Esper Reference

Version 4.10.0

by *Esper Team and EsperTech Inc.* [http://esper.codehaus.org]

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 5.1, "EPL Introduction"* for an introduction to event stream processing via EPL
6. Read *Section 6.1, "Event Pattern Overview"* for an overview over event patterns
7. Read *Section 7.1, "Overview"* for an overview over event patterns using the match recognize syntax.
8. Then glance over the examples *Section 19.1, "Examples Overview"*
9. Finally to test drive Esper performance, read *Chapter 20, 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 than 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 than 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 than 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.
- Apache commons logging is a logging API that works together with LOG4J and other logging APIs. While Apache commons logging is required, the LOG4J log component is not required and can be replaced with SLF4J or other loggers. 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

This section outlines the different means to model and represent events.

Esper uses the term *event type* to describe the type information available for an event representation.

Your application may configure predefined event types at startup time or dynamically add event types at runtime via API or EPL syntax. See *Section 15.4, “Configuration Items”* for startup-time configuration and *Section 14.3.7, “Runtime Configuration”* for the runtime configuration API.

The EPL `create schema` syntax allows declaring an event type at runtime using EPL, see *Section 5.16, “Declaring an Event Type: Create Schema”*.

In *Section 14.6, “Event and Event Type”* we explain how an event type becomes visible in EPL statements and output events delivered by the engine.

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.

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 implementations of the `java.util.Map` interface where each map entry is a propery value.
`Object[] (array of object)`	Object-array events are arrays of objects (type `Object[]`) where each array element is a property value.
`org.w3c.dom.Node`	XML document object model (DOM).
`org.apache.axiom.om.OMDocument or OMElement`	XML - Streaming API for XML (StAX) - Apache Axiom (provided by EsperIO package).
Application classes	Plug-in event representation via the extension API.

Esper provides multiple choices for representing an event. There is no absolute need for you to create new Java classes to represent an event.

Event representations have the following in common:

- All event representations support nested, indexed and mapped properties (aka. property expression), as explained in more detail below. There is no limitation to the nesting level.
- All event representations provide event type metadata. This includes type metadata for nested properties.
- All event representations allow transposing the event itself and parts of all of its property graph into new events. The term transposing refers to selecting the event itself or event properties that are themselves nestable property graphs, and then querying the event's properties or nested property graphs in further statements. The Apache Axiom event representation is an exception and does not currently allow transposing event properties but does allow transposing the event itself.
- The Java object, Map and Object-array representations allow supertypes.

The API behavior for all event representations is the same, with minor exceptions noted in this chapter.

The benefits of multiple event representations are:

- For applications that already have events in one of the supported representations, there is no need to transform events into a Java object before processing.
- Event representations are exchangeable, reducing or eliminating the need to change statements when the event representation changes.
- Event representations are interoperable, allowing all event representations to interoperate in same or different statements.
- The choice makes its possible to consciously trade-off performance, ease-of-use, the ability to evolve and effort needed to import or externalize events and use existing event type metadata.

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]`

Type	Description	Syntax	Example
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 `person.address('home').street[0]`.

You may use any expression as a mapped property key or indexed property index by putting the expression within parenthesis after the mapped or index property name. Please find examples below.

2.2.1. Escape Characters

If your application uses `java.util.Map`, `Object[]` (object-array) or XML to represent events, then event property names may themselves contain the dot ('.') character. The backslash ('\') character can be used to escape dot characters in property names, allowing a property name to contain dot characters.

For example, the EPL as shown below expects a property by name `part1.part2` to exist on event type `MyEvent`:

```
select part1\.part2 from MyEvent
```

Sometimes your event properties may overlap with EPL language keywords or contain spaces or other special characters. In this case you may use the backwards apostrophe ` (aka. back tick) character to escape the property name.

The next example assumes a `Quote` event that has a property by name `order`, while `order` is also a reserved keyword:

```
select `order`, price as `price.for.goods` from Quote
```

When escaping mapped or indexed properties, make sure the back tick character appears outside of the map key or index.

The next EPL selects event properties that have names that contain spaces (e.g. `candidate book`), have the tick special character (e.g. `children's books`), are an indexed property (e.g. `children's books[0]`) and a mapped property that has a reserved keyword as part of the property name (e.g. `book select('isbn')`):

```
select `candidate book` , `children's books`[0], `book select`('isbn') from
 MyEventType
```

2.2.2. Expression as Key or Index Value

The key or index expression must be placed in parenthesis. When using an expression as key for a mapped property, the expression must return a `String`-typed value. When using an expression as index for an indexed property, the expression must return an `int`-typed value.

This example below uses Java classes to illustrate;The same principles apply to all event representations.

Assume a class declares these properties (getters not shown for brevity):

```
public class MyEventType {
  String myMapKey;
  int myIndexValue;
  int myInnerIndexValue;
  Map<String, InnerType> innerTypesMap; // mapped property
  InnerType[] innerTypesArray; // indexed property
}

public class InnerType {
  String name;
  int[] ids;
}
```

A sample EPL statement demonstrating expressions as map keys or indexes is:

```
select innerTypesMap('somekey'),  // returns map value for 'somekey'
  innerTypesMap(myMapKey),         // returns map value for myMapKey value (an
 expression)
  innerTypesArray[1],              // returns array value at index 1
  innerTypesArray(myIndexValue)    // returns array value at index myIndexValue
 (an expression)
  from MyEventType
```

The dot-operator can be used to access methods on the value objects returned by the mapped or indexed properties. By using the dot-operator the syntax follows the chained method invocation described at *Section 8.6, “Dot Operator”*.

A sample EPL statement demonstrating the dot-operator as well as expressions as map keys or indexes is:

```
select innerTypesMap('somekey').ids[1],
 innerTypesMap(myMapKey).getIds(myIndexValue),
 innerTypesArray[1].ids[2],
 innerTypesArray(myIndexValue).getIds(myInnerIndexValue)
 from MyEventType
```

Please note the following limitations:

- The square brackets-syntax for indexed properties does now allow expressions and requires a constant index value.
- When using the dot-operator with mapped or indexed properties that have expressions as map keys or indexes you must follow the chained method invocation syntax.

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: they provide duck typing functionality.

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
```

Above is 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.
- The `typeof` function returns the string type name of a dynamic property.

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

2.4. Fragment and Fragment Type

Sometimes an event can have properties that are itself events. Esper uses the term *fragment* and *fragment type* for such event pieces. The best example is a pattern that matches two or more events and the output event contains the matching events as fragments. In other words, output events can be a composite event that consists of further events, the fragments.

Fragments have the same metadata available as their enclosing composite events. The metadata for enclosing composite events contains information about which properties are fragments, or have a property value that can be represented as a fragment and therefore as an event itself.

Fragments and type metadata can allow your application to navigate composite events without the need for using the Java reflection API and reducing the coupling to the underlying event representation. The API is further described in *Section 14.6, “Event and Event Type”*.

2.5. 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 or an accessor-style and accessor-methods may be defined via configuration.

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 15, Configuration* on options for naming event types represented by Java object event classes. 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 15.4.1.3, “Non-JavaBean and Legacy Java Event Classes”*.

2.5.1. 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 individually accessed by an integer-valued, non-negative index (or subscript).
- *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 `NewEmployeeEvent` 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` can themselves have properties that are nested within them, such as a street name in the `Address` object or a name of the employee in the `Employee` object.

```
public class NewEmployeeEvent {
 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, or a class that implements `Iterable`. An example is the `getAllSubordinates` getter method, which returns an array of Employee but could also return an `Iterable`. 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 (each line is a separate EPL statement):

```
every NewEmployeeEvent(firstName='myName')
every NewEmployeeEvent(address('home').streetName='Park Avenue')
every NewEmployeeEvent(subordinate[0].name='anotherName')
every NewEmployeeEvent(allSubordinates[1].name='thatName')
every         NewEmployeeEvent(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 NewEmployeeEvent
where address('work').streetName = 'Park Ave'
```

2.5.2. Property Names

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`

Method	Property Name	Example
`getqn()`	qn	`select qn from MyEvent`
`gets()`	s	`select s from MyEvent`

2.5.3. Parameterized Types

When your getter methods or accessor fields return a parameterized type, for example `Iterable<MyEventData>` for an indexed property or `Map<String, MyEventData>` for a mapped property, then property expressions may refer to the properties available through the class that is the type parameter.

An example event that has properties that are parameterized types is:

```
public class NewEmployeeEvent {
  public String getName();
  public Iterable<EducationHistory> getEducation();
  public Map<String, Address> getAddresses();
}
```

A sample of valid property expressions for this event is shown next:

```
select name, education, education[0].date, addresses('home').street
from NewEmployeeEvent
```

2.5.4. Setter Methods for Indexed and Mapped Properties

An EPL statement may update indexed or mapped properties of an event, provided the event class exposes the required setter method.

The setter method for indexed properties must be named `set`*PropertyName* and must take two parameters: the `int`-type index and the `Object` type new value.

The setter method for mapped properties must be named `set`*PropertyName* and must take two parameters: the `String`-type map key and the `Object` type new map value.

The following is an example event that features a setter method for the `props` mapped property and for the `array` indexed property:

```
public class MyEvent {
  private Map props = new HashMap();
  private Object[] array = new Object[10];
  public void setProps(String name, Object value) {
    props.put(name, value);
  }

  public void setArray(int index, Object value) {
    array[index] = value;
  }
  // ... also provide regular JavaBean getters and setters for all properties
```

This sample statement updates mapped and indexed property values:

```
update istream MyEventStream set props('key') = 'abc', array[2] = 100
```

2.5.5. Known Limitations

Esper employs byte code generation for fast access to event properties. When byte code generation is unsuccessful, the engine logs a warning and uses Java reflection to obtain property values instead.

A known limitation is that when an interface has an attribute of a particular type and the actual event bean class returns a subclass of that attribute, the engine logs a warning and uses reflection for that property.

2.6. `java.util.Map` Events

2.6.1. Overview

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.

Similar to the Object-array event type, the Map event type takes part in the comprehensive type system that can eliminate the need to use Java classes as event types, thereby making it easier to change types at runtime or generate type information from another source.

A given Map event type can have one or more supertypes that must also be Map event types. All properties available on any of the Map supertypes are available on the type itself. In addition, anywhere within EPL that an event type name of a Map supertype is used, any of its Map subtypes and their subtypes match that expression.

Your application can add properties to an existing Map event type during runtime using the configuration operation `updateMapEventType`. Properties may not be updated or deleted - properties can only be added, and nested properties can be added as well. The runtime configuration also allows removing Map event types and adding them back with new type information.

After your application configures a Map event type by providing a type name, the type name can be used when defining further Map or Object-array event types by specifying the type name as a property type or an array property type.

One-to-Many relationships in Map event types are represented via arrays. A property in a Map event type may be an array of primitive, an array of Java object, an array of Map or an an array of Object-array.

The engine can process `java.util.Map` events via the `sendEvent(Map map, String eventTypeName)` 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.

The engine does not validate Map event property names or values. Your application should ensure that objects passed in as event properties match the `create schema` property names and types, or the configured event type information when using runtime or static configuration.

2.6.2. Map Properties

Map event properties can be of any type. Map event properties that are Java application objects or that are of type `java.util.Map` (or arrays thereof) or that are of type `Object[]` (object-array) (or arrays thereof) 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 and arrays of Maps within Maps.
- Properties that are of type `Object[]` (object-array) allow object-arrays to be nested arbitrarily deep. The nested, indexed, mapped and dynamic property syntax can be used to query nested Maps and object-arrays alike.

In order to use `Map` events, the event type name and property names and types must be made known to the engine via Configuration or `create schema` EPL syntax. Please see examples in *Section 5.16, "Declaring an Event Type: Create Schema"* and *Section 15.4.2, "Events represented by java.util.Map"*.

The code snippet below defines a Map event type, creates a Map event and sends the event into the engine. The sample defines the `CarLocUpdateEvent` event type via runtime configuration interface (`create schema` or static configuration could have been used instead).

```
// Define CarLocUpdateEvent event type (example for runtime-configuration
 interface)
Map<String, Object> def = new HashMap<String, Object>;
def.put("carId", String.class);
def.put("direction", int.class);

epService.getEPAdministrator().getConfiguration().
  addEventType("CarLocUpdateEvent", def);
```

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

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

```
// Create a CarLocUpdateEvent event and send it into the engine for processing
Map<String, Object> event = new HashMap<String, Object>();
event.put("carId", carId);
event.put("direction", direction);

epRuntime.sendEvent(event, "CarLocUpdateEvent");
```

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
```

2.6.3. Map Supertypes

Your `Map` event type may declare one or more supertypes when configuring the type at engine initialization time or at runtime through the administrative interface.

Supertypes of a `Map` event type must also be Map event types. All property names and types of a supertype are also available on a subtype and override such same-name properties of the subtype. In addition, anywhere within EPL that an event type name of a Map supertype is used, any of its Map subtypes also matches that expression (similar to the concept of interface in Java).

This example assumes that the `BaseUpdate` event type has been declared and acts as a supertype to the `AccountUpdate` event type (both Map event types):

```
epService.getEPAdministrator().getConfiguration().
    addEventType("AccountUpdate", accountUpdateDef,
    new String[] {"BaseUpdate"});
```

Your application EPL statements may select `BaseUpdate` events and receive both `BaseUpdate` and `AccountUpdate` events, as well as any other subtypes of `BaseUpdate` and their subtypes.

```
// Receive BaseUpdate and any subtypes including subtypes of subtypes
select * from BaseUpdate
```

Your application Map event type may have multiple supertypes. The multiple inheritance hierarchy between Maps can be arbitrarily deep, however cyclic dependencies are not allowed. If using runtime configuration, supertypes must exist before a subtype to a supertype can be added.

See *Section 15.4.2, “Events represented by java.util.Map”* for more information on configuring Map event types.

2.6.4. Advanced Map Property Types

2.6.4.1. Nested Properties

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

Noteworthy points are:

- JavaBean (POJO) objects can appear as properties in `Map` event types.
- One may represent Map-within-Map and Map-Array within Map (same for object-array) using the name of a previously registered Map (or object-array) event type.
- 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 `UpdatedFieldType` structure as a property. Inside the `UpdatedFieldType` 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);
epService.getEPAdministrator().getConfiguration().
    addEventType("UpdatedFieldType", updatedFieldDef);

Map<String, Object> accountUpdateDef = new HashMap<String, Object>();
accountUpdateDef.put("accountId", long.class);
accountUpdateDef.put("fields", "UpdatedFieldType");
// the latter can also be:  accountUpdateDef.put("fields", updatedFieldDef);

epService.getEPAdministrator().getConfiguration().
    addEventType("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
```

2.6.4.2. One-to-Many Relationships

To model repeated properties within a Map, you may use arrays as properties in a Map. You may use an array of primitive types or an array of JavaBean objects or an array of a previously declared Map or object-array event type.

When using a previously declared Map event type as an array property, the literal `[]` must be appended after the event type name.

This following example defines a Map event type by name `Sale` to hold array properties of the various types. It assumes a `SalesPerson` Java class exists and a Map event type by name `OrderItem` was declared:

```
Map<String, Object> sale = new HashMap<String, Object>();
sale.put("userids", int[].class);
sale.put("salesPersons", SalesPerson[].class);
sale.put("items", "OrderItem[]");   // The property type is the name itself
 appended by []

epService.getEPAdministrator().getConfiguration().
    addEventType("SaleEvent", sale);
```

The three properties that the above example declares are:

- An integer array of user ids.
- An array of `SalesPerson` Java objects.
- An array of Maps for order items.

The next EPL statement is a sample query asking for property values held by arrays:

```
select userids[0], salesPersons[1].name,
    items[1], items[1].price.amount from SaleEvent
```

2.7. Object-array (`Object[]`) Events

2.7.1. Overview

An event can also be represented by an array of objects. Event properties of `Object[]` events are the array element values.

Similar to the Map event type, the object-array event type takes part in the comprehensive type system that can eliminate the need to use Java classes as event types, thereby making it easier to change types at runtime or generate type information from another source.

A given Object-array event type can have only a single supertype that must also be an Object-array event type. All properties available on the Object-array supertype is also available on the type itself. In addition, anywhere within EPL that an event type name of an Object-array supertype is used, any of its Object-array subtypes and their subtypes match that expression.

Your application can add properties to an existing Object-array event type during runtime using the configuration operation `updateObjectArrayEventType`. Properties may not be updated or deleted - properties can only be added, and nested properties can be added as well. The runtime configuration also allows removing Object-array event types and adding them back with new type information.

After your application configures an Object-array event type by providing a type name, the type name can be used when defining further Object-array or Map event types by specifying the type name as a property type or an array property type.

One-to-Many relationships in Object-array event types are represented via arrays. A property in an Object-array event type may be an array of primitive, an array of Java object, an array of Map or an array of Object-array.

The engine can process `Object[]` events via the `sendEvent(Object[] array, String eventTypeName)` method on the `EPRuntime` interface. Entries in the Object array represent event properties.

The engine does not validate Object array length or value types. Your application must ensure that Object array values match the declaration of the event type: The type and position of property values must match property names and types in the same exact order and object array length must match the number of properties declared via `create schema` or the static or runtime configuration.

2.7.2. Object-Array Properties

Object-array event properties can be of any type. Object-array event properties that are Java application objects or that are of type `java.util.Map` (or arrays thereof) or that are of type `Object-array` (or arrays thereof) 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 `Object[]` allow object-arrays 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 object-array within object-arrays and arrays of object-arrays within object-arrays.
- Properties that are of type `Map` allow Maps to be nested in object-array events and arbitrarily deep. The nested, indexed, mapped and dynamic property syntax can be used to query nested Maps and object-arrays alike.

In order to use `Object[]` (object-array) events, the event type name and property names and types, in a well-defined order that must match object-array event properties, must be made known to the engine via configuration or `create schema` EPL syntax. Please see examples in *Section 5.16, “Declaring an Event Type: Create Schema”* and *Section 15.4.3, “Events represented by Object[] (Object-array)”*.

The code snippet below defines an Object-array event type, creates an Object-array event and sends the event into the engine. The sample defines the `CarLocUpdateEvent` event type via the runtime configuration interface (`create schema` or static configuration could have been used instead).

```
// Define CarLocUpdateEvent event type (example for runtime-configuration
 interface)
String[] propertyNames = {"carId", "direction"};   // order is important
Object[] propertyTypes = {String.class, int.class};  // type order matches name
 order

epService.getEPAdministrator().getConfiguration().
  addEventType("CarLocUpdateEvent", propertyNames, propertyTypes);
```

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

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

```
// Send an event
Object[] event = {carId, direction};
epRuntime.sendEvent(event, "CarLocUpdateEvent");
```

The engine can also query Java objects as values in an `Object[]` event via the nested property syntax. Thus `Object[]` 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 `Object[]` event with a transaction and an account object.

```
epRuntime.sendEvent(new Object[] {txn, account}, "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
```

2.7.3. Object-Array Supertype

Your `Object[]` (object-array) event type may declare one supertype when configuring the type at engine initialization time or at runtime through the administrative interface.

The supertype of a `Object[]` event type must also be an object-array event type. All property names and types of a supertype are also available on a subtype and override such same-name properties of the subtype. In addition, anywhere within EPL that an event type name of an Object-array supertype is used, any of its Object-array subtypes also matches that expression (similar to the concept of interface or superclass).

The properties provided by the top-most supertype must occur first in the object array. Subtypes each append to the object array. The number of values appended must match the number of properties declared by the subtype.

For example, assume your application declares the following two types:

```
create objectarray schema SuperType (p0 string)
```

```
create objectarray schema SubType (p1 string) inherits SuperType
```

The object array event objects that your application can send into the engine are shown by the next code snippet:

```
epRuntime.sendEvent(new Object[] {"p0_value", "p1_value"}, "SubType");
epRuntime.sendEvent(new Object[] {"p0_value"}, "SuperType");
```

2.7.4. Advanced Object-Array Property Types

2.7.4.1. Nested Properties

Strongly-typed nested `Object[]`-within-`Object[]` events can be used to build rich, type-safe event types on the fly. Use the `addEventType` method on `Configuration` or `ConfigurationOperations` for initialization-time and runtime-time type definition, or the `create schema` EPL syntax.

Noteworthy points are:

- JavaBean (POJO) objects can appear as properties in `Object[]` event types.
- One may represent Object-array within Object-array and Object-Array-Array within Object-array (same for Map event types) using the name of a previously registered Object-array (or Map) event type.
- There is no limit to the number of nesting levels.
- Dynamic properties can be used to query `Object[]`-within-`Object[]` values 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 entries do not exist.

For demonstration, in this example our top-level event type is an `AccountUpdate` event, which has an `UpdatedFieldType` structure as a property. Inside the `UpdatedFieldType` 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:

```
String[] propertyNamesUpdField = {"name", "addressLine1", "history"};
Object[]     propertyTypesUpdField     =     {String.class,     String.class,
 UpdateHistory.class};
epService.getEPAdministrator().getConfiguration().
                    addEventType("UpdatedFieldType",     propertyNamesUpdField,
 propertyTypesUpdField);

String[] propertyNamesAccountUpdate = {"accountId", "fields"};
Object[] propertyTypesAccountUpdate = {long.class, "UpdatedFieldType"};
epService.getEPAdministrator().getConfiguration().
                  addEventType("AccountUpdate",     propertyNamesAccountUpdate,
 propertyTypesAccountUpdate);
```

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

```
Object[] updatedField = {"Joe Doe", "40 Popular Street", new UpdateHistory()};
Object[] accountUpdate = {10009901, 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
```

2.7.4.2. One-to-Many Relationships

To model repeated properties within an Object-array, you may use arrays as properties in an Object-array. You may use an array of primitive types or an array of JavaBean objects or an array of a previously declared Object-array or Map event type.

When using a previously declared Object-array event type as an array property, the literal `[]` must be appended after the event type name.

This following example defines an Object-array event type by name `Sale` to hold array properties of the various types. It assumes a `SalesPerson` Java class exists and an Object-array event type by name `OrderItem` was declared:

```
String[] propertyNames = {"userids", "salesPersons", "items"};
Object[] propertyTypes = {int[].class, SalesPerson[].class, "OrderItem[]");

epService.getEPAdministrator().getConfiguration().
    addEventType("SaleEvent", propertyNames, propertyTypes);
```

The three properties that the above example declares are:

- An integer array of user ids.
- An array of `SalesPerson` Java objects.
- An array of Object-array for order items.

The next EPL statement is a sample query asking for property values held by arrays:

```
select userids[0], salesPersons[1].name,
    items[1], items[1].price.amount from SaleEvent
```

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

Events can be represented as `org.w3c.dom.Node` instances and send into the engine via the `sendEvent` method on `EPRuntime` or via `EventSender`. Please note that configuration is required so the event type name and root element name is known. See *Chapter 15, Configuration*.

If a XML schema document (XSD file) can be made available as part of the configuration, then Esper can read the schema and appropriately present event type metadata and validate statements that use the event type and its properties. See *Section 2.8.1, “Schema-Provided XML Events”*.

When no XML schema document is provided, XML events can still be queried, however the return type and return values of property expressions are string-only and no event type metadata is available other then for explicitly configured properties. See *Section 2.8.2, “No-Schema-Provided XML Events”*.

In all cases Esper allows you to configure explicit XPath expressions as event properties. You can specify arbitrary XPath functions or expressions and provide a property name and type by which result values will be available for use in EPL statements. See *Section 2.8.3, “Explicitly-Configured Properties”*.

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

Only one event type per root element name may be configured. The engine recognizes each event by its root element name or you may use `EventSender` to send events.

This section uses the following XML document as an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<Sensor xmlns="SensorSchema">
  <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>
```

The schema for the example is:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="Sensor">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="ID" type="xs:string"/>
        <xs:element ref="Observation" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="Observation">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="ID" type="xs:string"/>
        <xs:element ref="Tag" maxOccurs="unbounded" />
      </xs:sequence>
      <xs:attribute name="Command" type="xs:string" use="required" />
    </xs:complexType>
  </xs:element>

  <xs:element name="Tag">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="ID" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

2.8.1. Schema-Provided XML Events

If you have a XSD schema document available for your XML events, Esper can interrogate the schema. The benefits are:

- New EPL statements that refer to event properties are validated against the types provided in the schema.
- Event type metadata becomes available for retrieval as part of the `EventType` interface.

2.8.1.1. Getting Started

The engine reads a XSD schema file from an URL you provide. Make sure files imported by the XSD schema file can also be resolved.

The configuration accepts a schema URL. This is a sample code snippet to determine a schema URL from a file in classpath:

```
URL schemaURL = this.getClass().getClassLoader().getResource("sensor.xsd");
```

Here is a sample use of the runtime configuration API, please see *Chapter 15, Configuration* for further examples.

```
epService = EPServiceProviderManager.getDefaultProvider();
ConfigurationEventTypeXMLDOM sensorcfg = new ConfigurationEventTypeXMLDOM();
sensorcfg.setRootElementName("Sensor");
sensorcfg.setSchemaResource(schemaURL.toString());
epService.getEPAdministrator().getConfiguration()
    .addEventType("SensorEvent", sensorcfg);
```

You must provide a root element name. This name is used to look up the event type for the `sendEvent(org.w3c.Node node)` method. An `EventSender` is a useful alternative method for sending events if the type lookup based on the root or document element name is not desired.

After adding the event type, you may create statements and send events. Next is a sample statement:

```
select ID, Observation.Command, Observation.ID,
  Observation.Tag[0].ID, Observation.Tag[1].ID
```

```
from SensorEvent
```

As you can see from the example above, property expressions can query property values held in the XML document's elements and attributes.

There are multiple ways to obtain a XML DOM document instance from a XML string. The next code snippet shows how to obtain a XML DOM `org.w3c.Document` instance:

```
InputSource source = new InputSource(new StringReader(xml));
DocumentBuilderFactory builderFactory = DocumentBuilderFactory.newInstance();
builderFactory.setNamespaceAware(true);
Document doc = builderFactory.newDocumentBuilder().parse(source);
```

Send the `org.w3c.Node` or `Document` object into the engine for processing:

```
epService.getEPRuntime().sendEvent(doc);
```

2.8.1.2. Property Expressions and Namespaces

By default, property expressions such as `Observation.Tag[0].ID` are evaluated by a fast DOM-walker implementation provided by Esper. This DOM-walker implementation is not namespace-aware.

Should you require namespace-aware traversal of the DOM document, you must set the `xpath-property-expr` configuration option to true (default is false). This flag causes Esper to generate namespace-aware XPath expressions from each property expression instead of the DOM-walker, as described next. Setting the `xpath-property-expr` option to true requires that you also configure namespace prefixes as described below.

When matching up the property names with the XSD schema information, the engine determines whether the attribute or element provides values. The algorithm checks attribute names first followed by element names. It takes the first match to the specified property name.

2.8.1.3. Property Expression to XPath Rewrite

By setting the `xpath-property-expr` option the engine rewrites each property expression as an XPath expression, effectively handing the evaluation over to the underlying XPath implementation available from classpath. Most JVM have a built-in XPath implementation and there are also optimized, fast implementations such as Jaxen that can be used as well.

Set the `xpath-property-expr` option if you need namespace-aware document traversal, such as when your schema mixes several namespaces and element names are overlapping.

The below table samples several property expressions and the XPath expression generated for each, without namespace prefixes to keep the example simple:

Table 2.5. Property Expression to XPath Expression

Property Expression	Equivalent XPath
`Observeration.ID`	`/Sensor/Observation/ID`
`Observeration.Command`	`/Sensor/Observation/@Command`
`Observeration.Tag[0].ID`	`/Sensor/Observation/Tag[position() = 1]/ID`

For mapped properties that are specified via the syntax `name('key')`, the algorithm looks for an attribute by name `id` and generates a XPath expression as `mapped[@id='key']`.

Finally, here is an example that includes all different types of properties and their XPath expression equivalent in one property expression:

```
select nested.mapped('key').indexed[1].attribute from MyEvent
```

The equivalent XPath expression follows, this time including `n0` as a sample namespace prefix:

```
/n0:rootelement/n0:nested/n0:mapped[@id='key']/n0:indexed[position() = 2]/
@attribute
```

2.8.1.4. Array Properties

All elements that are unbound or have max occurs greater then 1 in the XSD schema are represented as indexed properties and require an index for resolution.

For example, the following is not a valid property expression in the sample Sensor document: `Observeration.Tag.ID`. As no index is provided for `Tag`, the property expression is not valid.

Repeated elements within a parent element in which the repeated element is a simple type also are represented as an array.

Consider the next XML document:

```
<item>
  <book sku="8800090">
    <author>Isaac Asimov</author>
    <author>Robert A Heinlein</author>
  </book>
</item>
```

Here, the result of the expression `book.author` is an array of type String and the result of `book.author[0]` is a String value.

2.8.1.5. Dynamic Properties

Dynamic properties are not validated against the XSD schema information and their result value is always `org.w3c.Node`. You may use a user-defined function to process dynamic properties returning `Node`. As an alternative consider using an explicit property.

An example dynamic property is `Origin?.ID` which will look for an element by name `Origin` that contains an element or attribute node by name `LocationCode`:

```
select Origin?.LocationCode from SensorEvent
```

2.8.1.6. Transposing Properties

When providing a XSD document, the default configuration allows to transpose property values that are themselves complex elements, as defined in the XSD schema, into a new stream. This behavior can be controlled via the flag `auto-fragment`.

For example, consider the next query:

```
insert into ObservationStream
select ID, Observation from SensorEvent
```

The `Observation` as a property of the `SensorEvent` gets itself inserted into a new stream by name `ObservationStream`. The `ObservationStream` thus consists of a string-typed `ID` property and a complex-typed property named `Observation`, as described in the schema.

A further statement can use this stream to query:

```
select Observation.Command, Observation.Tag[0].ID from ObservationStream
```

Before continuing the discussion, here is an alternative syntax using the wildcard-select, that is also useful:

```
insert into TagListStream
select ID as sensorId, Observation.* from SensorEvent
```

The new `TagListStream` has a string-typed `ID` and `Command` property as well as an array of `Tag` properties that are complex types themselves as defined in the schema.

Next is a sample statement to query the new stream:

```
select sensorId, Command, Tag[0].ID from TagListStream
```

Please note the following limitations:

- The XPath standard prescribes that XPath expressions against `org.w3c.Node` are evaluated against the owner document of the `Node`. Therefore XPath is not relative to the current node but absolute against each node's owner document. Since Esper does not create new document instances for transposed nodes, transposing properties is not possible when the `xpath-property-expr` flag is set.
- Complex elements that have both simple element values and complex child elements are not transposed. This is to ensure their property value is not hidden. Use an explicit XPath expression to transpose such properties.

Esper automatically registers a new event type for transposed properties. It generates the type name of the new XML event type from the XML event type name and the property names used in the expression. The synposis is *type_name.property_name[.property_name...]*. The type name can be looked up, for example for use with `EventSender` or can be created in advance.

2.8.1.7. Event Sender

An `EventSender` sends events into the engine for a given type, saving a type lookup based on element name.

This brief example sends an event via `EventSender`:

```
EventSender sender = epRuntime.getEventSender("SensorEvent");
sender.sendEvent(node);
```

The XML DOM event sender checks the root element name before processing the event. Use the `event-sender-validates-root` setting to disable validation. This forces the engine to process XML documents according to any predefined type without validation of the root element name.

2.8.1.8. Limitations

The engine schema interrogation is based on the Xerces distribution packaged into Sun Java runtimes. Your application may not replace the JRE's Xerces version and use XML schemas, unless your application sets the DOM implementation registry as shown below before loading the engine configuration:

```
System.setProperty(DOMImplementationRegistry.PROPERTY,
  "com.sun.org.apache.xerces.internal.dom.DOMXSImplementationSourceImpl");
```

2.8.2. No-Schema-Provided XML Events

Without a schema document a XML event may still be queried. However there are important differences in the metadata available without a schema document and therefore the property expression results. These differences are outlined below.

All property expressions against a XML type without schema are assumed valid. There is no validation of the property expression other then syntax validation. At runtime, property expressions return string-type values or `null` if the expression did not yield a matching element or attribute result.

When asked for property names or property metadata, a no-schema type returns empty array.

In all other aspects the type behaves the same as the schema-provided type described earlier.

2.8.3. Explicitly-Configured Properties

Regardless of whether or not you provide a XSD schema for the XML event type, you can always fall back to configuring explicit properties that are backed by XPath expressions.

For further documentation on XPath, please consult the XPath standard or other online material. Consider using Jaxen or Apache Axiom, for example, to provide faster XPath evaluation then your Java VM built-in XPath provider may offer.

2.8.3.1. Simple Explicit Property

Shown below is an example configuration that adds an explicit property backed by a XPath expression and that defines namespace prefixes:

```
epService = EPServiceProviderManager.getDefaultProvider();
ConfigurationEventTypeXMLDOM sensorcfg = new ConfigurationEventTypeXMLDOM();
sensorcfg.addXPathProperty("countTags",       "count(/ss:Sensor/ss:Observation/
ss:Tag)",
    XPathConstants.NUMBER);
sensorcfg.addNamespacePrefix("ss", "SensorSchema");
sensorcfg.setRootElementName("Sensor");
epService.getEPAdministrator().getConfiguration()
    .addEventType("SensorEvent", sensorcfg);
```

The `countTags` property is now available for querying:

```
select countTags from SensorEvent
```

The XPath expression `count(...)` is a XPath built-in function that counts the number of nodes, for the example document the result is `2`.

2.8.3.2. Explicit Property Casting and Parsing

Esper can parse or cast the result of your XPath expression to the desired type. Your property configuration provides the type to cast to, like this:

```
sensorcfg.addXPathProperty("countTags",       "count(/ss:Sensor/ss:Observation/
ss:Tag)",
    XPathConstants.NUMBER, "int");
```

The type supplied to the property configuration must be one of the built-in types. Arrays of built-in type are also possible, requiring the `XPathConstants.NODESET` type returned by your XPath expression, as follows:

```
sensorcfg.addXPathProperty("idarray", "//ss:Tag/ss:ID",
    XPathConstants.NODESET, "String[]");
```

The XPath expression `//ss:Tag/ss:ID` returns all ID nodes under a Tag node, regardless of where in the node tree the element is located. For the example document the result is `2` array elements `urn:epc:1:2.24.400` and `urn:epc:1:2.24.40`.

2.8.3.3. Node and Nodeset Explicit Property

An explicit property may return `XPathConstants.NODE` or `XPathConstants.NODESET` and can provide the event type name of a pre-configured event type for the property. The method name to add such properties is `addXPathPropertyFragment`.

This code snippet adds two explicit properties and assigns an event type name for each property:

```
sensorcfg.addXPathPropertyFragment("tagOne", "//ss:Tag[position() = 1]",
    XPathConstants.NODE, "TagEvent");
sensorcfg.addXPathPropertyFragment("tagArray", "//ss:Tag",
    XPathConstants.NODESET, "TagEvent");
```

The configuration above references the `TagEvent` event type. This type must also be configured. Prefix the root element name with "//" to cause the lookup to search the nested schema elements for the definition of the type:

```
ConfigurationEventTypeXMLDOM tagcfg = new ConfigurationEventTypeXMLDOM();
tagcfg.setRootElementName("//Tag");
tagcfg.setSchemaResource(schemaURL);
epAdministrator.getConfiguration()
    .addEventType("TagEvent", tagcfg);
```

The `tagOne` and `tagArray` properties are now ready for selection and transposing to further streams:

```
insert into TagOneStream select tagOne.* from SensorEvent
```

Select from the new stream:

```
select ID from TagOneStream
```

An example with indexed properties is shown next:

```
insert into TagArrayStream select tagArray as mytags from SensorEvent
```

Select from the new stream:

```
select mytags[0].ID from TagArrayStream
```

2.9. Additional Event Representations

Part of the extension and plug-in features of Esper is an event representation API. This set of classes allow an application to create new event types and event instances based on information available elsewhere, statically or dynamically at runtime when EPL statements are created. Please see *Section 17.8, “Event Type And Event Object”* for details.

Creating a plug-in event representation can be useful when your application has existing Java classes that carry event metadata and event property values and your application does not want to (or cannot) extract or transform such event metadata and event data into one of the built-in event representations (POJO Java objects, Map, Object-array or XML DOM).

Further use of a plug-in event representation is to provide a faster or short-cut access path to event data. For example, access to event data stored in a XML format through the Streaming API for XML (StAX) is known to be very efficient. A plug-in event representation can also provide network lookup and dynamic resolution of event type and dynamic sourcing of event instances.

Currently, EsperIO provides the following additional event representations:

- Apache Axiom: Streaming API for XML (StAX) implementation

Please see the EsperIO documentation for details on the above.

The chapter on *Section 17.8, “Event Type And Event Object”* explains how to create your own custom event representation.

2.10. Updating, Merging and Versioning Events

To summarize, an event is an immutable record of a past occurrence of an action or state change, and event properties contain useful information about an event.

The length of time an event is of interest to the event processing engine (retention time) depends on your EPL statements, and especially the data window, pattern and output rate limiting clauses of your statements.

During the retention time of an event more information about the event may become available, such as additional properties or changes to existing properties. Esper provides three concepts for handling updates to events.

The first means to handle updating events is the `update istream` clause as further described in *Section 5.21, “Updating an Insert Stream: the Update IStream Clause”*. It is useful when you need to update events as they enter a stream, before events are evaluated by any particular consuming statement to that stream.

The second means to update events is the `on-merge` and `on-update` clauses, for use with named windows only, as further described in *Section 5.15.12, “Triggered Upsert using the On-Merge Clause”* and *Section 5.15.8, “Updating Named Windows: the On Update clause”*. On-merge is similar to the SQL `merge` clause and provides what is known as an "Upsert" operation: Update existing events or if no existing event(s) are found then insert a new event, all in one atomic operation provided by a single EPL statement. On-update can be used to update individual properties of events held in a named window.

The third means to handle updating events is the revision event types, for use with named windows only, as further described in *Section 5.15.14, “Versioning and Revision Event Type Use with Named Windows”*. With revision event types one can declare, via configuration only, multiple different event types and then have the engine present a merged event type that contains a superset of properties of all merged types, and have the engine merge events as they arrive without additional EPL statements.

Note that patterns do not reflect changes to past events. For the temporal nature of patterns, any changes to events that were observed in the past do not reflect upon current pattern state.

2.11. Coarse-Grained Events

Your application events may consist of fairly comprehensive, coarse-grained structures or documents. For example in business-to-business integration scenarios, XML documents or other event objects can be rich deeply-nested graphs of event properties.

To extract information from a coarse-grained event or to perform bulk operations on the rows of the property graph in an event, Esper provides a convenient syntax: When specifying a filter expression in a pattern or in a `select` clause, it may contain a contained-event selection syntax, as further described in *Section 5.20, “Contained-Event Selection”*.

2.12. Event Objects Instantiated and Populated by `Insert Into`

The `insert into` clause can populate instantiate new instances of Java object events, `java.util.Map` events and `Object[]` (object array) events directly from the results of `select` clause expressions and populate such instances. Simply use the event type name as the stream name in the `insert into` clause as described in *Section 5.10, “Merging Streams and Continuous Insertion: the Insert Into Clause”*.

If instead you have an existing instance of a Java object returned by an expression, such as a single-row function or static method invocation for example, you can transpose that expression result object to a stream. This is described further in *Section 5.10.7, “Transposing an Expression Result”* and *Section 9.4, “Select-Clause transpose Function”*.

The column names specified in the `select` and `insert into` clause must match available writable properties in the event object to be populated (the target event type). The expression result types of any expressions in the `select` clause must also be compatible with the property types of the target event type.

If populating a POJO-based event type and the class provides a matching constructor, the expression result types of expressions in the `select` clause must be compatible with the constructor parameters in the order listed by the constructor. The `insert into` clause column names are not relevant in this case.

Consider the following example statement:

```
insert into com.mycompany.NewEmployeeEvent
select fname as firstName, lname as lastName from HRSystemEvent
```

The above example specifies the fully-qualified class name of `NewEmployeeEvent`. The engine instantianes `NewEmployeeEvent` for each result row and populates the `firstName` and `lastName` properties of each instance from the result of `select` clause expressions. The `HRSystemEvent` in the example is assumed to have `lname` and `fname` properties, and either setter-methods and a default constructor, or a matching constructor.

Note how the example uses the `as`-keyword to assign column names that match the property names of the `NewEmployeeEvent` target event. If the property names of the source and target events are the same, the `as`-keyword is not required.

The next example is an alternate form and specifies property names within the `insert into` clause instead. The example also assumes that `NewEmployeeEvent` has been defined or imported via configuration since it does not specify the event class package name:

```
insert into NewEmployeeEvent(firstName, lastName)
```

```
select fname, lname from HRSystemEvent
```

Finally, this example populates `HRSystemEvent` events. The example populates the value of a `type` property where the event has the value 'NEW' and populates a new event object with the value 'HIRED', copying the `fname` and `lname` property values to the new event object:

```
insert into HRSystemEvent
select fname, lname, 'HIRED' as type from HRSystemEvent(type='NEW')
```

The matching of the `select` or `insert into`-clause column names to target event type's property names is case-sensitive. It is allowed to only populate a subset of all available columns in the target event type. Wildcard (`*`) is also allowed and copies all fields of the events or multiple events in a join.

For Java object events, your event class must provide setter-methods according to JavaBean conventions or, alternatively, a matching constructor. If the event class provides setter methods the class should also provide a default constructor taking no parameters. If the event class provides a matching constructor there is no need for setter-methods. If your event class does not have a default constructor and setter methods, or a matching constructor, your application may configure a factory method via `ConfigurationEventTypeLegacy`.

The engine follows Java standards in terms of widening, performing widening automatically in cases where widening type conversion is allowed without loss of precision, for both boxed and primitive types and including BigInteger and BigDecimal.

When inserting array-typed properties into a Java, Map-type or Object-array underlying event the event definition should declare the target property as an array.

Please note the following limitations:

- Event types that utilize XML `org.w3c.dom.Node` underlying event objects cannot be target of an `insert into` clause.

2.13. Comparing Event Representations

Each of the event representations of Java object, Map and XML document has advantages and disadvantages that are summarized in the table below:

Table 2.6. Comparing Event Representations

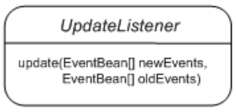
	Java Object (POJO/Bean or other)	Map	Object-array	XML Document
Performance	Good	Good	Very Good	Not comparable and depending on use of XPath
Memory Use	Small	Medium	Small	Depends on DOM and XPath implementation used, can be large
Call Method on Event	Yes	Yes, if contains Object(s)	Yes, if contains Object(s)	No
Nested, Indexed, Mapped and Dynamic Properties	Yes	Yes	Yes	Yes
Course-grained event syntax	Yes	Yes	Yes	Yes
Insert-into that Representation	Yes	Yes	Yes	No
Runtime Type Change	Reload class, yes	Yes	Yes	Yes
Create-schema Syntax	Yes	Yes	Yes	No, runtime and static configuration
Object is Self-Descriptive	Yes	Yes	No	Yes
Supertypes	Multiple	Multiple	Single	No

Chapter 3. Processing Model

3.1. Introduction

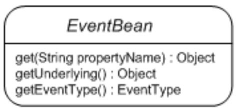
The Esper processing model is continuous: Update listeners and/or subscribers to statements receive updated data as soon as the engine processes events for that statement, according to the statement's choice of event streams, views, filters and output rates.

As outlined in *Chapter 14, 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:



A second, strongly-typed and native, highly-performant method of result delivery is provided: 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. Please see *Section 14.3.3, "Setting a Subscriber Object"*.

The engine provides statement results to update listeners by placing results in `com.espertech.esper.client.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* and *Section 14.6, "Event and Event Type"*

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.

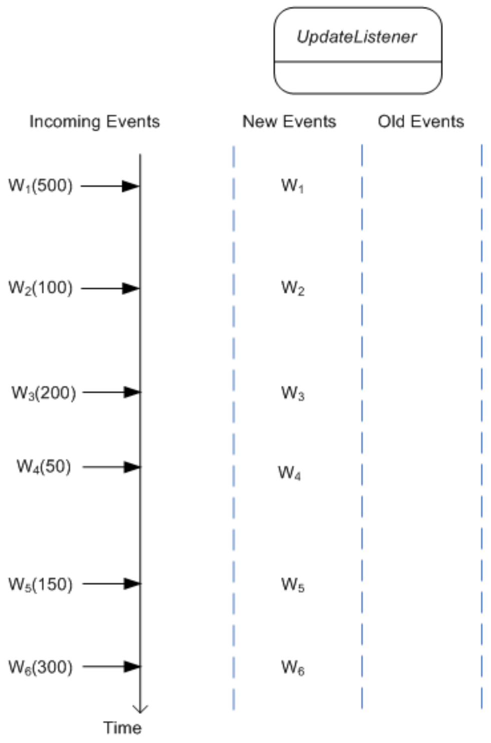


Figure 3.1. Output example for a simple statement

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.

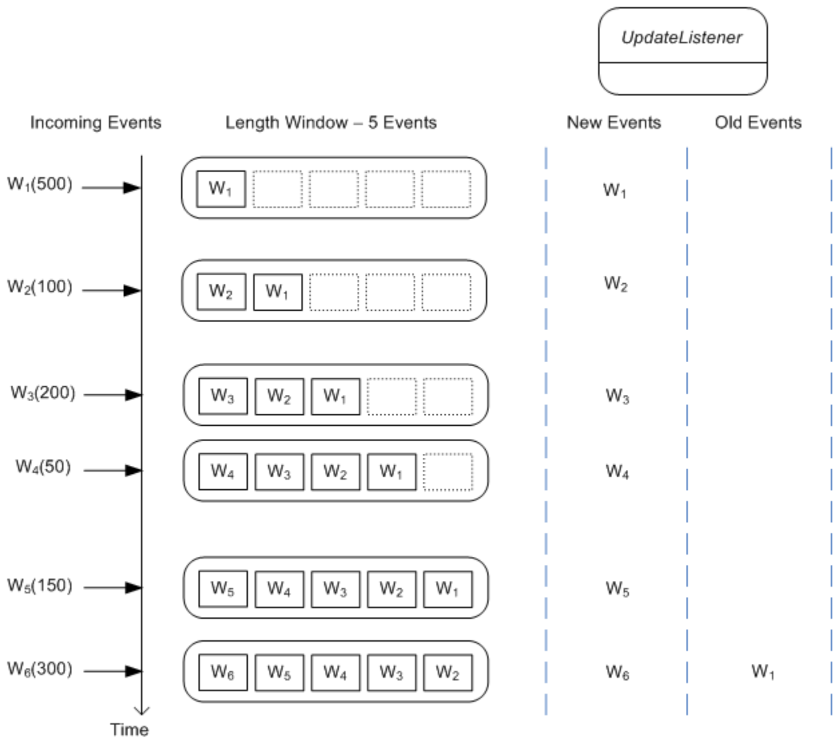


Figure 3.2. Output example for a length window

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

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 5.3.7, “Selecting insert and remove stream events”*. There is also a related, engine-wide configuration setting described in *Section 15.4.17, “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 5.4.1, “Filter-based Event Streams”* and *Section 6.4, “Filter Expressions In Patterns”*.

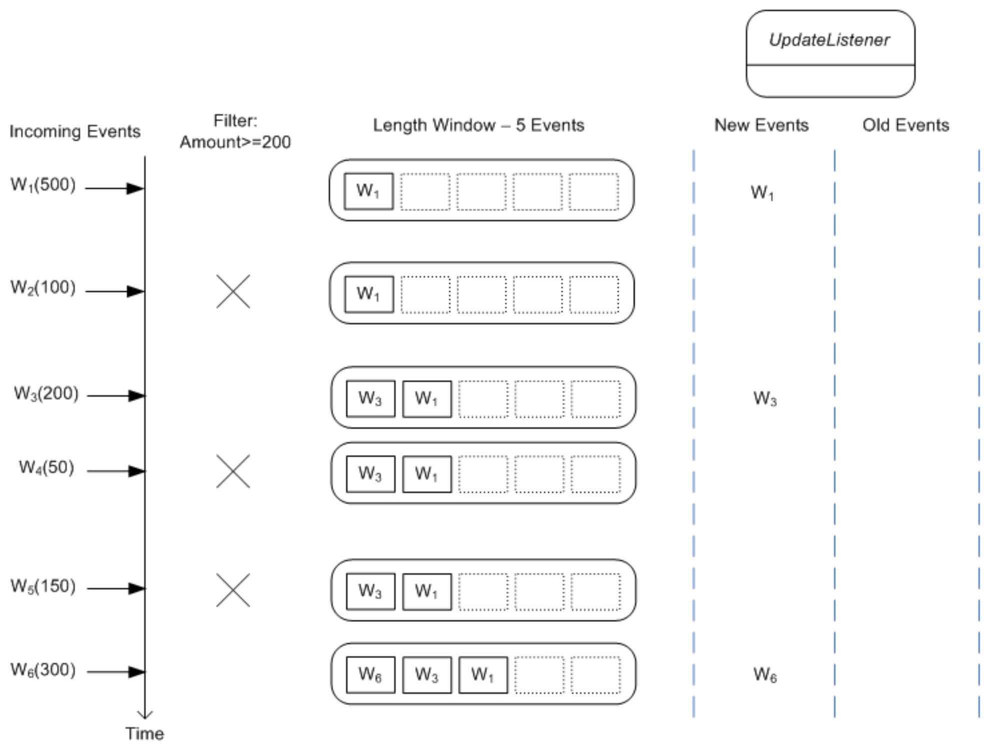


Figure 3.3. Output example for a statement with an event stream filter

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 5.5, “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.

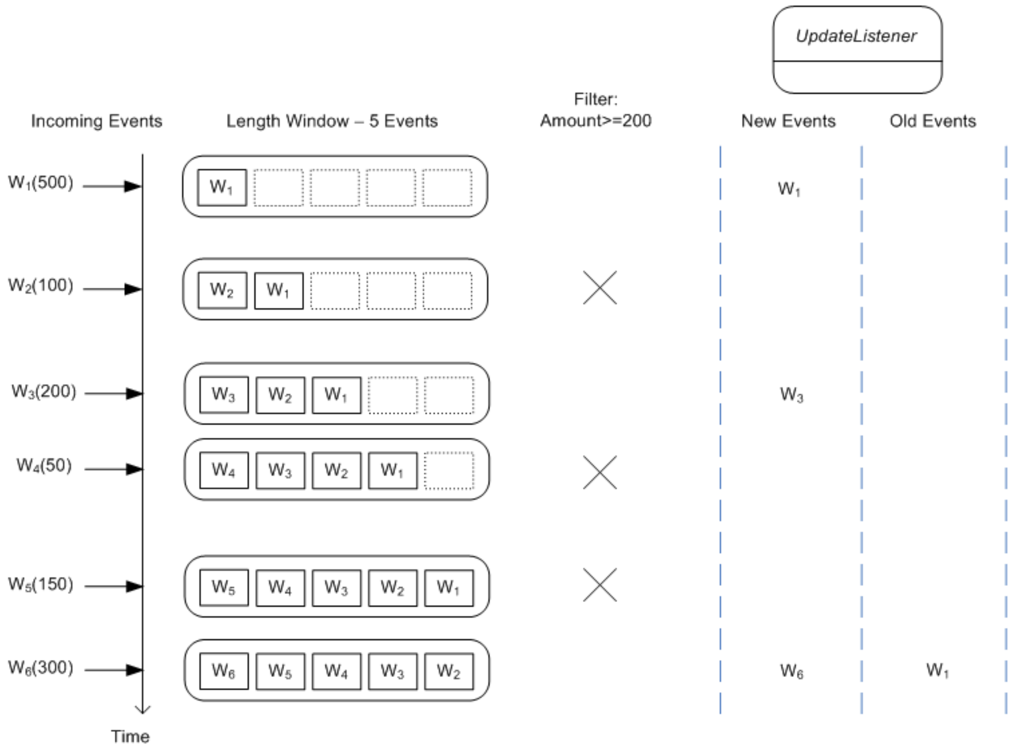


Figure 3.4. Output example for a statement with where-clause

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.

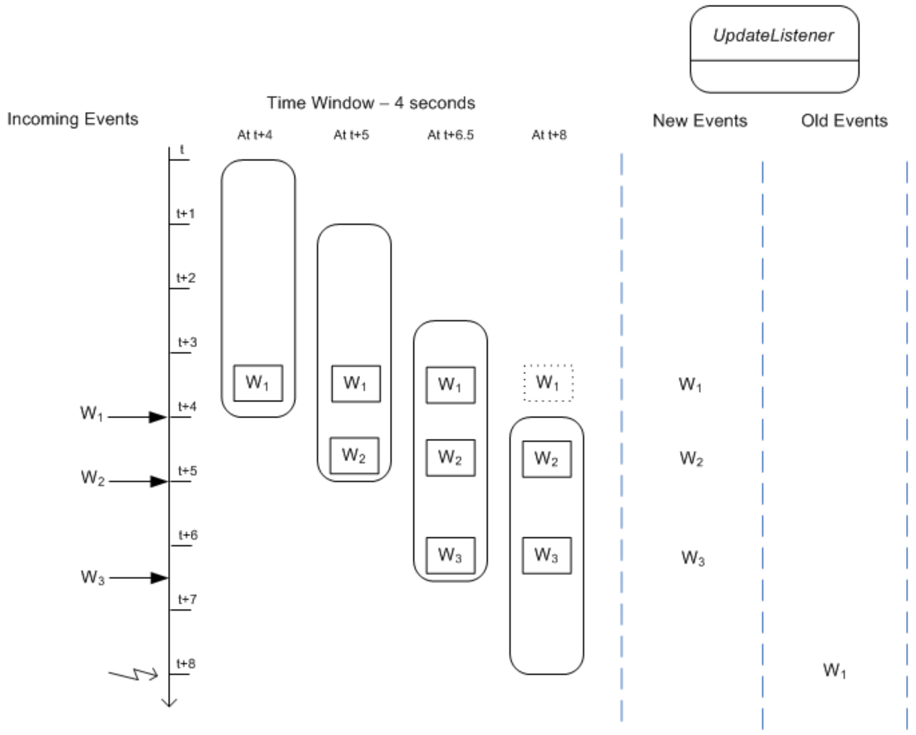


Figure 3.5. Output example for a statement with a time window

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.

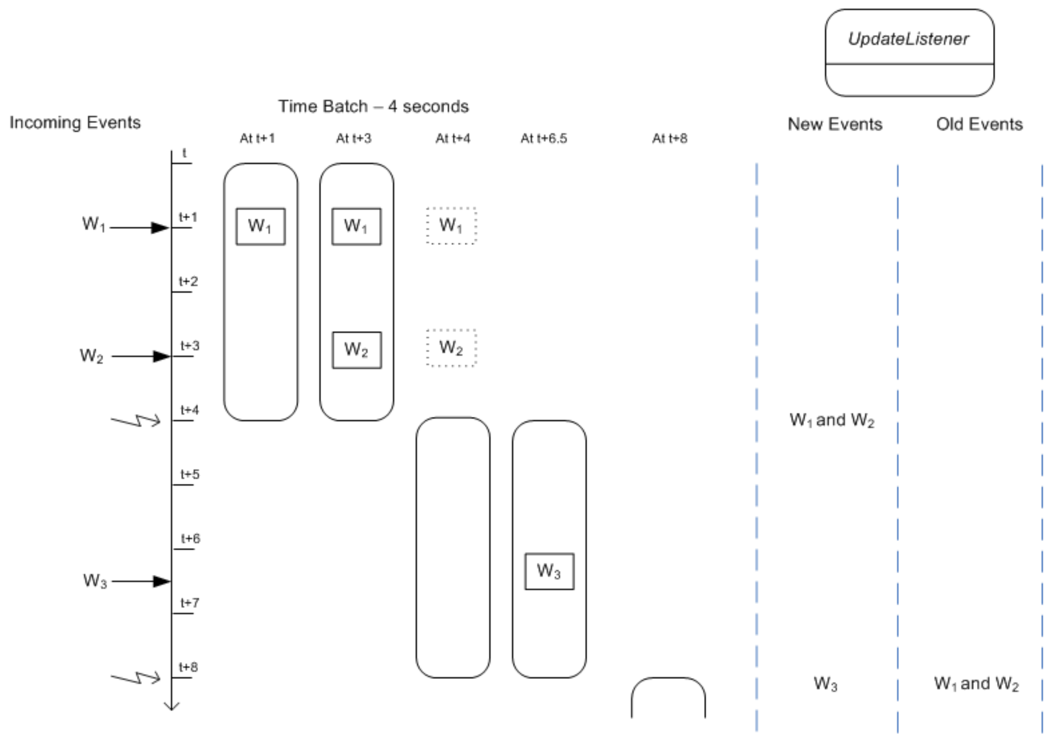


Figure 3.6. Output example for a statement with a time batch view

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, among others. 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.

Related to batch data windows is output rate limiting. While batch data windows retain events the `output` clause offered by output rate limiting can control or stabilize the rate at which events are output, see *Section 5.7, “Stabilizing and Controlling Output: the Output Clause”*.

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, if specifing the `rstream` keyword in the select clause to select the remove stream. The value of the "mycount" property on that old event is also 2.

Note the statement above does not specify a data window and thereby counts all arriving events since statement start. The statement above retains no events and its memory allocation is only the aggregation state, i.e. a single long value to represent `count(*)`.

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.

3.7.2.1. 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”*

3.7.2.2. 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"*

3.7.2.3. 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"*

3.7.2.4. 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"*

3.7.2.5. 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. Event Visibility and Current Time

An event sent by your application or generated by statements is visible to all other statements in the same engine instance. Similarly, current time (the time horizon) moves forward for all statements in the same engine instance. Please see the *Chapter 14, API Reference* chapter for how to send events and how time moves forward through system time or via simulated time, and the possible threading models.

Within an Esper engine instance you can additionally control event visibility and current time on a statement level, under the term *isolated service* as described in *Section 14.10, "Service Isolation"*.

An isolated service provides a dedicated execution environment for one or more statements. Events sent to an isolated service are visible only within that isolated service. In the isolated service you can move time forward at the pace and resolution desired without impacting other statements that reside in the engine runtime or other isolated services. You can move statements between the engine and an isolated service.

Chapter 4. Context and Context Partitions

4.1. Introduction

This section discusses the notion of context and its role in the Esper event processing language (EPL).

When you look up the word *context* in a dictionary, you may find: Context is the set of circumstances or facts that surround a particular event, situation, etc..

Context-dependent event processing occurs frequently: For example, consider a requirement that monitors banking transactions. For different customers your analysis considers customer-specific aggregations, patterns or data windows. In this example the context of detection is the customer. For a given customer you may want to analyze the banking transactions of that customer by using aggregations, data windows, patterns including other EPL constructs.

In a second example, consider traffic monitoring to detect speed violations. Assume the speed limit must be enforced only between 9 am and 5 pm. The context of detection is of temporal nature.

A context takes a cloud of events and classifies them into one or more sets. These sets are called *context partitions*. An event processing operation that is associated with a context operates on each of these context partitions independently. (Credit: Taken from the book "Event Processing in Action" by Opher Etzion and Peter Niblett.)

A context is a declaration of dimension and may thus result in one or more context partitions. In the banking transaction example there the context dimension is the customer and a context partition exists per customer. In the traffic monitoring example there is a single context partition that exists only between 9 am and 5 pm and does not exist outside of that daily time period.

In an event processing glossary you may find the term *event processing agent*. An EPL statement is an *event processing agent*. An alternative term for context partition is *event processing agent instance*.

Esper EPL allows you to declare contexts explicitly, offering the following benefits:

1. Context can apply to multiple statements thereby eliminating the need to duplicate context dimensional information between statements.
2. Context partitions can be temporally overlapping.
3. Context partitions provide a fine-grained lifecycle that is independent of the lifecycle of statement lifecycle.
4. Fine-grained lock granularity: The engine locks on the level of context partitions thereby allowing very high concurrency, with a maximum (theoretical) degree of parallelism at 2^31-1

(2,147,483,647) parallel threads working to process a single EPL statement under a hash segmented context.

5. EPL can become easier to read as common predicate expressions can be factored out into a context.

6. You may specify a nested context that is composed from two or more contexts. In particular a temporal context type is frequently used in combination with a segmentation-oriented context.

7. Using contexts your application can aggregate events over time periods (overlapping or non-overlapping) without retaining any events in memory.

8. Using contexts your application can coordinate time boundaries for multiple statements.

Esper EPL allows you to declare a context explicitly via the `create context` syntax introduced below.

After you have declared a context, one or more EPL statements can refer to that context by specifying `context` *name*. When an EPL statement refers to a context, all EPL-statement related state such as aggregations, patterns or data windows etc. exists once per context partition.

If an EPL statement does not declare a context, it implicitly has a single context partition. The single context partition lives as long as the EPL statement is started and ends when the EPL statement is stopped.

Variables are global state and are visible across context partitions. The same is true for event types and external data.

For more information on locking and threading please see *Section 14.7, “Engine Threading and Concurrency”*. For performance related information please refer to *Chapter 20, Performance*.

4.2. Context Declaration

The `create context` statement declares a context by specifying a context name and context dimension information.

A context declaration by itself does not consume any resources or perform any logic until your application starts at least one statement that refers to that context. Until then the context is inactive and not in use.

When your application creates or starts the first statement that refers to the context, the engine activates the context.

As soon as your application stops or destroys all statements that refer to the context, the context becomes inactive again.

When your application stops or destroys a statement that refers to a context, the context partitions associated to that statement also end (context partitions associated to other started statements live on).

When your application stops or destroys the statement that declared the context and does not also stop or destroy any statements that refer to the context, the context partitions associated to each such statement do not end.

When your application destroys the statement that declared the context and destroys all statements that refer to that context then the engine removes the context declaration entirely.

The `create context` statement posts no output events to listeners or subscribers and does not return any rows when iterated.

4.2.1. Context-Provided Properties

Each of the context declarations makes available a set of built-in context properties as well as initiating event or pattern properties, as applicable. You may select these context properties for output or use them in any of the statement expressions.

Refer to built-in context properties as `context.`*property_name*, wherein *property_name* refers to the name of the built-in context property.

Refer to initiating event or pattern match event properties as `context.`*stream_name.property_name*, wherein *stream_name* refers to the name assigned to the event or the tag name specified in a pattern and *property_name* refers to the name of the initiating event or pattern match event property.

4.2.2. Keyed Segmented Context

This context assigns events to context partitions based on the values of one or more event properties, using the value of these property(s) as a key that picks a unique context partition directly. Each event thus belongs to exactly one context partition or zero context partitions if the event does not match the optional filter predicate expression(s). Each context partition handles exactly one set of key values.

The syntax for creating a keyed segmented context is as follows:

```
create context context_name partition [by]
  event_property [and event_property [and ...]] from stream_def
  [, event_property [...] from stream_def]
  [, ...]
```

The *context_name* you assign to the context can be any identifier.

Following the context name is one or more lists of event properties and a stream definition for each entry, separated by comma (`,`).

The *event_property* is the name(s) of the event properties that provide the value(s) to pick a unique partition. Multiple event property names are separated by the `and` keyword.

The *stream_def* is a stream definition which consists of an event type name optionally followed by parenthesis that contains filter expressions. If providing filter expressions, only events matching the provided filter expressions for that event type are considered by context partitions.

You may list multiple event properties for each stream definition. You may list multiple stream definitions. Please refer to usage guidelines below when specifying multiple event properties and/ or multiple stream definitions.

The next statement creates a context `SegmentedByCustomer` that considers the value of the `custId` property of the `BankTxn` event type to pick the context partition to assign events to:

```
create context SegmentedByCustomer partition by custId from BankTxn
```

The following statement refers to the context created as above to compute a total withdrawal amount per account for each customer:

```
context SegmentedByCustomer
select custId, account, sum(amount) from BankTxn group by account
```

The following statement refers to the context created as above and detects a withdrawal of more then 400 followed by a second withdrawal of more then 400 that occur within 10 minutes of the first withdrawal, all for the same customer:

```
context SegmentedByCustomer
select * from pattern [
  every a=BankTxn(amount > 400) -> b=BankTxn(amount > 400) where timer:within(10
 minutes)
]
```

The EPL statement that refers to a keyed segmented context must have at least one filter expression, at any place within the EPL statement that looks for events of any of the event types listed in the context declaration.

For example, the following is not valid:

```
// Neither LoginEvent nor LogoutEvent are listed in the context declaration
context SegmentedByCustomer
select * from pattern [every a=LoginEvent -> b=LogoutEvent where timer:within(10
 minutes)]
```

4.2.2.1. Multiple Stream Definitions

If the context declaration lists multiple streams, each event type must be unrelated: You may not list the same event type twice and you may not list a sub- or super-type of any event type already listed.

The following is not a valid declaration since the `BankTxn` event type is listed twice:

```
// Not valid
create context SegmentedByCustomer partition by custId from BankTxn, account
 from BankTxn
```

If the context declaration lists multiple streams, the number of event properties provided for each event type must also be the same. The value type returned by event properties of each event type must match within the respective position it is listed in, i.e. the first property listed for each event type must have the same type, the second property listed for each event type must have the same type, and so on.

The following is not a valid declaration since the customer id of `BankTxn` and login time of `LoginEvent` is not the same type:

```
// Invalid: Type mismatch between properties
create context SegmentedByCustomer partition by custId from BankTxn, loginTime
 from LoginEvent
```

The next statement creates a context `SegmentedByCustomer` that also considers `LoginEvent` and `LogoutEvent`:

```
create context SegmentedByCustomer partition by
  custId from BankTxn, loginId from LoginEvent, loginId from LogoutEvent
```

As you may have noticed, the above example refers to `loginId` as the event property name for `LoginEvent` and `LogoutEvent` events. The assumption is that the `loginId` event property of the login and logout events has the same type and carries the same exact value as the `custId` of bank transaction events, thereby allowing all events of the three event types to apply to the same customer-specific context partition.

4.2.2.2. Filters

You may add a filter expression to each of the event types listed. The engine applies the filter expression to the EPL statement that refers to the context and to the same event type.

The next statement creates a context `SegmentedByCustomer` that does not consider login events that indicate that the login failed.

```
create context SegmentedByCustomer partition by
  custId from BankTxn, loginId from LoginEvent(failed=false)
```

4.2.2.3. Multiple Properties Per Event Type

You may assign events to context partitions based on the values of two or more event properties. The engine thus uses the combination of values of these properties to pick a context partition.

An example context declaration follows:

```
create context ByCustomerAndAccount partition by custId and account from BankTxn
```

The next statement refers to the context and computes a total withdrawal amount, per account and customer:

```
context ByCustomerAndAccount select custId, account, sum(amount) from BankTxn
```

As you can see, the above statement does not need to specify `group by` clause to aggregate per customer and account, since events of each unique combination of customer id and account are assigned to separate context partitions.

4.2.2.4. Built-In Context Properties

The following context properties are available in your EPL statement when it refers to a keyed segmented context:

Table 4.1. Keyed Segmented Context Properties

Name	Description
`name`	The string-type context name.
`id`	The integer-type internal context id that the engine assigns to the context partition.
`key1`	The event property value for the first key.
`key`*N*	The event property value for the Nth key.

Assume the keyed segmented context is declared as follows:

```
create context ByCustomerAndAccount partition by custId and account from BankTxn
```

You may, for example, select the context properties as follows:

```
context ByCustomerAndAccount
  select context.name, context.id, context.key1, context.key2 from BankTxn
```

4.2.2.5. Examples of Joins

This section discusses the impact of contexts on joins to provide further samples of use and deepen the understanding of context partitions.

Consider a context declared as follows:

```
create context ByCust partition by custId from BankTxn
```

The following statement matches, within the same customer id, the current event with the last 30 minutes of events to determine those events that match amounts:

```
context ByCust
  select * from BankTxn as t1 unidirectional, BankTxn.win:time(30) t2
  where t1.amount = t2.amount
```

Note that the `where`-clause in the join above does not mention customer id. Since each `BankTxn` applies to a specific context partition the join evaluates within that single context partition.

Consider the next statement that matches a security event with the last 30 minutes of transaction events for each customer:

```
context ByCust
  select * from SecurityEvent as t1 unidirectional, BankTxn.win:time(30) t2
  where t1.customerName = t2.customerName
```

When a security event comes in, it applies to all context partitions and not any specific context partition, since the `SecurityEvent` event type is not part of the context declaration.

4.2.3. Hash Segmented Context

This context assigns events to context partitions based on result of a hash function and modulo operation. Each event thus belongs to exactly one context partition or zero context partitions if the event does not match the optional filter predicate expression(s). Each context partition handles exactly one result of hash value modulo granularity.

The syntax for creating a hashed segmented context is as follows:

```
create context context_name coalesce [by]
  hash_func_name(hash_func_param) from stream_def
  [, hash_func_name(hash_func_param) from stream_def ]
  [, ...]
  granularity granularity_value
```

```
    [preallocate]
```

The *context_name* you assign to the context can be any identifier.

Following the context name is one or more lists of hash function name and parameters pairs and a stream definition for each entry, separated by comma (`,`).

The *hash_func_name* can either be `consistent_hash_crc32` or `hash_code` or a plug-in single-row function. The *hash_func_param* is a list of parameter expressions.

- If you specify `consistent_hash_crc32` the engine computes a consistent hash code using the CRC-32 algorithm.
- If you specify `hash_code` the engine uses the Java object hash code.
- If you specify the name of a plug-in single-row function your function must return an integer value that is the hash code. You may use the wildcard `(*)` character among the parameters to pass the underlying event to the single-row function.

The *stream_def* is a stream definition which consists of an event type name optionally followed by parenthesis that contains filter expressions. If providing filter expressions, only events matching the provided filter expressions for that event type are considered by context partitions.

You may list multiple stream definitions. Please refer to usage guidelines below when specifying multiple stream definitions.

The `granularity` is required and is an integer number that defines the maximum number of context partitions. The engine computes hash code modulo granularity `hash(`*params*`) mod` *granularity* to determine the context partition. When you specify the `hash_code` function the engine uses the object hash code and the computation is *params.*`hashCode() %`*granularity*.

Since the engine locks on the level of context partition to protect state, the granularity defines the maximum degree of parallelism. For example, a granularity of 1024 means that 1024 context partitions handle events and thus a maximum 1024 threads can process each assigned statement concurrently.

The optional `preallocate` keyword instructs the engine to allocate all context partitions at once at the time a statement refers to the context. This is beneficial for performance as the engine does not need to determine whether a context partition exists and dynamically allocate, but may require more memory.

The next statement creates a context `SegmentedByCustomerHash` that considers the CRC-32 hash code of the `custId` property of the `BankTxn` event type to pick the context partition to assign events to, with up to 16 different context partitions that are preallocated:

```
create context SegmentedByCustomerHash
    coalesce  by  consistent_hash_crc32(custId)  from  BankTxn  granularity  16
 preallocate
```

The following statement refers to the context created as above to compute a total withdrawal amount per account for each customer:

```
context SegmentedByCustomerHash
select custId, account, sum(amount) from BankTxn group by custId, account
```

Note that the statement above groups by `custId`: Since the events for different customer ids can be assigned to the same context partition, it is necessary that the EPL statement also groups by customer id.

The context declaration shown next assumes that the application provides a `computeHash` single-row function that accepts BankTxn as a parameter, wherein the result of this function must be an integer value that returns the context partition id for each event:

```
create context MyHashContext
  coalesce by computeHash(*) from BankTxn granularity 16 preallocate
```

The EPL statement that refers to a hash segmented context must have at least one filter expression, at any place within the EPL statement that looks for events of any of the event types listed in the context declaration.

For example, the following is not valid:

```
// Neither LoginEvent nor LogoutEvent are listed in the context declaration
context SegmentedByCustomerHash
select * from pattern [every a=LoginEvent -> b=LogoutEvent where timer:within(10
 minutes)]
```

4.2.3.1. Multiple Stream Definitions

If the context declaration lists multiple streams, each event type must be unrelated: You may not list the same event type twice and you may not list a sub- or super-type of any event type already listed.

If the context declaration lists multiple streams, the hash code function should return the same hash code for the related keys of all streams.

The next statement creates a context `HashedByCustomer` that also considers `LoginEvent` and `LogoutEvent`:

```
create context HashedByCustomer as coalesce
  consistent_hash_crc32(custId) from BankTxn,
  consistent_hash_crc32(loginId) from LoginEvent,
  consistent_hash_crc32(loginId) from LogoutEvent
  granularity 32 preallocate
```

4.2.3.2. Filters

You may add a filter expression to each of the event types listed. The engine applies the filter expression to the EPL statement that refers to the context and to the same event type.

The next statement creates a context `HashedByCustomer` that does not consider login events that indicate that the login failed.

