardParameters` method that is passed the parameters to the guard as supplied by the statement. It verifies the guard parameters, similar to the code snippet shown next. Our example counter guard takes a single numeric parameter:

```
public void setGuardParameters(List<Object> guardParameters) throws GuardParameterException {
  if (guardParameters.size() != 1) {
    throw new GuardParameterException("Count-to guard takes a single integer parameter");
  }
  if (!(guardParameters.get(0) instanceof Integer)) {
    throw new GuardParameterException("Count-to guard takes a single integer parameter");
  }
  numCountTo = (Integer) guardParameters.get(0);
}
```

The `makeGuard` method is called by the engine to create a new guard instance. The example `makeGuard` method shown below passes the maximum count of events to the guard instance. It also passes a `Quitable` implementation to the guard instance. The guard uses `Quitable` to indicate that the sub-expression contained within must stop (quit) listening for events.

```
public Guard makeGuard(PatternContext context, Quitable quitable,
  Object stateNodeId, Object guardState) {
    return new MyCountToPatternGuard(numCountTo, quitable);
}
```

11.4.2. Implementing a Guard Class

A guard class has the following responsibilities:

- Provides a `startGuard` method that initalizes the guard.
- Provides a `stopGuard` method that stops the guard, called by the engine when the whole pattern is stopped, or the sub-expression containing the guard is stopped.
- Provides an `inspect` method that the pattern engine invokes to determine if the guard lets matching events pass for further evaluation by the containing expression.

Guard classes subclass `com.espertech.esper.pattern.guard.GuardSupport` as shown here:

```
public abstract class GuardSupport implements Guard { ...
```

The engine invokes the guard factory class to construct an instance of the guard class for each new sub-expression instance within a statement.

A guard class must provide an implementation of the `startGuard` method that the pattern engine invokes to start a guard instance. In our example, the method resets the guard's counter to zero:

```
public void startGuard() {
  counter = 0;
}
```

The pattern engine invokes the `inspect` method for each time the sub-expression indicates a new event result. Our example guard needs to count the number of events matched, and quit if the maximum number is reached:

```
public boolean inspect(MatchedEventMap matchEvent) {
  counter++;
```

```
  if (counter > numCountTo) {
    quitable.guardQuit();
    return false;
  }
  return true;
}
```

The `inspect` method returns true for events that pass the guard, and false for events that should not pass the guard.

11.4.3. Configuring Guard Namespace and Name

The guard factory class name as well as the namespace and name for the new guard must be added to the engine configuration via the configuration API or using the XML configuration file. The configuration shown below is XML however the same options are available through the configuration API:

```
<esper-configuration
  <plugin-pattern-guard namespace="myplugin" name="count_to"
      factory-class="com.espertech.esper.regression.client.MyCountToPatternGuardFactory"/>
</esper-configuration>
```

The new guard is now ready to use in a statement. The next pattern statement detects the first 10 MyEvent events:

```
select * from pattern [(every MyEvent) where myplugin:count_to(10)]
```

Note that the `every` keyword was placed within parentheses to ensure the guard controls the repeated matching of events.

11.5. Custom Pattern Observer

Pattern observers are pattern objects that are executed as part of a pattern expression and can observe events or test conditions. Examples for built-in observers are `timer:at` and `timer:interval`. Some suggested uses of observer objects are:

- Implement custom scheduling logic using the engine's own scheduling and timer services
- Test conditions related to prior events matching an expression

The following steps are required to develop and use a custom observer object within pattern statements:

1. Implement an observer factory class, responsible for creating observer object instances.
2. Implement an observer class.
3. Register an observer factory class with the engine by supplying a namespace and name, via the engine configuration file or the configuration API.

The code for the example observer object as shown in this chapter can be found in the test source folder in package `com.espertech.esper.regression.client` by the name `MyFileExistsObserver`. The sample observer discussed here very simply checks if a file exists, using the filename supplied by the pattern statement, and via the `java.io.File` class.

11.5.1. Implementing an Observer Factory

An observer factory class is responsible for the following functions:

- Implement a `setObserverParameters` method that validates observer parameters.
- Implement a `makeObserver` method that constructs a new observer instance.

Observer factory classes subclass `com.espertech.esper.pattern.observer.ObserverFactorySupport`:

```
public class MyFileExistsObserverFactory extends ObserverFactorySupport { ...
```

The engine constructs one instance of the observer factory class for each time the observer is listed in a statement.

The observer factory class implements the `setObserverParameters` method that is passed the parameters to the observer as supplied by the statement. It verifies the observer parameters, similar to the code snippet shown next. Our example file-exists observer takes a single string parameter:

```
public void setObserverParameters(List<Object> observerParameters)
                        throws ObserverParameterException {
  String message = "File exists observer takes a single string filename parameter";
  if (observerParameters.size() != 1) {
    throw new ObserverParameterException(message);
  }
  if (!(observerParameters.get(0) instanceof String)) {
    throw new ObserverParameterException(message);
  }

  filename = observerParameters.get(0).toString();
}
```

The pattern engine calls the `makeObserver` method to create a new observer instance. The example `makeObserver` method shown below passes parameters to the observer instance:

```
public EventObserver makeObserver(PatternContext context,
                        MatchedEventMap beginState,
                        ObserverEventEvaluator observerEventEvaluator,
                        Object stateNodeId,
                        Object observerState) {
  return new MyFileExistsObserver(beginState, observerEventEvaluator, filename);
}
```

The `ObserverEventEvaluator` parameter allows an observer to indicate events, and to indicate change of truth value to permanently false. Use this interface to indicate when your observer has received or witnessed an event, or changed it's truth value to true or permanently false.

The `MatchedEventMap` parameter provides a Map of all matching events for the expression prior to the observer's start. For example, consider a pattern as below:

```
a=MyEvent -> myplugin:my_observer(...)
```

The above pattern tagged the MyEvent instance with the tag "a". The pattern engine starts an instance of `my_observer` when it receives the first MyEvent. The observer can query the `MatchedEventMap` using "a" as a key and obtain the tagged event.

11.5.2. Implementing an Observer Class

An observer class has the following responsibilities:

- Provides a `startObserve` method that starts the observer.
- Provides a `stopObserve` method that stops the observer, called by the engine when the whole pattern is stopped, or the sub-expression containing the observer is stopped.

Observer classes subclass `com.espertech.esper.pattern.observer.ObserverSupport` as shown here:

```
public class MyFileExistsObserver implements EventObserver { ...
```

The engine invokes the observer factory class to construct an instance of the observer class for each new sub-expression instance within a statement.

An observer class must provide an implementation of the `startObserve` method that the pattern engine invokes to start an observer instance. In our example, the observer checks for the presence of a file and indicates the truth value to the remainder of the expression:

```
public void startObserve() {
  File file = new File(filename);
  if (file.exists()) {
    observerEventEvaluator.observerEvaluateTrue(beginState);
  }
  else {
    observerEventEvaluator.observerEvaluateFalse();
  }
}
```

Note the observer passes the `ObserverEventEvaluator` an instance of `MatchedEventMap`. The observer can also create one or more new events and pass these events through the Map to the remaining expressions in the pattern.

11.5.3. Configuring Observer Namespace and Name

The observer factory class name as well as the namespace and name for the new observer must be added to the engine configuration via the configuration API or using the XML configuration file. The configuration shown below is XML however the same options are available through the configuration API:

```
<esper-configuration
  <plugin-pattern-observer namespace="myplugin" name="file_exists"
    factory-class="com.espertech.esper.regression.client.MyFileExistsObserverFactory" />
</esper-configuration>
```

The new observer is now ready to use in a statement. The next pattern statement checks every 10 seconds if the given file exists, and indicates to the listener when the file is found.

```
select * from pattern [every timer:interval(10 sec) -> myplugin:file_exists("myfile.txt")]
```

Chapter 12. Examples, Tutorials, Case Studies

12.1. Examples Overview

This chapter outlines the examples that come with Esper in the `examples/src` folder of the distribution. The code for examples can be found in the `com.espertech.esper.example` packages.

In order to compile and run the samples please follow the below instructions:

1. Make sure Java 1.5 or greater is installed and the JAVA_HOME environment variable is set.
2. Open a console window and change directory to examples/etc.
3. Run "setenv.bat" (Windows) or "setenv.sh" (Unix) to verify your environment settings.
4. Run "compile.bat" (Windows) or "compile.sh" (Unix) to compile the examples.
5. Now you are ready to run the examples. Some examples require mandatory parameters. Further information to running each example can be found in the "examples/etc" folder in file "readme.txt".
6. Modify the logger logging level in the "log4j.xml" configuration file changing DEBUG to INFO on a class or package level to reduce the volume of text output.

JUnit tests exist for the example code. The JUnit test source code for the examples can be found in the `examples/test` folder. To build and run the example JUnit tests, use the Maven 2 goal `test`. The JUnit test source code can also be helpful in understanding the example and in the use of Esper APIs.

12.2. Market Data Feed Monitor

This example processes a raw market data feed. It reports throughput statistics and detects when the data rate of a feed falls off unexpectedly. A rate fall-off may mean that the data is stale and we want to alert when there is a possible problem with the feed.

The classes for this example live in package `com.espertech.esper.example.marketdatafeed`. Run "run_mktdatafeed.bat" (Windows) or "run_mktdatafeed.sh" (Unix) in the `examples/etc` folder to start the market data feed simulator.

12.2.1. Input Events

The input stream consists of 1 event stream that contains 2 simulated market data feeds. Each individual event in the stream indicates the feed that supplies the market data, the security symbol and some pricing information:

```
String symbol;
FeedEnum feed;
double bidPrice;
double askPrice;
```

12.2.2. Computing Rates Per Feed

For the throughput statistics and to detect rapid fall-off we calculate a ticks per second rate for each market data feed.

We can use an EPL statement that specifies a view onto the market data event stream that batches together 1 second of events. We specify the feed and a count of events per feed as output values. To make this data available for further processing, we insert output events into the TicksPerSecond event stream:

```
insert into TicksPerSecond
select feed, count(*) as cnt
  from MarketDataEvent.win:time_batch(1 second)
 group by feed
```

12.2.3. Detecting a Fall-off

We define a rapid fall-off by alerting when the number of ticks per second for any second falls below 75% of the average number of ticks per second over the last 10 seconds.

We can compute the average number of ticks per second over the last 10 seconds simply by using the TicksPerSecond events computed by the prior statement and averaging the last 10 seconds. Next, we compare the current rate with the moving average and filter out any rates that fall below 75% of the average:

```
select feed, avg(cnt) as avgCnt, cnt as feedCnt
  from TicksPerSecond.win:time(10 seconds)
 group by feed
having cnt < avg(cnt) * 0.75
```

12.2.4. Event generator

The simulator generates market data events for 2 feeds, feed A and feed B. The first parameter to the simulator is a number of threads. Each thread sends events for each feed in an endless loop. Note that as the Java VM garbage collection kicks in, the example generates rate drop-offs during such pauses.

The second parameter is a rate drop probability parameter specifies the probability in percent that the simulator drops the rate for a randomly chosen feed to 60% of the target rate for that second. Thus rate fall-off alerts can be generated.

The third parameter defines the number of seconds to run the example.

12.3. JMS Server Shell and Client

12.3.1. Overview

The server shell is a Java Messaging Service (JMS) -based server that listens to messages on a JMS destination, and sends the received events into Esper. The example also demonstrates a Java Management Extension (JMX) MBean that allows remote dynamic statement management. This server has been designed to run with either Tibco (TM) Enterprise Messaging System (Tibco EMS), or with Apache ActiveMQ, controlled by a properties file.

The server shell has been created as an alternative to the EsperIO Spring JMSTemplate adapter. The server shell is a low-latency processor for byte messages. It employs JMS listeners to process message in multiple threads, this model reduces thread context switching for many JMS providers. The server is configurable and has been tested with two JMS providers. It consists of only 10 classes and is thus easy to understand.

The server shell sample comes with a client (server shell client) that sends events into the JMS-based server, and that also creates a statement on the server remotely through a JMX MBean proxy class.

The server shell classes for this example live in package `com.espertech.esper.example.servershell`. Configure the server to point to your JMS provider by changing the properties in the file `servershell_config.properties` in the `etc` folder. Make sure your JMS provider (ActiveMQ or Tibco EMS) is running, then run "run_servershell.bat" (Windows) or "run_servershell.sh" (Unix) to start the JMS server.

Start the server shell process first before starting the client, since the client also demonstrates remote statement management through JMX by attaching to the server process.

The client classes to the server shell can be found in package `com.espertech.esper.example.servershellclient`. The client shares the same configuration file as the server shell. Run "run_servershellclient.bat" (Windows) or "run_servershellclient.sh" (Unix) to start the JMS producer client that includes a JMX client as well.

12.3.2. JMS Messages as Events

The server shell starts a configurable number of JMS `MessageListener` instances that listen to a given JMS destination. The listeners expect a `BytesMessage` that contain a String payload. The payload consists of an IP address and a double-typed duration value separated by a comma.

Each listener extracts the payload of a message, constructs an event object and sends the event into the shared Esper engine instance.

At startup time, the server creates a single EPL statement with the Esper engine that prints out the average duration per IP address for the last 10 seconds of events, and that specifies an output rate of 2 seconds. By running the server and then the client, you can see the output of the averages every 2 seconds.

The server shell client acts as a JMS producer that sends 1000 events with random IP addresses and durations.

12.3.3. JMX for Remote Dynamic Statement Management

The server shell is also a JMX server providing an RMI-based connector. The server shell exposes a JMX MBean that allows remote statement management. The JMX MBean allows to create a statement remotely, and to attach a serializable listener to the statement remotely, and to destroy a statement remotely.

The server shell client, upon startup, obtains a remote instance of the management MBean exposed by the server shell. It creates a statement through the MBean that filters out all durations greater then the value 9.9. After sending 1000 events, the client then destroys the statement remotely on the server.

12.4. Transaction 3-Event Challenge

The classes for this example live in package `com.espertech.esper.example.transaction`. Run "run_txnsim.bat" (Windows) or "run_txnsim.sh" (Unix) to start the transaction simulator. Please see the readme file in the same folder for build instructions and command line parameters.

12.4.1. The Events

The use case involves tracking three components of a transaction. It‘s important that we use at least three components, since some engines have different performance or coding for only two events per transaction. Each component comes to the engine as an event with the following fields:

- Transaction ID
- Time stamp

In addition, we have the following extra fields:

In event A:

- Customer ID

In event C:

- Supplier ID (the ID of the supplier that the order was filled through)

12.4.2. Combined event

We need to take in events A, B and C and produce a single, combined event with the following fields:

- Transaction ID
- Customer ID
- Time stamp from event A
- Time stamp from event B
- Time stamp from event C

What we‘re doing here is matching the transaction IDs on each event, to form an aggregate event. If all these events were in a relational database, this could be done as a simple SQL join… except that with 10,000 events per second, you will need some serious database hardware to do it.

12.4.3. Real time summary data

Further, we need to produce the following:

- Min,Max,Average total latency from the events (difference in time between A and C) over the past 30 minutes.
- Min,Max,Average latency grouped by (a) customer ID and (b) supplier ID. In other words, metrics on the the latency of the orders coming from each customer and going to each supplier.
- Min,Max,Average latency between events A/B (time stamp of B minus A) and B/C (time stamp of C minus B).

12.4.4. Find problems

We need to detect a transaction that did not make it through all three events. In other words, a transaction with events A or B, but not C. Note that, in this case, what we care about is event C. The lack of events A or B could indicate a failure in the event transport and should be ignored. Although the lack of an event C could also be a transport failure, it merits looking into.

12.4.5. Event generator

To make testing easier, standard and to demonstrate how the example works, the example is including an event generator. The generator generates events for a given number of transactions, using the following rules:

- One in 5,000 transactions will skip event A
- One in 1,000 transactions will skip event B
- One in 10,000 transactions will skip event C.
- Transaction identifiers are randomly generated
- Customer and supplier identifiers are randomly chosen from two lists
- The time stamp on each event is based on the system time. Between events A and B as well as B and C, between 0 and 999 is added to the time. So, we have an expected time difference of around 500 milliseconds between each event
- Events are randomly shuffled as described below

To make things harder, we don‘t want transaction events coming in order. This code ensures that they come completely out of order. To do this, we fill in a bucket with events and, when the bucket is full, we shuffle it. The buckets are sized so that some transactions‘ events will be split between buckets. So, you have a fairly randomized flow of events, representing the worst case from a big, distributed infrastructure.

The generator lets you change the size of the bucket (small, medium, large, larger, largerer). The larger the bucket size, the more events potentially come in between two events in a given transaction and so, the more the performance characteristics like buffers, hashes/indexes and other structures are put to the test as the bucket size increases.

12.5. J2EE Self-Service Terminal Management

The example is about a J2EE-based self-service terminal managing system in an airport that gets a lot of events from connected terminals. The event rate is around 500 events per second. Some events indicate abnormal situations such as 'paper low' or 'terminal out of order'. Other events observe activity as customers use a terminal to check in and print boarding tickets.

12.5.1. Events

Each self-service terminal can publish any of the 6 events below.

- Checkin - Indicates a customer started a check-in dialog
- Cancelled - Indicates a customer cancelled a check-in dialog
- Completed - Indicates a customer completed a check-in dialog
- OutOfOrder - Indicates the terminal detected a hardware problem
- LowPaper - Indicates the terminal is low on paper
- Status - Indicates terminal status, published every 1 minute regardless of activity as a terminal heartbeat

All events provide information about the terminal that published the event, and a timestamp. The terminal information is held in a property named "term" and provides a terminal id. Since all events carry similar information, we model each event as a subtype to a base class BaseTerminalEvent, which will provide the terminal information that all events share. This enables us to treat all terminal events polymorphically, that is we can treat derived event types just like their parent event types. This helps simplify our queries.

All terminals publish Status events every 1 minute. In normal cases, the Status events indicate that a terminal is alive and online. The absence of status events may indicate that a terminal went offline for some reason and that may need to be investigated.

12.5.2. Detecting Customer Check-in Issues

A customer may be in the middle of a check-in when the terminal detects a hardware problem or when the network goes down. In that situation we want to alert a team member to help the customer. When the terminal detects a problem, it issues an OutOfOrder event. A pattern can find situations where the terminal indicates out-of-order and the customer is in the middle of the check-in process:

```
select * from pattern [ every a=Checkin ->
      ( OutOfOrder(term.id=a.term.id) and not
          (Cancelled(term.id=a.term.id) or Completed(term.id=a.term.id)) )]
```

12.5.3. Absence of Status Events

Since Status events arrive in regular intervals of 60 seconds, we can make us of temporal pattern matching using timer to find events that didn't arrive. We can use the every operator and timer:interval() to repeat an action every 60 seconds. Then we combine this with a not operator to check for absence of Status events. A 65 second interval during which we look for Status events allows 5 seconds to account for a possible delay in transmission or processing:

```
select 'terminal 1 is offline' from pattern
  [every timer:interval(60 sec) -> (timer:interval(65 sec) and not Status(term.id = 'T1'))]
output first every 5 minutes
```

12.5.4. Activity Summary Data

By presenting statistical information about terminal activity to our staff in real-time we enable them to monitor the system and spot problems. The next example query simply gives us a count per event type every 1 minute. We could further use this data, available through the CountPerType event stream, to join and compare against a recorded usage pattern, or to just summarize activity in real-time.

```
insert into CountPerType
select type, count(*) as countPerType
from BaseTerminalEvent.win:time(10 minutes)
group by type
output all every 1 minutes
```

12.5.5. Sample Application for J2EE Application Server

The example code in the distribution package implements a message-driven enterprise java bean (MDB EJB). We used an MDB as a convenient place for processing incoming events via a JMS message queue or topic. The example uses 2 JMS queues: One queue to receive events published by terminals, and a second queue to indicate situations detected via EPL statement and listener back to a receiving process.

This example has been packaged for deployment into a JBoss Java application server (see http://www.jboss.org) with default deployment configuration. JBoss is an open-source application server available under LGPL license. Of course the choice of application server does not indicate a requirement or preference for the use of Esper in a J2EE container. Other quality J2EE application servers are available and perhaps more suitable to run this example or a similar application.

The complete example code can be found in the "examples/terminalsvc" folder of the distribution. The Java package name is com.espertech.esper.example.terminalsvc.

Running the Example

The pre-build EAR file contains the MDB for deployment to a JBoss application server with default deployment options. The JBoss default configuration provides 2 queues that this example utilizes: queue/A and queue/

B. The queue/B is used to send events into the MDB, while queue/A is used to indicate back the any data received by listeners to EPL statements.

The application can be deployed by copying the ear file in the "examples/terminalsvc/terminalsvc-ear" folder to your JBoss deployment directory located under the JBoss home directory under "server/default/deploy".

The example contains an event simulator and an event receiver that can be invoked from the command line. See the folder "examples/terminalsvc/etc" folder readme file and start scripts for Windows and Unix, and the documentation set for further information on the simulator.

Building the Example

This example requires Maven 2 to build. To build the example, change directory to the folder "examples/terminalsvc" and type "mvn package". The instructions have been tested with JBoss AS 4.0.4.GA and Maven 2.0.4.

The Maven build packages the EAR file for deployment to a JBoss application server with default deployment options.

Running the Event Simulator and Receiver

The example also contains an event simulator that generates meaningful events. The simulator can be run from the directory "examples/terminalsvc/etc" via the command "run_terminalsvc_sender.bat" (Windows) and "run_terminalsvc_sender.sh" (Linux). The event simulator generates a batch of at least 200 events every 1 second. Randomly, with a chance of 1 in 10 for each batch of events, the simulator generates either an OutOfOrder or a LowPaper event for a random terminal. Each batch the simulator generates 100 random terminal ids and generates a Checkin event for each. It then generates either a Cancelled or a Completed event for each. With a chance of 1 in 1000, it generates an OutOfOrder event instead of the Cancelled or Completed event for a terminal.

The event receiver listens to the MDB-outcoming queue for alerts and prints these out to console. The receiver can be run from the directory "examples/terminalsvc/etc" via the command "run_terminalsvc_receiver.bat" (Windows) and "run_terminalsvc_receiver.sh" (Linux).

12.6. Assets Moving Across Zones - An RFID Example

This example out of the RFID domain processes location report events. Each location report event indicates an asset id and the current zone of the asset. The example solves the problem that when a given set of assets is not moving together from zone to zone, then an alert must be fired.

Each asset group is tracked by 2 statements. The two statements to track a single asset group consisting of assets identified by asset ids {1, 2, 3} are as follows:

```
insert into CountZone_G1
select 1 as groupId, zone, count(*) as cnt
from LocationReport(assetId in 1, 2, 3).std:unique(assetId)
group by zone

select Part.zone from pattern [
  every Part=CountZone_G1(cnt in (1,2)) ->
    (timer:interval(10 sec)  and not CountZone_G1(zone=Part.zone, cnt in (0,3)))]
```

The classes for this example can be found in package `com.espertech.esper.example.rfid`.

This example provides a Swing-based GUI that can be run from the command line. The GUI allows drag-and-drop of three RFID tags that form one asset group from zone to zone. Each time you move an asset across the screen the example sends an event into the engine indicating the asset id and current zone. The example detects if within 10 seconds the three assets do not join each other within the same zone, but stay split across zones. Run "run_rfid_swing.bat" (Windows) or "run_rfid_swing.sh" (Unix) to start the example's Swing GUI.

The example also provides a simulator that can be run from the command line. The simulator generates a number of asset groups as specified by a command line argument and starts a number of threads as specified by a command line argument to send location report events into the engine. Run "run_rfid_sim.bat" (Windows) or "run_rfid_sim.sh" (Unix) to start the RFID location report event simulator. Please see the readme file in the same folder for build instructions and command line parameters.

12.7. AutoID RFID Reader generating XML documents

In this example an array of RFID readers sense RFID tags as pallets are coming within the range of one of the readers. A reader generates XML documents with observation information such as reader sensor ID, observation time and tags observed. A statement computes the total number of tags per reader sensor ID within the last 60 seconds.

This example demonstrates how XML documents unmarshalled to `org.w3c.dom.Node` DOM document nodes can natively be processed by the engine without requiring Java object event representations. The example uses an XPath expression for an event property counting the number of tags observed by a sensor. The XML documents follow the AutoID (`http://www.autoid.org/`) organization standard.

The classes for this example can be found in package `com.espertech.esper.example.autoid`. As events are XML documents with no Java object representation, the example does not have event classes.

A simulator that can be run from the command line is also available for this example. The simulator generates a number of XML documents as specified by a command line argument and prints out the totals per sensor. Run "run_autoid.bat" (Windows) or "run_autoid.sh" (Unix) to start the autoid simulator. Please see the readme file in the same folder for build instructions and command line parameters.

The code snippet below shows the simple statement to compute the total number of tags per sensor. The statement is created by class `com.espertech.esper.example.autoid.RFIDTagsPerSensorStmt`.

```
select ID as sensorId, sum(countTags) as numTagsPerSensor
from AutoIdRFIDExample.win:time(60 seconds)
where Observation[0].Command = 'READ_PALLET_TAGS_ONLY'
group by ID
```

12.8. StockTicker

The StockTicker example comes from the stock trading domain. The example creates event patterns to filter stock tick events based on price and symbol. When a stock tick event is encountered that falls outside the lower or upper price limit, the example simply displays that stock tick event. The price range itself is dynamically created and changed. This is accomplished by an event patterns that searches for another event class, the price limit event.

The classes `com.espertech.esper.example.stockticker.event.StockTick` and `PriceLimit` represent our events. The event patterns are created by the class `com.espertech.esper.example.stockticker.monitor.StockTickerMonitor`.