```
create context HashedByCustomer
  coalesce consistent_hash_crc32(loginId) from LoginEvent(failed = false)
  granularity 1024 preallocate
```

4.2.3.3. Built-In Context Properties

The following context properties are available in your EPL statement when it refers to a keyed segmented context:

Table 4.2. Keyed Segmented Context Properties

Name	Description
`name`	The string-type context name.
`id`	The integer-type internal context id that the engine assigns to the context partition.

Assume the hashed segmented context is declared as follows:

```
create context ByCustomerHash coalesce consistent_hash_crc32(custId) from
 BankTxn granularity 1024
```

You may, for example, select the context properties as follows:

```
context ByCustomerHash
  select context.name, context.id from BankTxn
```

4.2.3.4. Performance Considerations

The `hash_code` function based on the Java object hash code is generally faster then the CRC32 algorithm. The CRC32 algorithm, when used with a non-String parameter or with multiple parameters, requires the engine to serialize all expression results to a byte array to compute the CRC32 hash code.

We recommend keeping the granularity small (1k and under) when using `preallocate`.

When specifying a granularity greater then `65536` (64k) the engine switches to a Map-based lookup of context partition state which can slow down statement processing.

4.2.4. Category Segmented Context

This context assigns events to context partitions based on the values of one or more event properties, using a predicate expression(s) to define context partition membership. Each event can thus belong to zero, one or many context partitions depending on the outcome of the predicate expression(s).

The syntax for creating a category segmented context is as follows:

```
create context context_name
  group [by] group_expression as category_label
  [, group [by] group_expression as category_label]
  [, ...]
  from stream_def
```

The *context_name* you assign to the context can be any identifier.

Following the context name is a list of groups separated by the `group` keyword. The list of group is followed by the `from` keyword and a stream definition.

The *group_expression* is an expression that categorizes events. Each group expression must be followed by the `as` keyword and a category label which can be any identifier.

Group expressions are predicate expression and must return a Boolean true or false when applied to an event. For a given event, any number of the group expressions may return true thus categories can be overlapping.

The *stream_def* is a stream definition which consists of an event type name optionally followed by parenthesis that contains filter expressions. If providing filter expressions, only events matching the provided filter expressions for that event type are considered by context partitions.

The next statement creates a context `CategoryByTemp` that consider the value of the `temperature` property of the `SensorEvent` event type to pick context partitions to assign events to:

```
create context CategoryByTemp
  group temp < 65 as cold,
  group temp between 65 and 85 as normal,
  group temp > 85 as large
  from SensorEvent
```

The following statement simply counts, for each category, the number of events and outputs the category label and count:

```
context CategoryByTemp select context.label, count(*) from SensorEvent
```

4.2.4.1. Built-In Context Properties

The following context properties are available in your EPL statement when it refers to a category segmented context:

Table 4.3. Category Segmented Context Properties

Name	Description
`name`	The string-type context name.
`id`	The integer-type internal context id that the engine assigns to the context partition.
`label`	The category label is the string identifier value after the `as` keyword that is specified for each group.

You may, for example, select the context properties as follows:

```
context CategoryByTemp
  select context.name, context.id, context.label from SensorEvent
```

4.2.5. Non-Overlapping Context

You may declare a non-overlapping context that exists once or that repeats in a regular fashion as controlled by start and end conditions. The number of context partitions is always either one or zero: Context partitions do not overlap.

The syntax for creating a non-overlapping context is as follows:

```
create context context_name
  start (@now | start_condition)
  end end_condition
```

The *context_name* you assign to the context can be any identifier.

Following the context name is the `start` keyword, either `@now` or a *start_condition*, the `end` keyword and an *end_condition*.

Both the start (if specified) and end condition can be an event filter, a pattern, a crontab or a time period. The syntax of start and end conditions is described in *Section 4.2.7, “Context Conditions”*.

Once the start condition occurs, the engine no longer observes the start condition and begins observing the end condition. Once the end condition occurs, the engine observes the start condition again. If you specified `@now` instead of a start condition, the engine begins observing the end condition instead.

If you specified an event filter as the start condition, then the event also counts towards the statement(s) that refer to that context. If you specified a pattern as the start condition, then the events that may constitute the pattern match can also count towards the statement(s) that refer to the context provided that `@inclusive` and event tags are both specified (see below).

At the time of context activation when your application creates a statement that utilizes the context, the engine checks whether the start and end condition are crontab expressions. The engine evaluates the start and end crontab expressions and determines whether the current time is a time between start and end. If the current time is between start and end times, the engine allocates the context partition and waits for observing the end time. Otherwise the engine waits to observe the start time and does not allocate a context partition.

The built-in context properties that are available are the same as described in *Section 4.2.6.1, "Built-In Context Properties"*.

The next statement creates a context `NineToFive` that declares a daily time period that starts at 9 am and ends at 5 pm:

```
create context NineToFive start (0, 9, *, *, *) end (0, 17, *, *, *)
```

The following statement outputs speed violations between 9 am and 5 pm, considering a speed of 100 or greater as a violation:

```
context NineToFive select * from TrafficEvent(speed >= 100)
```

The example that follows demonstrates the use of an event filter as the start condition and a pattern as the end condition.

The next statement creates a context `PowerOutage` that starts when the first `PowerOutageEvent` event arrives and that ends 5 seconds after a subsequent `PowerOnEvent` arrives:

```
create context PowerOutage start PowerOutageEvent end pattern [PowerOnEvent -
> timer:interval(5)]
```

The following statement outputs the temperature during a power outage and for 5 seconds after the power comes on:

```
context PowerOutage select * from TemperatureEvent
```

To output only the last value when a context partition ends (terminates, expires), please read on to the description of output rate limiting.

The next statement creates a context `Every15Minutes` that starts immediately and lasts for 15 minutes, repeatedly allocating a new context partition at the end of 15 minute intervals:

```
create context Every15Minutes start @now end after 15 minutes
```

Note

If you specified an event filter or pattern as the end condition for a context partition, and statements that refer to the context specify an event filter or pattern that matches the same conditions, use @Priority to instruct the engine whether the context management or the statement evaluation takes priority (see below for configuring prioritized execution).

For example, if your context declaration looks like this:

```
create context MyCtx start MyStartEvent end MyEndEvent
```

And a statement managed by the context is this:

```
context MyCtx select count(*) as cnt from MyEndEvent output when
 terminated
```

By using `@Priority(1)` for create-context and `@Priority(0)` for the counting statement the counting statement does not count the last `MyEndEvent` since context partition management takes priority.

By using `@Priority(0)` for create-context and `@Priority(1)` for the counting statement the counting statement will count the last `MyEndEvent` since the statement evaluation takes priority.

4.2.6. Overlapping Context

This context initiates a new context partition when an initiating condition occurs, and terminates one or more context partitions when the terminating condition occurs. The engine maintains as many context partitions as the initiating condition fired, and discards context partitions that terminate when the termination condition fires.

The syntax for creating an overlapping context is as follows:

```
create context context_name
  initiated [by] [@now and] initiating_condition
  terminated [by] terminating_condition
```

The *context_name* you assign to the context can be any identifier.

Following the context name is the `initiated` keyword, optionally followed by `@now and` and followed by the initiating condition. It follows the `terminated` keyword followed by the terminating condition.

Both the initiating and terminating condition can be an event filter, a pattern, a crontab or a time period. The syntax of initiating and terminating conditions is described in *Section 4.2.7, “Context Conditions”*.

If you specified `@now and` before the initiating condition then the engine initiates a new context partition immediately. The `@now` is only allowed in conjunction with initiation conditions that specify a pattern, crontab or time period and not with event filters.

If you specified an event filter for the initiating condition, then the event that initiates a new context partition also counts towards the statement(s) that refer to that context. If you specified a pattern to initiate a new context partition, then the events that may constitute the pattern match can also count towards the statement(s) that refer to the context provided that `@inclusive` and event tags are both specified (see below).

The next statement creates a context `CtxTrainEnter` that allocates a new context partition when a train enters a station, and that terminates each context partition 5 minutes after the time the context partition was allocated:

```
create context CtxTrainEnter
  initiated by TrainEnterEvent as te
  terminated after 5 minutes
```

The context declared above assigns the stream name `te`. Thereby the initiating event's properties can be accessed, for example, by specifying `context.te.trainId`.

The following statement detects when a train enters a station as indicated by a `TrainEnterEvent`, but does not leave the station within 5 minutes as would be indicated by a matching `TrainLeaveEvent`:

```
context CtxTrainEnter
select t1 from pattern [
   t1=TrainEnterEvent -> timer:interval(5 min) and not TrainLeaveEvent(trainId
 = context.te.trainId)
```

```
  ]
```

Since the `TrainEnterEvent` that initiates a new context partition also counts towards the statement, the first part of the pattern (the `t1=TrainEnterEvent`) is satisfied by that initiating event.

The next statement creates a context `CtxEachMinute` that allocates a new context partition immediately and every 1 minute, and that terminates each context partition 1 minute after the time the context partition was allocated:

```
create context CtxEachMinute
  initiated @now and pattern [every timer:interval(1 minute)]
  terminated after 1 minutes
```

The statement above specifies `@now` to instruct the engine to allocate a new context partition immediately as well as when the pattern fires. Without the `@now` the engine would only allocate a new context partition when the pattern fires after 1 minute and every minute thereafter.

The following statement averages the temperature, starting anew every 1 minute and outputs the aggregate value continuously:

```
context CtxEachMinute select avg(temp) from SensorEvent
```

To output only the last value when a context partition ends (terminates, expires), please read on to the description of output rate limiting.

Note

If you specified an event filter or pattern as the termination condition for a context partition, and statements that refer to the context specify an event filter or pattern that matches the same conditions, use @Priority to instruct the engine whether the context management or the statement evaluation takes priority (see below for configuring prioritized execution). See the note above for more information.

4.2.6.1. Built-In Context Properties

The following context properties are available in your EPL statement when it refers to a context:

Table 4.4. Context Properties

Name	Description
`name`	The string-type context name.

Name	Description
`startTime`	The start time of the context partition.
`endTime`	The end time of the context partition. This field is only available in the case that it can be computed from the crontab or time period expression that is provided.

You may, for example, select the context properties as follows:

```
context NineToFive
select context.name, context.startTime, context.endTime from TrafficEvent(speed
 >= 100)
```

The following statement looks for the next train leave event for the same train id and selects a few of the context properties:

```
context CtxTrainEnter
select *, context.te.trainId, context.id, context.name
from TrainLeaveEvent(trainId = context.te.trainId)
```

4.2.7. Context Conditions

Context start/initiating and end/terminating conditions are for use with overlapping and non-overlapping contexts. Any combination of conditions may be specified.

4.2.7.1. Filter Context Condition

Use the syntax described here to define the stream that starts/initiates a context partition or that ends/terminates a context partition.

The syntax is:

```
event_stream_name [(filter_criteria)] [as stream_name]
```

The *event_stream_name* is either the name of an event type or name of an event stream populated by an insert into statement. The *filter_criteria* is optional and consists of a list of expressions filtering the events of the event stream, within parenthesis after the event stream name.

Two examples are:

```
// A non-overlapping context that starts when MyStartEvent arrives and ends when
 MyEndEvent arrives
create context MyContext start MyStartEvent end MyEndEvent
```

```
// An overlapping context where each MyEvent with level greater zero
// initiates a new context partition that terminates after 10 seconds
create context MyContext initiated MyEvent(level > 0) terminated after 10 seconds
```

You may correlate the start/initiating and end/terminating streams by providing a stream name following the `as` keyword, and by referring to that stream name in the filter criteria of the end condition.

Two examples that correlate the start/initiating and end/terminating condition are:

```
// A non-overlapping context that starts when MyEvent arrives
// and ends when a matching MyEvent arrives (same id)
create context MyContext
start MyEvent as myevent
end MyEvent(id=myevent.id)
```

```
// An overlapping context where each MyInitEvent initiates a new context
 partition
// that terminates when a matching MyTermEvent arrives
create context MyContext
initiated by MyInitEvent as e1
terminated by MyTermEvent(id=e1.id, level <> e1.level)
```

4.2.7.2. Pattern Context Condition

You can define a pattern that starts/initiates a context partition or that ends/terminates a context partition.

The syntax is:

```
pattern [pattern_expression] [@inclusive]
```

The *pattern_expression* is a pattern at *Chapter 6, EPL Reference: Patterns*.

Specify `@inclusive` after the pattern to have those same events that constitute the pattern match also count towards any statements that are associated to the context. You must also provide a tag for each event in a pattern that should be included.

Examples are:

```
// A non-overlapping context that starts when either StartEventOne or
 StartEventTwo arrive
// and that ends after 5 seconds.
// Here neither StartEventOne or StartEventTwo count towards any statements
// that are referring to the context.
create context MyContext
  start pattern [StartEventOne or StartEventTwo]
  end after 5 seconds
```

```
// Same as above.
// Here both StartEventOne or StartEventTwo do count towards any statements
// that are referring to the context.
create context MyContext
  start pattern [a=StartEventOne or b=StartEventTwo] @inclusive
  end after 5 seconds
```

```
// An overlapping context where each distinct MyInitEvent initiates a new context
// and each context partition terminates after 20 seconds
// We use @inclusive to say that the same MyInitEvent that fires the pattern
// also applies to statements that are associated to the context.
create context MyContext
  initiated by pattern [every-distinct(a.id, 20 sec) a=MyInitEvent]@inclusive
  terminated after 20 sec
```

```
// An overlapping context where each pattern match initiates a new context
// and all context partitions terminate when MyTermEvent arrives.
// The MyInitEvent and MyOtherEvent that trigger the pattern are themselves not
 included
// in any statements that are associated to the context.
create context MyContext
  initiated by pattern [every MyInitEvent -> MyOtherEvent where timer:within(5)]
  terminated by MyTermEvent
```

You may correlate the start and end streams by providing tags as part of the pattern, and by referring to the tag name(s) in the filter criteria of the end condition.

An example that correlates the start and end condition is:

```
//  A  non-overlapping  context  that  starts  when  either  StartEventOne  or
 StartEventTwo arrive
// and that ends when either a matching EndEventOne or EndEventTwo arrive
create context MyContext
  start pattern [a=StartEventOne or b=StartEventTwo]@inclusive
  end pattern [EndEventOne(id=a.id) or EndEventTwo(id=b.id)]
```

4.2.7.3. Crontab Context Condition

Crontab expression are described in *Section 6.6.2.2, “timer:at”*.

Examples are:

```
// A non-overlapping context started daily between 9 am to 5 pm
// and not started outside of these hours:
create context NineToFive start (0, 9, *, *, *) end (0, 17, *, *, *)
```

```
// An overlapping context where crontab initiates a new context every 1 minute
// and each context partition terminates after 10 seconds:
create context MyContext initiated (*, *, *, *, *) terminated after 10 seconds
```

4.2.7.4. Time Period Context Condition

You may specify a time period that the engine observes before the condition fires. Time period expressions are described in *Section 5.2.1, “Specifying Time Periods”*.

The syntax is:

```
after time_period_expression
```

Examples are:

```
// A non-overlapping context started after 10 seconds
// that ends 1 minute after it starts and that again starts 10 seconds thereafter.
create context NonOverlap10SecFor1Min start after 10 seconds end after 1 minute
```

```
// An overlapping context that starts a new context partition every 5 seconds
// and each context partition lasts 1 minute
create context Overlap5SecFor1Min initiated after 5 seconds terminated after 1
 minute
```

4.3. Context Nesting

A nested context is a context that is composed from two or more contexts.

The syntax for creating a nested context is as follows:

```
create context context_name
  context nested_context_name [as] nested_context_definition ,
context nested_context_name [as] nested_context_definition [, ...]
```

The *context_name* you assign to the context can be any identifier.

Following the context name is a comma-separated list of nested contexts. For each nested context specify the `context` keyword followed a nested context name and the nested context declaration. Any of the context declarations as outlined in *Section 4.2, "Context Declaration"* are allowed for nested contexts. The order of nested context declarations matters as outlined below.

The next statement creates a nested context `NineToFiveSegmented` that, between 9 am and 5 pm, allocates a new context partition for each customer id:

```
create context NineToFiveSegmented
  context NineToFive start (0, 9, *, *, *) end (0, 17, *, *, *),
  context SegmentedByCustomer partition by custId from BankTxn
```

The following statement refers to the nested context to compute a total withdrawal amount per account for each customer but only between 9 am and 5 pm:

```
context NineToFiveSegmented
select custId, account, sum(amount) from BankTxn group by account
```

Esper implements nested contexts as a context tree: The context declared first controls the lifecycle of the context(s) declared thereafter. Thereby, in the above example, outside of the 9am-to-5pm time the engine has no memory and consumes no resources in relationship to bank transactions or customer ids.

When combining segmented contexts, the set of context partitions for the nested context effectively is the Cartesian product of the partition sets of the nested segmented contexts.

When combining temporal contexts with other contexts, since temporal contexts may overlap and may terminate, it is important to understand that temporal contexts control the lifecycle of sub-contexts (contexts declared thereafter). The order of declaration of contexts in a nested context can thereby change resource usage and output result.

The next statement creates a context that allocates context partition only when a train enters a station and then for each hash of the tag id of a passenger as indicated by PassengerScanEvent events, and terminates all context partitions after 5 minutes:

```
create context CtxNestedTrainEnter
  context InitCtx initiated by TrainEnterEvent as te terminated after 5 minutes,
      context    HashCtx    coalesce    by    consistent_hash_crc32(tagId)    from
 PassengerScanEvent
    granularity 16 preallocate
```

In the example above the engine does not start tracking PassengerScanEvent events or hash codes or allocate context partitions until a TrainEnterEvent arrives.

4.3.1. Built-In Nested Context Properties

Context properties of all nested contexts are available for use. Specify `context.`*nested_context_name*`.`*property_name* or if nested context declaration provided stream names or tags for patterns then `context.`*nested_context_name*`.`*stream_name*`.`*property_name*.

For example, consider the `CtxNestedTrainEnter` context declared earlier. The following statement selects a few of the context properties:

```
context CtxNestedTrainEnter
select context.InitCtx.te.trainId, context.HashCtx.id,
  tagId, count(*) from PassengerScanEvent group by tagId
```

In a second example, consider the `NineToFiveSegmented` context declared earlier. The following statement selects a few of the context properties:

```
context NineToFiveSegmented
select  context.NineToFive.startTime,  context.SegmentedByCustomer.key1  from
 BankTxn
```

The following context properties are available in your EPL statement when it refers to a nested context:

Table 4.5. Nested Context Properties

Name	Description
`name`	The string-type context name.
`id`	The integer-type internal context id that the engine assigns to the context partition.

This example selects the nested context name and context partition id:

```
context NineToFiveSegmented select context.name, context.id from BankTxn
```

4.4. Partitioning Without Context Declaration

You do not need to declare a context to partition data windows, aggregation values or patterns themselves individually. You may mix-and-match partitioning as needed.

The table below outlines other partitioning syntax supported by EPL:

Table 4.6. Partition in EPL without the use of Context Declaration

Partition Type	Description	Example

Grouped Data Window	Partitions at the level of dat window, only applies to appende data window(s).
	
	Syntax: `std:groupby(...)`
		`// Length window of 2 events per customer`
		`select * from`
		`BankTxn.std:groupwin(custId).win:length(2)`

Grouped Aggregation	Partitions at the level c aggregation, only applies to an aggregations.
	
	Syntax: `group by ....`
		`select avg(price), window(*)`
		`from BankTxn group by custId`

Pattern	Partitions pattern subexpressions.
	
	Syntax: `every` or `every-distinc`
		`select * from pattern [`
		`every a=BankTxn -> BankTxn(custId = a.custId)...]`

Match-Recognize	Partitions match-recogniz patterns.
	
	Syntax: `partition by`
		`select * from match_recognize`
		`... partition by custId`

Join and Subquery	Partitions join and subqueries. Syntax: `where ...`	`select * from ... where a.custId = b.custId`

4.5. Output When Context Partition Ends

You may use output rate limiting to trigger output when a context partition ends, as further described in *Section 5.7, “Stabilizing and Controlling Output: the Output Clause”*.

Consider the fixed temporal context: A new context partition gets allocated at the designated start time and the current context partition ends at the designated end time. To trigger output when the context partition ends and before it gets removed, read on.

The same is true for the initiated temporal context: That context starts a new context partition when trigger events arrive or when a pattern matches. Each context partition expires (ends, terminates) after the specified time period passed. To trigger output at the time the context partition expires, read on.

You may use the `when terminated` syntax with output rate limiting to trigger output when a context partition ends. The following example demonstrates the idea by declaring an initiated temporal context.

The next statement creates a context `CtxEachMinute` that initiates a new context partition every 1 minute, and that expires each context partition after 5 minutes:

```
create context CtxEachMinute
initiated by pattern [every timer:interval(1 min)]
terminated after 5 minutes
```

The following statement computes an ongoing average temperature however only outputs the last value of the average temperature after 5 minutes when a context partition ends:

```
context CtxEachMinute
select context.id, avg(temp) from SensorEvent output snapshot when terminated
```

The `when terminated` syntax can be combined with other output rates.

The next example outputs every 1 minute and also when the context partition ends:

```
context CtxEachMinute
select context.id, avg(temp) from SensorEvent output snapshot every 1 minute
 and when terminated
```

In the case that the end/terminating condition of the context partition is an event or pattern, the context properties contain the information of the tagged events in the pattern or the single event that ended/terminated the context partition.

For example, consider the following context wherein the engine initializes a new context partition for each arriving `MyStartEvent` event and that terminates a context partition when a matching `MyEndEvent` arrives:

```
create context CtxSample
initiated by MyStartEvent as startevent
terminated by MyEndEvent(id = startevent.id) as endevent
```

The following statement outputs the id property of the initiating and terminating event and only outputs when a context partition ends:

```
context CtxSample
select context.startevent.id, context.endevent.id, count(*) from MyEvent
output snapshot when terminated
```

You may in addition specify a termination expression that the engine evaluates when a context partition terminates. Only when the terminaton expression evaluates to true does output occur. The expression may refer to built-in properties as described in *Section 5.7.1.1, “Controlling Output Using an Expression”*. The syntax is as follows:

```
...output when terminated and termination_expression
```

The next example statement outputs when a context partition ends but only if at least two events are available for output:

```
context CtxEachMinute
select * from SensorEvent output when terminated and count_insert >= 2
```

The final example EPL outputs when a context partition ends and sets the variable `myvar` to a new value:

```
context CtxEachMinute
select * from SensorEvent output when terminated then set myvar=3
```

4.6. Context and Named Window

Named windows are globally-visible data window that may be referred to by multiple statements. You may refer to named windows in statements that declare a context without any special considerations.

You may also create a named window and declare a context for the named window. In this case the engine in effect manages separate named windows, one for each context partition. Limitations apply in this case that we discuss herein.

For example, consider the 9 am to 5 pm fixed temoral context as shown earlier:

```
create context NineToFive start (0, 9, *, *, *) end (0, 17, *, *, *)
```

You may create a named window that only exists between 9 am and 5 pm:

```
context NineToFive create window SpeedingEvents1Hour.win:time(30 min) as
 TrafficEvent
```

You can insert into the named window:

```
insert into SpeedingEvents1Hour select * from TrafficEvent(speed > 100)
```

Any on-merge, on-select, on-update and on-delete statements must however declare the same context.

The following is not a valid statement as it does not declare the same context that was used to declare the named window:

```
// You must declare the same context for on-trigger statements
on TruncateEvent delete from SpeedingEvents1Hour
```

The following is valid:

```
context NineToFive on TruncateEvent delete from SpeedingEvents1Hour
```

For context declarations that require specifying event types, such as the hash segmented context and keyed segmented context, please provide the named window underlying event type.

The following sample EPL statements define a type for the named window, declare a context and associate the named window to the context:

```
create schema ScoreCycle (userId string, keyword string, productId string, score
 long)
```

```
create context HashByUserCtx as
  coalesce by consistent_hash_crc32(userId) from ScoreCycle granularity 64
```

```
context  HashByUserCtx  create  window  ScoreCycleWindow.std:unique(productId,
 keyword) as ScoreCycle
```

4.7. Operations on Specific Context Partitions

Selecting specific context partitions and interrogating context partitions is useful for:

1. Iterating a specific context partition or a specific set of context partitions. Iterating a statement is described in *Section 14.3.5, “Using Iterators”*.

2. Executing an on-demand (fire-and-forget) query against specific context partition(s). On-demand queries are described in *Section 14.5, “On-Demand Fire-And-Forget Query Execution”*.

Esper provides APIs to identify, filter and select context partitions for statement iteration and on-demand queries. The APIs are described in detail at *Section 14.19, “Context Partition Selection”*.

For statement iteration, your application can provide context selector objects to the `iterate` and `safeIterate` methods on `EPStatement`. If your code does not provide context selectors the iteration considers all context partitions. At the time of iteration, the engine obtains the current set of context partitions and iterates each independently. If your statement has an order-by clause, the order-by clause orders within the context partition and does not order across context partitions.

For on-demand queries, your application can provide context selector objects to the `executeQuery` method on `EPRuntime` and to the `execute` method on `EPOnDemandPreparedQuery`. If your code does not provide context selectors the on-demand query considers all context partitions. At the time of on-demand query execution, the engine obtains the current set of context partitions and queries each independently. If the on-demand query has an order-by clause, the order-by clause orders within the context partition and does not order across context partitions.

Chapter 5. EPL Reference: Clauses

5.1. 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.

The purpose of the `UPDATE` clause is to update event properties. Update takes place before an event applies to any selecting statements or pattern statements.

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`, `win:ext_timed_batch`, `ext:sort`, `ext:rank`, `ext:time_order`, `std:unique`, `std:groupwin`, `std:lastevent`, `std:firstevent`, `std:firstunique`, `win:firstlength`, `win:firsttime`.
2. Views that derive statistics: `std:size`, `stat:uni`, `stat:linest`, `stat:correl`, `stat:weighted_avg`.

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 `ON MERGE` clause to atomically merge events into named window state, the `INSERT INTO` clause to insert data into a named window, the `ON DELETE` clause to remove events from a named window, the `ON UPDATE` clause to update events held by a named window and the `ON SELECT` clause to perform a query triggered by a pattern or arriving event 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.

EPL allows execution of on-demand (fire-and-forget, non-continuous, triggered by API) queries against named windows through the runtime API. The query engine automatically indexes named window data for fast access by `ON SELECT/UPDATE/INSERT/DELETE` without the need to create an index explicitly. For fast on-demand query execution via runtime API use the `CREATE INDEX` syntax to create an explicit index.

Use `CREATE SCHEMA` to declare an event type.

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.

5.2. 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.

```
[annotations]
[expression_declarations]
[context context_name]
[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]
[limit num_rows]
```

5.2.1. 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 : [year-part] [month-part] [week-part] [day-part] [hour-part]
      [minute-part] [seconds-part] [milliseconds-part]

year-part : (number|variable_name) ("years" | "year")
month-part : (number|variable_name) ("months" | "month")
week-part : (number|variable_name) ("weeks" | "week")
day-part : (number|variable_name) ("days" | "day")
hour-part : (number|variable_name) ("hours" | "hour")
minute-part : (number|variable_name) ("minutes" | "minute" | "min")
seconds-part : (number|variable_name) ("seconds" | "second" | "sec")
milliseconds-part : (number|variable_name) ("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
1 year
1 year 1 month
```

Variable names and substitution parameters '?' for prepared statements are also allowed as part of a time period expression.

A unit in the month part is equivalent to 30 days.

5.2.2. 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
```

5.2.3. Reserved Keywords

Certain words such as `select`, `delete` or `set` are reserved and may not be used as identifiers. Please consult *Appendix B, Reserved Keywords* for the list of reserved keywords and permitted keywords.

Names of built-in functions and certain auxiliary keywords are permitted as event property names and in the rename syntax of the `select` clause. For example, `count` is acceptable.

Consider the example below, which assumes that `'last'` is an event property of MyEvent:

```
// valid
select last, count(*) as count from MyEvent
```

This example shows an incorrect use of a reserved keyword:

```
// invalid
select insert from MyEvent
```

EPL offers an escape syntax for reserved keywords: Event properties as well as event or stream names may be escaped via the backwards apostrophe ` (ASCII 96) character.

The next example queries an event type by name `Order` (a reserved keyword) that provides a property by name `insert` (a reserved keyword):

```
// valid
select `insert` from `Order`
```

5.2.4. Escaping Strings

You may surround string values by either double-quotes (`"`) or single-quotes (`'`). When your string constant in an EPL statement itself contains double quotes or single quotes, you must escape the quotes.

Double and single quotes may be escaped by the backslash (`\`) character or by unicode notation. Unicode 0027 is a single quote (`'`) and 0022 is a double quote (`"`).

Escaping event property names is described in *Section 2.2.1, “Escape Characters”*.

The sample EPL below escapes the single quote in the string constant `John's`, and filters out order events where the name value matches:

```
select * from OrderEvent(name='John\'s')
// ...equivalent to...
select * from OrderEvent(name='John\u0027s')
```

The next EPL escapes the string constant `Quote "Hello"`:

```
select * from OrderEvent(description like "Quote \"Hello\"")
// is equivalent to
select * from OrderEvent(description like "Quote \u0022Hello\u0022")
```

When building an escape string via the API, escape the backslash, as shown in below code snippet:

```
epService.getEPAdministrator().createEPL("select * from OrderEvent(name='John\
\'s')");
// ... and for double quotes...
epService.getEPAdministrator().createEPL("select * from OrderEvent(
  description like \"Quote \\\"Hello\\\"\")");
```

5.2.5. Data Types

EPL honors all Java built-in primitive and boxed types, including `java.math.BigInteger` and `java.math.BigDecimal`.

EPL also follows Java standards in terms of widening, performing widening automatically in cases where widening type conversion is allowed without loss of precision, for both boxed and primitive types and including `BigInteger` and `BigDecimal`:

1. byte to short, int, long, float, double, BigInteger or BigDecimal
2. short to int, long, float, or double, BigInteger or BigDecimal
3. char to int, long, float, or double, BigInteger or BigDecimal
4. int to long, float, or double, BigInteger or BigDecimal
5. long to float or double, BigInteger or BigDecimal
6. float to double or BigDecimal
7. double to BigDecimal

In cases where loss of precision is possible because of narrowing requirements, EPL compilation outputs a compilation error.

EPL supports casting via the `cast` function.

EPL returns double-type values for division regardless of operand type. EPL can also be configured to follow Java rules for integer arithmetic instead as described in *Section 15.4.22, "Engine Settings related to Expression Evaluation"*.

Division by zero returns positive or negative infinity. Division by zero can be configured to return null instead.

5.2.5.1. Data Type of Constants

An EPL constant is a number or a character string that indicates a fixed value. Constants can be used as expressions in many EPL statements, including variable assignment and case-when statements. They can also be used as parameter values for many built-in objects and clauses. Constants are also called literals.

EPL supports the standard SQL constant notation as well as Java data type literals.

The following are types of EPL constants:

Table 5.1. Types of EPL constants

Type	Description	Examples

string	A single character to an unlimited number o characters. Valid delimiters are the single quot (') or double quote (").	`select 'volume' as field1,`
		`   "sleep" as field2,`
		`  "\u0041" as unicodeA`

boolean	A boolean value.	`select true as field1,`
		`  false as field2`

Type	Description	Examples

integer	An integer value (4 byte).	`select 1 as field1,`
		`  -1 as field2,`
		`  1e2 as field3`

long	A long value (8 byte). Use the "L" o "l" (lowercase L) suffix.	`select 1L as field1,`
		`  1l as field2`

double	A double-precision 64-bit IEEE 754 floatin point.	`select 1.67 as field1,`
		`  167e-2 as field2,`
		`  1.67d as field3`

float	A single-precision 32-bit IEEE 754 floating poin Use the "f" suffix.	`select 1.2f as field1,`
		`  1.2F as field2`

byte	A 8-bit signed two's complement integer.	`select 0x10 as field1`

EPL does not have a single-byte character data type for its literals. Single character literals are treated as string.

Internal byte representation and boundary values of constants follow the Java standard.

5.2.5.2. BigInteger and BigDecimal

EPL automatically performs widening of numbers to `BigInteger` and `BigDecimal` as required, and employs the respective `equals`, `compareTo` and arithmetic methods provided by `BigInteger` and `BigDecimal`.

To explicitly create `BigInteger` and `BigDecimal` constants in EPL, please use the cast syntax : `cast(`*value*`, BigInteger)`.

Note that since `BigDecimal.valueOf(1.0)` is not the same as `BigDecimal.valueOf(1)` (in terms of equality through `equals`), care should be taken towards the consistent use of scale.

When using aggregation functions for `BigInteger` and `BigDecimal` values, please note these limitations:

1. The `median`, `stddev` and `avedev` aggregation functions operate on the double value of the object and return a double value.
2. All other aggregation functions return `BigDecimal` or `BigInteger` values (except `count`).

For `BigDecimal` precision and rounding, please see *Section 15.4.22.6, “Math Context”*.

5.2.6. Using Constants and Enum Types

This chapter is about Java language constants and enum types and their use in EPL expressions.

Java language constants are public static final fields in Java that may 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 enum type value:

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

Event properties can also be passed to enum type functions or compared to an enum type method result:

```
select * from MyEvent where somevalue = EnumClass.ENUM_VALUE_1.getSomeValue()
  or EnumClass.ENUM_VALUE_2.analyze(someothervalue)
```

Enum types have a `valueOf` method that returns the enum type value:

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

If your application does not import, through configuration, the package that contains the enumeration class, then it must also specify the package name of the class. Enum types that are inner classes must be qualified with `$` following Java conventions.

For example, the Color enum type as an inner class to `MyEvent` in package `org.myorg` can be referenced as shown:

```
select * from MyEvent(enumProp=org.myorg.MyEvent$Color.GREEN).std:firstevent()
```

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
```

Chaining instance methods is supported as this example shows:

```
select myevent.getComputerFor('books', 'movies').calculate() as result
from MyEvent as myevent
```

5.2.7. Annotation

An annotation is an addition made to information in a statement. Esper provides certain built-in annotations for defining statement name, adding a statement description or for tagging statements such as for managing statements or directing statement output. Other then the built-in annotations, applications can provide their own annotation classes that the EPL compiler can populate.

An annotation is part of the statement text and precedes the EPL select or pattern statement. Annotations are therefore part of the EPL grammar. The syntax for annotations follows the host language (Java, .NET) annotation syntax:

```
@annotation_name [(annotation_parameters)]
```

An annotation consists of the annotation name and optional annotation parameters. The *annotation_name* is the simple class name or fully-qualified class name of the annotation class. The optional *annotation_parameters* are a list of key-value pairs following the syntax:

```
@annotation_name (attribute_name = attribute_value, [name=value, ...])
```

The *attribute_name* is an identifier that must match the attributes defined by the annotation class. An *attribute_value* is a constant of any of the primitive types or string, an array, an enum type value or another (nested) annotation. Null values are not allowed as annotation attribute values. Enumeration values are supported in EPL statements and not support in statements created via the `createPattern` method.