Summary:

- Good example to learn the API and get started with event patterns
- Dynamically creates and removes event patterns based on price limit events received
- Simple, highly-performant filter expressions for event properties in the stock tick event such as symbol and price

12.9. MatchMaker

In the MatchMaker example every mobile user has an X and Y location, a set of properties (gender, hair color, age range) and a set of preferences (one for each property) to match. The task of the event patterns created by this example is to detect mobile users that are within proximity given a certain range, and for which the properties match preferences.

The event class representing mobile users is `com.espertech.esper.example.matchmaker.event.MobileUserBean`. The `com.espertech.esper.example.matchmaker.monitor.MatchMakingMonitor` class contains the patterns for detecing matches.

Summary:

- Dynamically creates and removes event patterns based on mobile user events received
- Uses range matching for X and Y properties of mobile user events

12.10. QualityOfService

This example develops some code for measuring quality-of-service levels such as for a service-level agreement (SLA). A SLA is a contract between 2 parties that defines service constraints such as maximum latency for service operations or error rates.

The example measures and monitors operation latency and error counts per customer and operation. When one of our operations oversteps these constraints, we want to be alerted right away. Additionally, we would like to have some monitoring in place that checks the health of our service and provides some information on how the operations are used.

Some of the constraints we need to check are:

- That the latency (time to finish) of some of the operations is always less then X seconds.
- That the latency average is always less then Y seconds over Z operation invocations.

The `com.espertech.esper.example.qos_sla.events.OperationMeasurement` event class with its latency and status properties is the main event used for the SLA analysis. The other event `LatencyLimit` serves to set latency limits on the fly.

The `com.espertech.esper.example.qos_sla.monitor.AverageLatencyMonitor` creates an EPL statement that computes latency statistics per customer and operation for the last 100 events. The `DynaLatencySpikeMonitor` uses an event pattern to listen to spikes in latency with dynamically set limits. The `ErrorRateMonitor` uses the timer `'at'` operator in an event pattern that wakes up periodically and polls the error rate within the last 10 minutes. The `ServiceHealthMonitor` simply alerts when 3 errors occur, and the `SpikeAndErrorMonitor` alerts when a fixed latency is overstepped or an error status is reported.

Summary:

- This example combines event patterns with EPL statements for event stream analysis.
- Shows the use of the timer `'at'` operator and followed-by operator `->` in event patterns
- Outlines basic EPL statements
- Shows how to pull data out of EPL statements rather then subscribing to events a statement publishes

12.11. LinearRoad

The Linear Road example is a very incomplete implementation of the Stream Data Management Benchmark [3] by Standford University.

Linear Road simulates a toll system for the motor vehicle expressways of a large metropolitan area. The main event in this example is a car location report which the class `com.espertech.esper.example.linearroad.CarLocEvent` represents. Currently the event stream joins are performed by JUnit test classes in the `examples/test` folder. See the `com.espertech.esper.example.linearroad.TestAccidentNotify` and the `TestCarSegmentCount` classes. Please consider this a work in progress.

Summary:

- Shows more complex joins between event streams.

12.12. StockTick RSI

The RSI gives you the trend for a stock and for more complete explanation, you can visit the link: http://www.stockcharts.com/education/IndicatorAnalysis/indic_RSI.html.

After a definite number of stock events, or accumulation period, the first RSI is computed. Then for each subsequent stock event, the RSI calculations use the previous period’s Average Gain and Loss to determine the “smoothed RSI”.

Summary:

- Uses a simple event pattern with a filter which feeds a listener that computes the RSI, which publishes events containing the computed RSI.

Chapter 13. Performance

Esper has been highly optimized to handle very high throughput streams with very little latency between event receipt and output result posting. It is also possible to use Esper on a soft-real-time or hard-real-time JVM to maximize predictability even further.

This section describes performance best practices and explains how to assess Esper performance by using our provided performance kit.

13.1. Performance Results

For a complete understanding of those results, consult the next sections.

```
Esper exceeds over 500 000 event/s on a dual CPU 2GHz Intel based hardware,
with engine latency below 3 microseconds average (below 10us with more than
99% predictability) on a VWAP benchmark with 1000 statements registered in the system
- this tops at 70 Mbit/s at 85% CPU usage.

Esper also demonstrates linear scalability from 100 000 to 500 000 event/s on this
hardware, with consistent results accross different statements.

Other tests demonstrate equivalent performance results
(straight through processing, match all, match none, no statement registered,
VWAP with time based window or length based windows).

Tests on a laptop demonstrated about 5x time less performance - that is
between 70 000 event/s and 200 000 event/s - which still gives room for easy
testing on small configuration.
```

13.2. Performance Tips

13.2.1. Understand how to tune your Java virtual machine

Esper runs on a JVM and you need to be familiar with JVM tuning. Key parameters to consider include minimum and maximum heap memory and nursery heap sizes. Statements with time-based or length-based data windows can consume large amounts of memory as their size or length can be large.

For time-based data windows, one needs to be aware that the memory consumed depends on the actual event stream input throughput. Event pattern instances also consume memory, especially when using the "every" keyword in patterns to repeat pattern sub-expressions - which again will depend on the actual event stream input throughput.

13.2.2. Compare Esper to other solutions

If you compare Esper performance to the performance of another solution, you need to ensure that your statements have truly equivalent semantics. The is because between different vendors the event processing language can be seem fairly similar whoever may, for all similarities, produce different results.

For example some vendor solution mandates the use of "bounded streams". The next statement shows one vendor's event processing syntax:

```
// Other (name omitted) vendor solution statement:
```

```
select * from (select * from Market where ticker = 'GOOG') retain 1 event
// The above is NOT an Esper statement
```

The semantically equivalent statement in Esper is:

```
// Esper statement with the same semantics:
select * from MarketData(ticker='$').win:length(1)
```

As an example, a NOT semantically equivalent statement in Esper is:

```
// Esper statement that DOES ***NOT*** HAVE the same semantics
// No length window was used
select * from MarketData(ticker='$')
```

13.2.3. Select the underlying event rather than individual fields

By selecting the underlying event in the select-clause we can reduce load on the engine, since the engine does not need to generate a new output event for each input event.

For example, the following statement returns the underlying event to update listeners:

```
// Better performance
select * from RFIDEvent
```

In comparison, the next statement selects individual properties. This statement requires the engine to generate an output event that contains exactly the required properties:

```
// Less good performance
select assetId, zone, xlocation, ylocation from RFIDEvent
```

13.2.4. Prefer stream-level filtering over post-data-window filtering

Esper stream-level filtering is very well optimized, while filtering via the where-clause post any data windows is not optimized. In very simple statements that don't have data windows this distinction can make a performance difference.

Consider the example below, which performs stream-level filtering:

```
// Better performance : stream-level filtering
select * from MarketData(ticker = 'GOOG')
```

The example below is the equivalent (same semantics) statement and performs post-data-window filtering without a data window. The engine does not optimize statements that filter in the where-clause for the reason that data window views are generally present.

```
// Less good performance : post-data-window filtering
select * from Market where ticker = 'GOOG'
```

Thus this optimization technique applies to statements without any data window.

When a data window is used, the semantics change. Let's look at an example to better understand the difference: In the next statement only GOOG market events enter the length window:

```
select avg(price) from MarketData(ticker = 'GOOG').win:length(100)
```

The above statement computes the average price of GOOG market data events for the last 100 GOOG market data events.

Compare the filter position to a filter in the where clause. The following statement is NOT equivalent as all events enter the data window (not just GOOG events):

```
select avg(price) from Market.win:length(100) where ticker = 'GOOG'
```

The statement above computes the average price of all market data events for the last 100 market data events, and outputs results only for GOOG.

13.2.5. Reduce the use of arithmetic in expressions

Esper does not yet attempt to pre-evaluate arithmetic expressions that produce constant results.

Therefore, a filter expression as below is optimized:

```
// Better performance : no arithmetic
select * from MarketData(price>40)
```

While the engine cannot currently optimize this expression:

```
// Less good performance : with arithmetic
select * from MarketData(price+10>50)
```

13.2.6. Consider using EventPropertyGetter for fast access to event properties

The EventPropertyGetter interface is useful for obtaining an event property value without property name table lookup given an EventBean instance that is of the same event type that the property getter was obtained from.

When compiling a statement, the EPStatement instance lets us know the EventType via the getEventType() method. From the EventType we can obtain EventPropertyGetter instances for named event properties.

To demonstrate, consider the following simple statement:

```
select symbol, avg(price) from Market group by symbol
```

After compiling the statement, obtain the EventType and pass the type to the listener:

```
EPStatement stmt = epService.getEPAdministrator().createEPL(stmtText);
MyGetterUpdateListener listener = new MyGetterUpdateListener(stmt.getEventType());
```

The listener can use the type to obtain fast getters for property values of events for the same type:

```
public class MyGetterUpdateListener implements StatementAwareUpdateListener {
    private final EventPropertyGetter symbolGetter;
    private final EventPropertyGetter avgPriceGetter;

    public MyGetterUpdateListener(EventType eventType) {
        symbolGetter = eventType.getGetter("symbol");
        avgPriceGetter = eventType.getGetter("avg(price)");
    }
```

Last, the update method can invoke the getters to obtain event property values:

```
    public void update(EventBean[] eventBeans, EventBean[] oldBeans, EPStatement epStatement, EPServic
        String symbol = (String) symbolGetter.get(eventBeans[0]);
        long volume = (Long) volumeGetter.get(eventBeans[0]);
        // some more logic here
    }
```

13.2.7. Consider casting the underlying event

When an application requires the value of most or all event properties, it can often be best to simply select the underlying event via wildcard and cast the received events.

Let's look at the sample statement:

```
select * from MarketData(symbol regexp 'E[a-z]')
```

An update listener to the statement may want to cast the received events to the expected underlying event class:

```
    public void update(EventBean[] eventBeans, EventBean[] eventBeans) {
        MarketData md = (MarketData) eventBeans[0].getUnderlying();
        // some more logic here
    }
```

13.2.8. Turn off logging

Since Esper 1.10, even if you don't have a log4j configuration file in place, Esper will make sure to minimize execution path logging overhead. For prior versions, and to reduce logging overhead overall, we recommend the "WARN" log level or the "INFO" log level.

Please see the log4j configuration file in "etc/infoonly_log4j.xml" for example log4j settings.

13.2.9. Disable view sharing

By default, Esper compares streams and views in use with existing statement's streams and views, and then re-uses views to efficiently share resources between statements. The benefit is reduced resources usage, however the potential cost is that in multithreaded applications a shared view may mean excessive locking of multiple processing threads.

Consider disabling view sharing for better threading performance if your application overall uses fewer statements and statements have very similar streams, filters and views.

View sharing can be disabled via XML configuration or API, and the next code snippet shows how, using the API:

```
Configuration config = new Configuration();
config.getEngineDefaults().getViewResources().setShareViews(false);
```

13.2.10. Disable delivery order guarantees

If your application is not a multithreaded application, or you application is not sensitive to the order of delivery of result events to your application listeners, then consider disabling the delivery order guarantees the engine makes towards ordered delivery of results to listeners:

```
Configuration config = new Configuration();
```

```
config.getEngineDefaults().getThreading().setListenerDispatchPreserveOrder(false);
```

If your application is not a multithreaded application, or your application uses the `insert into` clause to make results of one statement available for further consuming statements but does not require ordered delivery of results from producing statements to consuming statements, you may disable delivery order guarantees between statements:

```
Configuration config = new Configuration();
config.getEngineDefaults().getThreading().setInsertIntoDispatchPreserveOrder(false);
```

13.2.11. Performance, JVM, OS and hardware

Performance will also depend on your JVM (Sun HotSpot, BEA JRockit, IBM J9), your operating system and your hardware. A JVM performance index such as specJBB at spec.org [http://www.spec.org] can be used. For memory intensive statement, you may want to consider 64bit architecture that can address more than 2GB or 3GB of memory, although a 64bit JVM usually comes with a slow performance penalty due to more complex pointer address management.

The choice of JVM, OS and hardware depends on a number of factors and therefore a definite suggestion is hard to make. The choice depends on the number of statements, and number of threads. A larger number of threads would benefit of more CPU and cores. If you have very low latency requirements, you should consider getting more GHz per core, and possibly soft real-time JVM to enforce GC determinism at the JVM level, or even consider dedicated hardware such as Azul. If your statements utilize large data windows, more RAM and heap space will be utilized hence you should clearly plan and account for that and possibly consider 64bit architectures or consider EsperHA [http://www.espertech.com/products/].

The number and type of statements is a factor that cannot be generically accounted for. The benchmark kit can help test out some requirements and establish baselines, and for more complex use cases a simulation or proof of concept would certainly works best. EsperTech' experts [http://www.espertech.com/support/services.php] can be available to help write interfaces in a consulting relationship.

13.3. Using the performance kit

13.3.1. How to use the performance kit

The benchmark application is basically an Esper event server build with Esper that listens to remote clients over TCP. Remote clients send MarketData(ticker, price, volume) streams to the event server. The Esper event server is started with 1000 statements of one single kind (unless otherwise written), with one statement per ticker symbol, unless the statement kind does not depend on the symbol. The statement prototype is provided along the results with a '$' instead of the actual ticker symbol value. The Esper event server is entirely multithreaded and can leverage the full power of 32bit or 64bit underlying hardware multi-processor multi-core architecture.

The kit also prints out when starting up the event size and the theoretical maximal throughput you can get on a 100 Mbit/s and 1 Gbit/s network. Keep in mind a 100 Mbit/s network will be overloaded at about 400 000 event/s when using our kit despite the small size of events.

Results are posted on our Wiki page at http://docs.codehaus.org/display/ESPER/Esper+performance. Reported results do not represent best ever obtained results. Reported results may help you better compare Esper to other solutions (for latency, throughput and CPU utilization) and also assess your target hardware and JVMs.

The Esper event server, client and statement prototypes are provided in the source repository `esper/`

`trunk/examples/benchmark/`. Refer to http://xircles.codehaus.org/projects/esper/repo for source access.

A built is provided for convenience (without sources) as an attachment to the Wiki page at http://docs.codehaus.org/pages/viewpageattachments.action?pageId=8356191. It contains Ant script to start client, server in simulation mode and server. For real measurement we advise to start from a shell script (because Ant is pipelining stdout/stderr when you invoke a JVM from Ant - which is costly). Sample scripts are provided for you to edit and customize.

If you use the kit you should:

1. Choose the statement you want to benchmark, add it to `etc/statements.properties` under your own KEY and use the `-mode KEY` when you start the Esper event server.

2. Prepare your runServer.sh/runServer.cmd and runClient.sh/runclient.cmd scripts. You'll need to drop required jar libraries in `lib/`, make sure the classpath is configured in those script to include `build` and `etc`. The required libraries are Esper (any compatible version, we have tested started with Esper 1.7.0) and its dependencies as in the sample below (with Esper 2.0) :

```
# classpath on Unix/Linux (on one single line)
etc:build:lib/esper-2.0.0.jar:lib/commons-logging-1.1.1.jar:lib/cglib-nodep-2.1_3.jar
   :lib/antlr-runtime-3.0.1.jar:lib/log4j-1.2.14.jar
@rem  classpath on Windows (on one single line)
etc;build;lib\esper-2.0.0.jar;lib\commons-logging-1.1.1.jar;lib\cglib-nodep-2.1_3.jar
   ;lib\antlr-runtime-3.0.1.jar;lib\log4j-1.2.14.jar
```

   Note that `./etc` and `./build` have to be in the classpath. At that stage you should also start to set min and max JVM heap. A good start is 1GB as in `-Xms1g -Xmx1g`

3. Write the statement you want to benchmark given that client will send a stream MarketData(String ticker, int volume, double price), add it to `etc/statements.properties` under your own KEY and use the `-mode KEY` when you start the Esper event server. Use `'$'` in the statement to create a prototype. For every symbol, a statement will get registered with all `'$'` replaced by the actual symbol value (f.e. `'GOOG'`)

4. Ensure client and server are using the same `-Desper.benchmark.symbol=1000` value. This sets the number of symbol to use (thus may set the number of statement if you are using a statement prototype, and governs how MarketData event are represented over the network. Basically all events will have the same size over the network to ensure predictability and will be ranging between `S0AA` and `S999A` if you use 1000 as a value here (prefix with S and padded with A up to a fixed length string. Volume and price attributes will be randomized.

5. Establish a performance baseline in simulation mode (without clients). Use the `-rate 1x5000` option to simulate one client (one thread) sending 5000 evt/s. You can ramp up both the number of client simulated thread and their emission rate to maximize CPU utilization. The right number should mimic the client emission rate you will use in the client/server benchmark and should thus be consistent with what your client machine and network will be able to send. On small hardware, having a lot of thread with slow rate will not help getting high throughput in this simulation mode.

6. Do performance runs with client/server mode. Remove the `-rate NxM` option from the runServer script or Ant task. Start the server with `-help` to display the possible server options (listen port, statistics, fan out options etc). On the remote machine, start one or more client. Use `-help` to display the possible client options (remote port, host, emission rate). The client will output the actual number of event it is sending to the server. If the server gets overloaded (or if you turned on `-queue` options on the server) the client will likely not be able to reach its target rate.

   Usually you will get better performance by using server side `-queue -1` option so as to have each client connection handled by a single thread pipeline. If you change to 0 or more, there will be intermediate structures to pass the event stream in an asynchronous fashion. This will increase context switching, although if you are using many clients, or are using the `-sleep xxx` (xxx in milliseconds) to simulate a listener delay you may get better performance.

The most important server side option is `-stat xxx` (xxx in seconds) to print out throughput and latency statistics aggregated over the last xxx seconds (and reset every time). It will produce both internal Esper latency (in nanosecond) and also end to end latency (in millisecond, including network time). If you are measuring end to end latency you should make sure your server and client machine(s) are having the same time with f.e. ntpd with a good enough precision. The stat format is like:

```
---Stats - engine (unit: ns)
  Avg: 2528 #4101107
        0 <    5000:  97.01%  97.01% #3978672
     5000 <   10000:   2.60%  99.62% #106669
    10000 <   15000:   0.35%  99.97% #14337
    15000 <   20000:   0.02%  99.99% #971
    20000 <   25000:   0.00%  99.99% #177
    25000 <   50000:   0.00% 100.00% #89
    50000 <  100000:   0.00% 100.00% #41
   100000 <  500000:   0.00% 100.00% #120
   500000 < 1000000:   0.00% 100.00% #2
  1000000 < 2500000:   0.00% 100.00% #7
  2500000 < 5000000:   0.00% 100.00% #5
  5000000 <    more:   0.00% 100.00% #18
---Stats - endToEnd (unit: ms)
  Avg: -2704829444341073400 #4101609
        0 <       1:  75.01%  75.01% #3076609
        1 <       5:   0.00%  75.01% #0
        5 <      10:   0.00%  75.01% #0
       10 <      50:   0.00%  75.01% #0
       50 <     100:   0.00%  75.01% #0
      100 <     250:   0.00%  75.01% #0
      250 <     500:   0.00%  75.01% #0
      500 <    1000:   0.00%  75.01% #0
     1000 <    more:  24.99% 100.00% #1025000
Throughput 412503 (active 0 pending 0 cnx 4)
```

This one reads as:

```
"Throughput is 412 503 event/s with 4 client connected. No -queue options
was used thus no event is pending at the time the statistics are printed.
Esper latency average is at 2528 ns (that is 2.5 us) for 4 101 107 events
(which means we have 10 seconds stats here). Less than 10us latency
was achieved for 106 669 events that is 99.62%. Latency between 5us
and 10us was achieved for those 2.60% of all the events in the interval."

"End to end latency was ... in this case likely due to client clock difference
we ended up with unusable end to end statistics."
```

Consider the second output paragraph on end-to-end latency:

```
---Stats - endToEnd (unit: ms)
  Avg: 15 #863396
        0 <       1:   0.75%   0.75% #6434
        1 <       5:   0.99%   1.74% #8552
        5 <      10:   2.12%   3.85% #18269
       10 <      50:  91.27%  95.13% #788062
       50 <     100:   0.10%  95.22% #827
      100 <     250:   4.36%  99.58% #37634
      250 <     500:   0.42% 100.00% #3618
      500 <    1000:   0.00% 100.00% #0
     1000 <    more:   0.00% 100.00% #0
```

This would read:

```
"End to end latency average is at 15 milliseconds for the 863 396 events
considered for this statistic report. 95.13% ie 788 062 events were handled
(end to end) below 50ms, and 91.27% were handled between 10ms and 50ms."
```

13.3.2. How we use the performance kit

We use the performance kit to track performance progress across Esper versions, as well as to implement optimizations. You can track our work on the Wiki at http://docs.codehaus.org/display/ESPER/Home

Chapter 14. References

14.1. Reference List

- Luckham, David. 2002. *The Power of Events.* Addison-Wesley.
- The Stanford Rapide (TM) Project. *http://pavg.stanford.edu/rapide*.
- Arasu, Arvind, et.al.. 2004. Linear Road: A Stream Data Management Benchmark, Stanford University *http://www.cs.brown.edu/research/aurora/Linear_Road_Benchmark_Homepage.html*.

Appendix A. Output Reference and Samples

This section specifies the output of a subset of EPL continuous queries, for two purposes: First, to help application developers understand streaming engine output in response to incoming events and in response to time passing. Second, to document and standardize output for EPL queries in a testable and trackable fashion.

The section focuses on a subset of features, namely the time window, aggregation, grouping, and output rate limiting. The section does not currently provide examples for many of the other language features, thus there is no example for other data windows (the time window is used here), joins, sub-selects or named windows etc.

Rather then just describe syntax and output, this section provides detailed examples for each of the types of queries presented. The input for each type of query is always the same set of events, and the same timing. Each event has three properties: symbol, volume and price. The property types are string, long and double, respectively.

The chapters are organized by the type of query: The presence or absence of aggregation functions, as well as the presence or absence of a `group by` clause change statement output as described in Section 3.7.2, “Output for Aggregation and Group-By”.

You will notice that some queries utilize the `order by` clause for sorting output. The reason is that when multiple output rows are produced at once, the output can be easier to read if it is sorted.

With output rate limiting, the engine invokes your listener even if there are no results to indicate when the output condition has been reached. Such is indicated as `(empty result)` in the output result columns.

The output columns show both insert and remove stream events. Insert stream events are delivered as an array of `EventBean` instances to listeners in the `newData` parameter, while remove stream events are delivered to the `oldData` parameter of listeners. Delivery to observers follows similar rules.

A.1. Introduction and Sample Data

For the purpose of illustration and documentation, the example data set demonstrates input and remove streams based on a time window of a 5.5 second interval. The statement utilizing the time window could look as follows:

```
select symbol, volume, price from MarketData.win:time(5.5 sec)
```

We have picked a time window to demonstrate the output for events entering and leaving a data window with an expiration policy. The time window provides a simple expiration policy based on time: if an event resides in the time window more then 5.5 seconds, the engine expires the event from the time window.

The input events and their timing are below. The table should be read, starting from top, as "The time starts at 0.2 seconds. Event E1 arrives at 0.2 seconds with properties [S1, 100, 25]. At 0.8 second event E2 arrives with properties [S2, 5000, 9.0]" and so on.

```
                       Input                                       Output
-----------------------------------------------  ----------------------------------
 Time Symbol  Volume   Price
  0.2
         S1     100    25.0   Event E1 arrives
```

```
  0.8
          S2    5000     9.0   Event E2 arrives
  1.0
  1.2
  1.5
          S1     150    24.0   Event E3 arrives
          S3   10000     1.0   Event E4 arrives
  2.0
  2.1
          S1     155    26.0   Event E5 arrives
  2.2
  2.5
  3.0
  3.2
  3.5
          S3   11000     2.0   Event E6 arrives
  4.0
  4.2
  4.3
          S1     150    22.0   Event E7 arrives
  4.9
          S3   11500     3.0   Event E8 arrives
  5.0
  5.2
  5.7                          Event E1 leaves the time window
  5.9
          S3   10500     1.0   Event E9 arrives
  6.0
  6.2
  6.3                          Event E2 leaves the time window
  7.0                          Event E3 and E4 leave the time window
  7.2
```

The event data set assumes a time window of 5.5 seconds. Thus at time 5.7 seconds the first arriving event (E1) leaves the time window.

The data set as above shows times between 0.2 seconds and 7.2 seconds. Only a couple of time points have been picked for the table to keep the set of time points constant between statements, and thus make the test data and output easier to understand.

A.2. Output for Un-aggregated and Un-grouped Queries

This chapter provides sample output for queries that do not have aggregation functions and do not have a `group by` clause.

A.2.1. No Output Rate Limiting

Without an `output` clause, the engine dispatches to listeners as soon as events arrive, or as soon as time passes such that events leave data windows.

The statement for this sample reads:

```
select irstream symbol, volume, price from MarketData.win:time(5.5 sec)
```

The output is as follows:

```
                       Input                                       Output
                                                          Insert Stream    Remove Stream
-----------------------------------------------  -----------------------------------
 Time Symbol  Volume   Price
  0.2
```

```
         IBM     100    25.0   Event E1 arrives
                                                  [IBM, 100, 25.0]
  0.8
        MSFT    5000     9.0   Event E2 arrives
                                                  [MSFT, 5000, 9.0]
  1.0
  1.2
  1.5
         IBM     150    24.0   Event E3 arrives
                                                  [IBM, 150, 24.0]
         YAH   10000     1.0   Event E4 arrives
                                                  [YAH, 10000, 1.0]
  2.0
  2.1
         IBM     155    26.0   Event E5 arrives
                                                  [IBM, 155, 26.0]
  2.2
  2.5
  3.0
  3.2
  3.5
         YAH   11000     2.0   Event E6 arrives
                                                  [YAH, 11000, 2.0]
  4.0
  4.2
  4.3
         IBM     150    22.0   Event E7 arrives
                                                  [IBM, 150, 22.0]
  4.9
         YAH   11500     3.0   Event E8 arrives
                                                  [YAH, 11500, 3.0]
  5.0
  5.2
  5.7                          Event E1 leaves the time window
                                                                   [IBM, 100, 25.0]
  5.9
         YAH   10500     1.0   Event E9 arrives
                                                  [YAH, 10500, 1.0]
  6.0
  6.2
  6.3                          Event E2 leaves the time window
                                                                   [MSFT, 5000, 9.0]
  7.0                          Event E3 and E4 leave the time window
                                                                   [IBM, 150, 24.0]
                                                                   [YAH, 10000, 1.0]
  7.2
```

A.2.2. Output Rate Limiting - Default

With an `output` clause, the engine dispatches to listeners when the output condition occurs. Here, the output condition is a 1-second time interval. The engine thus outputs every 1 second, starting from the first event, even if there are no new events or no expiring events to output.