Use the `getAnnotations` method of `EPStatement` to obtain annotations provided via statement text.

5.2.7.1. Application-Provided Annotations

Your application may provide its own annotation classes. The engine detects and populates annotation instances for application annotation classes.

To enable the engine to recognize application annotation classes, your annotation name must include the package name (i.e. be fully-qualified) or your engine configuration must import the annotation class or package via the configuration API.

For example, assume that your application defines an annotation in its application code as follows:

```
public @interface ProcessMonitor {
  String processName();
  boolean isLongRunning default false;
  int[] subProcessIds;
}
```

Shown next is an EPL statement text that utilizes the annotation class defined earlier:

```
@ProcessMonitor(processName='CreditApproval',
  isLongRunning=true, subProcessIds = {1, 2, 3} )

select count(*) from ProcessEvent(processId in (1, 2, 3).win:time(30)
```

Above example assumes the `ProcessMonitor` annotation class is imported via configuration XML or API. Here is an example API call to import annotations provided by a package `com.mycompany.myannotations`:

```
epService.getEPAdministrator().getConfiguration().addImport("com.mycompany.myannotations.*");
```

5.2.7.2. Built-In Annotations

The list of built-in EPL annotations is:

Table 5.2. Built-In EPL Annotations

Name	Purpose and Attributes	Example
Name	Provides a statement name. Attribute are: value : Statement name.	`@Name("MyStatementName")`
Description	Provides a statement textual descriptior Attributes are: value : Statement description.	`@Description("Place statement description here.")`
Tag	For tagging a statement with additiona information. Attributes are: name : Tag name. value : Tag value.	`@Tag(name="MyTagName", value="MyTagValue")`
Priority	Applicable when an event (or schedule matches filter criteria for multipl statements: Defines the order c statement processing (requires an engine-level setting).	`@Priority(10)`

Name	Purpose and Attributes	Example
	Attributes are: value : priority value.	
Drop	Applicable when an event (or schedule matches filter criteria for multipl statements, drops the event afte. processing the statement (requires an engine-level setting). No attributes.	`@Drop`
Hint	For providing one or more hints toward how the engine should execute statement. Attributes are: value : A comma-separated list of one or more case-insensitive keywords.	`@Hint('ITERATE_ONLY')`

Hook	Use this annotation to register one c more statement-specific hooks providin a hook type for each individual hook, suc as for SQL parameter, column or rov conversion.
	
	Attributes are the hook `type` and the `hook` itself (usually a import or class name):
		`@Hook(type=HookType.SQLCOL,`
		
		`hook='MyDBTypeConvertor')`

Audit	Causes the engine to output detaile processing information for a statement. optional value : A comma-separated list of one or more case-insensitive keywords.	`@Audit`
EventRepresentation	Causes the engine to use object-arra event representation, if possible, fc output and internal event types.	`@EventRepresentation(array=true)`

The following example statement text specifies some of the built-in annotations in combination:

```
@Name("RevenuePerCustomer")
@Description("Outputs revenue per customer considering all events encountered
 so far.")
@Tag(name="grouping", value="customer")

select customerId, sum(revenue) from CustomerRevenueEvent
```

5.2.7.3. @Name

Use the @Name EPL annotation to specify a statement name within the EPL statement itself, as an alternative to specifying the statement name via API.

If your application is also providing a statement name through the API, the statement name provided through the API overrides the annotation-provided statement name.

Example:

```
@Name("SecurityFilter1") select * from SecurityFilter(ip="127.0.0.1")
```

5.2.7.4. @Description

Use the @Description EPL annotation to add a statement textual description.

Example:

```
@Description('This statement filters localhost.') select * from
 SecurityFilter(ip="127.0.0.1")
```

5.2.7.5. @Tag

Use the @Tag EPL annotation to tag statements with name-value pairs, effectively adding a property to the statement. The attributes `name` and `value` are of type string.

Example:

```
@Tag(name='ip_sensitive', value='Y')
@Tag(name='author', value='Jim')
select * from SecurityFilter(ip="127.0.0.1")
```

5.2.7.6. @Priority

This annotation only takes effect if the engine-level setting for prioritized execution is set via configuration, as described in *Section 15.4.23, “Engine Settings related to Execution of Statements”*.

Use the @Priority EPL annotation to tag statements with a priority value. The default priority value is zero (0) for all statements. When an event (or single timer execution) requires processing the event for multiple statements, processing begins with the highest priority statement and ends with the lowest-priority statement.

Example:

```
@Priority(10) select * from SecurityFilter(ip="127.0.0.1")
```

5.2.7.7. @Drop

This annotation only takes effect if the engine-level setting for prioritized execution is set via configuration, as described in *Section 15.4.23, “Engine Settings related to Execution of Statements”*.

Use the @Drop EPL annotation to tag statements that preempt all other same or lower-priority statements. When an event (or single timer execution) requires processing the event for multiple statements, processing begins with the highest priority statement and ends with the first statement marked with @Drop, which becomes the last statement to process that event.

Unless a different priority is specified, the statement with the @Drop EPL annotation executes at priority 1. Thereby @Drop alone is an effective means to remove events from a stream.

Example:

```
@Drop select * from SecurityFilter(ip="127.0.0.1")
```

5.2.7.8. @Hint

A hint can be used to provide tips for the engine to affect statement execution. Hints change performance or memory-use of a statement but generally do not change its output.

The string value of a `Hint` annotation contains a keyword or a comma-separated list of multiple keywords. Hint keywords are case-insensitive. A list of hints is available in *Section 20.2.23, “Consider using Hints”*.

Example:

```
@Hint('disable_reclaim_group')
select  ipaddress,  count(*)  from  SecurityFilter.win:time(60  sec)  group  by
 ipaddress
```

5.2.7.9. @Hook

A hook is for attaching a callback to a statement.

The type value of a `@Hook` annotation defines the type of hook and the `hook` value is an imported or fully-qualified class name providing the callback implementation.

5.2.7.10. @Audit

Causes the engine to output detailed information about the statements processing. Described in more detail at *Section 16.3.1, “@Audit Annotation”*.

5.2.7.11. @EventRepresentation

Use the `@EventRepresentation` annotation with `create schema` and `create window` statements to instruct the engine to use a specific event representation for the schema or named window.

Use the `@EventRepresentation` annotation with `select` statements to instruct the engine to use a specific event representation for output events.

When no `@EventRepresentation` annotation is specified, the engine uses the default event representation as configured, see *Section 15.4.11.1, “Default Event Representation”*.

Use `@EventRepresentation(array=true)` to instruct the engine to use object-array events.

Use `@EventRepresentation(array=false)` to instruct the engine to use Map events.

5.2.8. Expression Declaration

An EPL statement can contain expression declarations. Expressions that are common to multiple places in the same EPL statement can be moved to a named expression declaration and reused within the same statement without duplicating the expression itself.

For declaring expressions that are visible across multiple EPL statements i.e. globally visible expressions please consult *Section 5.19.1, “Declaring a Global Expression”* that explains the `create expression` clause.

An expression declaration follows the lambda-style expression syntax. This syntax was chosen as it typically allows for a shorter and more concise expression body that can be easier to read then most procedural code.

The syntax for an expression declaration is:

```
expression expression_name { expression_body }
```

An expression declaration consists of the expression name and an expression body. The *expression_name* is any identifier. The *expression_body* contains optional parameters and the expression. The parameter types and the return type of the expression is determined by the engine and do not need to be specified.

Parameters to a declared expression can be a stream name, pattern tag name or wildcard (`*`). Use wildcard to pass the event itself to the expression. In a join or subquery, or more generally in an expression where multiple streams or pattern tags are available, the EPL must specify the stream name or pattern tag name and cannot use wildcard.

In the expression body the => lambda operator reads as "goes to". The left side of the lambda operator specifies the input parameters (if any) and the right side holds the expression. The lambda expression `x => x * x` is read "x goes to x times x".

In the expression body, if your expression takes no parameters, you may simply specify the expression and do not need the => lambda operator.

If your expression takes one parameters, specify the input parameter name followed by the => lambda operator and followed by the expression. The synopsis for use with a single input parameter is:

```
expression_body:   input_param_name => expression
```

If your expression takes two or more parameters, specify the input parameter names in parenthesis followed by the => lambda operator followed by the expression. The synopsis for use with a multiple input parameter is:

```
expression_body:   (input_param [,input_param [,...]]) => expression
```

The following example declares an expression that returns two times PI (ratio of the circumference of a circle to its diameter) and demonstrates its use in a select-clause:

```
expression twoPI { Math.PI * 2} select twoPI() from SampleEvent
```

The next example declares an expression that accepts one parameter: a MarketData event. The expression computes a new "mid" price based on the buy and sell price:

```
expression midPrice { x => (x.buy + x.sell) / 2 }
select midPrice(md) from MarketDataEvent as md
```

The variable name can be left off if event property names resolve without ambiguity.

This example EPL removes the variable name `x`:

```
expression midPrice { x => (buy + sell) / 2 }
select midPrice(md) from MarketDataEvent as md
```

The next example EPL specifies wildcard instead:

```
expression midPrice { x => (buy + sell) / 2 }
select midPrice(*) from MarketDataEvent
```

A further example that demonstrates two parameters is listed next. The example joins two streams and uses the price value from MarketDataEvent and the sentiment value of NewsEvent to compute a weighted sentiment:

```
expression weightedSentiment { (x, y) => x.price * y.sentiment }
select weightedSentiment(md, news)
from MarketDataEvent.std:lastevent() as md, NewsEvent.std:lastevent() news
```

Any expression can be used in the expression body including aggregations, variables, subqueries or further declared expressions. Sub-queries, when used without `in` or `exists`, must be placed within parenthesis.

An example subquery within a declared expression is shown next:

```
expression newsSubq(md) {
   (select sentiment from NewsEvent.std:unique(symbol) where symbol = md.symbol)
}
select newsSubq(mdstream)
from MarketDataEvent mdstream
```

When using declared expressions please note these limitations:

1. Parameters to a declared expression can only be a stream name, pattern tag name or wildcard (`*`).

The following scope rules apply for declared expressions:

1. The scope of the expression body of a declared expression only includes the parameters explicitly listed.

5.2.9. Script Declaration

Esper allows the use of scripting languages within EPL. Any scripting language that supports JSR 223 and also the MVEL scripting language can be specified in EPL.

Please see *Chapter 18, Script Support* for more information.

5.2.10. Referring to a Context

You may refer to a context in the EPL text by specifying the `context` keyword followed by a context name. Context are described in more detail at *Chapter 4, Context and Context Partitions*

The effect of referring to a context is that your statement operates according to the context dimensional information as declared for the context.

The synopsis is:

```
... context context_name ...
```

You may refer to a context in all statements except for the following types of statements:

1. `create schema` for declaring event types.
2. `create variable` for declaring a variable.
3. `create index` for creating an index on a named window.
4. `update istream` for updating insert stream events.

5.3. 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 15.4.17, “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] [distinct] * | 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.

The `distinct` keyword outputs only unique rows depending on the column list you have specified after it. It must occur after the `select` and after the optional stream keywords, as described in more detail below.

5.3.1. 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)
```

You may well be asking: Why does the statement specify a time window here? First, the statement is meant to demonstrate the use of `*` wildcard. When the engine pushes statement results to your listener and as the statement does not select remove stream events via `rstream` keyword, the listener receives only new events and the time window could be left off. By adding the time window the pull API (iterator API or JDBC driver) returns the last 30 seconds of events.

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
```

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 name in the `from` clause.

5.3.2. 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, stddev, stddevpa
from StockTick(symbol='IBM').win:length(100).stat:uni(volume)
```

5.3.3. 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)
```

5.3.4. 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 resulting column.

```
select volume * price as volPrice from StockTick
```

Identifiers cannot contain the "." (dot) character, i.e. "vol.price" is not a valid identifier for the rename syntax.

5.3.5. 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 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 name as a prefix:

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

```
where news.symbol = tick.symbol
```

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 name] 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.

The stream name itself, as further described in *Section 5.4.5, “Using the Stream Name”*, may be used within expressions or alone.

This example passes events to a user-defined function named `compute` and also shows `insert-into` to populate an event stream of combined events:

```
insert into TickNewStream select tick, news, MyLib.compute(news, tick) as result
from StockTick.win:time(10) as tick, News.win:time(10) as news
where tick.symbol = news.symbol
```

```
// second statement that uses the TickNewStream stream
select tick.price, news.text, result from TickNewStream
```

In summary, the *stream_name.** streamname wildcard syntax can be used to select a stream as the underlying event or as a property, but cannot appear within an expression. While the *stream_name* syntax (without wildcard) always selects a property (and not as an underlying event), and can occur anywhere within an expression.

5.3.6. 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)]
```

A tag name as specified in a pattern is a valid expression itself. This example uses the `insert into` clause to make available the events matched by a pattern to further statements:

```
// make a new stream of ticks and news available
insert into StockTickAndNews
select    tick,    news    from    pattern    [every    tick=StockTick    ->
 news=News(symbol=tick.symbol)]
```

```
// second statement to select from the stream of ticks and news
select tick.symbol, tick.price, news.text from StockTickAndNews
```

5.3.7. 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 the `irstream` keyword occurs in the `select` clause. This behavior can be changed via engine-wide configuration as described in *Section 15.4.17, “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.

5.3.8. Qualifying property names and stream names

Property or column names can optionally be qualified by a stream name and the provider URI. The syntax is:

```
[[provider_URI.]stream_name.]property_name
```

The *provider_URI* is the URI supplied to the `EPServiceProviderManager` class, or the string `default` for the default provider.

This example assumes the provider is the default provider:

```
select MyEvent.myProperty from MyEvent
// ... equivalent to ...
select default.MyEvent.myProperty from MyEvent
```

Stream names can also be qualified by the provider URI. The syntax is:

```
[provider_URI.]stream_name
```

The next example assumes a provider URI by name of `Processor`:

```
select Processor.MyEvent.myProperty from Processor.MyEvent
```

5.3.9. Select `Distinct`

The optional `distinct` keyword removes duplicate output events from output. The keyword must occur after the `select` keyword and after the optional `irstream` keyword.

The `distinct` keyword in your `select` instructs the engine to consolidate, at time of output, the output event(s) and remove output events with identical property values. Duplicate removal only takes place when two or more events are output together at any one time, therefore `distinct` is typically used with a batch data window, output rate limiting, on-demand queries, on-select or iterator pull API.

If two or more output event objects have same property values for all properties of the event, the `distinct` removes all but one duplicated event before outputting events to listeners. Indexed, nested and mapped properties are considered in the comparison, if present in the output event.

The next example outputs sensor ids of temperature sensor events, but only every 10 seconds and only unique sensor id values during the 10 seconds:

```
select distinct sensorId from TemperatureSensorEvent output every 10 seconds
```

Use `distinct` with wildcard (`*`) to remove duplicate output events considering all properties of an event.

This example statement outputs all distinct events either when 100 events arrive or when 10 seconds passed, whichever occurs first:

```
select distinct * from TemperatureSensorEvent.win:time_length_batch(10, 100)
```

When selecting nested, indexed, mapped or dynamic properties in a `select` clause with `distinct`, it is relevant to know that the comparison uses hash code and the Java `equals` semantics.

5.3.10. Transposing an Expression Result to a Stream

For transposing an instance of a Java object returned by an expression to a stream use the transpose function as described in *Section 9.4, “Select-Clause transpose Function”*.

5.3.11. Selecting EventBean instead of Underlying Event

By default, for certain select-clause expressions that output events or a collection of events, the engine outputs the underlying event objects. With outputs we refer to the data passed to listeners, subscribers and inserted-into into another stream via insert-into.

The select-clause expressions for which underlying event objects are output by default are:

- Event Aggregation Functions (including extension API)
- The `previous` family of single-row functions
- Subselects that select events
- Declared expressions and enumeration methods that operate on any of the above

To have the engine output `EventBean` instance(s) instead, add `@eventbean` to the relevant expressions of the `select`-clause.

The sample EPL shown below outputs current data window contents as `EventBean` instances into the stream `OutStream`, thereby statements consuming the stream may operate on such instances:

```
insert into OutStream
select prevwindow(s0) @eventbean as win
from MyEvent.win:length(2) as s0
```

The next EPL consumes the stream and selects the last event:

```
select win.lastOf() from OutStream
```

It is not necessary to use `@eventbean` if an event type by the same name (`OutStream` in the example) is already declared and a property exist on the type by the same name (`win` in this example) and the type of the property is the event type (`MyEvent` in the example) returned by the expression. This is further described in *Section 5.10.8, “Select-Clause Expression And Inserted-Into Column Event Type”*.

5.4. 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` keyword.

```
from stream_def [as name] [unidirectional] [retain-union | retain-
intersection]
    [, 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 and the `unidirectional` keyword are described in more detail in *Section 5.12, “Joining Event Streams”*.

Esper supports joins against relational databases for access to historical or reference data as explained in *Section 5.13, “Accessing Relational Data via SQL”*. Esper can also join results returned by an arbitrary method invocation, as discussed in *Section 5.14, “Accessing 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.

For all streams with the exception of historical sources your query may employ data window views as outlined below. The `retain-intersection` (the default) and `retain-union` keywords build a union or intersection of two or more data windows as described in *Section 5.4.4, “Multiple Data Window Views”*.

5.4.1. Filter-based Event Streams

The *stream_def* syntax for a filter-based event stream is as below:

```
event_stream_name [(filter_criteria)] [contained_selection] [.view_spec]
 [.view_spec] [...]
```

The *event_stream_name* is either the name of an event type or name of an event stream populated by an `insert into` statement or the name of a named window.

The *filter_criteria* is optional and consists of a list of expressions filtering the events of the event stream, within parenthesis after the event stream name.

The *contained_selection* is optional and is for use with coarse-grained events that have properties that are themselves one or more events, see *Section 5.20, “Contained-Event Selection”* for the synopsis and examples.

The *view_spec* are optional view specifications, which are combinable definitions for retaining events and for deriving information from events.

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.

5.4.1.1. 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
```

5.4.1.2. 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 and especially in the presence of a larger number of filters or statements:

- 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
- single-row functions that have been registered and are invoked via function name (see user-defined functions) and that either return a boolean value or that have their return value compared to a constant

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, especially if your application creates a large number of statements or requires many similar filters. 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.

5.4.1.3. 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)
```

5.4.1.4. 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'))
```

5.4.1.5. Filter Limitations

The following restrictions apply to filter criteria:

- Range and comparison operators require the event property to be of a numeric or string 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.

5.4.2. 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))`.

5.4.3. Specifying Views

Views are used to specify an expiry policy for events (data window views) and also to derive data. Views can be staggered onto each other. See the section *Chapter 12, EPL Reference: Views* on the views available that also outlines the different types of views: Data Window views and Derived-Value views.

Views can optionally take one or more parameters. These parameters are expressions themselves that may consist of any combination of variables, arithmetic, user-defined function or substitution parameters for prepared statements, for example.

The example statement below outputs a count per expressway for car location events (contains information about the location of a car on a highway) of the last 60 seconds:

```
select expressway, count(*) from CarLocEvent.win:time(60)
group by expressway
```

The next example serves to show staggering of views. It uses the `std:groupwin` view to create a separate length window per car id:

```
select cardId, expressway, direction, segment, count(*)
from CarLocEvent.std:groupwin(carId).win:length(4)
group by carId, expressway, direction, segment
```

The first view `std:groupwin(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 separate length window for each car id. The 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.

Note that the `group by` syntax is generally preferable over `std:groupwin` for grouping information as it is SQL-compliant, easier to read and does not create a separate data window per group. The `std:groupwin` in above example creates a separate data window (length window in the example) per group, demonstrating staggering views.

When views are staggered onto each other as a chain of views, then the insert and remove stream received by each view is the insert and remove stream made available by the view (or stream) earlier in the chain.

The special keep-all view keeps all events: It does not provide a remove stream, i.e. events are not removed from the keep-all view unless by means of the `on-delete` syntax or by revision events.

5.4.4. Multiple Data Window Views

Data window views provide an expiry policy that indicates when to remove events from the data window, with the exception of the keep-all data window which has no expiry policy and the `std:groupwin` grouped-window view for allocating a new data window per group.

EPL allows the freedom to use multiple data window views onto a stream and thus combine expiry policies. Combining data windows into an intersection (the default) or a union can achieve a useful strategy for retaining events and expiring events that are no longer of interest. Named windows and the `on-delete` syntax provide an additional degree of freedom.

In order to combine two or more data window views there is no keyword required. The *retain-intersection* keyword is the default and the *retain-union* keyword may instead be provided for a stream.

The concept of union and intersection come from Set mathematics. In the language of Set mathematics, two sets A and B can be "added" together: The intersection of A and B is the set of all things which are members of both A and B, i.e. the members two sets have "in common". The union of A and B is the set of all things which are members of either A or B.

Use the *retain-intersection* (the default) keyword to retain an intersection of all events as defined by two or more data windows. All events removed from any of the intersected data windows are entered into the remove stream. This is the default behavior if neither retain keyword is specified.

Use the *retain-union* keyword to retain a union of all events as defined by two or more data windows. Only events removed from all data windows are entered into the remove stream.

The next example statement totals the price of OrderEvent events in a union of the last 30 seconds and unique by product name:

```
select  sum(price)  from  OrderEvent.win:time(30  sec).std:unique(productName)
 retain-union
```

In the above statement, all OrderEvent events that are either less then 30 seconds old or that are the last event for the product name are considered.

Here is an example statement totals the price of OrderEvent events in an intersection of the last 30 seconds and unique by product name:

```
select  sum(price)  from  OrderEvent.win:time(30  sec).std:unique(productName)
 retain-intersection
```

In the above statement, only those OrderEvent events that are both less then 30 seconds old and are the last event for the product name are considered. The number of events that the engine retains is the number of unique events per product name in the last 30 seconds (and not the number of events in the last 30 seconds).

For an intersection the engine retains the minimal number of events representing that intersection. Thus when combining a time window of 30 seconds and a last-event window, for example, the number of events retained at any time is zero or one event (and not 30 seconds of events).

When combining a batch window into an intersection with another data window the combined data window gains batching semantics: Only when the batch criteria is fulfilled does the engine provide the batch of intersecting insert stream events. Multiple batch data windows may not be combined into an intersection.

In below table we provide additional examples for data window intersections:

Table 5.3. Intersection Data Window Examples

Example	Description
`win:time(30).std:firstunique(keys)`	Retains 30 seconds of events unique per `keys` value (first event per value).
`win:firstlength(3).std:firstunique(keys)`	Retains the first 3 events that are also unique per `keys` value.
`win:time_batch(N seconds).std:unique(keys)`	Posts a batch every N seconds that contains the last of each unique event per `keys` value.
`win:time_batch(N seconds).std:firstunique(keys)`	Posts a batch every N seconds that contains the first of each unique event per `keys` value.
`win:length_batch(N).std:unique(keys)`	Posts a batch of unique events (last event per value) when N unique events per `keys` value are encountered.
`win:length_batch(N).std:firstunique(keys)`	Posts a batch of unique events (first event per value) when N unique events per `keys` value are encountered.

For advanced users and for backward compatibility, it is possible to configure Esper to allow multiple data window views without either of the `retain` keywords, as described in *Section 15.4.12.2, "Configuring Multi-Expiry Policy Defaults"*.

5.4.5. 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
```

Instance methods may also be chained: Your EPL may invoke a method on the result returned by a method invocation.

Assume that your product event exposes a method `getZone` which returns a zone object. Assume that the Zone class declares a method `checkZone`. This example statement invokes a method chain:

```
select prod.getZone().checkZone("zone 1") from ProductEvent as prod
```

5.5. 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 5.12, “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
```

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

5.6.1. 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.

5.6.2. 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 aggregate_free_expression [, aggregate_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
3. When grouping an unbound stream, i.e. no data window is specified onto the stream providing groups, or when using output rate limiting with the ALL keyword, you should ensure your group-by expression does not return an unlimited number of values. If, for example, your group-by expression is a fine-grained timestamp, group state that accumulates for an unlimited number of groups potentially reduces available memory significantly. Use a @Hint as described below to instruct the engine when to discard group state.

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
```

5.6.2.1. Hints Pertaining to Group-By

The Esper engine reclaims aggregation state agressively when it determines that a group has no data points, based on the data in the data windows. When your application data creates a large number of groups with a small or zero number of data points then performance may suffer as state is reclaimed and created anew. Esper provides the `@Hint('disable_reclaim_group')` hint that you can specify as part of an EPL statement text to avoid group reclaim.

When aggregating values over an unbound stream (i.e. no data window is specified onto the stream) and when your group-by expression returns an unlimited number of values, for example when a timestamp expression is used, then please note the next hint.

A sample statement that aggregates stock tick events by timestamp, assuming the event type offers a property by name `timestamp` that, reflects time in high resolution, for example arrival or system time:

```
// Note the below statement could lead to an out-of-memory problem:
select symbol, sum(price) from StockTickEvent group by timestamp
```

As the engine has no means of detecting when aggregation state (sums per symbol) can be discarded, you may use the following hints to control aggregation state lifetime.

The @Hint("`reclaim_group_aged=`*age_in_seconds*") hint instructs the engine to discard aggregation state that has not been updated for *age_in_seconds* seconds.

The optional @Hint("`reclaim_group_freq=`*sweep_frequency_in_seconds*") can be used in addition to control the frequency at which the engine sweeps aggregation state to determine aggregation state age and remove state that is older then *age_in_seconds* seconds. If the hint is not specified, the frequency defaults to the same value as *age_in_seconds*.

The updated sample statement with both hints:

```
// Instruct engine to remove state older then 10 seconds and sweep every 5 seconds
@Hint('reclaim_group_aged=10,reclaim_group_freq=5')
select symbol, sum(price) from StockTickEvent group by timestamp
```

Variables may also be used to provide values for *age_in_seconds* and *sweep_frequency_in_seconds*.

This example statement uses a variable named `varAge` to control how long aggregation state remains in memory, and the engine defaults the sweep frequency to the same value as the variable provides:

```
@Hint('reclaim_group_aged=varAge')
select symbol, sum(price) from StockTickEvent group by timestamp
```

5.6.3. 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
```

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)
```

5.6.4. 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.

5.6.5. Comparing Keyed Segmented Context, the *Group By* clause and the *std:groupwin* view

The keyed segmented context *create context ... partition by* and the *group by* clause as well as the built-in *std:groupwin* view are similar in their ability to group events but very different in their semantics. This section explains the key differences in their behavior and use.

The keyed segmented context as declared with *create context ... partition by* and *context .... select ...* creates a new context partition per key value(s). The engine maintains separate data window views as well as separate aggregations per context partition; thereby the keyed segmented context applies to both. See *Section 4.2.2, “Keyed Segmented Context”* for additional examples.

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:groupwin* view is a built-in view that groups events into data windows. The view is described in greater detail in *Section 12.3.2, “Grouped Data Window (std:groupwin)”*. 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 table below summarizes the point:

Table 5.4. Grouping Options

Option	Description
Keyed Segmented Context	Separate context partition per key value.

Option	Description
	Affects all of data windows, aggregations, patterns, etc. (except variables which are global).
Grouped Data Window (*std:groupwin*)	Separate data window per key value. Affects only the data window that is declared next to it.
Group By Clause (*group by*)	Separate aggregation values per key value. Affects only aggregation values.

Please review the performance section for advice related to performance or memory-use.

The next example shows 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:groupwin(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:groupwin(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.

5.7. Stabilizing and Controlling Output: the *Output* Clause

5.7.1. Output Clause Options

The `output` clause is optional in Esper and is used to control or stabilize the rate at which events are output and to suppress output events. The EPL language provides for several different ways to control output rate.

Here is the syntax for the `output` clause that specifies a rate in time interval or number of events:

```
output [after suppression_def]
  [[all | first | last | snapshot] every output_rate [seconds | events]]
[and when terminated]
```

An alternate syntax specifies the time period between output as outlined in *Section 5.2.1, “Specifying Time Periods”* :

```
output [after suppression_def]
  [[all | first | last | snapshot] every time_period]
[and when terminated]
```

A crontab-like schedule can also be specified. The schedule parameters follow the pattern observer parameters and are further described in *Section 6.6.2.2, “timer:at”* :

```
output [after suppression_def]
  [[all | first | last | snapshot] at
   (minutes, hours, days of month, months, days of week [, seconds])]
[and when terminated]
```

For use with contexts, in order to trigger output only when a context partition terminates, specify `when terminated` as further described in *Section 4.5, “Output When Context Partition Ends”*:

```
output [after suppression_def]
  [[all | first | last | snapshot] when terminated
  [and termination_expression]
  [then set variable_name = assign_expression [, variable_name =
 assign_expression [,...]]]
  ]
```

Last, output can be controlled by an expression that may contain variables, user-defined functions and information about the number of collected events. Output that is controlled by an expression is discussed in detail below.

The `after` keyword and *suppression_def* can appear alone or together with further output conditions and suppresses output events.

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
```

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. For statements that group via the `group by` clause, the `first` keywords provides special behavior as below.

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. Its output is equivalent to the `iterator` method provided by a statement. The `snapshot` keyword requires a data window declaration if not used within a join and outputs only the last event if used without a data window.

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` clause keywords.

For use with contexts you may append the keywords `and when terminated` to trigger output at the rate defined and in addition trigger output when the context partition terminates. Please see *Section 4.5, “Output When Context Partition Ends”* for details.

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

```
select * from StockTickEvent 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:

```
select * from StockTickEvent 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 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 where rate<100 output first every 60 seconds
```

A sample statement using the Unix "crontab"-command schedule is shown next. See *Section 6.6.2.2, “timer:at”* for details on schedule syntax. Here, output occurs every 15 minutes from 8am to 5:45pm (hours 8 to 17 at 0, 15, 30 and 45 minutes past the hour):

```
select symbol, sum(price) from StockTickEvent group by symbol output at
 (*/15, 8:17, *, *, *)
```

5.7.1.1. Controlling Output Using an Expression

Output can also be controlled by an expression that may check variable values, use user-defined functions and query built-in properties that provide additional information. The synopsis is as follows:

```
output [after suppression_def]
  [[all | first | last | snapshot] when trigger_expression
    [then set variable_name = assign_expression [, variable_name
 = assign_expression [,...]]]
  [and when terminated
    [and termination_expression]
    [then set variable_name = assign_expression [, variable_name =
 assign_expression [,...]]]
  ]
```

The `when` keyword must be followed by a trigger expression returning a boolean value of true or false, indicating whether to output. Use the optional `then` keyword to change variable values after the trigger expression evaluates to true. An assignment expression assigns a new value to variable(s).

For use with contexts you may append the keywords `and when terminated` to also trigger output when the context partition terminates. Please see *Section 4.5, “Output When Context Partition Ends”* for details. You may optionally specify a termination expression. If that expression is provided the engine evaluates the expression when the context partition terminates: The evaluation result of `true` means output occurs when the context partition terminates, `false` means no output occurs when the context partition terminates. You may specify `then set` followed by a list of assignments to assign variables. Assignments are executed on context partition termination regardless of the termination expression, if present.

Lets consider an example. The next statement assumes that your application has defined a variable by name OutputTriggerVar of boolean type. The statement outputs rows only when the OutputTriggerVar variable has a boolean value of true:

```
select sum(price) from StockTickEvent output when OutputTriggerVar = true
```

The engine evaluates the trigger expression when streams and data views post one or more insert or remove stream events after considering the `where` clause, if present. It also evaluates the trigger expression when any of the variables used in the trigger expression, if any, changes value. Thus output occurs as follows:

1. When there are insert or remove stream events and the `when` trigger expression evaluates to true, the engine outputs the resulting rows.
2. When any of the variables in the `when` trigger expression changes value, the engine evaluates the expression and outputs results. Result output occurs within the minimum time interval of timer resolution (100 milliseconds).

By adding a `then` part to the EPL, we can reset any variables after the trigger expression evaluated to true:

```
select sum(price) from StockTickEvent
  output when OutputTriggerVar = true
  then set OutputTriggerVar = false
```

Expressions in the `when` and `then` may, for example, use variables, user defined functions or any of the built-in named properties that are described in the below list.

The following built-in properties are available for use:

Table 5.5. Built-In Properties for Use with Output When

Built-In Property Name	Description
`last_output_timestamp`	Timestamp when the last output occurred for the statement; Initially set to time of statement creation
`count_insert`	Number of insert stream events
`count_insert_total`	Number of insert stream events in total (not reset when output occurs).
`count_remove`	Number of remove stream events

Built-In Property Name	Description
`count_remove_total`	Number of remove stream events in total (not reset when output occurs).

The values provided by `count_insert` and `count_remove` are non-continues: The number returned for these properties may 'jump' up rather then count up by 1. The counts reset to zero upon output.

The following restrictions apply to expressions used in the output rate clause:

- Event property names cannot be used in the output clause.
- Aggregation functions cannot be used in the output clause.
- The `prev` previous event function and the `prior` prior event function cannot be used in the output clause.

5.7.1.2. Suppressing Output With `After`

The `after` keyword and its time period or number of events parameters is optional and can occur after the `output` keyword, either alone or with output conditions as listed above.

The synopsis of `after` is as follows:

```
output after time_period | number events [...]
```

When using `after` either alone or together with further output conditions, the engine discards all output events until the time period passed as measured from the start of the statement, or until the number of output events are reached. The discarded events are not output and do not count towards any further output conditions if any are specified.

For example, the following statement outputs every minute the total price for all orders in the 30-minute time window but only after 30 minutes have passed:

```
select sum(price) from OrderEvent.win:time(30 min) output after 30 min snapshot
 every 1 min
```

An example in which `after` occur alone is below, in a statement that outputs total price for all orders in the last minute but only after 1 minute passed, each time an event arrives or leaves the data window:

```
select sum(price) from OrderEvent.win:time(1 min) output after 1 min
```

To demonstrate `after` when used with an event count, this statement find pairs of orders with the same id but suppresses output for the first 5 pairs:

```
select * from pattern[every o=OrderEvent->p=OrderEvent(id=o.id)] output after
 5 events
```

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

Remove stream events can also be useful in conjunction with aggregation and the `output` clause: 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`, `all` and `first` 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 or when using group-by then an output per group only for those groups present in the batch, following *Section 3.7.2, “Output for Aggregation and Group-By”*.
- 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.
- The `first` keyword specifies that only groups whose aggregate values have been updated with the most recent batch of events should be output following the defined frequency, keeping frequency state for each group.

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.

The next statement outputs total price per symbol cumulatively (no data window was used here). As it specifies the `all` keyword, the statement outputs the current value for all groups seen so far, regardless of whether the group was updated in the last interval. Output occurs after an interval of 5 seconds passed and at the end of each subsequent interval:

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

The below statement outputs total price per symbol considering events in the last 3 minutes. When events leave the 3-minute data window output also occurs as new aggregation values are computed. The `last` keyword instructs the engine to output only those groups that had changes. Output occurs after an interval of 10 seconds passed and at the end of each subsequent interval:

```
select symbol, sum(price) from StockTickEvent.win:time(3 min)
group by symbol output last every 10 seconds
```

This statement also outputs total price per symbol considering events in the last 3 minutes. The `first` keyword instructs the engine to output as soon as there is a new value for a group. After output for a given group the engine suppresses output for the same group for 10 seconds and does not suppress output for other groups. Output occurs again for that group after the interval when the group has new value(s):

```
select symbol, sum(price) from StockTickEvent.win:time(3 min)
group by symbol output first every 10 seconds
```

5.7.3. 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.

5.8. 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:groupwin` 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 name defined in the `select` clause, is also valid in the order by clause.

By default all sort operations on string values are performed via the `compare` method and are thus not locale dependent. To account for differences in language or locale, see *Section 15.4.21, "Engine Settings related to Language and Locale"* to change this setting.

5.9. Limiting Row Count: the *Limit* Clause

The `limit` clause is typically used together with the `order by` and `output` clause to limit your query results to those that fall within a specified range. You can use it to receive the first given number of result rows, or to receive a range of result rows.