The default (no keyword) and the `ALL` keyword result in the same output.

The statement for this sample reads:

```
select irstream symbol, volume, price from MarketData.win:time(5.5 sec)
output every 1 seconds
```

The output is as follows:

```
                       Input                                       Output
                                                  Insert Stream    Remove Stream
-----------------------------------------------   ----------------------------------
```

```
 Time Symbol  Volume   Price
  0.2
        IBM     100    25.0   Event E1 arrives
  0.8
       MSFT    5000     9.0   Event E2 arrives
  1.0
  1.2
                                                [IBM, 100, 25.0]
                                                [MSFT, 5000, 9.0]
  1.5
        IBM     150    24.0   Event E3 arrives
        YAH   10000     1.0   Event E4 arrives
  2.0
  2.1
        IBM     155    26.0   Event E5 arrives
  2.2
                                                [IBM, 150, 24.0]
                                                [YAH, 10000, 1.0]
                                                [IBM, 155, 26.0]
  2.5
  3.0
  3.2
                                                (empty result)     (empty result)
  3.5
        YAH   11000     2.0   Event E6 arrives
  4.0
  4.2
                                                [YAH, 11000, 2.0]
  4.3
        IBM     150    22.0   Event E7 arrives
  4.9
        YAH   11500     3.0   Event E8 arrives
  5.0
  5.2
                                                [IBM, 150, 22.0]
                                                [YAH, 11500, 3.0]
  5.7                         Event E1 leaves the time window
  5.9
        YAH   10500     1.0   Event E9 arrives
  6.0
  6.2
                                                [YAH, 10500, 1.0]  [IBM, 100, 25.0]
  6.3                         Event E2 leaves the time window
  7.0                         Event E3 and E4 leave the time window
  7.2
                                                                   [MSFT, 5000, 9.0]
                                                                   [IBM, 150, 24.0]
                                                                   [YAH, 10000, 1.0]
```

A.2.3. Output Rate Limiting - Last

Using the `LAST` keyword in the `output` clause, the engine dispatches to listeners only the last event of each insert and remove stream.

The statement for this sample reads:

```
select irstream symbol, volume, price from MarketData.win:time(5.5 sec)
output last every 1 seconds
```

The output is as follows:

```
                       Input                                      Output
                                                Insert Stream    Remove Stream
-----------------------------------------------  ----------------------------------
 Time Symbol  Volume   Price
  0.2
```

```
         IBM     100    25.0   Event E1 arrives
 0.8
        MSFT    5000     9.0   Event E2 arrives
 1.0
 1.2
                                                  [MSFT, 5000, 9.0]
 1.5
         IBM     150    24.0   Event E3 arrives
         YAH   10000     1.0   Event E4 arrives
 2.0
 2.1
         IBM     155    26.0   Event E5 arrives
 2.2
                                                  [IBM, 155, 26.0]
 2.5
 3.0
 3.2
                                                  (empty result)     (empty result)
 3.5
         YAH   11000     2.0   Event E6 arrives
 4.0
 4.2
                                                  [YAH, 11000, 2.0]
 4.3
         IBM     150    22.0   Event E7 arrives
 4.9
         YAH   11500     3.0   Event E8 arrives
 5.0
 5.2
                                                  [YAH, 11500, 3.0]
 5.7                           Event E1 leaves the time window
 5.9
         YAH   10500     1.0   Event E9 arrives
 6.0
 6.2
                                                  [YAH, 10500, 1.0]  [IBM, 100, 25.0]
 6.3                           Event E2 leaves the time window
 7.0                           Event E3 and E4 leave the time window
 7.2
                                                                     [YAH, 10000, 1.0]
```

A.2.4. Output Rate Limiting - First

Using the `FIRST` keyword in the `output` clause, the engine dispatches to listeners only the first event of each insert or remove stream, and does not output further events until the output condition is reached.

The statement for this sample reads:

```
select irstream symbol, volume, price from MarketData.win:time(5.5 sec)
output first every 1 seconds
```

The output is as follows:

```
                     Input                                     Output
                                                  Insert Stream    Remove Stream
----------------------------------------------    ----------------------------------
 Time Symbol  Volume   Price
 0.2
         IBM     100    25.0   Event E1 arrives
                                                  [IBM, 100, 25.0]
 0.8
        MSFT    5000     9.0   Event E2 arrives
 1.0
 1.2
 1.5
         IBM     150    24.0   Event E3 arrives
```

```
                                             [IBM, 150, 24.0]
       YAH   10000     1.0   Event E4 arrives
  2.0
  2.1
       IBM     155    26.0   Event E5 arrives
  2.2
  2.5
  3.0
  3.2
                                             (empty result)     (empty result)
  3.5
       YAH   11000     2.0   Event E6 arrives
                                             [YAH, 11000, 2.0]
  4.0
  4.2
  4.3
       IBM     150    22.0   Event E7 arrives
                                             [IBM, 150, 22.0]
  4.9
       YAH   11500     3.0   Event E8 arrives
  5.0
  5.2
  5.7                        Event E1 leaves the time window
                                                                [IBM, 100, 25.0]
  5.9
       YAH   10500     1.0   Event E9 arrives
  6.0
  6.2
  6.3                        Event E2 leaves the time window
                                                                [MSFT, 5000, 9.0]
  7.0                        Event E3 and E4 leave the time window
  7.2
```

A.2.5. Output Rate Limiting - Snapshot

Using the `SNAPSHOT` keyword in the `output` clause, the engine posts data window contents when the output condition is reached.

The statement for this sample reads:

```
select irstream symbol, volume, price from MarketData.win:time(5.5 sec)
output snapshot every 1 seconds
```

The output is as follows:

```
                        Input                                    Output
                                             Insert Stream    Remove Stream
-----------------------------------------------  ---------------------------------
 Time Symbol  Volume   Price
  0.2
       IBM     100    25.0   Event E1 arrives
  0.8
      MSFT    5000     9.0   Event E2 arrives
  1.0
  1.2
                                             [IBM, 100, 25.0]
                                             [MSFT, 5000, 9.0]
  1.5
       IBM     150    24.0   Event E3 arrives
       YAH   10000     1.0   Event E4 arrives
  2.0
  2.1
       IBM     155    26.0   Event E5 arrives
  2.2
                                             [IBM, 100, 25.0]
                                             [MSFT, 5000, 9.0]
```

```
                                               [IBM, 150, 24.0]
                                               [YAH, 10000, 1.0]
                                               [IBM, 155, 26.0]
 2.5
 3.0
 3.2
                                               [IBM, 100, 25.0]
                                               [MSFT, 5000, 9.0]
                                               [IBM, 150, 24.0]
                                               [YAH, 10000, 1.0]
                                               [IBM, 155, 26.0]
 3.5
        YAH   11000     2.0   Event E6 arrives
 4.0
 4.2
                                               [IBM, 100, 25.0]
                                               [MSFT, 5000, 9.0]
                                               [IBM, 150, 24.0]
                                               [YAH, 10000, 1.0]
                                               [IBM, 155, 26.0]
                                               [YAH, 11000, 2.0]
 4.3
        IBM     150    22.0   Event E7 arrives
 4.9
        YAH   11500     3.0   Event E8 arrives
 5.0
 5.2
                                               [IBM, 100, 25.0]
                                               [MSFT, 5000, 9.0]
                                               [IBM, 150, 24.0]
                                               [YAH, 10000, 1.0]
                                               [IBM, 155, 26.0]
                                               [YAH, 11000, 2.0]
                                               [IBM, 150, 22.0]
                                               [YAH, 11500, 3.0]
 5.7                          Event E1 leaves the time window
 5.9
        YAH   10500     1.0   Event E9 arrives
 6.0
 6.2
                                               [MSFT, 5000, 9.0]
                                               [IBM, 150, 24.0]
                                               [YAH, 10000, 1.0]
                                               [IBM, 155, 26.0]
                                               [YAH, 11000, 2.0]
                                               [IBM, 150, 22.0]
                                               [YAH, 11500, 3.0]
                                               [YAH, 10500, 1.0]
 6.3                          Event E2 leaves the time window
 7.0                          Event E3 and E4 leave the time window
 7.2
                                               [IBM, 155, 26.0]
                                               [YAH, 11000, 2.0]
                                               [IBM, 150, 22.0]
                                               [YAH, 11500, 3.0]
                                               [YAH, 10500, 1.0]
```

A.3. Output for Fully-aggregated and Un-grouped Queries

This chapter provides sample output for queries that have aggregation functions, and that do not have a `group by` clause, and in which all event properties are under aggregation.

A.3.1. No Output Rate Limiting

The statement for this sample reads:

```
select irstream sum(price) from MarketData.win:time(5.5 sec)
```

The output is as follows:

```
                       Input                                   Output
                                                    Insert Stream    Remove Stream
-----------------------------------------------  ----------------------------------
 Time Symbol  Volume   Price
  0.2
        IBM      100    25.0   Event E1 arrives
                                                    [25.0]            [null]
  0.8
       MSFT     5000     9.0   Event E2 arrives
                                                    [34.0]            [25.0]
  1.0
  1.2
  1.5
        IBM      150    24.0   Event E3 arrives
                                                    [58.0]            [34.0]
        YAH    10000     1.0   Event E4 arrives
                                                    [59.0]            [58.0]
  2.0
  2.1
        IBM      155    26.0   Event E5 arrives
                                                    [85.0]            [59.0]
  2.2
  2.5
  3.0
  3.2
  3.5
        YAH    11000     2.0   Event E6 arrives
                                                    [87.0]            [85.0]
  4.0
  4.2
  4.3
        IBM      150    22.0   Event E7 arrives
                                                    [109.0]           [87.0]
  4.9
        YAH    11500     3.0   Event E8 arrives
                                                    [112.0]           [109.0]
  5.0
  5.2
  5.7                          Event E1 leaves the time window
                                                    [87.0]            [112.0]
  5.9
        YAH    10500     1.0   Event E9 arrives
                                                    [88.0]            [87.0]
  6.0
  6.2
  6.3                          Event E2 leaves the time window
                                                    [79.0]            [88.0]
  7.0                          Event E3 and E4 leave the time window
                                                    [54.0]            [79.0]
  7.2
```

A.3.2. Output Rate Limiting - Default

Output occurs when the output condition is reached after each 1-second time interval. For each event arriving, the new aggregation value is output as part of the insert stream. As part of the remove stream, the prior aggregation value is output. This is useful for getting a delta-change for each event or group. If there is a `having` clause, the filter expression applies to each row.

Here also the default (no keyword) and the `ALL` keyword result in the same output.

The statement for this sample reads:

```
select irstream sum(price) from MarketData.win:time(5.5 sec)
output every 1 seconds
```

The output is as follows:

```
                       Input                                    Output
                                                 Insert Stream    Remove Stream
-----------------------------------------------  ----------------------------------
 Time Symbol  Volume   Price
  0.2
        IBM     100    25.0   Event E1 arrives
  0.8
       MSFT    5000     9.0   Event E2 arrives
  1.0
  1.2
                                                 [25.0]             [null]
                                                 [34.0]             [25.0]
  1.5
        IBM     150    24.0   Event E3 arrives
        YAH   10000     1.0   Event E4 arrives
  2.0
  2.1
        IBM     155    26.0   Event E5 arrives
  2.2
                                                 [58.0]             [34.0]
                                                 [59.0]             [58.0]
                                                 [85.0]             [59.0]
  2.5
  3.0
  3.2
                                                 [85.0]             [85.0]
  3.5
        YAH   11000     2.0   Event E6 arrives
  4.0
  4.2
                                                 [87.0]             [85.0]
  4.3
        IBM     150    22.0   Event E7 arrives
  4.9
        YAH   11500     3.0   Event E8 arrives
  5.0
  5.2
                                                 [109.0]            [87.0]
                                                 [112.0]            [109.0]
  5.7                         Event E1 leaves the time window
  5.9
        YAH   10500     1.0   Event E9 arrives
  6.0
  6.2
                                                 [87.0]             [112.0]
                                                 [88.0]             [87.0]
  6.3                         Event E2 leaves the time window
  7.0                         Event E3 and E4 leave the time window
  7.2
                                                 [79.0]             [88.0]
                                                 [54.0]             [79.0]
```

A.3.3. Output Rate Limiting - Last

With the LAST keyword, the insert stream carries one event that holds the last aggregation value, and the remove stream carries the prior aggregation value.

The statement for this sample reads:

```
select irstream sum(price) from MarketData.win:time(5.5 sec)
output last every 1 seconds
```

The output is as follows:

```
                       Input                                     Output
                                                 Insert Stream    Remove Stream
-----------------------------------------------  ----------------------------------
 Time Symbol  Volume   Price
  0.2
         IBM     100    25.0   Event E1 arrives
  0.8
        MSFT    5000     9.0   Event E2 arrives
  1.0
  1.2
                                                 [34.0]            [null]
  1.5
         IBM     150    24.0   Event E3 arrives
         YAH   10000     1.0   Event E4 arrives
  2.0
  2.1
         IBM     155    26.0   Event E5 arrives
  2.2
                                                 [85.0]            [34.0]
  2.5
  3.0
  3.2
                                                 [85.0]            [85.0]
  3.5
         YAH   11000     2.0   Event E6 arrives
  4.0
  4.2
                                                 [87.0]            [85.0]
  4.3
         IBM     150    22.0   Event E7 arrives
  4.9
         YAH   11500     3.0   Event E8 arrives
  5.0
  5.2
                                                 [112.0]           [87.0]
  5.7                          Event E1 leaves the time window
  5.9
         YAH   10500     1.0   Event E9 arrives
  6.0
  6.2
                                                 [88.0]            [112.0]
  6.3                          Event E2 leaves the time window
  7.0                          Event E3 and E4 leave the time window
  7.2
                                                 [54.0]            [88.0]
```

A.3.4. Output Rate Limiting - First

The statement for this sample reads:

```
select irstream sum(price) from MarketData.win:time(5.5 sec)
output first every 1 seconds
```

The output is as follows:

```
                       Input                                     Output
                                                 Insert Stream    Remove Stream
-----------------------------------------------  ----------------------------------
 Time Symbol  Volume   Price
  0.2
         IBM     100    25.0   Event E1 arrives
                                                 [25.0]            [null]
  0.8
        MSFT    5000     9.0   Event E2 arrives
```

```
 1.0
 1.2
 1.5
        IBM     150    24.0   Event E3 arrives
                                                 [58.0]            [34.0]
        YAH   10000     1.0   Event E4 arrives
 2.0
 2.1
        IBM     155    26.0   Event E5 arrives
 2.2
 2.5
 3.0
 3.2
                                                 [85.0]            [85.0]
 3.5
        YAH   11000     2.0   Event E6 arrives
                                                 [87.0]            [85.0]
 4.0
 4.2
 4.3
        IBM     150    22.0   Event E7 arrives
                                                 [109.0]           [87.0]
 4.9
        YAH   11500     3.0   Event E8 arrives
 5.0
 5.2
 5.7                          Event E1 leaves the time window
                                                 [87.0]            [112.0]
 5.9
        YAH   10500     1.0   Event E9 arrives
 6.0
 6.2
 6.3                          Event E2 leaves the time window
                                                 [79.0]            [88.0]
 7.0                          Event E3 and E4 leave the time window
 7.2
```

A.3.5. Output Rate Limiting - Snapshot

The statement for this sample reads:

```
select irstream sum(price) from MarketData.win:time(5.5 sec)
output snapshot every 1 seconds
```

The output is as follows:

```
                       Input                                     Output
                                                 Insert Stream    Remove Stream
-----------------------------------------------  ----------------------------------
 Time Symbol  Volume   Price
 0.2
        IBM     100    25.0   Event E1 arrives
 0.8
       MSFT    5000     9.0   Event E2 arrives
 1.0
 1.2
                                                 [34.0]
 1.5
        IBM     150    24.0   Event E3 arrives
        YAH   10000     1.0   Event E4 arrives
 2.0
 2.1
        IBM     155    26.0   Event E5 arrives
 2.2
                                                 [85.0]
 2.5
 3.0
```

```
  3.2
                                                [85.0]
  3.5
         YAH   11000     2.0   Event E6 arrives
  4.0
  4.2
                                                [87.0]
  4.3
         IBM     150    22.0   Event E7 arrives
  4.9
         YAH   11500     3.0   Event E8 arrives
  5.0
  5.2
                                                [112.0]
  5.7                          Event E1 leaves the time window
  5.9
         YAH   10500     1.0   Event E9 arrives
  6.0
  6.2
                                                [88.0]
  6.3                          Event E2 leaves the time window
  7.0                          Event E3 and E4 leave the time window
  7.2
                                                [54.0]
```

A.4. Output for Aggregated and Un-grouped Queries

This chapter provides sample output for queries that have aggregation functions, and that do not have a `group by` clause, and in which there are event properties that are not under aggregation.

A.4.1. No Output Rate Limiting

The statement for this sample reads:

```
select irstream symbol, sum(price) from MarketData.win:time(5.5 sec)
```

The output is as follows:

```
                       Input                                     Output
                                                Insert Stream    Remove Stream
-----------------------------------------------  ----------------------------------
 Time Symbol  Volume   Price
  0.2
         IBM     100    25.0   Event E1 arrives
                                                [IBM, 25.0]
  0.8
        MSFT    5000     9.0   Event E2 arrives
                                                [MSFT, 34.0]
  1.0
  1.2
  1.5
         IBM     150    24.0   Event E3 arrives
                                                [IBM, 58.0]
         YAH   10000     1.0   Event E4 arrives
                                                [YAH, 59.0]
  2.0
  2.1
         IBM     155    26.0   Event E5 arrives
                                                [IBM, 85.0]
  2.2
  2.5
  3.0
  3.2
```

```
  3.5
         YAH   11000     2.0   Event E6 arrives
                                                  [YAH, 87.0]
  4.0
  4.2
  4.3
         IBM     150    22.0   Event E7 arrives
                                                  [IBM, 109.0]
  4.9
         YAH   11500     3.0   Event E8 arrives
                                                  [YAH, 112.0]
  5.0
  5.2
  5.7                          Event E1 leaves the time window
                                                                   [IBM, 87.0]
  5.9
         YAH   10500     1.0   Event E9 arrives
                                                  [YAH, 88.0]
  6.0
  6.2
  6.3                          Event E2 leaves the time window
                                                                   [MSFT, 79.0]
  7.0                          Event E3 and E4 leave the time window
                                                                   [IBM, 54.0]
                                                                   [YAH, 54.0]
  7.2
```

A.4.2. Output Rate Limiting - Default

The statement for this sample reads:

```
select irstream symbol, sum(price) from MarketData.win:time(5.5 sec)
output every 1 seconds
```

The output is as follows:

```
                       Input                                    Output
                                                  Insert Stream    Remove Stream
-----------------------------------------------  ----------------------------------
 Time Symbol  Volume   Price
  0.2
         IBM     100    25.0   Event E1 arrives
  0.8
        MSFT    5000     9.0   Event E2 arrives
  1.0
  1.2
                                                  [IBM, 25.0]
                                                  [MSFT, 34.0]
  1.5
         IBM     150    24.0   Event E3 arrives
         YAH   10000     1.0   Event E4 arrives
  2.0
  2.1
         IBM     155    26.0   Event E5 arrives
  2.2
                                                  [IBM, 58.0]
                                                  [YAH, 59.0]
                                                  [IBM, 85.0]
  2.5
  3.0
  3.2
                                                  (empty result)     (empty result)
  3.5
         YAH   11000     2.0   Event E6 arrives
  4.0
  4.2
                                                  [YAH, 87.0]
```

```
  4.3
         IBM     150    22.0   Event E7 arrives
  4.9
         YAH   11500     3.0   Event E8 arrives
  5.0
  5.2
                                                 [IBM, 109.0]
                                                 [YAH, 112.0]
  5.7                          Event E1 leaves the time window
  5.9
         YAH   10500     1.0   Event E9 arrives
  6.0
  6.2
                                                 [YAH, 88.0]        [IBM, 87.0]
  6.3                          Event E2 leaves the time window
  7.0                          Event E3 and E4 leave the time window
  7.2
                                                                   [MSFT, 79.0]
                                                                   [IBM, 54.0]
                                                                   [YAH, 54.0]
```

A.4.3. Output Rate Limiting - Last

The statement for this sample reads:

```
select irstream symbol, sum(price) from MarketData.win:time(5.5 sec)
output last every 1 seconds
```

The output is as follows:

```
                       Input                                       Output
                                                 Insert Stream    Remove Stream
-----------------------------------------------  ---------------------------------
 Time Symbol  Volume   Price
  0.2
         IBM     100    25.0   Event E1 arrives
  0.8
        MSFT    5000     9.0   Event E2 arrives
  1.0
  1.2
                                                 [MSFT, 34.0]
  1.5
         IBM     150    24.0   Event E3 arrives
         YAH   10000     1.0   Event E4 arrives
  2.0
  2.1
         IBM     155    26.0   Event E5 arrives
  2.2
                                                 [IBM, 85.0]
  2.5
  3.0
  3.2
                                                 (empty result)     (empty result)
  3.5
         YAH   11000     2.0   Event E6 arrives
  4.0
  4.2
                                                 [YAH, 87.0]
  4.3
         IBM     150    22.0   Event E7 arrives
  4.9
         YAH   11500     3.0   Event E8 arrives
  5.0
  5.2
                                                 [YAH, 112.0]
  5.7                          Event E1 leaves the time window
  5.9
```

```
        YAH   10500     1.0   Event E9 arrives
  6.0
  6.2
                                                  [YAH, 88.0]        [IBM, 87.0]
  6.3                         Event E2 leaves the time window
  7.0                         Event E3 and E4 leave the time window
  7.2
                                                                     [YAH, 54.0]
```

A.4.4. Output Rate Limiting - First

The statement for this sample reads:

```
select symbol, sum(price) from MarketData.win:time(5.5 sec)
output first every 1 seconds
```

The output is as follows:

```
                      Input                                      Output
                                                  Insert Stream    Remove Stream
-----------------------------------------------  ----------------------------------
 Time Symbol  Volume   Price
  0.2
        IBM      100    25.0   Event E1 arrives
                                                  [IBM, 25.0]
  0.8
       MSFT     5000     9.0   Event E2 arrives
  1.0
  1.2
  1.5
        IBM      150    24.0   Event E3 arrives
                                                  [IBM, 58.0]
        YAH    10000     1.0   Event E4 arrives
  2.0
  2.1
        IBM      155    26.0   Event E5 arrives
  2.2
  2.5
  3.0
  3.2
                                                  (empty result)     (empty result)
  3.5
        YAH    11000     2.0   Event E6 arrives
                                                  [YAH, 87.0]
  4.0
  4.2
  4.3
        IBM      150    22.0   Event E7 arrives
                                                  [IBM, 109.0]
  4.9
        YAH    11500     3.0   Event E8 arrives
  5.0
  5.2
  5.7                         Event E1 leaves the time window
                                                                     [IBM, 87.0]
  5.9
        YAH    10500     1.0   Event E9 arrives
  6.0
  6.2
  6.3                         Event E2 leaves the time window
                                                                     [MSFT, 79.0]
  7.0                         Event E3 and E4 leave the time window
  7.2
```

A.4.5. Output Rate Limiting - Snapshot

The statement for this sample reads:

```
select irstream symbol, sum(price) from MarketData.win:time(5.5 sec)
output snapshot every 1 seconds
```

The output is as follows:

```
                       Input                                     Output
                                                 Insert Stream    Remove Stream
-----------------------------------------------  ----------------------------------
 Time Symbol  Volume   Price
  0.2
         IBM     100    25.0   Event E1 arrives
  0.8
        MSFT    5000     9.0   Event E2 arrives
  1.0
  1.2
                                                 [IBM, 34.0]
                                                 [MSFT, 34.0]
  1.5
         IBM     150    24.0   Event E3 arrives
         YAH   10000     1.0   Event E4 arrives
  2.0
  2.1
         IBM     155    26.0   Event E5 arrives
  2.2
                                                 [IBM, 85.0]
                                                 [MSFT, 85.0]
                                                 [IBM, 85.0]
                                                 [YAH, 85.0]
                                                 [IBM, 85.0]
  2.5
  3.0
  3.2
                                                 [IBM, 85.0]
                                                 [MSFT, 85.0]
                                                 [IBM, 85.0]
                                                 [YAH, 85.0]
                                                 [IBM, 85.0]
  3.5
         YAH   11000     2.0   Event E6 arrives
  4.0
  4.2
                                                 [IBM, 87.0]
                                                 [MSFT, 87.0]
                                                 [IBM, 87.0]
                                                 [YAH, 87.0]
                                                 [IBM, 87.0]
                                                 [YAH, 87.0]
  4.3
         IBM     150    22.0   Event E7 arrives
  4.9
         YAH   11500     3.0   Event E8 arrives
  5.0
  5.2
                                                 [IBM, 112.0]
                                                 [MSFT, 112.0]
                                                 [IBM, 112.0]
                                                 [YAH, 112.0]
                                                 [IBM, 112.0]
                                                 [YAH, 112.0]
                                                 [IBM, 112.0]
                                                 [YAH, 112.0]
  5.7                          Event E1 leaves the time window
  5.9
         YAH   10500     1.0   Event E9 arrives
  6.0
```

```
  6.2
                                                [MSFT, 88.0]
                                                [IBM, 88.0]
                                                [YAH, 88.0]
                                                [IBM, 88.0]
                                                [YAH, 88.0]
                                                [IBM, 88.0]
                                                [YAH, 88.0]
                                                [YAH, 88.0]
  6.3                             Event E2 leaves the time window
  7.0                             Event E3 and E4 leave the time window
  7.2
                                                [IBM, 54.0]
                                                [YAH, 54.0]
                                                [IBM, 54.0]
                                                [YAH, 54.0]
                                                [YAH, 54.0]
```

A.5. Output for Fully-aggregated and Grouped Queries

This chapter provides sample output for queries that have aggregation functions, and that have a `group by` clause, and in which all event properties are under aggregation or appear in the `group by` clause.