There are two syntaxes for the `limit` clause, each can be parameterized by integer constants or by variable names. The first syntax is shown below:

```
limit row_count [offset offset_count]
```

The required *row_count* parameter specifies the number of rows to output. The *row_count* can be an integer constant and can also be the name of the integer-type variable to evaluate at runtime.

The optional *offset_count* parameter specifies the number of rows that should be skipped (offset) at the beginning of the result set. A variable can also be used for this parameter.

The next sample EPL query outputs the top 10 counts per property 'uri' every 1 minute.

```
select uri, count(*) from WebEvent
group by uri
```

```
output snapshot every 1 minute
order by count(*) desc
limit 10
```

The next statement demonstrates the use of the `offset` keyword. It outputs ranks 3 to 10 per property 'uri' every 1 minute:

```
select uri, count(*) from WebEvent
group by uri
output snapshot every 1 minute
order by count(*) desc
limit 8 offset 2
```

The second syntax for the `limit` clause is for SQL standard compatibility and specifies the offset first, followed by the row count:

```
limit offset_count[, row_count]
```

The following are equivalent:

```
limit 8 offset 2
// ...equivalent to
limit 2, 8
```

A negative value for *row_count* returns an unlimited number or rows, and a zero value returns no rows. If variables are used, then the current variable value at the time of output dictates the row count and offset. A variable returning a null value for *row_count* also returns an unlimited number or rows.

A negative value for offset is not allowed. If your variable returns a negative or null value for offset then the value is assumed to be zero (i.e. no offset).

The `iterator` pull API also honors the `limit` clause, if present.

5.10. 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 | irstream | 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).

If your application specifies `irstream`, the engine inserts into the new stream both the insert and remove stream. This is often useful in connection with the `istream` built-in function that returns an inserted/removed boolean indicator for each event, see *Section 9.1.7, "The Istream Function"*.

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.

```
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)]
```

5.10.1. Transposing a Property To a Stream

Sometimes your events may carry properties that are themselves event objects. Therefore EPL offers a special syntax to insert the value of a property itself as an event into a stream:

```
insert into stream_name select property_name.* from ...
```

This feature is only supported for JavaBean events and for `Map` and Object-array (`Object[]`) event types that associate an event type name with the property type. It is not supported for `XML` events. Nested property names are also not supported.

In this example, the class `Summary` with properties `bid` and `ask` that are of type `Quote` is:

```
public class Summary {
  private Quote bid;
  private Quote ask;
  ...
```

The statement to populate a stream of `Quote` events is thus:

```
insert into MyBidStream select bid.* from Summary
```

5.10.2. 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 as follows: A conversion function must be the only selected column, and it must return either a Java object or `java.util.Map` or `Object[]` (object array). Your EPL should not use the `as` keyword to assign a column name.

5.10.3. Merging Disparate Types of Events: Variant Streams

A *variant stream* is a predefined stream into which events of multiple disparate event types can be inserted.

A variant stream name may appear anywhere in a pattern or `from` clause. In a pattern, a filter against a variant stream matches any events of any of the event types inserted into the variant stream. In a `from` clause including for named windows, views declared onto a variant stream may hold events of any of the event types inserted into the variant stream.

A variant stream is thus useful in problems that require different types of event to be treated the same.

Variant streams can be declared by means of `create variant schema` or can be predefined via runtime or initialization-time configuration as described in *Section 15.4.27, “Variant Stream”*. Your application may declare or predefine variant streams to carry events of a limited set of event types, or you may choose the variant stream to carry any and all types of events. This choice affects what event properties are available for consuming statements or patterns of the variant stream.

Assume that an application predefined a variant stream named `OrderStream` to carry only `ServiceOrder` and `ProductOrder` events. An `insert into` clause inserts events into the variant stream:

```
insert into OrderStream select * from ServiceOrder
```

```
insert into OrderStream select * from ProductOrder
```

Here is a sample statement that consumes the variant stream and outputs a total price per customer id for the last 30 seconds of `ServiceOrder` and `ProductOrder` events:

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

If your application predefines the variant stream to hold specific type of events, as the sample above did, then all event properties that are common to all specified types are visible on the variant stream, including nested, indexed and mapped properties. For access to properties that are only available on one of the types, the dynamic property syntax must be used. In the example above, the `customerId` and `price` were properties common to both `ServiceOrder` and `ProductOrder` events.

For example, here is a consuming statement that selects a `service duraction` property that only `ServiceOrder` events have, and that must therefore be casted to double and null values removed in order to aggregate:

```
select customerId, sum(coalesce(cast(serviceDuraction?, double), 0))
from OrderStream.win:time(30 sec) group by customerId
```

If your application predefines a variant stream to hold any type of events (the `any` type variance), then all event properties of the variant stream are effectively dynamic properties.

For example, an application may define an `OutgoingEvents` variant stream to hold any type of event. The next statement is a sample consumer of the `OutgoingEvents` variant stream that looks for the `destination` property and fires for each event in which the property exists with a value of `'email'`:

```
select * from OutgoingEvents(destination = 'email')
```

5.10.4. Decorated Events

Your `select` clause may use the '*' wildcard together with further expressions to populate a stream of events. A sample statement is:

```
insert into OrderStream select *, price*units as linePrice from PurchaseOrder
```

When using wildcard and selecting additional expression results, the engine produces what is called *decorating* events for the resulting stream. Decorating events add additional property values to an underlying event.

In the above example the resulting OrderStream consists of underlying PurchaseOrder events *decorated* by a `linePrice` property that is a result of the `price*units` expression.

In order to use `insert into` to insert into an existing stream of decorated events, your underlying event type must match, and all additional decorating property names and types of the `select` clause must also match.

5.10.5. Event as a Property

Your `select` clause may use the stream name to populate a stream of events in which each event has properties that are itself an event. A sample statement is:

```
insert into CompositeStream select order, service, order.price+service.price as
 totalPrice
from PurchaseOrder.std:lastevent() as order, ServiceEvent:std:lastevent() as
 service
```

When using the stream name (or tag in patterns) in the select-clause, the engine produces composite events: One or more of the properties of the composite event are events themselves.

In the above example the resulting CompositeStream consists of 3 columns: the PurchaseOrder event, the ServiceEvent event and the `totalPrice` property that is a result of the `order.price +service.price` expression.

In order to use `insert into` to insert into an existing stream of events in which properties are themselves events, each event column's event type must match, and all additional property names and types of the `select` clause must also match.

5.10.6. Instantiating and Populating an Underlying Event Object

Your `insert into` clause may also directly instantiate and populate application underlying event objects or `Map` or `Object[]` event objects. This is described in greater detail in *Section 2.12, “Event Objects Instantiated and Populated by Insert Into”*.

If instead you have an expression that returns an event object, please read on to the next section.

5.10.7. Transposing an Expression Result

You can transpose an object returned as an expression result into a stream using the `transpose` function as described further in *Section 9.4, “Select-Clause transpose Function”*.

5.10.8. Select-Clause Expression And Inserted-Into Column Event Type

When you declare the inserted-into event type in advance to the statement that inserts, the engine compares the inserted-into event type information to the return type of expressions in the select-clause. The comparison uses the column alias assigned to each select-clause expression using the `as` keyword.

When the inserted-into column type is an event type and when using a subquery or the `new` operator, the engine compares column names assigned to subquery columns or `new` operator columns.

For example, assume a `PurchaseOrder` event type that has a property called `items` that consists of `Item` rows:

```
create schema Item(name string, price double)
```

```
create schema PurchaseOrder(orderId string, items Item[])
```

Declare a statement that inserts into the `PurchaseOrder` stream:

```
insert into PurchaseOrder
select '001' as orderId, new {name='i1', price=10} as items
from TriggerEvent
```

The alias assigned to the first and second expression in the select-clause, namely `orderId` and `items`, both match the event property names of the `Purchase Order` event type. The column names provided to the `new` operator also both match the event property names of the `Item` event type.

When the event type declares the column as a single value (and not an array) and when the select-clause expression produces a multiple rows, the engine only populate the first row.

Consider a `PurchaseOrder` event type that has a property called `item` that consists of a single `Item` event:

```
create schema PurchaseOrder(orderId string, items Item)
```

The sample subquery below populates only the very first event, discarding remaining subquery result events, since the `items` property above is declared as holding a single `Item`-typed event only (versus `Item[]` to hold multiple `Item`-typed events).

```
insert into PurchaseOrder select
(select 'i1' as name, 10 as price from HistoryEvent.win:length(2)) as items
from TriggerEvent
```

Consider using a subquery with filter, or one of the enumeration methods to select a specific subquery result row.

5.11. Subqueries

A subquery is a `select` within another statement. Esper supports subqueries in the `select` clause, `where` clause, `having` clause and in stream and pattern filter expressions. 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:groupwin` or aggregation functions or the multi-row and multi-column selects as described below.

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.

When your subquery returns multiple rows, you must use an aggregation function in the `select` clause of the subselect, as a subquery can only return a single row and single value object. To return multiple values from a subquery, consider writing a custom aggregation function that returns an array or collection of values.

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 (by default), 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.

When using aggregation functions in a correlated subselect the engine computes the aggregation based on data window or named window contents matching the where-clause.

The following example compares the quantity value provided by the current order event against the total quantity of all order events in the last 1 hour for the same client.

```
select * from OrderEvent oe
where qty >
  (select sum(qty) from OrderEvent.win:time(1 hour) pd
  where pd.client = oe.client)
```

Filter expressions in a pattern or stream may also employ subqueries. Subqueries can be uncorrelated or can be correlated to properties of the stream or to properties of tagged events in a pattern. Subqueries may reference named windows as well.

The following example filters `BarData` events that have a close price less then the last moving average (field `movAgv`) as provided by stream `SMA20Stream` (an uncorrelated subquery):

```
select * from BarData(ticker='MSFT', closePrice <
    (select movAgv from SMA20Stream(ticker='MSFT').std:lastevent()))
```

A few generic examples follow to demonstrate the point. The examples use short event and property names so they are easy to read. Assume `A` and `B` are streams and `DNamedWindow` is a named window, and properties `a_id, b_id, d_id, a_val, b_val, d_val` respectively:

```
// Sample correlated subquery as part of stream filter criteria
select * from A(a_val in
  (select b_val from B.std:unique(b_val) as b where a.a_id = b.b_id)) as a
```

```
// Sample correlated subquery against a named window
select * from A(a_val in
  (select b_val from DNamedWindow as d where a.a_id = d.d_id)) as a
```

```
// Sample correlated subquery in the filter criteria as part of a pattern,
 querying a named window
select * from pattern [
  a=A -> b=B(bvalue =
     (select d_val from DNamedWindow as d where d.d_id = b.b_id and d.d_id =
 a.a_id))
]
```

Subquery state starts to accumulate as soon as a statement starts (and not only when a pattern-subexpression activates).

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.

The order of evaluation of subqueries relative to the containing statement is guaranteed: If the containing statement and its subqueries are reacting to the same type of event, the subquery will receive the event first before the containing statement's clauses are evaluated. This behavior can be changed via configuration. The order of evaluation of subqueries is not guaranteed between subqueries.

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, 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.

5.11.1. 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.

The synopsis for the `exists` keyword is as follows:

```
exists (subquery)
```

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.

5.11.2. The '`in`' and '`not in`' Keywords

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 synopsis for the `in` keyword is as follows:

```
expression in (subquery)
```

The right-hand side subquery must return exactly one column.

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

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

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

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand row yields null, the result of the `in` construct will be null, not false (or true for `not-in`). This is in accordance with SQL's normal rules for Boolean combinations of null values.

5.11.3. The '`any`' and '`some`' Keywords

The `any` subquery condition is true if the expression returns true for one or more of the values returned by the subquery.

The synopsis for the `any` keyword is as follows:

```
expression operator any (subquery)
expression operator some (subquery)
```

The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of `any` is "true" if any true result is obtained. The result is "false" if no true result is found (including the special case where the subquery returns no rows).

The *operator* can be any of the following values: `=, !=, <>, <, <=, >, >=`.

The `some` keyword is a synonym for `any`. The `in` construct is equivalent to `= any`.

The right-hand side subquery must return exactly one column.

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

```
select * from ProductOrder as ord
  where quantity < any
    (select minimumQuantity from MinimumQuantity.win:keepall())
```

The above query compares ProductOrder event's quantity value with all rows from the MinimumQuantity stream of events and returns only those ProductOrder events that have a quantity that is less then any of the minimum quantity values of the MinimumQuantity events.

Note that if there are no successes and at least one right-hand row yields null for the operator's result, the result of the `any` construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

5.11.4. The '`all`' Keyword

The `all` subquery condition is true if the expression returns true for all of the values returned by the subquery.

The synopsis for the `all` keyword is as follows:

```
expression operator all (subquery)
```

The left-hand expression is evaluated and compared to each row of the subquery result using the given operator, which must yield a Boolean result. The result of `all` is "true" if all rows yield true (including the special case where the subquery returns no rows). The result is "false" if any false result is found. The result is `null` if the comparison does not return false for any row, and it returns `null` for at least one row.

The *operator* can be any of the following values: `=, !=, <>, <, <=, >, >=`.

The `not in` construct is equivalent to `!= all`.

The right-hand side subquery must return exactly one column.

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

```
select * from ProductOrder as ord
  where quantity < all
```

```
    (select minimumQuantity from MinimumQuantity.win:keepall())
```

The above query compares ProductOrder event's quantity value with all rows from the MinimumQuantity stream of events and returns only those ProductOrder events that have a quantity that is less then all of the minimum quantity values of the MinimumQuantity events.

5.11.5. Multi-Column Selection

Your subquery may select multiple columns in the `select` clause including multiple aggregated values from a data window or named window.

The following example is a correlated subquery that selects wildcard and in addition selects the `bid` and `offer` properties of the last `MarketData` event for the same symbol as the arriving `OrderEvent`:

```
select *,
  (select bid, offer from MarketData.std:unique(symbol) as md
   where md.symbol = oe.symbol) as bidoffer
from OrderEvent oe
```

Output events for the above query contain all properties of the original `OrderEvent` event. In addition each output event contains a `bidoffer` nested property that itself contains the `bid` and `offer` properties. You may retrieve the bid and offer from output events directly via the `bidoffer.bid` property name syntax for nested properties.

The next example is similar to the above query but instead selects aggregations and selects from a named window by name `OrderNamedWindow` (creation not shown here). For each arriving `OrderEvent` it selects the total quantity and count of all order events for the same client, as currently held by the named window:

```
select *,
  (select sum(qty) as sumPrice, count(*) as countRows
   from OrderNamedWindow as onw
   where onw.client = oe.client) as pastOrderTotals
from OrderEvent as oe
```

The next EPL statement computes a prorated quantity considering the maximum and minimum quantity for the last 1 minute of order events:

```
expression subq {
 (select max(quantity) as maxq, min(quantity) as minq from OrderEvent.win:time(1
 min))
}
```

```
select (quantity - minq) / (subq().maxq  - subq().minq) as prorated
from OrderEvent
```

Output events for the above query contain all properties of the original `OrderEvent` event. In addition each output event contains a `pastOrderTotals` nested property that itself contains the `sumPrice` and `countRows` properties.

5.11.6. Multi-Row Selection

While a subquery cannot change the cardinality of the selected stream, a subquery can return multiple values from the selected data window or named window. This section shows examples of the `window` aggregation function as well as the use of enumeration methods with subselects.

Consider using an inner join, outer join or unidirectional join instead to achieve a 1-to-many cardinality in the number of output events.

The next example is an uncorrelated subquery that selects all current `ZoneEvent` events considering the last `ZoneEvent` per zone for each arriving `RFIDEvent`.

```
select assetId,
 (select window(z.*) as winzones from ZoneEvent.std:unique(zone) as z) as zones
 from RFIDEvent
```

Output events for the above query contain two properties: the `assetId` property and the `zones` property. The latter property is a nested property that contains the `winzones` property. You may retrieve the zones from output events directly via the `zones.winzones` property name syntax for nested properties.

In this example for a correlated subquery against a named window we assume that the `OrderNamedWindow` has been created and contains order events. The query returns for each `MarketData` event the list of order ids for orders with the same symbol:

```
select price,
 (select window(orderId) as winorders
  from OrderNamedWindow onw
  where onw.symbol = md.symbol) as orderIds
 from MarketData md
```

Output events for the above query contain two properties: the `price` property and the `orderIds` property. The latter property is a nested property that contains the `winorders` property of type array.

Another option to reduce selected rows to a single value is through the use of enumeration methods.

```
select price,
 (select *  from OrderNamedWindow onw
  where onw.symbol = md.symbol).selectFrom(v => v) as ordersSymbol
 from MarketData md
```

Output events for the above query also contain a Collection of underlying events in the `ordersSymbol` property.

5.11.7. Hints Related to Subqueries

The following hints are available to tune performance and memory use of subqueries.

Use the `@Hint('set_noindex')` hint for a statement that utilizes one or more subqueries. It instructs the engine to always perform a full table scan. The engine does not build an implicit index or use an explicitly-created index when this hint is provided. Use of the hint may result in reduced memory use but poor statement performance.

The following hints are available to tune performance and memory use of subqueries that select from named windows.

Named windows are globally-visible data windows. As such an application may create explicit indexes as discussed in *Section 5.15.13, “Explicitly Indexing Named Windows”*. The engine may also elect to create implicit indexes (no create-index EPL required) for index-based lookup of rows when executing `on-select`, `on-merge`, `on-update` and `on-delete` statements and for statements that subquery a named window.

By default and without specifying a hint, each statement that subqueries a named window also maintains its own index for looking up events held by the named window. The engine maintains the index by consuming the named window insert and remove stream. When the statement is destroyed it releases that index.

Specify the `@Hint('enable_window_subquery_indexshare')` hint to enable subquery index sharing for named windows. When using this hint, indexes for subqueries are maintained by the named window itself (and not each statement), are shared between one or more statements and may also utilize explicit indexes. Specify the hint once as part of the `create window` statement.

This sample EPL statement creates a named window with subquery index sharing enabled:

```
@Hint('enable_window_subquery_indexshare')
create window AllOrdersNamedWindow.win:keepall() as OrderMapEventType
```

When subquery index sharing is enabled, performance may increase as named window stream consumption is no longer needed. You may also expect reduced memory use especially if a large number of EPL statements perform similar subqueries against a named window. Subquery index sharing may require additional short-lived object creation and may slightly increase lock held time for named windows.

The following statement performs a correlated subquery against the named window above. When a settlement event arrives it select the order detail for the same order id as provided by the settlement event:

```
select
  (select * from AllOrdersNamedWindow as onw
    where onw.orderId = se.orderId) as orderDetail
  from SettlementEvent as se
```

With subquery index sharing enabled the engine maintains an index of order events by order id for the named window, and shares that index between additional statements until the time all utilizing statements are destroyed.

You may disable subquery index sharing for a specific statement by specifying the `@Hint('disable_window_subquery_indexshare')` hint, as this example shows, causing the statement to maintain its own index:

```
@Hint('disable_window_subquery_indexshare')
select
  (select * from AllOrdersNamedWindow as onw
    where onw.orderId = se.orderId) as orderDetail
  from SettlementEvent as se
```

5.12. Joining Event Streams

5.12.1. Introducing Joins

Two or more event streams can be part of the `from`-clause and thus both (all) streams determine the resulting events. This section summarizes the important concepts. The sections that follow present more detail on each topic.

The default join is an inner join which produces output events only when there is at least one match in all streams.

Consider the sample statement shown next:

```
select * from TickEvent.std:lastevent(), NewsEvent.std:lastevent()
```

The above statement outputs the last TickEvent and the last NewsEvent in one output event when either a TickEvent or a NewsEvent arrives. If no TickEvent was received before a NewsEvent arrives, no output occurs. Similarly when no NewsEvent was received before a TickEvent arrives, no output occurs.

The `where`-clause lists the join conditions that Esper uses to relate events in the two or more streams.

The next example statement retains the last TickEvent and last NewsEvent per symbol, and joins the two streams based on their symbol value:

```
select * from TickEvent.std:unique(symbol) as t, NewsEvent.std:unique(symbol)
 as n
where t.symbol = n.symbol
```

As before, when a TickEvent arrives for a symbol that has no matching NewsEvent then there is no output event.

An outer join does not require each event in either stream to have a matching event. The full outer join is useful when output is desired when no match is found. The different outer join types (full, left, right) are explained in more detail below.

This example statement is an outer-join and also returns the last TickEvent and last NewsEvent per symbol:

```
select * from TickEvent.std:unique(symbol) as t
full outer join NewsEvent.std:unique(symbol) as n on t.symbol = n.symbol
```

In the sample statement above, when a TickEvent arrives for a symbol that has no matching NewsEvent, or when a NewsEvent arrives for a symbol that has no matching TickEvent, the statement still produces an output event with a null column value for the missing event.

Note that each of the sample queries above defines a data window. The sample queries above use the last-event data window (std:lastevent) or the unique data window (std:unique). A data window serves to indicate the subset of events to join from each stream and may be required depending on the join.

In above queries, when either a TickEvent arrives or when a NewsEvent arrives then the query evaluates and there is output. The same holds true if additional streams are added to the `from`-clause: Each of the streams in the `from`-clause trigger the join to evaluate.

The `unidirectional` keyword instructs the engine to evaluate the join only when an event arrives from the single stream that was marked with the `unidirectional` keyword. In this case no data window should be specified for the stream marked as `unidirectional` since the keyword implies that the current event of that stream triggers the join.

Here is the sample statement above with `unidirectional` keyword, so that output occurs only when a TickEvent arrives and not when a NewsEvent arrives:

```
select * from TickEvent as t unidirectional, NewsEvent.std:unique(symbol) as n
where t.symbol = n.symbol
```

It is oftentimes the case that an aggregation (count, sum, average) only needs to be calculated in the context of an arriving event or timer. Consider using the `unidirectional` keyword when aggregating over joined streams.

An EPL pattern is a normal citizen also providing a stream of data consisting of pattern matches. A time pattern, for example, can be useful to evaluate a join and produce output upon each interval.

This sample statement includes a pattern that fires every 5 seconds and thus triggers the join to evaluate and produce output, computing an aggregated total quantity per symbol every 5 seconds:

```
select   symbol,   sum(qty)   from   pattern[every   timer:interval(5   sec)]
 unidirectional,
  TickEvent.std:unique(symbol) t, NewsEvent.std:unique(symbol) as n
where t.symbol = n.symbol group by symbol
```

Named windows as well as reference and historical data such as stored in your relational database, and data returned by a method invocation, can also be included in joins as discussed in *Section 5.13, “Accessing Relational Data via SQL”* and *Section 5.14, “Accessing Non-Relational Data via Method Invocation”*.

Related to joins are subqueries: A subquery is a `select` within another statement, see *Section 5.11, “Subqueries”*

The engine performs extensive query analysis and planning, building internal indexes and strategies as required to allow fast evaluation of many types of queries.

5.12.2. Inner (Default) Joins

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 when matching events are found for all joined streams.

This example joins 2 event streams. The first event stream consists of fraud warning events for which we keep the last 30 minutes. 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)].std:lastevent() 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. The last-event view instucts the join to only consider the last pattern match.

In a join and outer join, your statement must declare a data window view or other view onto each stream. Streams that are marked as unidirectional and named windows as well as database or methods in a join are an exception and do not require a view to be specified. If you are joining an event to itself via contained-event selection, views also do not need to be specified. The reason that a data window must be declared is that a data window specifies which events are considered for the join (i.e. last event, last 10 events, all events, last 1 second of events etc.).

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.win:keepall()      as      fraud,
 PINChangeEvent.win:time(20 sec) as pin
 where fraud.accountNumber = pin.accountNumber
```

The above example employed the special keep-all view that retains all events.

5.12.3. Outer, Left and Right Joins

Esper supports left outer joins, right outer joins, full outer joins and inner joins in any combination between an unlimited number of event streams. Outer and inner joins can also join reference and historical data as explained in *Section 5.13, “Accessing Relational Data via SQL”*, as well as join data returned by a method invocation as outlined in *Section 5.14, “Accessing Non-Relational Data via Method Invocation”*.

The keywords `left, right, full` and `inner` control the type of the join between two streams. The optional `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) | inner) join stream_def
  [on property = property [and property = property ...] ]
  [ ((left|right|full outer) | inner) 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 next 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
```

The last type of join is an inner join. In an inner join, the engine produces an output event for each event of the stream on the left-hand side that matches at least one event on the right hand side considering the join properties. For example, in the inner join shown below we will get output for each event in the RfidEvent stream that matches one or more events in the OrderList data window:

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

Patterns as streams in a join follow this rule: If no data window view is declared for the pattern then the pattern stream retains the last match. Thus a pattern must have matched at least once for the last row to become available in a join. Multiple rows from a pattern stream may be retained by declaring a data window view onto a pattern using the `pattern [...].`*view_specification* syntax.

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
```

If the optional `on` clause is specified, it may only employ the = equals operator and property names. Any other operators must be placed in the `where`-clause. The stream names that appear in the `on` clause may refer to any stream in the `from`-clause.

Your EPL may also provide no `on` clause. This is useful when the streams that are joined do not provide any properties to join on, for example when joining with a time-based pattern.

The next example employs a unidirectional left outer join such that the engine, every 10 seconds, outputs a count of the number of RfidEvent events in the 60-second time window.

```
select count(*) from
  pattern[every timer:interval(1)] unidirectional
  left outer join
  RfidEvent.win:time(60 sec)
```

5.12.4. Unidirectional 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
```

Aggregation functions in a `unidirectional` join aggregate within the context of each unidirectional event evaluation and are not cumulative. Thereby aggregation functions when used with `unidirectional` may evaluate faster as they do not need to consider a remove stream (data removed from data windows or named windows).

The count function in the next query returns, for each TickEvent, the number of matching NewEvent events:

```
select count(*) 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.

5.12.5. Hints Related to Joins

When joining 3 or more streams (including any relational or non-relational sources as below) it can sometimes help to provide the query planner instructions how to best execute the join. The engine compiles a query plan for the EPL statement at statement creation time. You can output the query plan to logging (see configuration).

An outer join that specifies only `inner` keywords for all streams is equivalent to an default (inner) join. The following two statements are equivalent:

```
select * from TickEvent.std:lastevent(),
    NewsEvent.std:lastevent() where tick.symbol = news.symbol
```

Equivalent to:

```
select * from TickEvent.std:lastevent()
 inner join NewsEvent.std:lastevent() on tick.symbol = news.symbol
```

For all types of joins, the query planner determines a query graph: The term is used here for all the information regarding what properties or expressions are used to join the streams. The query graph thus includes the where-clause expressions as well as outer-join on-clauses if this statement is an outer join. The query planner also computes a dependency graph which includes information about all historical data streams (relational and non-relational as below) and their input needs.

For default (inner) joins the query planner first attempts to find a path of execution as a nested iteration. For each stream the query planner selects the best order of streams available for the nested iteration considering the query graph and dependency graph. If the full depth of the join is achievable via nested iteration for all streams without full table scan then the query planner uses that nested iteration plan. If not, then the query planner re-plans considering a merge join (Cartesian) approach instead.

Specify the @Hint('PREFER_MERGE_JOIN') to instruct the query planner to prefer a merge join plan instead of a nested iteration plan. Specify the @Hint('FORCE_NESTED_ITER') to instruct the query planner to always use a nested iteration plan.

For example, consider the below statement. Depending on the number of matching rows in OrderBookOne and OrderBookTwo (named windows in this example, and assumed to be defined elsewhere) the performance of the join may be better using the merge join plan.

```
@Hint('PREFER_MERGE_JOIN')
select * from TickEvent.std:lastevent() t,
 OrderBookOne ob1, OrderBookOne ob2
where ob1.symbol = t.symbol and ob2.symbol = t.symbol
and ob1.price between t.buy and t.sell and ob2.price between t.buy and t.sell
```

For outer joins the query planner considers nested iteration and merge join (Cartesian) equally and above hints don't apply.

5.13. Accessing 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 access via join and outer join as well as via iterator (poll) API 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 15.4.9, "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 or 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 and caches the appropriate index and lookup strategies for fast row matching against indexes.

Joins or outer joins in which only SQL statements or method invocations are listed in the `from` clause and no other event streams are termed *passive* joins. A passive join does not produce an insert or remove stream and therefore does not invoke statement listeners with results. A passive join can be iterated on (polled) using a statement's `safeIterator` and `iterator` methods.

There are no restrictions to the number of SQL statements or types of streams joined. The following restrictions currently apply:

- 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).

5.13.1. 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 15.4.9, “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 `${`*expression*`}`. The engine resolves *expression* at statement execution time to the actual expression result by evaluating the events in the joined event stream or current variable values, if any event property references or variables occur in the expression. An *expression* may not contain EPL substitution parameters.

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.

Since any expression may be placed within the `${...}` syntax, you may use variables or user-defined functions as well.

The next example assumes that a variable by name `varLowerLimit` is defined and that a user-defined function `getLimit` exists on the `MyLib` imported class that takes a `LimitEvent` as a parameter:

```
select * from LimitEvent le,
  sql:MyCustomerDB [' select cust_name from Customer where
      amount > ${max(varLowerLimit, MyLib.getLimit(le))} ']
```

The example above takes the higher of the current variable value or the value returned by the user-defined function to return only those customer names where the amount exceeds the computed limit.

5.13.2. 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 and outer-join `on` 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.

The SQL result caches operate on the level of all result rows for a given parameter set. For example, if your query returns 10 rows for a certain set of parameter values then the cache treats all 10 rows as a single entry keyed by the parameter values, and the expiry policy applies to all 10 rows and not to each individual row.

It is also possible to join multiple autonomous database systems in a single query, for example:

```
select symbol, symbolDesc from OrderEvent as orders,
  sql:My_Oracle_DB ['select symbolDesc from SymbolReference'] as reference,
  sql:My_MySQL_DB ['select orderList from orderHistory'] as history
  where reference.symbol = orders.symbol
  and history.symbol = orders.symbol
```

5.13.3. 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.

5.13.4. 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 or following the schedule of the crontab-like `timer:at`.

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']
```

5.13.5. Polling SQL Queries via Iterator

Usually your SQL query will take part in a join and thus be triggered by an event or pattern occurrence. Instead, your application may need to poll a SQL query and thus use Esper query execution and caching facilities and obtain event data and metadata.

Your EPL statement can specify an SQL statement without a join. Such a stand-alone SQL statement does not post new events, and may only be queried via the `iterator` poll API. Your EPL and SQL statement may still use variables.

The next statement assumes that a `price_var` variable has been declared. It selects from the relational database table named `NewOrder` all rows in which the `price` column is greater then the current value of the `price_var` EPL variable:

```
select * from sql:MyCustomerDB ['select * from NewOrder where ${price_var} >
 price']
```

Use the `iterator` and `safeIterator` methods on `EPStatement` to obtain results. The statement does not post events to listeners, it is strictly passive in that sense.

5.13.6. 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.

5.13.7. 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 15, 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, you 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 ....'], ...
```

If your *parameterized_sql_query* SQL query contains vendor-specific SQL syntax, generation of the metadata query may fail to produce a valid SQL statement. If you experience an SQL error while fetching metadata, use any of the above workarounds with the Oracle JDBC driver.

5.13.8. SQL Input Parameter and Column Output Conversion

As part of database access configuration you may optionally specify SQL type mappings. These mappings apply to all queries against the same database identified by name.

If your application must perform SQL-query-specific or EPL-statement-specific mapping or conversion between types, the facility to register a conversion callback exists as follows.

Use the `@Hook` instruction and `HookType.SQLCOL` as part of your EPL statement text to register a statement SQL parameter or column conversion hook. Implement the interface `com.espertech.esper.client.hook.SQLColumnTypeConversion` to perform the input parameter or column value conversion.

A sample statement with annotation is shown:

```
@Hook(type=HookType.SQLCOL, hook='MyDBTypeConvertor')
select * from sql:MyDB ['select * from MyEventTable]
```

The engine expects `MyDBTypeConvertor` to resolve to a class (considering engine imports) and instantiates one instance of MyDBTypeConvertor for each statement.

5.13.9. SQL Row POJO Conversion

Your application may also directly convert a SQL result row into a Java class which is an opportunity for your application to interrogate and transform the SQL row result data freely before packing the data into a Java class. Your application can additionally indicate to skip SQL result rows.

Use the `@Hook` instruction and `HookType.SQLROW` as part of your EPL statement text to register a statement SQL output row conversion hook. Implement the interface `com.espertech.esper.client.hook.SQLOutputRowConversion` to perform the output row conversion.

A sample statement with annotation is shown:

```
@Hook(type=HookType.SQLROW, hook='MyDBRowConvertor')
select * from sql:MyDB ['select * from MyEventTable]
```

The engine expects `MyDBRowConvertor` to resolve to a class (considering engine imports) and instantiates one MyDBRowConvertor instance for each statement.

5.14. Accessing 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.

The results of such a method invocation in the `from` clause plays the same role as a relational database table in an inner and outer join in SQL. The role is thus dissimilar to the role of a user-defined function, which may occur in any expression such as in the `select` clause or the `where` clause. Both are backed by one or more static methods provided by your class library.

Esper can join and outer join an unlimited number and 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.

Joins or outer joins in which only SQL statements or method invocations are listed in the `from` clause and no other event streams are termed *passive* joins. A passive join does not produce an insert or remove stream and therefore does not invoke statement listeners with results. A passive join can be iterated on (polled) using a statement's `safeIterator` and `iterator` methods.

The following restrictions currently apply:

- 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.

5.14.1. 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 or outer-join `on` 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 15.4.7, "Cache Settings for From-Clause Method Invocations"* describes.

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

5.14.2. Polling Method Invocation Results via Iterator

Usually your method invocation will take part in a join and thus be triggered by an event or pattern occurrence. Instead, your application may need to poll a method invocation and thus use Esper query execution and caching facilities and obtain event data and metadata.

Your EPL statement can specify a method invocation in the `from` clause without a join. Such a stand-alone method invocation does not post new events, and may only be queried via the `iterator` poll API. Your EPL statement may still use variables.

The next statement assumes that a `category_var` variable has been declared. It polls the `getAssetDescriptions` method passing the current value of the `category_var` EPL variable:

```
select * from method:MyLookupLib.getAssetDescriptions(category_var)]
```

Use the `iterator` and `safeIterator` methods on `EPStatement` to obtain results. The statement does not post events to listeners, it is strictly passive in that sense.

5.14.3. 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 15.4.6, “Class and package imports”*.

5.14.4. 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.

5.15. 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. A named window holds events of the same type or supertype, unless used with a variant stream.

The `create window` clause declares a new named window. The named window starts up empty unless populated from an existing named window at time of creation. 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.

A named window may also decorate an event to preserve original events as described in *Section 5.10.4, “Decorated Events”* and *Section 5.15.2.1, “Named Windows Holding Decorated Events”*. In addition, columns of a named window are allowed to hold events themselves, as further explained in *Section 5.10.5, “Event as a Property”* and *Section 5.15.2.2, “Named Windows Holding Events As Property”*.

To tune subquery performance when the subquery selects from a named window, consider the hints discussed in *Section 5.11.7, “Hints Related to Subqueries”*.

5.15.1. 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.

There are two syntaxes for creating a named window: The first syntax allows to model a named window after an existing event type or an existing named window. The second syntax is similar to the SQL create-table syntax and provides a list of column names and column types.

A new named window starts up empty. It must be explicitly inserted into by one or more statements, as discussed below. A named window can also be populated at time of creation from an existing named window.

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.

5.15.1.1. Creation by Modelling after an Existing Type

The benefit of modelling a named window after an existing event type is that event properties can be nested, indexed, mapped or other types that your event objects may provide as properties, including the type of the underlying event itself. Also, using the wildcard (*) operator means your EPL does not need to list each individual property explicitly.

The syntax for creating a named window by modelling the named window after an existing event type, is as follows:

```
[context context_name]
create window window_name.view_specifications
    [as] [select list_of_properties from] event_type_or_windowname
    [insert [where filter_expression]]
```

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. Data window views are listed in *Chapter 12,* *EPL Reference: Views*. View parameterization and staggering are described in *Section 5.4.3, “Specifying Views”*.

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 than column names are not allowed in the `select` list of properties. Wildcards (*) and wildcards with additional properties can also be used.

The *event_type_or_windowname* is required if using the model-after syntax. It 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 name of an (existing) other named window is also allowed here. Please find more details in *Section 5.15.7, “Populating a Named Window from an Existing Named Window”*.

Finally, the `insert` clause and optional *filter_expression* are used if the new named windows is modelled after an existing named window, and the data of the new named window is to be populated from the existing named window upon creation. The optional *filter_expression* can be used to exclude events.

You may refer to a context by specifying the `context` keyword followed by a context name. Context are described in more detail at *Chapter 4, Context and Context Partitions*. The effect of referring to a context is that your named window operates according to the context dimensional information as declared for the context. For usage and limitations please see the respective chapter.

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, and constants may occur in the select-clause as well:

```
create window OrdersTimeWindow.win:time(30 sec) as
  select symbol as sym, volume as vol, price, 1 as alertId from OrderEvent
```

5.15.1.2. Creation By Defining Columns Names and Types

The second syntax for creating a named window is by supplying column names and types:

```
[context context_name]
create window window_name.view_specifications [as] (column_name column_type
  [,column_name column_type [,...])
```

The *column_name* is an identifier providing the event property name. The *column_type* is also required for each column. Valid column types are listed in *Section 5.18.1, “Creating Variables: the Create Variable clause”* and are the same as for variable types.

For attributes that are array-type append `[]` (left and right brackets).

The next statement creates a named window:

```
create window SecurityEvent.win:time(30 sec)
    (ipAddress string, userId String, numAttempts int, properties String[])
```

Named window columns can hold events by declaring the column type as the event type name. Array-type in combination with event-type is also supported.

The next two statements declare an event type and create a named window with a column of the defined event type:

```
create schema SecurityData (name String, roles String[])
```

```
create window SecurityEvent.win:time(30 sec)
      (ipAddress string, userId String, secData SecurityData, historySecData
 SecurityData[])
```

Whether the named window uses a Map or Object-array event representation for the rows can be specified as follows. If the create-window statement provides the `@EventRepresentation(array=true)` annotation the engine maintains named window rows as object array. If the statement provides the `@EventRepresentation(array=false)` annotation the engine maintains named window rows using Map objects. If neither annotation is provided, the engine uses the configured default event representation as discussed in *Section 15.4.11.1, “Default Event Representation”*.

The following EPL statement instructs the engine to represent FooWindow rows as object arrays:

```
@EventRepresentation(array=true) create window FooWindow.win:time(5 sec) as
 (string prop1)
```

5.15.1.3. Dropping or Removing Named Windows

There is no syntax to drop or remove a named window.

The `destroy` method on the `EPStatement` that created the named window removes the named window. However the implicit event type associated with the named window remains active since further statements may continue to use that type. Therefore a named window of the same name can only be created again if the type information matches the prior declaration for a named window.

5.15.2. 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.

For inserting into a named window and for simulateously checking if the inserted row already exists in the named window or for atomic update-insert operation on a named window, consider using `on-merge` as described in *Section 5.15.12, “Triggered Upsert using the On-Merge Clause”*. On-merge is similar to the SQL `merge` clause and provides what is known as an "Upsert" operation: Update existing events or if no existing event(s) are found then insert a new event, all in one atomic operation provided by a single EPL statement.

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
```

An alternative form is shown next:

```
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 statements create a named window for an event type backed by a Java class and insert into the window any 'OrderEvent' where the symbol value is IBM:

```
create window OrdersWindow.win:time(30) 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.win:time(30) 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.std:unique(name) 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 inserted in a named window and atomically check if a row already exists, use `on-merge` as outlined in *Section 5.15.12, “Triggered Upsert using the On-Merge Clause”*. An example:

```
on ServiceProductEvent as spe merge OrdersWindow as win
where win.id = spe.id when not matched then insert select *
```

5.15.2.1. Named Windows Holding Decorated Events

Decorated events hold an underlying event and add additional properties to the underlying event, as described further in *Section 5.10.4, “Decorated Events”*.

Here we create a named window that decorates OrderEvent events by adding an additional property named `priceTotal` to each OrderEvent. A matching `insert into` statement is also part of the sample:

```
create window OrdersWindow.win:time(30) as select *, price as priceTotal from
 OrderEvent
```

```
insert into OrdersWindow select *, price * unit as priceTotal from
 ServiceOrderEvent
```

The property type of the additional `priceTotal` column is the property type of the existing `price` property of OrderEvent.

5.15.2.2. Named Windows Holding Events As Property

Columns in a named window may also hold an event itself. More information on the `insert into` clause providing event columns is in *Section 5.10.5, “Event as a Property”*.

The next sample creates a named window that specifies two columns: A column that holds an OrderEvent, and a column by name `priceTotal`. A matching `insert into` statement is also part of the sample:

```
create window OrdersWindow.win:time(30) as select this, price as priceTotal from
 OrderEvent
```

```
insert into OrdersWindow select order, price * unit as priceTotal
from ServiceOrderEvent as order
```

Note that the `this` proprerty must exist on the event and must return the event class itself (JavaBean events only). The property type of the additional `priceTotal` column is the property type of the existing `price` property.

5.15.3. Inserting Into Named Windows Using Fire-And-Forget Queries

Your application can insert rows into a named window using on-demand (fire-and-forget, non-continuous) queries as described in *Section 14.5, “On-Demand Fire-And-Forget Query Execution”*.

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

```
insert into window_name [(property_names)]
select value_expressions
```

The *window_name* is the name of the named window to insert events into.

After the named window name you can optionally provide a comma-separated list of event property names. When providing property names, the order of expressions in the select-clause must match the order of property names specified (select-clause property names are ignored). When not providing property names, the select-clause expressions must name the event properties to be inserted into by assigning a property name.

It follows the `select` keyword and a list of value expressions that provide values for the event properties of the event to be inserted into the named window.

The example code snippet inserts a new order event into the `OrdersWindow` named window:

```
String query =
  "insert into OrdersWindow(orderId, symbol, price) select '001', 'GE', 100";
epService.getEPRuntime().executeQuery(query);
```

If you do not specify event property names, the select-clause expressions must name the event property names.

The following EPL inserts the same values as above but specifies property names as part of the select-clause expressions:

```
insert into OrdersWindow
select '001' as orderId, 'GE' as symbol, 100 as price
```

5.15.4. 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 stream of the named window and reports that stream to its listeners. The `iterator` method returns returns all events in the named window, if any.

If your application desires to obtain the events removed from the named window, use the `rstream` keyword as this statement shows:

```
select rstream * from AllOrdersNamedWindow
```

The next statement derives an average price per symbol for the events held by the named window:

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

A statement that consumes from a named window, like the one above, receives the insert and remove stream of the named window. The insert stream represents the events inserted into the named window. The remove stream represents the events expired from the named window data window and the events explicitly deleted via `on-delete` for on-demand (fire-and-forget) `delete`.

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. A consuming statement also sees the remove stream of an already populated named window, if any.

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 `std:groupwin` grouped-window 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)
```

5.15.5. 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 stream_name]
[insert into insert_into_def]
select select_list
from window_name [as stream_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 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 stream name. Patterns or named windows can also be specified in the `on` clause, see the samples in *Section 5.15.10.1, “Using Patterns in the On Delete Clause”*.

The *insert into* clause works as described in *Section 5.10, “Merging Streams and Continuous Insertion: the Insert Into Clause”*. The *select* clause is described in *Section 5.3, “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 a stream name 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 streamname.*` .

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.

When using wildcard (*) to select from streams in an on-select clause, each stream, that is the the triggering stream and the selected-upon named window, are selected, similar to a join. Therefore your wildcard select returns two columns: the triggering event and the selection result event, for each row.

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

The query above returns a `queryEvent` column and a `win` column for each event. If only a single stream's event is desired in the result, use `select win.*` instead.

To trigger an on-select when an update to the selected named window occurs or when the triggering event is the same event that is being inserted into the named window, specify the named window name as the event type.

The next query fires the select for every change to the named window OrdersNamedWindow:

```
on OrdersNamedWindow as trig
select onw.symbol, sum(onw.volume)
from OrdersNamedWindow as onw
where onw.symbol = trig.symbol
```

5.15.5.1. On-Select Compared To Join

`On-select` and the unidirectional join can be compared as follows.

`On-select`, `on-merge`, `on-insert`, `on-delete`, `on-update` and `on-select-and-delete` operate only on named windows. Unidirectional joins however can operate on any stream. If the unidirectional join is between a single named window and a triggering event or pattern and that triggering event or pattern is marked unidirectional, the unidirectional join is equivalent to `on-select`.

Execution aspects differ in terms of locking. An `on-select` statement executes under a shareable named window context partition lock. A unidirectional join does not execute under such named window context partition lock and instead is a consumer relationship to the named window (see above).

5.15.6. Triggered Select+Delete on Named Windows: the `On Select Delete` clause

The `on select delete` clause performs a one-time, non-continuous query on a named window every time a triggering event arrives or a triggering pattern matches, similar to `on-select` as described in the previous section. In addition, any selected rows are also deleted.

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

```
on trigger
select [and] delete select_list...
... (please see on-select for insert into, from, group by, having, order
 by)...
```

The syntax follows the syntax of `on-select` as described earlier. The `select` clause follows the optional `and` keyword and the `delete` keyword.

The example statement below selects and deletes all rows from OrdersNamedWindow when a QueryEvent arrives:

```
on QueryEvent select and delete window(win.*) as rows from OrdersNamedWindow
 as win
```

The sample EPL above also shows the use of the `window` aggregation function. It specifies the `window` aggregation function to instruct the engine to output a single event, regardless of the number of rows in the named window, and that contains a column `rows` that contains a collection of the selected event's underlying objects.

5.15.7. Populating a Named Window from an Existing Named Window

Your EPL statement may specify the name of an existing named window when creating a new named window, and may use the `insert` keyword to indicate that the new named window is to be populated from the events currently held by the existing named window.

For example, and assuming the named window `OrdersNamedWindow` already exists, this statement creates a new named window `ScratchOrders` and populates all orders in `OrdersNamedWindow` into the new named window:

```
create window ScratchOrders.win:keepall() as OrdersNamedWindow insert
```

The `where` keyword is also available to perform filtering, for example:

```
create window ScratchBuyOrders.win:time(10) as OrdersNamedWindow insert where
 side = 'buy'
```

5.15.8. Updating Named Windows: the `On Update` clause

An `on update` clause updates events held by a named window. The clause can be used to update all events, or only events that match certain criteria, or events that correlate with an arriving event or a pattern of arriving events.

For updating a named window and for simulateously checking if the updated row exists in the named window or for atomic update-insert operation on a named window, consider using `on-merge` as described in *Section 5.15.12, "Triggered Upsert using the On-Merge Clause"*. On-merge is similar to the SQL `merge` clause and provides what is known as an "Upsert" operation: Update existing events or if no existing event(s) are found then insert a new event, all in one atomic operation provided by a single EPL statement.

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

```
on event_type[(filter_criteria)] [as stream_name]
update window_name [as stream_name]
set mutation_expression [, mutation_expression [,...]]
[where criteria_expression]
```

The *event_type* is the name of the type of events that trigger an update of rows in a 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 a name for use in expressions and the `where` clause. Patterns and named windows can also be specified in the `on` clause.

The *window_name* is the name of the named window to update events. The `as` keyword is also available to assign a name to the named window.

After the `set` keyword follows a list of comma-separated *mutation_expression* expressions. A mutation expression is any valid EPL expression. Subqueries may by part of expressions however aggregation functions and the `prev` or `prior` function may not be used in expressions.

The below table shows some typical mutation expessions:

Table 5.6. Mutation Expressions in Named Window Update And Merge

Description	Syntax and Examples

Assignment	`property_name = value_expression`
	
	`price = 10, side = 'BUY'`

Event Method Invocation	`alias_or_windowname.methodname(...)`
	
	`orderWindow.clear()`

User-Defined Function Call	`functionname(...)`
	
	`clearQuantities(orderRow)`

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

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

Statements that reference the named window receive the new event in the insert stream and the event prior to the update in the remove stream.

Let's look at a couple of examples. In the simplest form, this statement updates all events in the named window 'AllOrdersNamedWindow' when any 'UpdateOrderEvent' event arrives, setting the price property to zero for all events currently held by the named window:

```
on UpdateOrderEvent update AllOrdersNamedWindow set price = 0
```

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

```
on ZeroVolumeEvent update AllOrdersNamedWindow set price = 0 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
update AllOrdersNamedWindow as myNamedWindow
set price = myNewOrders.price
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 updated (their price property is set to that of the arriving event). The statement also showcases the `as` keyword to assign a name 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 update of events.

Your application can subscribe a listener to your `on update` statements to determine update events. The statement post any events that are updated to all listeners attached to the statement as new data, and the events prior to the update as old data. Upon iteration, the statement provides the last update event, if any.

The following example shows the use of tags and a pattern. It sets the price value of orders to that of either a 'FlushOrderEvent' or 'OrderUpdateEvent' depending on which arrived:

```
on pattern [every ord=OrderUpdateEvent(volume>0) or every flush=FlushOrderEvent]
update AllOrdersNamedWindow as win
set price = case when ord.price is null then flush.price else ord.price end
where ord.id = win.id or flush.id = win.id
```

When updating indexed properties use the syntax *propertyName*`[`*index*`] = ` *value* with the index value being an integer number. When updating mapped properties use the syntax *propertyName*`(`*key*`) = ` *value* with the key being a string value.

The engine executes assignments in the order they are listed. When performing multiple assignments, the engine takes the most recent named window event property value according to the last assignment, if any. To instruct the engine to use the initial property value before update, prefix the event property name with the literal `initial`.

The following statement illustrates:

```
on UpdateEvent as upd
update MyWindow as win
set field_a = 1,
  field_b = win.field_a, // assigns the value 1
  field_c = initial.field_a // assigns the field_a original value before update
```

The next example assumes that your application provides a user-defined function `copyFields` that receives 3 parameters: The update event, the new row and the initial state before-update row.

```
on UpdateEvent as upd update MyWindow as win set copyFields(win, upd, initial)
```

The following example assumes that your event type provides a method by name `populateFrom` that receives the update event as a parameter:

```
on UpdateEvent as upd update MyWindow as win set win.populateFrom(upd)
```

The following restrictions apply:

1. Each property to be updated via assignment must be writable.
2. For underlying event representations that are Java objects, a event object class must implement the java.io.Serializable interface as discussed in *Section 5.21.1, “Immutability and Updates”* and must provide setter methods for updated properties.
3. When using an XML underlying event type, event properties in the XML document representation are not available for update.
4. Nested properties are not supported for update. Revision event types and variant streams may also not be updated.

5.15.9. Updating Named Windows Using Fire-And-Forget Queries

Your application can update named window rows using on-demand (fire-and-forget, non-continuous) queries as described in *Section 14.5, “On-Demand Fire-And-Forget Query Execution”*.

The syntax for the `update` clause is as follows:

```
update window_name [as stream_name]
set mutation_expression [, mutation_expression [,...]]
[where criteria_expression]
```

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

After the `set` keyword follows a comma-separated list of mutation expressions. For fire-and-forget queries the following restriction applies: Subqueries, aggregation functions and the `prev` or `prior` function may not be used in expressions. Mutation expressions are detailed in *Section 5.15.8, “Updating Named Windows: the On Update clause”*.

The optional `where` clause contains a *criteria_expression* that identifies events to be updated.

The example code snippet updates those rows of the named window that have a negative value for volume:

```
String query = "update AllOrdersNamedWindow set volume = 0 where volumne = 0";
epService.getEPRuntime().executeQuery(query);
```

To instruct the engine to use the initial property value before update, prefix the event property name with the literal `initial`.

5.15.10. 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 stream_name]
delete from window_name [as stream_name]
[where criteria_expression]
```

The *event_type* is the name 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 a name for use in the `where` clause. Patterns and named windows 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 a name 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 a name 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.

5.15.10.1. 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 stream_name]
delete from window_name [as stream_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.

5.15.11. Deleting From Named Windows Using Fire-And-Forget Queries

Your application can delete rows from a named window using on-demand (fire-and-forget, non-continuous) queries as described in *Section 14.5, “On-Demand Fire-And-Forget Query Execution”*.

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

```
delete from window_name [as stream_name]
[where criteria_expression]
```

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

The optional `where` clause contains a *criteria_expression* that identifies events to be removed from the named window.

The example code snippet deletes from a named window all rows that have a negative value for volume:

```
String query = "delete from AllOrdersNamedWindow where volume <= 0";
epService.getEPRuntime().executeQuery(query);
```

5.15.12. Triggered Upsert using the `On-Merge` Clause

The `on merge` clause is similar to the SQL `merge` clause. It provides what is known as an "Upsert" operation: Update existing events or if no existing event(s) are found then insert a new event, all in an atomic operation provided by a single EPL statement.

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

```
on event_type[(filter_criteria)] [as stream_name]
merge [into] window_name [as stream_name]
[where criteria_expression]
  when [not] matched [and search_condition]
    then [
      insert [into streamname]
          [ (property_name [, property_name] [,...]) ]
          select select_expression [, select_expression[,...]]
          [where filter_expression]
      |
      update set mutation_expression [, mutation_expression [,...]]
          [where filter_expression]
      |
      delete
          [where filter_expression]
    ]
    [then [insert|update|delete]] [,then ...]
  [when ...  then ... [...]]
```

The *event_type* is the name of the type of events that trigger the merge. 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 a name for use in the `where` clause. Patterns and named windows can also be specified in the `on` clause as described in prior sections.

The *window_name* is the name of the named window to insert, update or delete events. The `as` keyword is also available to assign a name to the named window.

The optional `where` clause contains a *criteria_expression* that correlates the arriving (triggering) event to the events to be considered of the named window. We recommend specifying a criteria expression that is as specific as possible.

Following the `where` clause is one or more `when matched` or `when not matched` clauses in any order. Each may have an additional search condition associated.

After each `when [not] matched` follow one or more `then` clauses that each contain the action to take: Either an `insert`, `update` or `delete` keyword.

After `when not matched` only `insert` action(s) are available. After `when matched` any `insert`, `update` and `delete` action(s) are available.

After `insert` follows, optionally, the `into` keyword followed by the stream name or named window to insert-into. If no `into` and stream name is specified, the insert applies to the current named window. It follows an optional list of columns inserted. It follows the required `select` keyword and one or more select-clause expressions. The wildcard (`*`) is available in the select-clause as well. It follows an optional where-clause that may return Boolean false to indicate that the action should not be applied.

After `update` follows the `set` keyword and one or more mutation expressions. For mutation expressions please see *Section 5.15.8, "Updating Named Windows: the On Update clause"*. It follows an optional where-clause that may return Boolean false to indicate that the action should not be applied.

After `delete` follows an optional where-clause that may return Boolean false to indicate that the action should not be applied.

When according to the where-clause *criteria_expression* the engine finds no events in the named window that match the condition, the engine evaluates each *when not matched* clause. If the optional search condition returns true or no search condition was provided then the engine performs all of the actions listed after each `then`.

When according to the where-clause *criteria_expression* the engine finds one or more events in the named window that match the condition, the engine evaluates each *when matched* clause. If the optional search condition returns true or no search condition was provided the engine performs all of the actions listed after each `then`.

The engine executes `when matched` and `when not matched` in the order specified. If the optional search condition returns true or no search condition was specified then the engine takes the associated action (or multiple actions for multiple `then` keywords). When the block of actions completed the engine proceeds to the next matching event, if any. After completing all matching events the engine continues to the next triggering event if any.

In the first example we declare a schema that provides a product id and that holds a total price:

```
create schema ProductTotalRec as (productId string, totalPrice double)
```

We create a named window that holds a row for each unique product:

```
create window ProductWindow.std:unique(productId) as ProductTotalRec
```

The events for this example are order events that hold an order id, product id, price, quantity and deleted-flag declared by the next schema:

```
create schema OrderEvent as (orderId string, productId string, price double,
    quantity int, deletedFlag boolean)
```

The following EPL statement utilizes `on-merge` to total up the price for each product based on arriving order events:

```
on OrderEvent oe
  merge ProductWindow pw
  where pw.productId = oe.productId
  when matched
    then update set totalPrice = totalPrice + oe.price
  when not matched
    then insert select productId, price as totalPrice
```

In the above example, when an order event arrives, the engine looks up in the product named window the matching row or rows for the same product id as the arriving event. In this example the engine always finds no row or one row as the product named window is declared with a unique data window based on product id. If the engine finds a row in the named window, it performs the update action adding up the price as defined under `when matched`. If the engine does not find a row in the named window it performs the insert action as defined under `when not matched`, inserting a new row.

The `insert` keyword may be followed by a list of columns as shown in this EPL snippet:

```
// equivalent to the insert shown in the last 2 lines in above EPL
...when not matched
    then insert(productId, totalPrice) select productId, price
```

The second example demonstrates the use of a select-clause with wildcard, a search condition and the `delete` keyword. It creates a named window that holds order events and employs on-merge to insert order events for which no corresponding order id was found, update quantity to the quantity provided by the last arriving event and delete order events that are marked as deleted:

```
create window OrderWindow.win:keepall() as OrderEvent
```

```
on OrderEvent oe
  merge OrderWindow pw
  where pw.orderId = oe.orderId
  when not matched
    then insert select *
  when matched and oe.deletedFlag=true
    then delete
  when matched
    then update set pw.quantity = oe.quantity, pw.price = oe.price
```

In the above example the `oe.deletedFlag=true` search condition instructs the engine to take the delete action only if the deleted-flag is set.

You may specify multiple actions by providing multiple `then` keywords each followed by an action. Each of the `insert`, `update` and `delete` actions can itself have a where-clause as well. If a where-clause exists for an action, the engine evaluates the where-clause and applies the action only if the where-clause returns Boolean true.

This example specifies two update actions and uses the where-clause to trigger different update behavior depending on whether the order event price is less than zero. This example assumes that the host application defined a `clearorder` user-defined function, to demonstrate calling a user-defined function as part of the update mutation expressions:

```
on OrderEvent oe
  merge OrderWindow pw
  where pw.orderId = oe.orderId
  when matched
    then update set clearorder(pw) where oe.price < 0
   then update set pw.quantity = oe.quantity, pw.price = oe.price where oe.price
 >= 0
```

To insert events into another stream and not the named window, use `insert into` *streamname*.

In the next example each matched-clause contains two actions, one action to insert a log event and a second action to insert, delete or update:

```
on OrderEvent oe
  merge OrderWindow pw
  where pw.orderId = oe.orderId
  when not matched
    then insert into LogEvent select 'this is an insert' as name
    then insert select *
  when matched and oe.deletedFlag=true
    then insert into LogEvent select 'this is a delete' as name
    then delete
  when matched
    then insert into LogEvent select 'this is a update' as name
    then update set pw.quantity = oe.quantity, pw.price = oe.price
```

While the engine evaluates and executes all actions listed under the same matched-clause in order, you may not rely on updated field values of an earlier action to trigger the where-clause of a later action. Similarly you should avoid simultaneous update and delete actions for the same match: the engine does not guarantee whether the update or the delete take final affect.

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 update and removal of events especially from named windows that hold large numbers of events.

Your application can subscribe a listener to `on merge` statements to determine inserted, updated and removed events. Statements post any events that are inserted to, updated or deleted from a named window to all listeners attached to the statement as new data and removed data. Upon iteration, the statement provides the last inserted events, if any.

The following limitations apply to on-merge statements:

1. Aggregation functions and the `prev` and `prior` operators are not available in conditions and the `select`-clause.

5.15.13. Explicitly Indexing Named Windows

You may explicitly create an index on a named window. The engine considers explicitly-created as well as implicitly-allocated indexes in query planning and execution of the following types of usages of named windows:

1. On-demand (fire-and-forget, non-continuous) queries as described in *Section 14.5, “On-Demand Fire-And-Forget Query Execution”*.

   `On-select`, `on-merge`, `on-update`, `on-delete` and `on-insert`.

   Subqueries against named windows.

   For joins (including outer joins) with named windows the engine considers the filter criteria listed in parenthesis using the syntax

   ```
   name_window_name(filter_criteria)
   ```

   for index access.

Please use the following syntax to create an explicit index on a named window:

```
create [unique] index index_name on named_window_name (property [hash|
btree]
    [, property] [hash|btree] [,...] )
```

The optional *unique* keyboard indicates that the property or properties uniquely identify rows. If *unique* is not specified the index allows duplicate rows.

The *index_name* is the name assigned to the index. The name uniquely identifies the index and is used in engine query plan logging.

The *named_window_name* is the name of an existing named window. If the named window has data already, the engine builds an index for the data in the named window.

The list of *property* names are the properties of events within the named window to include in the index. Following each property name you may specify the optional `hash` or `btree` keyword.

If you specify no keyword or the `hash` keyword for a property, the index will be a hash-based (unsorted) index in respect to that property. If you specify the `btree` keyword, the index will be a binary-tree-based sorted index in respect to that property. You may combine `hash` and `btree` properties for the same index. Specify `btree` for a property if you expect to perform numerical or string comparison using relational operators (<, >, >=, <=), the `between` or the `in` keyword for ranges and inverted ranges. Use `hash` (the default) instead of `btree` if you expect to perform exact comparison using `=`.

We list a few example EPL statements next that create a named window and create a single index:

```
// create a named window
create window UserProfileWindow.win:time(1 hour) select * from UserProfile
```

```
// create a non-unique index (duplicates allowed) for the user id property only
create index UserProfileIndex on UserProfileWindow(userId)
```

Next, execute an on-demand fire-and-forget query as shown below, herein we use the prepared version to demonstrate:

```
String query = "select * from UserProfileWindow where userId='Joe'";
EPOnDemandPreparedQuery prepared = epRuntime.prepareQuery(query);
// query performance excellent in the face of large number of rows
EPOnDemandQueryResult result = prepared.execute();
// ...later ...
prepared.execute(); // execute a second time
```

A unique index is generally preferable over non-unique indexes. If your data window declares a unique data window (`std:unique`, `std:firstunique`, including intersections and grouped unique data windows) it is not necessary to create a unique index unless index sharing is enabled, since the engine considers the unique data window declaration in query planning.

The engine enforces uniqueness (e.g. unique constraint) for unique indexes. If your application inserts a duplicate row the engine raises a runtime exception when processing the statement and discards the row. The default error handler logs such an exception and continues.

For example, if the user id together with the profile id uniquely identifies an entry into the named window, your application can create a unique index as shown below:

```
// create a unique index on user id and profile id
create unique index UserProfileIndex on UserProfileWindow(userId, profileId)
```

By default, the engine builds a hash code -based index useful for direct comparison via equals (=). Filter expressions that look for ranges or use `in`, `between` do not benefit from the hash-based index and should use the `btree` keyword. For direct comparison via equals (=) then engine does not use `btree` indexes.

The next example creates a composite index over two fields `symbol` and `buyPrice`:

```
// create a named window
create window TickEventWindow.win:time(1 hour) as (symbol string, buyPrice
 double)
```

```
// create a non-unique index
create index idx1 on TickEventWindow(symbol hash, buyPrice btree)
```

A sample fire-and-forget query is shown below (this time the API calls are not shown):

```
// query performance excellent in the face of large number of rows
select * from TickEventWindow where symbol='GE' and buyPrice between 10 and 20
```

5.15.14. Versioning and Revision Event Type Use with Named Windows

As outlined in *Section 2.10, “Updating, Merging and Versioning Events”*, revision event types process updates or new versions of events held by a named window.

A revision event type is simply one or more existing pre-configured event types whose events are related, as configured by static configuration, by event properties that provide same key values. The purpose of key values is to indicate that arriving events are related: An event amends, updates or adds properties to an earlier event that shares the same key values. No additional EPL is needed when using revision event types for merging event data.

Revision event types can be useful in these situations:

1. Some of your events carry only partial information that is related to a prior event and must be merged together.

2. Events arrive that add additional properties or change existing properties of prior events.
3. Events may carry properties that have null values or properties that do no exist (for example events backed by Map or XML), and for such properties the earlier value must be used instead.

To better illustrate, consider a revision event type that represents events for creation and updates to user profiles. Lets assume the user profile creation events carry the user id and a full profile. The profile update events indicate only the user id and the individual properties that actually changed. The user id property shall serve as a key value relating profile creation events and update events.

A revision event type must be configured to instruct the engine which event types participate and what their key properties are. Configuration is described in *Section 15.4.26, "Revision Event Type"* and is not shown here.

Assume that an event type `UserProfileRevisions` has been configured to hold profile events, i.e. creation and update events related by user id. This statement creates a named window to hold the last 1 hour of current profiles per user id:

```
create window UserProfileWindow.win:time(1 hour) select * from
 UserProfileRevisions
```

```
insert into UserProfileWindow select * from UserProfileCreation
```

```
insert into UserProfileWindow select * from UserProfileUpdate
```

In revision event types, the term *base* event is used to describe events that are subject to update. Events that update, amend or add additional properties to base events are termed *delta* events. In the example, base events are profile creation events and delta events are profile update events.

Base events are expected to arrive before delta events. In the case where a delta event arrives and is not related by key value to a base event or a revision of the base event currently held by the named window the engine ignores the delta event. Thus, considering the example, profile update events for a user id that does not have an existing profile in the named window are not applied.

When a base or delta event arrives, the insert and remove stream output by the named window are the current and the prior version of the event. Let's come back to the example. As creation events arrive that are followed by update events or more creation events for the same user id, the engine posts the current version of the profile as insert stream (new data) and the prior version of the profile as remove stream (old data).

Base events are also implicitly delta events. That is, if multiple base events of the same key property values arrive, then each base event provides a new version. In the example, if multiple profile creation events arrive for the same user id then new versions of the current profile for that user id are output by the engine for each base event, as it does for delta events.

The expiry policy as specified by view definitions applies to each distinct key value, or multiple distinct key values for composite keys. An expiry policy re-evaluates when new versions arrive. In the example, user profile events expire from the time window when no creation or update event for a given user id has been received for 1 hour.

Tip

It usually does not make sense to configure a revision event type without delta event types. Use the unique data window (`std:unique`) or unique data window in intersection with other data windows instead (i.e. `std:unique(field).win:time(1 hour)`).

Several strategies are available for merging or overlaying events as the configuration chapter describes in greater detail.

Any of the Map, XML and JavaBean event representations as well as plug-in event representations may participate in a revision event type. For example, profile creation events could be JavaBean events, while profile update events could be `java.util.Map` events.

Delta events may also add properties to the revision event type. For example, one could add a new event type with security information to the revision event type and such security-related properties become available on the resulting revision event type.

The following restrictions apply to revision event types:

- Nested properties are only supported for the JavaBean event representation. Nested properties are not individually versioned; they are instead versioned by the containing property.
- Dynamic, indexed and mapped properties are only supported for nested properties and not as properties of the revision event type itself.

5.16. Declaring an Event Type: *Create Schema*

EPL allows declaring an event type via the `create schema` clause and also by means of the static or runtime configuration API `addEventType` functions. The term schema and event type has the same meaning in EPL.

Your application can declare an event type by providing the property names and types or by providing a class name. Your application may also declare a variant stream schema.

When using the `create schema` syntax to declare an event type, the engine automatically removes the event type when there are no started statements referencing the event type, including the statement that declared the event type. When using the configuration API, the event type stays cached even if there are no statements that refer to the event type and until explicitly removed via the runtime configuration API.

5.16.1. Declare an Event Type by Providing Names and Types

The synopsis of the `create schema` syntax providing property names and types is:

```
create [map | objectarray] schema schema_name [as]
    (property_name property_type [,property_name property_type [,...])
  [inherits inherited_event_type[, inherited_event_type] [,...]]
  [starttimestamp timestamp_property_name]
  [endtimestamp timestamp_property_name]
  [copyfrom copy_type_name [, copy_type_name] [,...]]
```

The `create` keyword can be followed by `map` to instruct the engine to represent events of that type by the Map event representation, or `objectarray` to denote an Object-array event type. If neither the `map` or `objectarray` keywords are provided, the engine-wide default event representation applies.

After `create schema` follows a *schema_name*. The schema name is the event type name.

The *property_name* is an identifier providing the event property name. The *property_type* is also required for each property. Valid property types are listed in *Section 5.18.1, "Creating Variables: the Create Variable clause"* and in addition include:

1. Any Java class name, fully-qualified or the simple class name if imports are configured.
2. Add left and right square brackets `[]` to any type to denote an array-type event property.
3. Use an event type name as a property type.

The optional `inherits` keywords is followed by a comma-separated list of event type names that are the supertypes to the declared type.

The optional `starttimestamp` keyword is followed by a property name. Use this to tell the engine that your event has a timestamp. The engine checks that the property name exists on the declared type and returns a date-time value. Declare a timestamp property if you want your events to implicitly carry a timestamp value for convenient use with interval algebra methods as a start timestamp.

The optional `endtimestamp` keyword is followed by a property name. Use this together with starttimestamp to tell the engine that your event has a duration. The engine checks that the property name exists on the declared type and returns a date-time value. Declare an endtimestamp property if you want your events to implicitly carry a duration value for convenient use with interval algebra methods.

The optional `copyfrom` keyword is followed by a comma-separate list of event type names. For each event type listed, the engine looks up that type and adds all event property definitions to the newly-defined type, in addition to those listed explicitly (if any).

A few example event type declarations follow:

```
// Declare type SecurityEvent
create schema SecurityEvent as (ipAddress string, userId String, numAttempts int)

// Declare type AuthorizationEvent with the roles property being an array of
 String
```

```
// and the hostinfo property being a POJO object
create schema AuthorizationEvent(group String, roles String[], hostinfo
 com.mycompany.HostNameInfo)

// Declare type CompositeEvent in which the innerEvents property is an array
 of SecurityEvent
create schema CompositeEvent(group String, innerEvents SecurityEvent[])

// Declare type WebPageVisitEvent that inherits all properties from PageHitEvent
create schema WebPageVisitEvent(userId String) inherits PageHitEvent

// Declare a type with start and end timestamp (i.e. event with duration).
create schema RoboticArmMovement (robotId string, startts long, endts long)
  starttimestamp startts endtimestamp endts

// Create a type that has all properties of SecurityEvent plus a userName property
create schema ExtendedSecurityEvent (userName string) copyfrom SecurityEvent

// Create a type that has all properties of SecurityEvent
create schema SimilarSecurityEvent () copyfrom SecurityEvent

// Create a type that has all properties of SecurityEvent and WebPageVisitEvent
 plus a userName property
create schema WebSecurityEvent (userName string) copyfrom SecurityEvent,
 WebPageVisitEvent
```

To elaborate on the `inherits` keyword, consider the following two schema definitions:

```
create schema Foo as (string prop1)
```

```
create schema Bar() inherits Foo
```

Following above schema, Foo is a supertype or Bar and therefore any Bar event also fulfills Foo and matches where Foo matches. An EPL statement such as `select * from Foo` returns any Foo event as well as any event that is a subtype of Foo such as all Bar events. When your EPL queries don't use any Foo events there is no cost, thus `inherits` is generally an effective way to share properties between types. The start and end timestamp are also inherited from any supertype that has the timestamp property names defined.

The optional `copyfrom` keyword is for defining a schema based on another schema. This keyword causes the engine to copy property definitions: There is no inherits, extends, supertype or subtype relationship between the types listed.

To define an event type `Bar` that has the same properties as `Foo`:

```
create schema Foo as (string prop1)
```

```
create schema Bar() copyfrom Foo
```

To define an event type `Bar` that has the same properties as `Foo` and that adds its own property `prop2`:

```
create schema Foo as (string prop1)
```

```
create schema Bar(string prop2) copyfrom Foo
```

If neither the `map` or `objectarray` keywords are provided, and if the create-schema statement provides the `@EventRepresentation(array=true)` annotation the engine expects object array events. If the statement provides the `@EventRepresentation(array=false)` annotation the engine expects Map objects as events. If neither annotation is provided, the engine uses the configured default event representation as discussed in *Section 15.4.11.1, “Default Event Representation”*.

The following two EPL statements both instructs the engine to represent Foo events as object arrays. When sending Foo events into the engine use the `sendEvent(Object[] data, String typeName)` footprint.