A.5.1. No Output Rate Limiting

The statement for this sample reads:

```
select irstream symbol, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
order by symbol
```

The output is as follows:

```
                       Input                                         Output
                                                Insert Stream    Remove Stream
-----------------------------------------------  ----------------------------------
 Time Symbol  Volume   Price
  0.2
         IBM     100    25.0   Event E1 arrives
                                                [IBM, 25.0]       [IBM, null]
  0.8
        MSFT    5000     9.0   Event E2 arrives
                                                [MSFT, 9.0]       [MSFT, null]
  1.0
  1.2
  1.5
         IBM     150    24.0   Event E3 arrives
                                                [IBM, 49.0]       [IBM, 25.0]
         YAH   10000     1.0   Event E4 arrives
                                                [YAH, 1.0]        [YAH, null]
  2.0
  2.1
         IBM     155    26.0   Event E5 arrives
                                                [IBM, 75.0]       [IBM, 49.0]
  2.2
  2.5
  3.0
  3.2
  3.5
         YAH   11000     2.0   Event E6 arrives
                                                [YAH, 3.0]        [YAH, 1.0]
  4.0
  4.2
```

```
  4.3
         IBM     150    22.0   Event E7 arrives
                                                  [IBM, 97.0]       [IBM, 75.0]
  4.9
         YAH   11500     3.0   Event E8 arrives
                                                  [YAH, 6.0]        [YAH, 3.0]
  5.0
  5.2
  5.7                          Event E1 leaves the time window
                                                  [IBM, 72.0]       [IBM, 97.0]
  5.9
         YAH   10500     1.0   Event E9 arrives
                                                  [YAH, 7.0]        [YAH, 6.0]
  6.0
  6.2
  6.3                          Event E2 leaves the time window
                                                  [MSFT, null]      [MSFT, 9.0]
  7.0                          Event E3 and E4 leave the time window
                                                  [IBM, 48.0]       [IBM, 72.0]
                                                  [YAH, 6.0]        [YAH, 7.0]
  7.2
```

A.5.2. Output Rate Limiting - Default

The default (no keyword) and the `ALL` keyword do not result in the same output. The default generates an output row per input event, while the `ALL` keyword generates a row for all groups.

The statement for this sample reads:

```
select irstream symbol, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output every 1 seconds
```

The output is as follows:

```
                                                Input                                       Output
                                                  Insert Stream    Remove Stream
-----------------------------------------------  ----------------------------------
 Time Symbol  Volume   Price
  0.2
         IBM     100    25.0   Event E1 arrives
  0.8
        MSFT    5000     9.0   Event E2 arrives
  1.0
  1.2
                                                  [IBM, 25.0]       [IBM, null]
                                                  [MSFT, 9.0]       [MSFT, null]
  1.5
         IBM     150    24.0   Event E3 arrives
         YAH   10000     1.0   Event E4 arrives
  2.0
  2.1
         IBM     155    26.0   Event E5 arrives
  2.2
                                                  [IBM, 49.0]       [IBM, 25.0]
                                                  [YAH, 1.0]        [YAH, null]
                                                  [IBM, 75.0]       [IBM, 49.0]
  2.5
  3.0
  3.2
                                                  (empty result)    (empty result)
  3.5
         YAH   11000     2.0   Event E6 arrives
  4.0
  4.2
                                                  [YAH, 3.0]        [YAH, 1.0]
```

```
  4.3
         IBM     150    22.0   Event E7 arrives
  4.9
         YAH   11500     3.0   Event E8 arrives
  5.0
  5.2
                                                  [IBM, 97.0]        [IBM, 75.0]
                                                  [YAH, 6.0]         [YAH, 3.0]
  5.7                          Event E1 leaves the time window
  5.9
         YAH   10500     1.0   Event E9 arrives
  6.0
  6.2
                                                  [IBM, 72.0]        [IBM, 97.0]
                                                  [YAH, 7.0]         [YAH, 6.0]
  6.3                          Event E2 leaves the time window
  7.0                          Event E3 and E4 leave the time window
  7.2
                                                  [MSFT, null]       [MSFT, 9.0]
                                                  [YAH, 6.0]         [YAH, 7.0]
                                                  [IBM, 48.0]        [IBM, 72.0]
```

A.5.3. Output Rate Limiting - All

The statement for this sample reads:

```
select irstream symbol, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output all every 1 seconds
order by symbol
```

The output is as follows:

```
                       Input                                       Output
                                                  Insert Stream    Remove Stream
-----------------------------------------------  --------------------------------
 Time Symbol  Volume   Price
  0.2
         IBM     100    25.0   Event E1 arrives
  0.8
        MSFT    5000     9.0   Event E2 arrives
  1.0
  1.2
                                                  [IBM, 25.0]        [IBM, null]
                                                  [MSFT, 9.0]        [MSFT, null]
  1.5
         IBM     150    24.0   Event E3 arrives
         YAH   10000     1.0   Event E4 arrives
  2.0
  2.1
         IBM     155    26.0   Event E5 arrives
  2.2
                                                  [IBM, 75.0]        [IBM, 25.0]
                                                  [MSFT, 9.0]        [MSFT, 9.0]
                                                  [YAH, 1.0]         [YAH, null]
  2.5
  3.0
  3.2
                                                  [IBM, 75.0]        [IBM, 75.0]
                                                  [MSFT, 9.0]        [MSFT, 9.0]
                                                  [YAH, 1.0]         [YAH, 1.0]
  3.5
         YAH   11000     2.0   Event E6 arrives
  4.0
  4.2
                                                  [IBM, 75.0]        [IBM, 75.0]
                                                  [MSFT, 9.0]        [MSFT, 9.0]
```

```
                                                  [YAH, 3.0]         [YAH, 1.0]
  4.3
         IBM     150    22.0   Event E7 arrives
  4.9
         YAH   11500     3.0   Event E8 arrives
  5.0
  5.2
                                                  [IBM, 97.0]        [IBM, 75.0]
                                                  [MSFT, 9.0]        [MSFT, 9.0]
                                                  [YAH, 6.0]         [YAH, 3.0]
  5.7                          Event E1 leaves the time window
  5.9
         YAH   10500     1.0   Event E9 arrives
  6.0
  6.2
                                                  [IBM, 72.0]        [IBM, 97.0]
                                                  [MSFT, 9.0]        [MSFT, 9.0]
                                                  [YAH, 7.0]         [YAH, 6.0]
  6.3                          Event E2 leaves the time window
  7.0                          Event E3 and E4 leave the time window
  7.2
                                                  [IBM, 48.0]        [IBM, 72.0]
                                                  [MSFT, null]       [MSFT, 9.0]
                                                  [YAH, 6.0]         [YAH, 7.0]
```

A.5.4. Output Rate Limiting - Last

The statement for this sample reads:

```
select irstream symbol, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output last every 1 seconds
order by symbol
```

The output is as follows:

```
                       Input                                    Output
                                                  Insert Stream    Remove Stream
-----------------------------------------------  ----------------------------------
 Time Symbol  Volume   Price
  0.2
         IBM     100    25.0   Event E1 arrives
  0.8
        MSFT    5000     9.0   Event E2 arrives
  1.0
  1.2
                                                  [IBM, 25.0]        [IBM, null]
                                                  [MSFT, 9.0]        [MSFT, null]
  1.5
         IBM     150    24.0   Event E3 arrives
         YAH   10000     1.0   Event E4 arrives
  2.0
  2.1
         IBM     155    26.0   Event E5 arrives
  2.2
                                                  [IBM, 75.0]        [IBM, 25.0]
                                                  [YAH, 1.0]         [YAH, null]
  2.5
  3.0
  3.2
                                                  (empty result)     (empty result)
  3.5
         YAH   11000     2.0   Event E6 arrives
  4.0
  4.2
                                                  [YAH, 3.0]         [YAH, 1.0]
  4.3
```

```
        IBM     150    22.0   Event E7 arrives
  4.9
        YAH   11500     3.0   Event E8 arrives
  5.0
  5.2
                                                  [IBM, 97.0]       [IBM, 75.0]
                                                  [YAH, 6.0]        [YAH, 3.0]
  5.7                         Event E1 leaves the time window
  5.9
        YAH   10500     1.0   Event E9 arrives
  6.0
  6.2
                                                  [IBM, 72.0]       [IBM, 97.0]
                                                  [YAH, 7.0]        [YAH, 6.0]
  6.3                         Event E2 leaves the time window
  7.0                         Event E3 and E4 leave the time window
  7.2
                                                  [IBM, 48.0]       [IBM, 72.0]
                                                  [MSFT, null]      [MSFT, 9.0]
                                                  [YAH, 6.0]        [YAH, 7.0]
```

A.5.5. Output Rate Limiting - First

The statement for this sample reads:

```
select irstream symbol, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output first every 1 seconds
```

The output is as follows:

```
                       Input                                     Output
                                                  Insert Stream    Remove Stream
-----------------------------------------------   ---------------------------------
 Time Symbol  Volume   Price
  0.2
        IBM     100    25.0   Event E1 arrives
                                                  [IBM, 25.0]       [IBM, null]
  0.8
       MSFT    5000     9.0   Event E2 arrives
  1.0
  1.2
  1.5
        IBM     150    24.0   Event E3 arrives
                                                  [IBM, 49.0]       [IBM, 25.0]
        YAH   10000     1.0   Event E4 arrives
  2.0
  2.1
        IBM     155    26.0   Event E5 arrives
  2.2
  2.5
  3.0
  3.2
                                                  (empty result)    (empty result)
  3.5
        YAH   11000     2.0   Event E6 arrives
                                                  [YAH, 3.0]        [YAH, 1.0]
  4.0
  4.2
  4.3
        IBM     150    22.0   Event E7 arrives
                                                  [IBM, 97.0]       [IBM, 75.0]
  4.9
        YAH   11500     3.0   Event E8 arrives
  5.0
  5.2
```

```
  5.7                         Event E1 leaves the time window
                                                  [IBM, 72.0]       [IBM, 97.0]
  5.9
         YAH   10500     1.0   Event E9 arrives
  6.0
  6.2
  6.3                         Event E2 leaves the time window
                                                  [MSFT, null]      [MSFT, 9.0]
  7.0                         Event E3 and E4 leave the time window
  7.2
```

A.5.6. Output Rate Limiting - Snapshot

The statement for this sample reads:

```
select irstream symbol, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output snapshot every 1 seconds
order by symbol
```

The output is as follows:

```
                       Input                                       Output
                                                  Insert Stream    Remove Stream
-----------------------------------------------  ---------------------------------
 Time Symbol  Volume   Price
  0.2
         IBM     100    25.0   Event E1 arrives
  0.8
        MSFT    5000     9.0   Event E2 arrives
  1.0
  1.2
                                                  [IBM, 25.0]
                                                  [MSFT, 9.0]
  1.5
         IBM     150    24.0   Event E3 arrives
         YAH   10000     1.0   Event E4 arrives
  2.0
  2.1
         IBM     155    26.0   Event E5 arrives
  2.2
                                                  [IBM, 75.0]
                                                  [MSFT, 9.0]
                                                  [YAH, 1.0]
  2.5
  3.0
  3.2
                                                  [IBM, 75.0]
                                                  [MSFT, 9.0]
                                                  [YAH, 1.0]
  3.5
         YAH   11000     2.0   Event E6 arrives
  4.0
  4.2
                                                  [IBM, 75.0]
                                                  [MSFT, 9.0]
                                                  [YAH, 3.0]
  4.3
         IBM     150    22.0   Event E7 arrives
  4.9
         YAH   11500     3.0   Event E8 arrives
  5.0
  5.2
                                                  [IBM, 97.0]
                                                  [MSFT, 9.0]
                                                  [YAH, 6.0]
  5.7                         Event E1 leaves the time window
```

```
  5.9
         YAH   10500     1.0   Event E9 arrives
  6.0
  6.2
                                                  [IBM, 72.0]
                                                  [MSFT, 9.0]
                                                  [YAH, 7.0]
  6.3                          Event E2 leaves the time window
  7.0                          Event E3 and E4 leave the time window
  7.2
                                                  [IBM, 48.0]
                                                  [YAH, 6.0]
```

A.6. Output for Aggregated and Grouped Queries

This chapter provides sample output for queries that have aggregation functions, and that have a `group by` clause, and in which some event properties are not under aggregation.