```
create objectarray schema Foo as (string prop1)
```

```
@EventRepresentation(array=true) create schema Foo as (string prop1)
```

The next two EPL statements both instructs the engine to represent Foo events as Maps. When sending Foo events into the engine use the `sendEvent(Map data, String typeName)` footprint.

```
create map schema Foo as (string prop1)
```

```
@EventRepresentation(array=false) create schema Foo as (string prop1)
```

5.16.2. Declare an Event Type by Providing a Class Name

When using Java classes as the underlying event representation your application may simply provide the class name:

```
create schema schema_name [as] class_name
  [starttimestamp timestamp_property_name]
  [endtimestamp timestamp_property_name]
```

The *class_name* must be a fully-qualified class name (including the package name) if imports are not configured. If you application configures imports then the simple class name suffices without package name.

The optional `starttimestamp` and `endtimestamp` keywords have a meaning as defined earlier.

The next example statements declare an event type based on a class:

```
// Shows the use of a fully-qualified class name to declare the LoginEvent
 event type
create schema LoginEvent as com.mycompany.LoginValue

// When the configuration includes imports, the declaration does not need a
 package name
create schema LogoutEvent as SignoffValue
```

5.16.3. Declare a Variant Stream

A variant stream is a predefined stream into which events of multiple disparate event types can be inserted. Please see *Section 5.10.3, “Merging Disparate Types of Events: Variant Streams”* for rules regarding property visibility and additional information.

The synopsis is:

```
create variant schema schema_name [as] eventtype_name|* [, eventtype_name|*]
 [,...]
```

Provide the `variant` keyword to declare a variant stream.

The '`*`' wildcard character declares a variant stream that accepts any type of event inserted into the variant stream.

Provide *eventtype_name* if the variant stream should hold events of the given type only. When using `insert into` to insert into the variant stream the engine checks to ensure the inserted event type or its supertypes match the required event type.

A few examples are shown below:

```
// Create a variant stream that accepts only LoginEvent and LogoutEvent event
 types
create variant schema SecurityVariant as LoginEvent, LogoutEvent

// Create a variant stream that accepts any event type
create variant schema AnyEvent as *
```

5.17. Splitting and Duplicating Streams

EPL offers a convenient syntax to splitting, routing or duplicating events into multiple streams, and for receiving unmatched events among a set of filter criteria.

For splitting a single event that acts as a container and expose child events as a property of itself consider the contained-event syntax as described in *Section 5.20, “Contained-Event Selection”*.

You may define a triggering event or pattern in the `on`-part of the statement followed by multiple `insert into`, `select` and `where` clauses.

The synopsis is:

```
[context context_name]
on event_type[(filter_criteria)] [as stream_name]
insert into insert_into_def select select_list [where condition]
[insert into insert_into_def select select_list [where condition]]
[insert into...]
[output first | all]
```

The *event_type* is the name of the type of events that trigger the split stream. 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 a stream name. Patterns and named windows can also be specified in the `on` clause.

Following the `on`-clause is one or more *insert into* clauses as described in *Section 5.10, “Merging Streams and Continuous Insertion: the Insert Into Clause”* and *select* clauses as described in *Section 5.3, “Choosing Event Properties And Events: the Select Clause”*.

Each `select` clause may be followed by a `where` clause containing a condition. If the condition is true for the event, the engine transforms the event according to the `select` clause and inserts it into the corresponding stream.

At the end of the statement can be an optional `output` clause. By default the engine inserts into the first stream for which the `where` clause condition matches if one was specified, starting from the top. If you specify the `output all` keywords, then the engine inserts into each stream (not only the first stream) for which the `where` clause condition matches or that do not have a `where` clause.

If, for a given event, none of the `where` clause conditions match, the statement listener receives the unmatched event. The statement listener only receives unmatched events and does not receive any transformed or inserted events. The `iterator` method to the statement returns no events.

You may specify an optional context name to the effect that the split-stream operates according to the context dimensional information as declared for the context. See *Chapter 4, Context and Context Partitions* for more information.

In the below sample statement, the engine inserts each `OrderEvent` into the `LargeOrders` stream if the order quantity is 100 or larger, or into the `SmallOrders` stream if the order quantity is smaller then 100:

```
on OrderEvent
  insert into LargeOrders select * where orderQty >= 100
  insert into SmallOrders select *
```

The next example statement adds a new stream for medium-sized orders. The new stream receives orders that have an order quantity between 20 and 100:

```
on OrderEvent
  insert into LargeOrders select orderId, customer where orderQty >= 100
  insert into MediumOrders select orderId, customer where orderQty between 20
 and 100
  insert into SmallOrders select orderId, customer where orderQty > 0
```

As you may have noticed in the above statement, orders that have an order quantity of zero don't match any of the conditions. The engine does not insert such order events into any stream and the listener to the statement receives these unmatched events.

By default the engine inserts into the first `insert into` stream without a `where` clause or for which the `where` clause condition matches. To change the default behavior and insert into all matching streams instead (including those without a `where` clause), the `output all` keywords may be added to the statement.

The sample statement below shows the use of the `output all` keywords. The statement populates both the `LargeOrders` stream with large orders as well as the `VIPCustomerOrders` stream with orders for certain customers based on customer id:

```
on OrderEvent
  insert into LargeOrders select * where orderQty >= 100
  insert into VIPCustomerOrders select * where customerId in (1001, 1002)
  output all
```

Since the `output all` keywords are present, the above statement inserts each order event into either both streams or only one stream or none of the streams, depending on order quantity and customer id of the order event. The statement delivers order events not inserted into any of the streams to the listeners and/or subscriber to the statement.

The following limitations apply to split-stream statements:

1. Aggregation functions and the `prev` and `prior` operators are not available in conditions and the `select`-clause.

5.18. Variables and Constants

A *variable* is a scalar, object or event 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 runtime or static 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.

A variable can be declared constant. A constant variable always has the initial value and cannot be assigned a new value. A constant variable can be used like any other variable and can be used wherever a constant is required. By declaring a variable constant you enable the Esper engine to optimize and perform query planning knowing that the variable value cannot change.

When declaring a class-type or an event type variable you may read or set individual properties within the same variable.

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

Variables can also be removed, at runtime, by destroying all referencing statements including the statement that created the variable, or by means of the runtime configuration API.

5.18.1. 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 [constant] variable variable_type [[]] variable_name
 [ = assignment_expression ]
```

Specify the optional `constant` keyword when the variable is a constant whose associated value cannot be altered. Your EPL design should prefer constant variables over non-constant variables.

The *variable_type* can be any of the following:

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

All variable types accept null values. The `object` type is for an untyped variable that can be assigned any value. You can provide a class name (use imports) or a fully-qualified class name to declare a variable of that Java class type including an enumeration class. You can also supply the name of an event type to declare a variable that holds an event of that type.

Append `[]` to the variable type to declare an array variable. A limitation is that if your variable type is an event type then array is not allowed.

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
```

The next statement declares a constant string-type variable:

```
create constant variable string const_filter_symbol = 'GE'
```

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");
```

The following example declares a constant that is an array of string:

```
create constant variable string[] const_filters = {'GE', 'MSFT'}
```

The next example declares a constant that is an array of enumeration values. It assumes the `Color` enumeration class was imported:

```
create constant variable Color[] const_colors = {Color.RED, Color.BLUE}
```

The engine removes the variable if the statement that created the variable is destroyed and all statements that reference the variable are also destroyed. The `getVariableNameUsedBy` and the `removeVariable` methods, both part of the runtime `ConfigurationOperations` API, provide use information and can remove a variable. If the variable was added via configuration, it can only be removed via the configuration API.

5.18.2. 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 stream_name]
  set variable_name = expression [, variable_name = expression [,...]]
```

The *event_type* is the name 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 stream name. Patterns and named windows 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. Subqueries may by part of expressions however aggregation functions and the `prev` or `prior` function may not be used in expressions.

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
```

5.18.3. 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.

5.18.4. Object-Type Variables

A variable of type `object` (or `java.lang.Object` via the API) can be assigned any value including null. When using an object-type variable in an expression, your statement may need to cast the value to the desired type.

The following sample EPL creates a variable by name `varobj` of type object:

```
create variable object varobj
```

5.18.5. Class and Event-Type Variables

The `create variable` syntax and the API accept a fully-qualified class name or alternatively the name of an event type. This is useful when you want a single variable to have multiple property values to read or set.

The next statement assumes that the event type `PageHitEvent` is declared:

```
create variable PageHitEvent varPageHitZero
```

These example statements show two ways of assigning to the variable:

```
// You may assign the complete event
on PageHitEvent(ip='0.0.0.0') pagehit set varPageHitZero = pagehit
```

```
// Or assign individual properties of the event
on PageHitEvent(ip='0.0.0.0') pagehit set varPageHitZero.userId = pagehit.userId
```

When using class or event-type variables, in order for the engine to assign property values, the underlying event type must allow writing property values. If using JavaBean event classes the class must have setter methods and a default constructor. The underlying event type must also be copy-able i.e. implement `Serializable` or configure a copy method.

Similarly statements may use properties of class or event-type variables as this example shows:

```
select * from FirewallEvent(userId=varPageHitZero.userId)
```

Instances method can also be invoked:

```
create variable com.example.StateChecker stateChecker
```

```
select * from TestEvent as e where stateChecker.checkState(e)
```

5.19. Declaring Global Expressions And Scripts: *Create Expression*

Your application can declare an expression or script using the `create expression` clause. Such expressions or scripts become available globally to any EPL statement.

The synopsis of the `create expression` syntax is:

```
create expression expression_or_script
```

Use the `create expression` keywords and append the expression or scripts.

At the time your application creates the `create expression` statement the expression or script becomes globally visible.

At the time your application destroys the `create expression` statement the expression or script are no longer visible. Existing statements that use the global expression or script are unaffected.

5.19.1. Declaring a Global Expression

The syntax and additional examples for declaring an expression is outlined in *Section 5.2.8, “Expression Declaration”*, which discusses declaring expressions that are visible within the same EPL statement i.e. visible locally only.

When using the `create expression` syntax to declare an expression the engine remembers the expression and allows the expression to be referenced in all other EPL statements.

The below EPL declares a globally visible expression that computes a mid-price:

```
create expression midPrice { in => (buy + sell) / 2 }
```

The next EPL returns mid-price for each event:

```
select midPrice(md) from MarketDataEvent as md
```

The expression name must be unique for global expressions. It is not possible to declare the same global expression twice with the same name.

Your application can declare an expression of the same name local to a given EPL statement as well as globally using `create expression`. The locally-declared expression overrides the globally declared expression.

The engine validates globally declared expressions at the time your application creates a statement that references the global expression. When a statement references a global expression, the engine uses that statement's type information to validate the global expressions.

Global expressions can therefore be dynamically typed and type information does not need to be the same for all statements that reference the global expression.

This example shows a sequence of EPL, that can be created in the order shown, and that demonstrates expression validation at time of referral:

```
create expression minPrice {(select min(price) from OrderWindow)}
```

```
create window OrderWindow.win:time(30) as OrderEvent
```

```
insert into OrderWindow select * from OrderEvent
```

```
// Validates and incorporates the declared global expression
select minPrice() as minprice from MarketData
```

5.19.2. Declaring a Global Script

The syntax and additional examples for declaring scripts is outlined in *Chapter 18, Script Support*, which discusses declaring scripts that are visible within the same EPL statement i.e. visible locally only.

When using the `create expression` syntax to declare a script the engine remembers the script and allows the script to be referenced in all other EPL statements.

The below EPL declares a globally visible script in the JavaScript dialect that computes a mid-price:

```
create expression midPrice(buy, sell) [ (buy + sell) / 2 ]
```

The next EPL returns mid-price for each event:

```
select midPrice(buy, sell) from MarketDataEvent
```

The engine validates globally declared scripts at the time your application creates a statement that references the global script. When a statement references a global script, the engine uses that statement's type information to determine parameter types. Global scripts can therefore be dynamically typed and type information does not need to be the same for all statements that reference the global script.

The script name in combination with the number of parameters must be unique for global scripts. It is not possible to declare the same global script twice with the same name and number of parameters.

Your application can declare a script of the same name and number of parameters that is local to a given EPL statement as well as globally using `create expression`. The locally-declared script overrides the globally declared script.

5.20. Contained-Event Selection

Contained-event selection is for use when an event contains properties that are themselves events, or more generally when your application needs to split an event into multiple events. One example is when application events are coarse-grained structures and you need to perform bulk operations on the rows of the property graph in an event.

Use the contained-event selection syntax in a filter expression such as in a pattern, `from` clause, subselect, on-select and on-delete. This section provides the synopsis and examples.

To review, in the `from` clause a *contained_selection* may appear after the event stream name and filter criteria, and before any view specifications.

The synopsis for *contained_selection* is as follows:

```
[select select_expressions from]
  contained_expression [@type(eventtype_name)] [as alias_name]
  [where filter_expression]
```

The `select` clause and *select_expressions* are optional and may be used to select specific properties of contained events.

The *contained_expression* is required and returns individual events. The expression can, for example, be an event property name that returns an event fragment, i.e. a property that can itself be represented as an event by the underlying event representation. Simple values such as integer or string are not fragments. The expression can also be any other expression such as a single-row function or a script that returns either an array or a `java.util.Collection` of events.

Provide the `@type(name)` annotation after the contained expression to name the event type of events returned by the expression. The annotation is optional and not needed when the contained-expression is an event property that returns a class or other event fragment.

The *alias_name* can be provided to assign a name to the expression result value rows.

The `where` clause and *filter_expression* is optional and may be used to filter out properties.

As an example event, consider a media order. A media order consists of order items as well as product descriptions. A media order event can be represented as an object graph (POJO event representation), or a structure of nested Maps (Map event representation) or a XML document (XML DOM or Axiom event representation) or other custom plug-in event representation.

To illustrate, a sample media order event in XML event representation is shown below. Also, a XML event type can optionally be strongly-typed with an explicit XML XSD schema that we don't show here. Note that Map and POJO representation can be considered equivalent for the purpose of this example.

Let us now assume that we have declared the event type `MediaOrder` as being represented by the root node `<mediaorder>` of such XML snip:

```
<mediaorder>
  <orderId>PO200901</orderId>
  <items>
    <item>
      <itemId>100001</itemId>
      <productId>B001</productId>
      <amount>10</amount>
      <price>11.95</price>
    </item>
  </items>
  <books>
    <book>
      <bookId>B001</bookId>
      <author>Heinlein</author>
      <review>
        <reviewId>1</reviewId>
        <comment>best book ever</comment>
      </review>
    </book>
    <book>
      <bookId>B002</bookId>
      <author>Isaac Asimov</author>
    </book>
  </books>
</mediaorder>
```

The next query utilizes the contained-event selection syntax to return each book:

```
select * from MediaOrder[books.book]
```

The result of the above query is one event per book. Output events contain only the book properties and not any of the mediaorder-level properties.

Note that, when using listeners, the engine delivers multiple results in one invocation of each listener. Therefore listeners to the above statement can expect a single invocation passing all book events within one media order event as an array.

To better illustrate the position of the contained-event selection syntax in a statement, consider the next two queries:

```
select * from MediaOrder(orderId='PO200901')[books.book]
```

The above query the returns each book only for media orders with a given order id. This query illustrates a contained-event selection and a view:

```
select count(*) from MediaOrder[books.book].std:unique(bookId)
```

The sample above counts each book unique by book id.

Contained-event selection can be staggered. When staggering multiple contained-event selections the staggered contained-event selection is relative to its parent.

This example demonstrates staggering contained-event selections by selecting each review of each book:

```
select * from MediaOrder[books.book][review]
```

Listeners to the query above receive a row for each review of each book. Output events contain only the review properties and not the book or media order properties.

The following is not valid:

```
// not valid
select * from MediaOrder[books.book.review]
```

The `book` property in an indexed property (an array or collection) and thereby requires an index in order to determine which book to use. The expression `books.book[1].review` is valid and means all reviews of the second (index 1) book.

The contained-event selection syntax is part of the filter expression and may therefore occur in patterns and anywhere a filter expression is valid.

A pattern example is below. The example assumes that a `Cancel` event type has been defined that also has an `orderId` property:

```
select * from pattern [c=Cancel -> books=MediaOrder(orderId = c.orderId)
[books.book] ]
```

When used in a pattern, a filter with a contained-event selection returns an array of events, similar to the match-until clause in patterns. The above statement returns, in the `books` property, an array of book events.

5.20.1. Select-Clause in a Contained-Event Selection

The optional `select` clause provides control over which fields are available in output events. The expressions in the select-clause apply only to the properties available underneath the property in the `from` clause, and the properties of the enclosing event.

When no `select` is specified, only the properties underneath the selected property are available in output events.

In summary, the `select` clause may contain:

1. Any expressions, wherein properties are resolved relative to the property in the `from` clause.
2. Use the wildcard (`*`) to provide all properties that exist under the property in the `from` clause.
3. Use the *alias_name*`.*` syntax to provide all properties that exist under a property in the `from` clause.

The next query's `select` clause selects each review for each book, and the order id as well as the book id of each book:

```
select * from MediaOrder[select orderId, bookId from books.book][select * from
 review]
// ... equivalent to ...
select * from MediaOrder[select orderId, bookId from books.book][review]]
```

Listeners to the statement above receive an event for each review of each book. Each output event has all properties of the review row, and in addition the `bookId` of each book and the `orderId` of the order. Thus `bookId` and `orderId` are found in each result event, duplicated when there are multiple reviews per book and order.

The above query uses wildcard `(*)` to select all properties from reviews. As has been discussed as part of the `select` clause, the wildcard (`*`) and *property_alias*`.*` do not copy properties for performance reasons. The wildcard syntax instead specifies the underlying type, and additional properties are added onto that underlying type if required. Only one wildcard (`*`) and *property_alias*`.*` (unless used with a column rename) may therefore occur in the `select` clause list of expressions.

All the following queries produce an output event for each review of each book. The next sample queries illustrate the options available to control the fields of output events.

The output events produced by the next query have all properties of each review and no other properties available:

```
select * from MediaOrder[books.book][review]
```

The following query is not a valid query, since the order id and book id are not part of the contained-event selection:

```
// Invalid select-clause: orderId and bookId not produced.
select orderId, bookId from MediaOrder[books.book][review]
```

This query is valid. Note that output events carry only the `orderId` and `bookId` properties and no other data:

```
select orderId, bookId from MediaOrder[books.book][select orderId, bookId from
 review]
//... equivalent to ...
select * from MediaOrder[select orderId, bookId from books.book][review]
```

This variation produces output events that have all properties of each book and only `reviewId` and `comment` for each review:

```
select * from MediaOrder[select * from books.book][select reviewId, comment from
 review]
// ... equivalent to ...
select * from MediaOrder[books.book as book][select book.*, reviewId, comment
 from review]
```

The output events of the next EPL have all properties of the order and only `bookId` and `reviewId` for each review:

```
select * from MediaOrder[books.book as book]
    [select mediaOrder.*, bookId, reviewId from review] as mediaOrder
```

This EPL produces output events with 3 columns: a column named `mediaOrder` that is the order itself, a column named `book` for each book and a column named `review` that holds each review:

```
insert into ReviewStream select * from MediaOrder[books.book as book]
  [select mo.* as mediaOrder, book.* as book, review.* as review from review
 as review] as mo
```

```
// .. and a sample consumer of ReviewStream...
select mediaOrder.orderId, book.bookId, review.reviewId from ReviewStream
```

Please note these limitations:

1. Sub-selects, aggregation functions and the `prev` and `prior` operators are not available in contained-event selection.
2. Expressions in the `select` and `where` clause of a contained-event selection can only reference properties relative to the current event and property.

5.20.2. Where Clause in a Contained-Event Selection

The optional `where` clause may be used to filter out properties at the same level that the where-clause occurs.

The properties in the filter expression must be relative to the property in the `from` clause or the enclosing event.

This query outputs all books with a given author:

```
select * from MediaOrder[books.book where author = 'Heinlein']
```

This query outputs each review of each book where a review comment contains the word 'good':

```
select * from MediaOrder[books.book][review where comment like 'good']
```

5.20.3. Contained-Event Selection and Joins

This section discusses contained-event selection in joins.

When joining within the same event it is not required that views are specified. Recall, in a join or outer join there must be views specified that hold the data to be joined. For self-joins, no views are required and the join executes against the data returned by the same event.

This query inner-joins items to books where book id matches the product id:

```
select book.bookId, item.itemId
from MediaOrder[books.book] as book,
      MediaOrder[items.item] as item
where productId = bookId
```

Query results for the above query when sending the media order event as shown earlier are:

book.bookId	item.itemId
B001	100001

The next example query is a left outer join. It returns all books and their items, and for books without item it returns the book and a `null` value:

```
select book.bookId, item.itemId
from MediaOrder[books.book] as book
  left outer join
    MediaOrder[items.item] as item
  on productId = bookId
```

Query results for the above query when sending the media order event as shown earlier are:

book.bookId	item.itemId
B001	100001
B002	null

A full outer join combines the results of both left and right outer joins. The joined table will contain all records from both tables, and fill in `null` values for missing matches on either side.

This example query is a full outer join, returning all books as well as all items, and filling in `null` values for book id or item id if no match is found:

```
select orderId, book.bookId,item.itemId
from MediaOrder[books.book] as book
  full outer join
     MediaOrder[select orderId, * from items.item] as item
  on productId = bookId
order by bookId, item.itemId asc
```

As in all other continuous queries, aggregation results are cumulative from the time the statement was created.

The following query counts the cumulative number of items in which the product id matches a book id:

```
select count(*)
from MediaOrder[books.book] as book,
      MediaOrder[items.item] as item
where productId = bookId
```

The `unidirectional` keyword in a join indicates to the query engine that aggregation state is not cumulative. The next query counts the number of items in which the product id matches a book id for each event:

```
select count(*)
from MediaOrder[books.book] as book unidirectional,
      MediaOrder[items.item] as item
where productId = bookId
```

5.20.4. Sentence and Word Example

The next example splits an event representing a sentence into multiple events in which each event represents a word. It represents all events and the logic to split events into contained events as Java code. The next chapter has additional examples that use Map-type events and put contained-event logic into a separate expression or script.

The sentence event in this example is represented by a class declared as follows:

```
public class SentenceEvent {
  private final String sentence;

  public SentenceEvent(String sentence) {
    this.sentence = sentence;
  }

  public WordEvent[] getWords() {
    String[] split = sentence.split(" ");
    WordEvent[] words = new WordEvent[split.length];
    for (int i = 0; i < split.length; i++) {
      words[i] = new WordEvent(split[i]);
    }
    return words;
  }
}
```

The sentence event as above provides an event property `words` that returns each word event.

The declaration of word event is also a class:

```
public class WordEvent {
  private final String word;

  public WordEvent(String word) {
    this.word = word;
  }
  public String getWord() {
    return word;
  }
}
```

The EPL statement to populate a stream of words from a sentence event is:

```
insert into WordStream select * from SentenceEvent[words]
```

Finally, the API call to send a sentence event to the engine is shown here:

```
epService.getEPRuntime().sendEvent(new  SentenceEvent("Hello  Word  Contained
 Events"));
```

5.20.5. More Examples

The examples herein are not based on the POJO events of the prior example. They are meant to demonstrate different types of contained-event expressions and the use of `@type(`*type_name*`)` to identify the event type of the return values of the contained-event expression.

The example first defines a few sample event types:

```
create schema SentenceEvent(sentence String)
```

```
create schema WordEvent(word String)
```

```
create schema CharacterEvent(char String)
```

The following EPL assumes that your application defined a plug-in single-row function by name `splitSentence` that returns an array of Map, producting output events that are `WordEvent` events:

```
insert         into         WordStream         select         *         from
 SentenceEvent[splitSentence(sentence)@type(WordEvent)]
```

The example EPL shown next invokes a JavaScript function which returns some events of type `WordEvent`:

```
expression Collection js:splitSentenceJS(sentence) [
importPackage(java.util);
var words = new ArrayList();
words.add(Collections.singletonMap('word', 'wordOne'));
words.add(Collections.singletonMap('word', 'wordTwo'));
words;
]
select * from SentenceEvent[splitSentenceJS(sentence)@type(WordEvent)]
```

In the next example the sentence event first gets split into words and then each word event gets split into character events via an additional `splitWord` single-row function, producing events of type `CharacterEvent`:

```
select * from SentenceEvent
  [splitSentence(sentence)@type(WordEvent)]
  [splitWord(word)@type(CharacterEvent)]
```

5.20.6. Contained-Event Limitations

The following restrictions apply to contained-event selection:

- When selecting contained events from a named window in a join, the stream must be marked as `unidirectional`.
- Selecting contained events from a named window in a correlated subquery is not allowed.

5.21. Updating an Insert Stream: the Update IStream Clause

The `update istream` statement allows declarative modification of event properties of events entering a stream. Update is a pre-processing step to each new event, modifying an event before the event applies to any statements.

The synopsis of `update istream` is as follows:

```
update istream event_type [as stream_name]
  set property_name = set_expression [, property_name = set_expression]
 [,...]
  [where where_expression]
```

The *event_type* is the name of the type of events that the `update` applies to. The optional `as` keyword can be used to assign a name to the event type for use with subqueries, for example. Following the `set` keyword is a comma-separated list of property names and expressions that provide the event properties to change and values to set.

The optional `where` clause and expression can be used to filter out events to which to apply updates.

Listeners to an `update` statement receive the updated event in the insert stream (new data) and the event prior to the update in the remove stream (old data). Note that if there are multiple update statements that all apply to the same event then the engine will ensure that the output events delivered to listeners or subscribers are consistent with the then-current updated properties of the event (if necessary making event copies, as described below, in the case that listeners are attached to update statements). Iterating over an update statement returns no events.

As an example, the below statement assumes an `AlertEvent` event type that has properties named `severity` and `reason`:

```
update istream AlertEvent
  set severity = 'High'
  where severity = 'Medium' and reason like '%withdrawal limit%'
```

The statement above changes the value of the `severity` property to "High" for `AlertEvent` events that have a medium severity and contain a specific reason text.

Update statements apply the changes to event properties before other statements receive the event(s) for processing, e.g. "`select * from AlertEvent`" receives the updated `AlertEvent`. This is true regardless of the order in which your application creates statements.

When multiple update statements apply to the same event, the engine executes updates in the order in which update statements are created. We recommend the `@Priority` EPL annotation to define a deterministic order of processing updates, especially in the case where update statements get created and destroyed dynamically or multiple update statements update the same fields. The update statement with the highest `@Priority` value applies last.

The `update` clause can be used on streams populated via `insert into`, as this example utilizing a pattern demonstrates:

```
insert into DoubleWithdrawalStream
select a.id, b.id, a.account as account, 0 as minimum
from pattern [a=Withdrawal -> b=Withdrawal(id = a.id)]
```

```
update istream DoubleWithdrawalStream set minimum = 1000 where account in (10002,
 10003)
```

When using `update` with named windows, any changes to event properties apply before an event enters the named window.

Consider the next example (shown here with statement names in @Name EPL annotation, multiple EPL statements):

```
@Name("CreateWindow") create window MyWindow.win:time(30 sec) as AlertEvent

@Name("UpdateStream") update istream MyWindow set severity = 'Low' where reason
 = '%out of paper%'

@Name("InsertWindow") insert into MyWindow select * from AlertEvent

@Name("SelectWindow") select * from MyWindow
```

The `UpdateStream` statement specifies an `update` clause that applies to all events entering the named window. Note that `update` does not apply to events already in the named window at the time an application creates the `UpdateStream` statement, it only applies to new events entering the named window (after an application created the `update` statement).

Therefore, in the above example listeners to the `SelectWindow` statement as well as the `CreateWindow` statement receive the updated event, while listeners to the `InsertWindow` statement receive the original `AlertEvent` event (and not the updated event).

Subqueries can also be used in all expressions including the optional `where` clause.

This example demonstrates a correlated subquery in an assignment expression and also demonstrates the optional `as` keyword. It assigns the `phone` property of an `AlertEvent` event a new value based on the lookup within all unique `PhoneEvent` events (according to an `empid` property) correlating the `AlertEvent` property `reporter` with the `empid` property of `PhoneEvent`:

```
update istream AlertEvent as ae
  set phone =
    (select phone from PhoneEvent.std:unique(empid) where empid = ae.reporter)
```

When updating indexed properties use the syntax *propertyName*`[`*index*`] =` *value* with the index value being an integer number. When updating mapped properties use the syntax *propertyName*`(`*key*`) =` *value* with the key being a string value.

When using `update`, please note these limitations:

1. Expressions may not use aggregation functions.
2. The `prev` and `prior` functions may not be used.
3. For underlying event representations that are Java objects, a event object class must implement the `java.io.Serializable` interface as discussed below.
4. When using an XML underlying event type, event properties in the XML document representation are not available for update.

5. Nested properties are not supported for update. Revision event types and variant streams may also not be updated.

5.21.1. Immutability and Updates

When updating event objects the engine maintains consistency across statements. The engine ensures that an update to an event does not impact the results of statements that look for or retain the original un-updated event. As a result the engine may need to copy an event object to maintain consistency.

In the case your application utilizes Java objects as the underlying event representation and an `update` statement updates properties on an object, then in order to maintain consistency across statements it is necessary for the engine to copy the object before changing properties (and thus not change the original object).

For Java application objects, the copy operation is implemented by serialization. Your event object must therefore implement the `java.io.Serializable` interface to become eligible for update. As an alternative to serialization, you may instead configure a copy method as part of the event type configuration via `ConfigurationEventTypeLegacy`.

5.22. Controlling Event Delivery : The `For` Clause

The engine delivers all result events of a given statement to the statement's listeners and subscriber (if any) in a single invocation of each listener and subscriber's `update` method passing an array of result events. For example, a statement using a time-batch view may provide many result events after a time period passes, a pattern may provide multiple matching events or in a join the join cardinality could be multiple rows.

For statements that typically post multiple result events to listeners the `for` keyword controls the number of invocations of the engine to listeners and subscribers and the subset of all result events delivered by each invocation. This can be useful when your application listener or subscriber code expects multiple invocations or expects that invocations only receive events that belong together by some additional criteria.

The `for` keyword is a reserved keyword. It is followed by either the `grouped_delivery` keyword for grouped delivery or the `discrete_delivery` keyword for discrete delivery. The `for` clause is valid after any EPL select statement.

The synopsis for grouped delivery is as follows:

```
... for grouped_delivery (group_expression [, group_expression] [,...])
```

The *group_expression* expression list provides one or more expressions to apply to result events. The engine invokes listeners and subscribers once for each distinct set of values returned by *group_expression* expressions passing only the events for that group.

The synopsis for discrete delivery is as follows:

```
... for discrete_delivery
```

With discrete delivery the engine invokes listeners and subscribers once for each result event passing a single result event in each invocation.

Consider the following example without `for`-clause. The time batch data view collects RFIDEvent events for 10 seconds and posts an array of result events:

```
select * from RFIDEvent.win:time_batch(10 sec)
```

Let's consider an example event sequence as follows:

Table 5.7. Sample Sequence of Events for `For` Keyword

Event
RFIDEvent(id:1, zone:'A')
RFIDEvent(id:2, zone:'B')
RFIDEvent(id:3, zone:'A')

Without `for`-clause and after the 10-second time period passes, the engine delivers an array of 3 events in a single invocation to listeners and the subscriber.

The next example specifies the `for`-clause and grouped delivery by zone:

```
select * from RFIDEvent.win:time_batch(10 sec) for grouped_delivery (zone)
```

With grouped delivery and after the 10-second time period passes, the above statement delivers result events in two invocations to listeners and the subscriber: The first invocation delivers an array of two events that contains zone A events with id 1 and 3. The second invocation delivers an array of 1 event that contains a zone B event with id 2.

The next example specifies the `for`-clause and discrete delivery:

```
select * from RFIDEvent.win:time_batch(10 sec) for discrete_delivery
```

With discrete delivery and after the 10-second time period passes, the above statement delivers result events in three invocations to listeners and the subscriber: The first invocation delivers an array of 1 event that contains the event with id 1, the second invocation delivers an array of 1 event that contains the event with id 2 and the third invocation delivers an array of 1 event that contains the event with id 3.

Remove stream events are also delivered in multiple invocations, one for each group, if your statement selects remove stream events explicitly via `irstream` or `rstream` keywords.

The `insert into` for inserting events into a stream is not affected by the `for`-clause.

The delivery order respects the natural sort order or the explicit sort order as provided by the `order by` clause, if present.

The following are known limitations:

1. The engine validates *group_expression* expressions against the output event type, therefore all properties specified in *group_expression* expressions must occur in the `select` clause.

Chapter 6. EPL Reference: Patterns

6.1. 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 consist of pattern atoms and pattern operators:

1. Pattern *atoms* are the basic building blocks of patterns. Atoms are filter expressions, observers for time-based events and plug-in custom observers that observe external events not under the control of the engine.

2. Pattern *operators* control expression lifecycle and combine atoms logically or temporally.

The below table outlines the different pattern atoms available:

Table 6.1. Pattern Atoms

Pattern Atom	Example
Filter expressions specify an event t look for.	`StockTick(symbol='ABC', price > 100)`

Time-based event observers specify tim intervals or time schedules.	`timer:interval(10 seconds)`
	 `timer:at(*, 16, *, *, *)`

Custom plug-in observers can ad pattern language syntax for observin application-specific events.	`myapplication:myobserver("http:// someResource")`

There are 4 types of pattern operators:

1. Operators that control pattern sub-expression repetition: `every`, `every-distinct`, `[num]` and `until`
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`, `timer:withinmax` and `while`-expression. Custom plug-in guards may also be used.

Pattern expressions can be nested arbitrarily deep by including the nested expression(s) in `()` round parenthesis.

Underlying the pattern matching is a state machine that transitions between states based on arriving events and based on time advancing. A single event or advancing time may cause a reaction in multiple parts of your active pattern state.

6.2. How to use Patterns

6.2.1. 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 6.4, “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 5.4.2, “Pattern-based Event Streams”*.

Patterns can also contain comments within the pattern as outlined in *Section 5.2.2, “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 5.4.2, “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))");
```

6.2.2. Patterns in EPL

A pattern may appear anywhere in the `from` clause of an EPL statement including joins and subqueries. Patterns may therefore be used in combination with the `where` clause, `group by` clause, `having` clause as well as output rate limiting and `insert into`.

In addition, a data window view can be declared onto a pattern. A data window declared onto a pattern only serves to retain pattern matches. A data window declared onto a pattern does not limit, cancel, remove or delete intermediate pattern matches of the pattern when pattern matches leave the data window.

This example statement demonstrates the idea by selecting a total price per customer over pairs of events (ServiceOrder followed by a ProductOrder event for the same customer id within 1 minute), occuring in the last 2 hours, in which the sum of price is greater than 100, and using a `where` clause to filter on name:

```
select a.custId, sum(a.price + b.price)
from pattern [every a=ServiceOrder ->
   b=ProductOrder(custId = a.custId) where timer:within(1 min)].win:time(2 hour)
where a.name in ('Repair', b.name)
group by a.custId
having sum(a.price + b.price) > 100
```

6.2.3. 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`.

6.2.4. 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
}
```

Further, if a data window is defined onto a pattern, the iterator returns the pattern matches according to the data window expiry policy.

This pattern specifies a length window of 10 elements that retains the last 10 matches of A and B events, for use via iterator or for use in a join or subquery:

```
select * from pattern [every (A or B).win:length(10)
```

6.2.5. Pattern Error Reporting

While the pattern compiler analyzes your pattern and verifies its integrity, it may not detect certain pattern errors that may occur at runtime. Sections of this pattern documentation point out common cases where the pattern engine will log a runtime error. We recommend turning on the log warning level at project development time to inspect and report on warnings logged. If a statement name is assigned to a statement then the statement name is logged as well.