A.6.1. No Output Rate Limiting

The statement for this sample reads:

```
select irstream symbol, volume, sum(price) from MarketData.win:time(5.5 sec) group by symbol
```

The output is as follows:

```
                       Input                                       Output
                                                  Insert Stream    Remove Stream
-----------------------------------------------   ----------------------------------
 Time Symbol  Volume   Price
  0.2
         IBM     100    25.0   Event E1 arrives
                                                  [IBM, 100, 25.0]
  0.8
        MSFT    5000     9.0   Event E2 arrives
                                                  [MSFT, 5000, 9.0]
  1.0
  1.2
  1.5
         IBM     150    24.0   Event E3 arrives
                                                  [IBM, 150, 49.0]
         YAH   10000     1.0   Event E4 arrives
                                                  [YAH, 10000, 1.0]
  2.0
  2.1
         IBM     155    26.0   Event E5 arrives
                                                  [IBM, 155, 75.0]
  2.2
  2.5
  3.0
  3.2
  3.5
         YAH   11000     2.0   Event E6 arrives
                                                  [YAH, 11000, 3.0]
  4.0
  4.2
  4.3
         IBM     150    22.0   Event E7 arrives
                                                  [IBM, 150, 97.0]
  4.9
         YAH   11500     3.0   Event E8 arrives
                                                  [YAH, 11500, 6.0]
  5.0
```

```
  5.2
  5.7                         Event E1 leaves the time window
                                                                     [IBM, 100, 72.0]
  5.9
         YAH   10500     1.0   Event E9 arrives
                                                  [YAH, 10500, 7.0]
  6.0
  6.2
  6.3                         Event E2 leaves the time window
                                                                     [MSFT, 5000, null]
  7.0                         Event E3 and E4 leave the time window
                                                                     [IBM, 150, 48.0]
                                                                     [YAH, 10000, 6.0]
  7.2
```

A.6.2. Output Rate Limiting - Default

The default (no keyword) and the `ALL` keyword do not result in the same output. The default generates an output row per input event, while the `ALL` keyword generates a row for all groups based on the last new event for each group.

The statement for this sample reads:

```
select irstream symbol, volume, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output every 1 seconds
```

The output is as follows:

```
                       Input                                       Output
                                                  Insert Stream    Remove Stream
-----------------------------------------------   ----------------------------------
 Time Symbol  Volume   Price
  0.2
         IBM     100    25.0   Event E1 arrives
  0.8
        MSFT    5000     9.0   Event E2 arrives
  1.0
  1.2
                                                  [IBM, 100, 25.0]
                                                  [MSFT, 5000, 9.0]
  1.5
         IBM     150    24.0   Event E3 arrives
         YAH   10000     1.0   Event E4 arrives
  2.0
  2.1
         IBM     155    26.0   Event E5 arrives
  2.2
                                                  [IBM, 150, 49.0]
                                                  [YAH, 10000, 1.0]
                                                  [IBM, 155, 75.0]
  2.5
  3.0
  3.2
                                                  (empty result)     (empty result)
  3.5
         YAH   11000     2.0   Event E6 arrives
  4.0
  4.2
                                                  [YAH, 11000, 3.0]
  4.3
         IBM     150    22.0   Event E7 arrives
  4.9
         YAH   11500     3.0   Event E8 arrives
  5.0
  5.2
```

```
                                                  [IBM, 150, 97.0]
                                                  [YAH, 11500, 6.0]
  5.7                         Event E1 leaves the time window
  5.9
         YAH   10500     1.0  Event E9 arrives
  6.0
  6.2
                                                  [YAH, 10500, 7.0]  [IBM, 100, 72.0]
  6.3                         Event E2 leaves the time window
  7.0                         Event E3 and E4 leave the time window
  7.2
                                                                     [MSFT, 5000, null]
                                                                     [IBM, 150, 48.0]
                                                                     [YAH, 10000, 6.0]
```

A.6.3. Output Rate Limiting - All

The statement for this sample reads:

```
select irstream symbol, volume, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output all every 1 seconds
order by symbol
```

The output is as follows:

```
                        Input                                    Output
                                                  Insert Stream    Remove Stream
-----------------------------------------------  ----------------------------------
 Time Symbol  Volume   Price
  0.2
         IBM     100    25.0  Event E1 arrives
  0.8
        MSFT    5000     9.0  Event E2 arrives
  1.0
  1.2
                                                  [IBM, 100, 25.0]
                                                  [MSFT, 5000, 9.0]
  1.5
         IBM     150    24.0  Event E3 arrives
         YAH   10000     1.0  Event E4 arrives
  2.0
  2.1
         IBM     155    26.0  Event E5 arrives
  2.2
                                                  [IBM, 150, 49.0]
                                                  [IBM, 155, 75.0]
                                                  [MSFT, 5000, 9.0]
                                                  [YAH, 10000, 1.0]
  2.5
  3.0
  3.2
                                                  [IBM, 155, 75.0]
                                                  [MSFT, 5000, 9.0]
                                                  [YAH, 10000, 1.0]
  3.5
         YAH   11000     2.0  Event E6 arrives
  4.0
  4.2
                                                  [IBM, 155, 75.0]
                                                  [MSFT, 5000, 9.0]
                                                  [YAH, 11000, 3.0]
  4.3
         IBM     150    22.0  Event E7 arrives
  4.9
         YAH   11500     3.0  Event E8 arrives
  5.0
```

```
  5.2
                                                [IBM, 150, 97.0]
                                                [MSFT, 5000, 9.0]
                                                [YAH, 11500, 6.0]
  5.7                          Event E1 leaves the time window
  5.9
         YAH   10500     1.0   Event E9 arrives
  6.0
  6.2
                                                [IBM, 150, 72.0]   [IBM, 100, 72.0]
                                                [MSFT, 5000, 9.0]
                                                [YAH, 10500, 7.0]
  6.3                          Event E2 leaves the time window
  7.0                          Event E3 and E4 leave the time window
  7.2
                                                [IBM, 150, 48.0]   [IBM, 150, 48.0]
                                                [MSFT, 5000, null] [MSFT, 5000, null]
                                                [YAH, 10500, 6.0]  [YAH, 10000, 6.0]
```

A.6.4. Output Rate Limiting - Last

The statement for this sample reads:

```
select irstream symbol, volume, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output last every 1 seconds
order by symbol
```

The output is as follows:

```
                       Input                                       Output
                                                Insert Stream    Remove Stream
----------------------------------------------  ----------------------------------
 Time Symbol  Volume   Price
  0.2
         IBM     100    25.0   Event E1 arrives
  0.8
        MSFT    5000     9.0   Event E2 arrives
  1.0
  1.2
                                                [IBM, 100, 25.0]
                                                [MSFT, 5000, 9.0]
  1.5
         IBM     150    24.0   Event E3 arrives
         YAH   10000     1.0   Event E4 arrives
  2.0
  2.1
         IBM     155    26.0   Event E5 arrives
  2.2
                                                [IBM, 155, 75.0]
                                                [YAH, 10000, 1.0]
  2.5
  3.0
  3.2
                                                (empty result)     (empty result)
  3.5
         YAH   11000     2.0   Event E6 arrives
  4.0
  4.2
                                                [YAH, 11000, 3.0]
  4.3
         IBM     150    22.0   Event E7 arrives
  4.9
         YAH   11500     3.0   Event E8 arrives
  5.0
  5.2
                                                [IBM, 150, 97.0]
```

```
                                                    [YAH, 11500, 6.0]
  5.7                        Event E1 leaves the time window
  5.9
         YAH   10500     1.0   Event E9 arrives
  6.0
  6.2
                                                    [YAH, 10500, 7.0]  [IBM, 100, 72.0]
  6.3                        Event E2 leaves the time window
  7.0                        Event E3 and E4 leave the time window
  7.2
                                                                       [IBM, 150, 48.0]
                                                                       [MSFT, 5000, null]
                                                                       [YAH, 10000, 6.0]
```

A.6.5. Output Rate Limiting - First

The statement for this sample reads:

```
select irstream symbol, volume, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output first every 1 seconds
```

The output is as follows:

```
                        Input                                      Output
                                                    Insert Stream    Remove Stream
-----------------------------------------------  ----------------------------------
 Time Symbol  Volume   Price
  0.2
         IBM     100    25.0   Event E1 arrives
                                                    [IBM, 100, 25.0]
  0.8
        MSFT    5000     9.0   Event E2 arrives
  1.0
  1.2
  1.5
         IBM     150    24.0   Event E3 arrives
                                                    [IBM, 150, 49.0]
         YAH   10000     1.0   Event E4 arrives
  2.0
  2.1
         IBM     155    26.0   Event E5 arrives
  2.2
  2.5
  3.0
  3.2
                                                    (empty result)     (empty result)
  3.5
         YAH   11000     2.0   Event E6 arrives
                                                    [YAH, 11000, 3.0]
  4.0
  4.2
  4.3
         IBM     150    22.0   Event E7 arrives
                                                    [IBM, 150, 97.0]
  4.9
         YAH   11500     3.0   Event E8 arrives
  5.0
  5.2
  5.7                        Event E1 leaves the time window
                                                                       [IBM, 100, 72.0]
  5.9
         YAH   10500     1.0   Event E9 arrives
  6.0
  6.2
  6.3                        Event E2 leaves the time window
                                                                       [MSFT, 5000, null]
```

```
  7.0                          Event E3 and E4 leave the time window
  7.2
```

A.6.6. Output Rate Limiting - Snapshot

The statement for this sample reads:

```
select irstream symbol, volume, sum(price) from MarketData.win:time(5.5 sec)
group by symbol
output snapshot every 1 seconds
```

The output is as follows:

```
                       Input                                   Output
                                                 Insert Stream    Remove Stream
-----------------------------------------------  ----------------------------------
 Time Symbol  Volume   Price
  0.2
         IBM     100    25.0   Event E1 arrives
  0.8
        MSFT    5000     9.0   Event E2 arrives
  1.0
  1.2
                                                 [IBM, 100, 25.0]
                                                 [MSFT, 5000, 9.0]
  1.5
         IBM     150    24.0   Event E3 arrives
         YAH   10000     1.0   Event E4 arrives
  2.0
  2.1
         IBM     155    26.0   Event E5 arrives
  2.2
                                                 [IBM, 100, 75.0]
                                                 [MSFT, 5000, 9.0]
                                                 [IBM, 150, 75.0]
                                                 [YAH, 10000, 1.0]
                                                 [IBM, 155, 75.0]
  2.5
  3.0
  3.2
                                                 [IBM, 100, 75.0]
                                                 [MSFT, 5000, 9.0]
                                                 [IBM, 150, 75.0]
                                                 [YAH, 10000, 1.0]
                                                 [IBM, 155, 75.0]
  3.5
         YAH   11000     2.0   Event E6 arrives
  4.0
  4.2
                                                 [IBM, 100, 75.0]
                                                 [MSFT, 5000, 9.0]
                                                 [IBM, 150, 75.0]
                                                 [YAH, 10000, 3.0]
                                                 [IBM, 155, 75.0]
                                                 [YAH, 11000, 3.0]
  4.3
         IBM     150    22.0   Event E7 arrives
  4.9
         YAH   11500     3.0   Event E8 arrives
  5.0
  5.2
                                                 [IBM, 100, 97.0]
                                                 [MSFT, 5000, 9.0]
                                                 [IBM, 150, 97.0]
                                                 [YAH, 10000, 6.0]
                                                 [IBM, 155, 97.0]
                                                 [YAH, 11000, 6.0]
```

```
                                                    [IBM, 150, 97.0]
                                                    [YAH, 11500, 6.0]
  5.7                          Event E1 leaves the time window
  5.9
         YAH   10500     1.0   Event E9 arrives
  6.0
  6.2
                                                    [MSFT, 5000, 9.0]
                                                    [IBM, 150, 72.0]
                                                    [YAH, 10000, 7.0]
                                                    [IBM, 155, 72.0]
                                                    [YAH, 11000, 7.0]
                                                    [IBM, 150, 72.0]
                                                    [YAH, 11500, 7.0]
                                                    [YAH, 10500, 7.0]
  6.3                          Event E2 leaves the time window
  7.0                          Event E3 and E4 leave the time window
  7.2
                                                    [IBM, 155, 48.0]
                                                    [YAH, 11000, 6.0]
                                                    [IBM, 150, 48.0]
                                                    [YAH, 11500, 6.0]
                                                    [YAH, 10500, 6.0]
```

Index

Symbols

A

B

C

D

E