6.3. Operator Precedence

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

Table 6.2. Pattern Operator Precedence

Precedenc	Operator	Description	Example

1	guard postfix	`where timer:withi` and `whil` `(expression)` (inc withinmax and plug-i pattern guard)	`MyEvent where timer:within(1 sec)`
			
			`a=MyEvent while (a.price between 1 and 10)`

2	unary	`every, not, every distinct`	`every MyEvent`
			`timer:interval(5 min) and not MyEvent`

3	repeat	`[num]`, `until`	`[5] MyEvent`
			
			`[1..3] MyEvent until MyOtherEvent`

4	and	`and`	`every (MyEvent and MyOtherEvent)`
5	or	`or`	`every (MyEvent or MyOtherEvent)`
6	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.

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

Table 6.3. 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 -> B or B -> A	A -> (B or B) -> A	The `or` operator has higher precedence then the `followed-by` operator, specify as (A -> B) or (B -> A) instead.
A and B or C	(A and B) or C	The `and` operator has higher precedence then the `or` operator.
A -> B until C -> D	A -> (B until C) -> D	The `until` operator has higher precedence then the `followed-by` operator.
[5] A or B	([5] A) or B	The `[num]` repeat operator has higher precedence then the `or` operator.
every A where timer:within(10)	every (A where timer:within(10))	The `where` postfix has higher precedence then the `every` operator.

6.4. 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 name 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 5.4.1, "Filter-based Event Streams"*. Contained-event selection on filters in patterns is further described in *Section 5.20, "Contained-Event Selection"*.

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 e1=RfidEvent -> e2=RfidEvent(assetId=e1.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. The tag names in the above example were `e1` and `e2`. This syntax can be used in all filter operators or expressions including ranges and the `in` set-of-values check:

```
every e1=RfidEvent ->
  e2=RfidEvent(MyLib.isInRadius(e1.x, e1.y, x, y) and zone in (1, e1.zone))
```

An arriving event changes the truth value of all expressions that look for the event. Consider the pattern as follows:

```
every (RfidEvent(zone > 1) and RfidEvent(zone < 10))
```

The pattern above is satisfied as soon as only one event with zone in the interval [2, 9] is received.

6.4.1. Controlling Event Consumption

An arriving event applies to all filter expressions for which the event matches. In other words, an arriving event is not consumed by any specify filter expression(s) but applies to all active filter expressions of all pattern sub-expressions.

You may provide the `@consume` annotation as part of a filter expression to control consumption of an arriving event. If an arriving event matches the filter expression marked with `@consume` it is no longer available to other filter expressions of the same pattern that also match the arriving event.

The `@consume` can include a level number in parenthesis. A higher level number consumes the event first. The default level number is 1. Multiple filter expressions with the same level number for `@consume` all match the event.

Consider the next sample pattern:

```
a=RfidEvent(zone='Z1') and b=RfidEvent(assetId='0001')
```

This pattern fires when a single RfidEvent event arrives that has zone 'Z1' and assetId '0001'. The pattern also matches when two RfidEvent events arrive, in any order, wherein one has zone 'Z1' and the other has assetId '0001'.

Mark a filter expression with `@consume` to indicate that if an arriving event matches multiple filter expressions that the engine prefers the marked filter expression and does not match any other filter expression.

This updated pattern statement uses `@consume` to indicate that a match against zone is preferred:

```
a=RfidEvent(zone='Z1')@consume and b=RfidEvent(assetId='0001')
```

This pattern no longer fires when a single RfidEvent arrives that has zone 'Z1' and assetId '0001', because when the first filter expression matches the pattern engine consumes the event. The pattern only matches when two RfidEvent events arrive in any order. One event must have zone 'Z1' and the other event must have a zone other then 'Z1' and an assetId '0001'.

The next sample pattern provides a level number for each `@consume`:

```
a=RfidEvent(zone='Z1')@consume(2)
  or b=RfidEvent(assetId='0001')@consume(1)
  or c=RfidEvent(category='perishable'))
```

The pattern fires when an RfidEvent arrives with zone 'Z1'. In this case the output event populates property 'a' but not properties 'b' and 'c'. The pattern also fires when an RfidEvent arrives with a zone other then 'Z1' and an asset id of '0001'. In this case the output event populates property 'b' but not properties 'a' and 'c'. The pattern also fires when an RfidEvent arrives with a zone other then 'Z1' and an asset id other then '0001' and a category of 'perishable'. In this case the output event populates property 'c' but not properties 'a' and 'b'.

6.5. Pattern Operators

6.5.1. Every

The `every` operator indicates that the pattern sub-expression should restart when the subexpression qualified by the `every` keyword evaluates to true or false. Without the `every` operator the pattern sub-expression stops when the pattern sub-expression evaluates to true or false.

As a side note, please be aware that a single invocation to the `UpdateListener` interface may deliver multiple events in one invocation, since the interface accepts an array of values.

Thus the `every` operator works like a factory for the pattern sub-expression contained within. When the pattern sub-expression within it fires and thus quits checking for events, the `every` causes the start of a new pattern sub-expression listening for more occurrences of the same event or set of events.

Every time a pattern sub-expression 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 sub-expression. If the `every` operator is not specified for a subexpression, the subexpression stops after the first match was found.

This pattern fires when encountering an A event and then stops looking.

```
A
```

This pattern keeps firing when encountering A events, and doesn't stop looking.

```
every A
```

When using `every` operator with the `->` followed-by operator, each time the `every` operator restarts it also starts a new subexpression instance looking for events in the followed-by subexpression.

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 6.4. 'Every' operator examples

Example	Description
`every ( A -> B )`	Detect an A event followed by a B event. At the time when B occurs the pattern matches, then the pattern matcher restarts and looks for the next A event. 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 A event followed by a B event. 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 A event followed by every B event. 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 A event followed by every B event. 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 A events. When an A event arrives, it looks for a B event. After the B event arrives, the pattern looks for a C event. Finally, when the C event arrives the pattern fires. The engine then starts looking for an A event again.

Assume that between the B event and the C event a second A_2 event arrives. The pattern would ignore the A_2 event entirely since it's then looking for a C event. 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 each A event that is followed by a B event and then a C event, regardless of when the A event arrives. Note that for each A event that arrives the pattern engine starts a new subexpression looking for a B event and then a C event, outputting each combination of matching events.

6.5.1.1. `Every` Operator Equivalence

A pattern that only has the `every` operator and a single filter expression is equivalent to selecting the same filter in the `from` clause:

```
select * from StockTickEvent(symbol='GE')      // Prefer this
// ... equivalent to ...
select * from pattern[every StockTickEvent(symbol='GE')]
```

6.5.1.2. Limiting Subexpression Lifetime

As the introduction of the `every` operator states, the operator starts new subexpression instances and can cause multiple matches to occur for a single arriving event.

New subexpressions also take a very small amount of system resources and thereby your application should carefully consider when subexpressions must end when designing patterns. Use the `timer:within` construct and the `and not` constructs to end active subexpressions. The data window onto a pattern stream does not serve to limit pattern sub-expression lifetime.

Lets look at a concrete example. Consider the following sequence of events arriving:

A_1 A_2 B_1

This pattern matches on arrival of B_1 and outputs two events (an array of length 2 if using a listener). The two events are the combinations {A_1, B_1} and {A_2, B_1}:

```
every a=A -> b=B
```

The `and not` operators are used to end an active subexpression.

The next pattern matches on arrival of B_1 and outputs only the last A event which is the combination {A_2, B_1}:

```
every a=A -> (b=B and not A)
```

The `and not` operators cause the subexpression looking for {A_1, B?} to end when A_2 arrives.

Similarly, in the pattern below the engine starts a new subexpression looking for a B event every 1 second. After 5 seconds there are 5 subexpressions active looking for a B event and 5 matches occur at once if a B event arrives after 5 seconds.

```
every timer:interval(1 sec) -> b=B
```

Again the `and not` operators can end subexpressions that are not intended to match any longer:

```
every timer:interval(1 sec) -> (b=B and not timer:interval(1 sec))
// equivalent to
every timer:interval(1 sec) -> (b=B where timer:within(1 sec))
```

6.5.1.3. `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 event and followed by a C event, in which both the B event and the C event 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}.

6.5.1.4. 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.

6.5.2. Every-Distinct

Similar to the `every` operator in most aspects, the `every-distinct` operator indicates that the pattern sub-expression should restart when the subexpression qualified by the `every-distinct` keyword evaluates to true or false. In addition, the `every-distinct` eliminates duplicate results received from an active subexpression according to its distinct-value expressions.

The synopsis for the `every-distinct` pattern operator is:

```
every-distinct(distinct_value_expr [, distinct_value_exp[...]
[, expiry_time_period])
```

Within parenthesis are one or more *distinct_value_expr* expressions that return the values by which to remove duplicates.

You may optionally specify an *expiry_time_period* time period. If present, the pattern engine expires and removes distinct key values that are older then the time period, removing their associated memory and allowing such distinct values to match again. When your distinct value expressions return an unlimited number of values, for example when your distinct value is a timestamp or auto-increment column, you should always specify an expiry time period.

When specifying properties in the distinct-value expression list, you must ensure that the event types providing properties are tagged. Only properties of event types within filter expressions that are sub-expressions to the `every-distinct` may be specified.

For example, this pattern keeps firing for every A event with a distinct value for its `aprop` property:

```
every-distinct(a.aprop) a=A
```

Note that the pattern above assigns the `a` tag to the A event and uses `a.prop` to identify the `prop` property as a value of the `a` event A.

A pattern that returns the first Sample event for each sensor, assuming sensor is a field that returns a unique id identifying the sensor that originated the Sample event, is:

```
every-distinct(s.sensor) s=Sample
```

The next pattern looks for pairs of A and B events and returns only the first pair for each combination of `aprop` of an A event and `bprop` of a B event:

```
every-distinct(a.aprop, b.bprop) (a=A and b=B)
```

The following pattern looks for A events followed by B events for which the value of the `aprop` of an A event is the same value of the `bprop` of a B event but only for each distinct value of `aprop` of an A event:

```
every-distinct(a.aprop) a=A -> b=B(bprop = a.aprop)
```

When specifying properties as part of distinct-value expressions, properties must be available from tagged event types in sub-expressions to the `every-distinct`.

The following patterns are not valid:

```
// Invalid: event type in filter not tagged
every-distinct(aprop) A

// Invalid: property not from a sub-expression of every-distinct
a=A -> every-distinct(a.aprop) b=B
```

When an active subexpression to `every-distinct` becomes permanently false, the distinct-values seen from the active subexpression are removed and the sub-expression within is restarted.

For example, the below pattern detects each A event distinct by the value of `aprop`.

```
every-distinct(a.aprop) (a=A and not B)
```

In the pattern above, when a B event arrives, the subexpression becomes permanently false and is restarted anew, detecting each A event distinct by the value of `aprop` without considering prior values.

When your distinct key is a timestamp or other non-unique property, specify an expiry time period.

The following example returns every distinct A event according to the timestamp property on the A event, retaining each timestamp value for 10 seconds:

```
every-distinct(a.timestamp, 10 seconds) a=A
```

In the example above, if for a given A event and its timestamp value the same timestamp value occurs again for another A event before 10 seconds passed, the A event is not a match. If 10 seconds passed the pattern indicates a second match.

You may not use every-distinct with a timer-within guard to expire keys: The expiry time notation as above is the recommended means to expire keys.

```
// This is not the same as above; It does not expire transaction ids and is
 not recommended
every-distinct(a.timestamp) a=A where timer:within(10 sec)
```

6.5.3. Repeat

The repeat operator fires when a pattern sub-expression evaluates to true a given number of times. The synopsis is as follows:

```
[match_count] repeating_subexpr
```

The repeat operator is very similar to the `every` operator in that it restarts the *repeating_subexpr* pattern sub-expression up to a given number of times.

match_count is a positive number that specifies how often the *repeating_subexpr* pattern sub-expression must evaluate to true before the repeat expression itself evaluates to true, after which the engine may indicate a match.

For example, this pattern fires when the last of five A events arrives:

```
[5] A
```

Parenthesis must be used for nested pattern sub-expressions. This pattern fires when the last of a total of any five A or B events arrives:

```
[5] (A or B)
```

Without parenthesis the pattern semantics change, according to the operator precedence described earlier. This pattern fires when the last of a total of five A events arrives or a single B event arrives, whichever happens first:

```
[5] A or B
```

Tags can be used to name events in filter expression of pattern sub-expressions. The next pattern looks for an A event followed by a B event, and a second A event followed by a second B event. The output event provides indexed and array properties of the same name:

```
[2] (a=A -> b=B)
```

Using tags with repeat is further described in *Section 6.5.4.6, “Tags and the Repeat Operator”*.

Consider the following pattern that demonstrates the behavior when a pattern sub-expression becomes permanently false:

```
[2] (a=A and not C)
```

In the case where a C event arrives before 2 A events arrive, the pattern above becomes permanently false.

Lets add an `every` operator to restart the pattern and thus keep matching for all pairs of A events that arrive without a C event in between each pair:

```
every [2] (a=A and not C)
```

Since pattern matches return multiple A events, your select clause should use tag `a` as an array, for example:

```
select a[0].id, a[1].id from pattern [every [2] (a=A and not C)]
```

6.5.4. Repeat-Until

The repeat `until` operator provides additional control over repeated matching.

The repeat until operator takes an optional range, a pattern sub-expression to repeat, the `until` keyword and a second pattern sub-expression that ends the repetition. The synopsis is as follows:

```
[range] repeated_pattern_expr until end_pattern_expr
```

Without a *range*, the engine matches the *repeated_pattern_expr* pattern sub-expression until the *end_pattern_expr* evaluates to true, at which time the expression turns true.

An optional *range* can be used to indicate the minimum number of times that the *repeated_pattern_expr* pattern sub-expression must become true.

The optional *range* can also specify a maximum number of times that *repeated_pattern_expr* pattern sub-expression evaluates to true and retains tagged events. When this number is reached, the engine stops the *repeated_pattern_expr* pattern sub-expression.

The `until` keyword is always required when specifying a range and is not required if specifying a fixed number of repeat as discussed in the section before.

6.5.4.1. Unbound Repeat

In the unbound repeat, without a *range*, the engine matches the *repeated_pattern_expr* pattern sub-expression until the *end_pattern_expr* evaluates to true, at which time the expression turns true. The synopsis is:

```
repeated_pattern_expr until end_pattern_expr
```

This is a pattern that keeps looking for A events until a B event arrives:

```
A until B
```

Nested pattern sub-expressions must be placed in parenthesis since the `until` operator has precedence over most operators. This example collects all A or B events for 10 seconds and places events received in indexed properties 'a' and 'b':

```
(a=A or b=B) until timer:interval(10 sec)
```

6.5.4.2. Bound Repeat Overview

The synopsis for the optional *range* qualifier is:

```
[ [low_endpoint] : [high_endpoint] ]
```

low_endpoint is an optional number that appears on the left of a colon (:), after which follows an optional *high_endpoint* number.

A range thus consists of a *low_endpoint* and a *high_endpoint* in square brackets and separated by a colon (:) characters. Both endpoint values are optional but either one or both must be supplied. The *low_endpoint* can be omitted to denote a range that starts at zero. The *high_endpoint* can be omitted to denote an open-ended range.

Some examples for valid ranges might be:

```
[3 : 10]
[:3]    // range starts at zero
[2:]    // open-ended range
```

The *low_endpoint*, if specified, defines the minimum number of times that the *repeated_pattern_expr* pattern sub-expression must become true in order for the expression to become true.

The *high_endpoint*, if specified, is the maximum number of times that the *repeated_pattern_expr* pattern sub-expression becomes true. If the number is reached, the engine stops the *repeated_pattern_expr* pattern sub-expression.

In all cases, only at the time that the *end_pattern_expr* pattern sub-expression evaluates to true does the expression become true. If *end_pattern_expr* pattern sub-expression evaluates to false, then the expression evaluates to false.

6.5.4.3. Bound Repeat - Open Ended Range

An open-ended range specifies only a low endpoint and not a high endpoint.

Consider the following pattern which requires at least three A events to match:

```
[3:] A until B
```

In the pattern above, if a B event arrives before 3 A events occurred, the expression ends and evaluates to false.

6.5.4.4. Bound Repeat - High Endpoint Range

A high-endpoint range specifies only a high endpoint and not a low endpoint.

In this sample pattern the engine will be looking for a maximum of 3 A events. The expression turns true as soon as a single B event arrives regardless of the number of A events received:

```
[:3] A until B
```

The next pattern matches when a C or D event arrives, regardless of the number of A or B events that occurred:

```
[:3] (a=A or b=B) until (c=C or d=D)
```

In the pattern above, if more then 3 A or B events arrive, the pattern stops looking for additional A or B events. The 'a' and 'b' tags retain only the first 3 (combined) matches among A and B events. The output event contains these tagged events as indexed properties.

6.5.4.5. Bound Repeat - Bounded Range

A bounded range specifies a low endpoint and a high endpoint.

The next pattern matches after at least one A event arrives upon the arrival of a single B event:

```
[1:3] a=A until B
```

If a B event arrives before the first A event, then the pattern does not match. Only the first 3 A events are returned by the pattern.

6.5.4.6. Tags and the Repeat Operator

The tags assigned to events in filter subexpressions within a repeat operator are available for use in filter expressions and also in any EPL clause.

This sample pattern matches 2 A events followed by a B event. Note the filter on B events: only a B event that has a value for the "beta" property that equals any of the "id" property values of the two A events is considered:

```
[2] A -> B(beta in (a[0].id, a[1].id))
```

The next EPL statement returns pairs of A events:

```
select a, a[0], a[0].id, a[1], a[1].id
from pattern [ every [2] a=A ]
```

The `select` clause of the statement above showcases different ways of accessing tagged events:

- The tag itself can be used to select an array of underlying events. For example, the 'a' expression above returns an array of underlying events of event type A.
- The tag as an indexed property returns the underlying event at that index. For instance, the 'a[0]' expression returns the first underlying A event, or null if no such A event was matched by the repeat operator.
- The tag as a nested, indexed property returns a property of the underlying event at that index. For example, the 'a[1].id' expression returns the 'id' property value of the second A event, or null if no such second A event was matched by the repeat operator.

6.5.4.7. Note on Indexed Tags

You may not use indexed tags defined in the sub-expression to the repeat operator in the same subexpression. For example, in the following pattern the subexpression to the repeat operator is `(a=A() -> b=B(id=a[0].id))` and the tag `a` cannot be used in its indexed form in the filter for event B:

```
// invalid
every [2] (a=A() -> b=B(id=a[0].id))
```

You can use tags without an index:

```
// valid
every [2] (a=A() -> b=B(id=a.id))
```

6.5.5. 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).

This pattern matches when both an A event and a B event arrive, at the time the last of the two events arrive:

```
A and B
```

This pattern matches on any sequence of an A event followed by a B event and then a C event followed by a D event, or a C event followed by a D and an A event followed by a B event:

```
(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))
```

Consider a pattern that looks for the same event:

```
A and A
```

The pattern above fires when a single A event arrives. The first arriving A event triggers a state transition in both the left and the right hand side expression.

In order to match after two A events arrive in any order, there are two options to express this pattern. The followed-by operator is one option and the repeat operator is the second option, as the next two patterns show:

```
A -> A
// ... or ...
[2] A
```

6.5.6. 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 an A event or a B event. 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)
```

6.5.7. 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 conjunction with the `and` operator or subexpressions as the below examples show.

This pattern matches only when an A event is encountered followed by a B event but only if no C event was encountered before either an A event and a B event, counting from the time the pattern is started:

```
(A -> B) and not C
```

Assume we'd like to detect when an A event is followed by a D event, 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).

To test for absence of an event, use `timer:interval` together with `and not` operators. The sample statement reports each 10-second interval during which no A event occurred:

```
every (timer:interval(10 sec) and not A)
```

In many cases the `not` operator, when used alone, does not make sense. The following example is invalid and will log a warning when the engine is started:

```
// not a sensible pattern
(not a=A) -> B(id=a.id)
```

6.5.8. 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 an A event and if encountered, look for a B event. 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')
```

A pattern that takes all A events that are not followed by a B event within 5 minutes:

```
every A -> (timer:interval(5 min) and not B)
```

A pattern that takes all A events that are not preceded by B within 5 minutes:

```
every (timer:interval(5 min) and not B -> A)
```

6.5.8.1. Limiting Sub-Expression Count

The followed-by `->` operator can optionally be provided with an expression that limits the number of sub-expression instances of the right-hand side pattern sub-expression.

The synopsis for the followed-by operator with limiting expression is:

```
lhs_expression -[limit_expression]> rhs_expression
```

Each time the *lhs_expression* pattern sub-expression turns true the pattern engine starts a new *rhs_expression* pattern sub-expression. The *limit_expression* returns an integer value that defines a maximum number of pattern sub-expression instances that can simultaneously be present for the same *rhs_expression*.

When the limit is reached the pattern engine issues a `com.espertech.esper.client.hook.ConditionPatternSubexpressionMax` notification object to any condition handlers registered with the engine as described in *Section 14.12, “Condition Handling”* and does not start a new pattern sub-expression instance for the right-hand side pattern sub-expression.

For example, consider the following pattern which returns for every A event the first B event that matches the `id` field value of the A event:

```
every a=A -> b=B(id = a.id)
```

In the above pattern, every time an A event arrives (lhs) the pattern engine starts a new pattern sub-expression (rhs) consisting of a filter for the first B event that has the same value for the `id` field as the A event.

In some cases your application may want to limit the number of right-hand side sub-expressions because of memory concerns or to reduce output. You may add a limit expression returning an integer value as part of the operator.

This example employs the followed-by operator with a limit expression to indicate that maximally 2 filters for B events (the right-hand side pattern sub-expression) may be active at the same time:

```
every a=A -[2]> b=B(id = a.id)
```

Note that the limit expression in the example above is not a limit per value of `id` field, but a limit counting all right-hand side pattern sub-expression instances that are managed by that followed-by sub-expression instance.

If your followed-by operator lists multiple sub-expressions with limits, each limit applies to the immediate right-hand side. For example, the pattern below limits the number of filters for B events to 2 and the number of filters for C events to 3:

```
every a=A -[2]> b=B(id = a.id) -[3]> c=C(id = a.id)
```

6.5.8.2. Limiting Engine-wide Sub-Expression Count

Esper allows setting a maximum number of pattern sub-expressions in the configuration, applicable to all followed-by operators of all statements.

If your application has patterns in multiple EPL statements and all such patterns should count towards a total number of pattern sub-expression counts, you may consider setting a maximum number of pattern sub-expression instances, engine-wide, via the configuration described in *Section 15.4.15.1, “Followed-By Operator Maximum Subexpression Count”*.

When the limit is reached the pattern engine issues a notification object to any condition handlers registered with the engine as described in *Section 14.12, “Condition Handling”*. Depending on your configuration the engine can prevent the start of a new pattern sub-expression instance for the right-hand side pattern sub-expression, until pattern sub-expression instances end or statements are stopped or destroyed.

The notification object issued to condition handlers is an instance of `com.espertech.esper.client.hook.ConditionPatternEngineSubexpressionMax`. The notification object contains information which statement triggered the limit and the pattern counts per statement for all statements.

For information on static and runtime configuration, please consult *Section 15.4.15.1, “Followed-By Operator Maximum Subexpression Count”*. The limit can be changed and disabled or enabled at runtime via the runtime configuration API.

6.5.9. Pattern Guards

Guards are where-conditions that control the lifecycle of subexpressions. Custom guard functions can also be used. The section *Chapter 17, Integration and Extension* outlines guard plug-in development in greater detail.

The pattern guard where-condition has no relationship to the EPL `where` clause that filters sets of events.

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 occurred within 10 seconds after pattern start:

```
every ( (A and B) where timer.within(10 sec) )
```

6.5.9.1. The `timer:within` Pattern Guard

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 synopsis for `timer:within` is as follows:

```
timer:within(time_period_expression)
```

The *time_period_expression* is a time period (see *Section 5.2.1, “Specifying Time Periods”*) or an expression providing a number of seconds as a parameter. The interval expression may contain references to properties of prior events in the same pattern as well as variables and substitution parameters.

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 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)
```

6.5.9.2. The `timer:withinmax` Pattern Guard

The `timer:withinmax` guard is similar to the `timer:within` guard and acts as a stopwatch that additionally has a counter that counts the number of matches. It ends the subexpression when either the stopwatch ends or the match counter maximum value is reached.

The synopsis for `timer:withinmax` is as follows:

```
timer:withinmax(time_period_expression, max_count_expression)
```

The *time_period_expression* is a time period (see *Section 5.2.1, “Specifying Time Periods”*) or an expression providing a number of seconds.

The *max_count_expression* provides the maximum number of matches before the guard ends the subexpression.

Each parameter expression may also contain references to properties of prior events in the same pattern as well as variables and substitution parameters.

This pattern fires for every A event that arrives within 5 seconds after statement creation but only up to the first two A events:

```
(every A) where timer:withinmax (5 seconds, 2)
```

If the result of the *max_count_expression* is 1, the guard ends the subexpression after the first match and indicates the first match.

This pattern fires for the first A event that arrives within 5 seconds after statement creation:

```
(every A) where timer:withinmax (5 seconds, 1)
```

If the result of the *max_count_expression* is zero, the guard ends the subexpression upon the first match and does no indicate any matches.

This example receives every A event followed by every B event (as each B event arrives) until the 5-second subexpression timer ends or X number of B events have arrived (assume X was declared as a variable):

```
every A -> (every B) where timer:withinmax (5 seconds, X)
```

6.5.9.3. The `while` Pattern Guard

The `while` guard is followed by an expression that the engine evaluates for every match reported by the guard pattern sub-expression. When the expression returns false the pattern sub-expression ends.

The synopsis for `while` is as follows:

```
while (guard_expression)
```

The *guard_expression* is any expression that returns a boolean true or false. The expression may contain references to properties of prior events in the same pattern as well as variables and substitution parameters.

Each time the subexpression indicates a match, the engine evaluates *guard_expression* and if true, passes the match and when false, ends the subexpression.

This pattern fires for every A event until an A event arrives that has a value of zero or less for its `size` property (assuming A events have an integer `size` property).

```
(every a=A) while (a.size > 0)
```

Note the parenthesis around the `every` subexpression. They ensure that, following precedence rules, the guard applies to the `every` operator as well.

6.5.9.4. Guard Time Interval Expressions

The `timer:within` and `timer:withinmax` guards may be parameterized by an expression that contains one or more references to properties of prior events in the same pattern.

As a simple example, this pattern matches every A event followed by a B event that arrives within `delta` seconds after the A event:

```
every a=A -> b=B where timer:within (a.delta seconds)
```

Herein A event is assumed to have a `delta` property that provides the number of seconds to wait for B events. Each arriving A event may have a different value for `delta` and the guard is therefore parameterized dynamically based on the prior A event received.

When multiple events accumulate, for example when using the match-until or repeat pattern elements, an index must be provided:

```
[2] a=A -> b=B where timer:within (a[0].delta + a[1].delta)
```

The above pattern matches after 2 A events arrive followed by a B event within a time interval after the A event that is defined by the sum of the `delta` properties of both A events.

6.5.9.5. Combining Guard Expressions

You can combine guard expression by using parenthesis around each subexpression.

The below pattern matches for each A event while A events of size greater then zero arrive and only within the first 20 seconds:

```
((every a=A) while (a.size > 0)) where timer:within(20)
```

6.6. Pattern Atoms

6.6.1. Filter Atoms

Filter atoms have been described in section *Section 6.4, “Filter Expressions In Patterns”*.

6.6.2. Time-based Observer Atoms

Observers observe time-based events for which the thread-of-control originates by the engine timer or external timer event. Custom observers can also be developed that observe timer events or other engine-external application events such as a file-exists check. The section *Chapter 17, Integration and Extension* outlines observer plug-in development in greater detail.

6.6.2.1. 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 5.2.1, “Specifying Time Periods”*) as a parameter, or an expression that returns the number of seconds.

The observer may be parameterized by an expression that contains one or more references to properties of prior events in the same pattern, or may also reference variables, substitution parameters or any other expression returning a numeric value.

After an A event 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 A event that is not followed by a B event within 60 seconds after the A event arrived. The B event must have the same "id" property value as the A event.

```
every a=A -> (timer:interval(60 sec) and not B(id=a.id))
```

Consider the next example, which assumes that the A event has a property `waittime`:

```
every a=A -> (timer:interval(a.waittime + 2) and not B(id=a.id))
```

In the above pattern the logic waits for 2 seconds plus the number of seconds provided by the value of the `waittime` property of the A event.

6.6.2.2. 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 [, time zone]])`.

The value for seconds and time zone 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 ([]).

The `timer:at` observer may also be parameterized by an expression that contains one or more references to properties of prior events in the same pattern, or may also reference variables, substitution parameters or any other expression returning a numeric value. The frequency division operator `*/x` and parameters lists within brackets ([]) are an exception: they may only contain variables, substitution parameters or numeric values.

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 5:45pm (hours 8 to 17 at 0, 15, 30 and 45 minutes past the hour) 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 6.5. Crontab Fields

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 (required if specifying a time zone)	0 - 59	

Field Name	Mandatory?	Allowed Values	Additional Keywords
Time Zone	no	any string (not validated, see TimeZone javadoc)	

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 for year 2007 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).

The time zone is a string-type value that specifies the time zone of the schedule. You must specify a value for seconds when specifying a time zone. Esper relies on the `java.util.TimeZone` to interpret the time zone value. Note that `TimeZone` does not validate time zone strings.

The following `timer:at` pattern matches at 5:00 pm Pacific Standard Time (PST):

```
timer:at (0, 17, *, *, *, *, 'PST')
```

Any expression may occur among the parameters. This example invokes a user-defined function `computeHour` to return an hour:

```
timer:at (0, computeHour(), *, *, *, *)
```

The following restrictions apply to crontab parameters:

- It is not possible to specify both Days Of Month and Days Of Week.

6.6.2.2.1. timer:at and the `every` Operator

When using `timer:at` with the `every` operator the crontab-like timer computes the next time at which the timer should fire based on the specification and the current time. When using `every`, the current time is the time the timer fired or the statement start time if the timer has not fired once.

For example, this pattern fires every 1 minute starting at 1:00pm and ending at 1:59pm, every day:

```
every timer:at(*, 13, *, *, *)
```

Assume the above statement gets started at 1:05pm and 20 seconds. In such case the above pattern fires every 1 minute starting at 1:06pm and ending at 1:59pm for that day and 1:00pm to 1:59pm every following day.

To get the pattern to fire only once at 1pm every day, explicitly specify the minute to start. The pattern below fires every day at 1:00pm:

```
every timer:at(0, 13, *, *, *)
```

By specifying a second resolution the timer can be made to fire every second, for instance:

```
every timer:at(*, *, *, *, *, *)
```

Chapter 7. EPL Reference: Match Recognize

7.1. Overview

Using *match recognize* patterns are defined in the familiar syntax of regular expressions.

The match recognize syntax presents an alternative way to specify pattern detection as compared to the EPL pattern language described in the previous chapter. A comparison of match recognize and EPL patterns is below.

The match recognize syntax is a proposal for incorporation into the SQL standard. It is thus subject to change as the standard evolves and finalizes (it has not finalized yet). Please consult *row-pattern-recogniton-11-public* [http://dist.codehaus.org/esper//row-pattern-recogniton-11-public.pdf] for further information.

You may be familiar with regular expressions in the context of finding text of interest in a string, such as particular characters, words, or patterns of characters. Instead of matching characters, match recognize matches sequences of events of interest.

Esper can apply match-recognize patterns in real-time upon arrival of new events in a stream of events (also termed incrementally, streaming or continuous). Esper can also match patterns on-demand via the `iterator` pull-API, if specifying a named window or data window on a stream.

7.2. Comparison of Match Recognize and EPL Patterns

This section compares pattern detection via match recognize and via the EPL pattern language.

Table 7.1. Comparison Match Recognize to EPL Patterns

Category	EPL Patterns	Match Recognize
Purpose	Pattern detection in sequences of events.	Same.
Standards	Not standardized, similar to Rapide pattern language.	Proposal for incorporation into the SQL standard.
Real-time Processing	Yes.	Yes.
On-Demand query via Iterator	No.	Yes.
Language	Nestable expressions consisting of boolean `AND`, `OR`, `NOT` and time or arrival-based constructs such as `->` (followed-by), `timer:within` and `timer:interval`.	Regular expression consisting of variables each representing conditions on events.

Category	EPL Patterns	Match Recognize
Event Types	An EPL pattern may react to multiple different types of events.	The input is a single type of event (unless used with variant streams).
Data Window Interaction	Disconnected, i.e. an event leaving a data window does not change pattern state.	Connected, i.e. an event leaving a data window removes the event from match selection.
Semantic Evaluation	Truth-value based: A EPL pattern such as `(A and B)` can fire when a single event arrives that satisfies both A and B conditions.	Sequence-based: A regular expression `(A B)` requires at least two events to match.
Time Relationship Between Events	The `timer:within`, `timer:interval` and `NOT` operator can expressively search for absence of events or other more complex timing relationships.	Some support for detecting absence of events using the `interval` clause.
Extensibility	Custom pattern objects, user-defined functions.	User-defined functions, custom aggregation functions.
Memory Use	Likely between 500 bytes to 2k per open sequence, depends on pattern.	Likely between 100 bytes to 1k per open sequence, depends on pattern.

7.3. Syntax

The synopsis is as follows:

```
match_recognize (
  [ partition by partition_expression [, partition_expression] [,...]  ]
  measures  measure_expression as col_name [, measure_expression as col_name
 ] [,...]
  [ all matches ]
  [ after match skip (past last row | to next row | to current row) ]
  pattern ( variable_regular_expr [, variable_regular_expr] [,...] )
  [ interval time_period ]
  define  variable as variable_condition [, variable as variable_condition]
 [,...]
)
```

The `match_recognize` keyword starts the match recognize definition and occurs right after the `from` clause in an EPL `select` statement. It is followed by parenthesis that surround the match recognize definition.

`Partition by` is optional and may be used to specify that events are to be partitioned by one or more event properties or expressions. If there is no `Partition by` then all rows of the table constitute a single partition. The regular expression applies to events in the same partition and not across partitions.

The `measures` clause defines columns that contain expressions over the pattern variables. The expressions can reference partition columns, singleton variables, aggregates as well as indexed properties on the group variables. Each *measure_expression* expression must be followed by the `as` keyword and a *col_name* column name.

The `all matches` keywords are optional and instructs the engine to find all possible matches. By default matches are ranked and the engine returns a single match following an algorithm to eliminate duplicate matches, as described below. When specifying `all matches`, matches may overlap and may start at the same row.

The `after match skip` keywords are optional and serve to determine the resumption point of pattern matching after a match has been found. By default the behavior is `after match skip past last row`. This means that after eliminating duplicate matches, the logic skips to resume pattern matching at the next event after the last event of the current match.

The `pattern` component is used to specify a regular expression. The regular expression is built from variable names, and may use the operators such as `*`, `+`, `?`, `*?`, `+?`, `??` quantifiers and `|` alteration (concatenation is indicated by the absence of any operator sign between two successive items in a pattern).

With the optional `interval` keyword and time period you can control how long the engine should wait for further events to arrive that may be part of a matching event sequence, before indicating a match (or matches) (not applicable to on-demand pattern matching).

`Define` is a mandatory component, used to specify the boolean condition that defines a variable name that is declared in the pattern. A variable name does not require a definition and if there is no definition, the default is a predicate that is always true. Such a variable name can be used to match any row.

7.3.1. Syntax Example

For illustration, the examples in this chapter use the `TemperatureSensorEvent` event. Each event has 3 properties: the `id` property is a unique event id, the `device` is a sensor device number and the `temp` property is a temperature reading. An event described as `"id=E1, device=1, temp=100"` is a `TemperatureSensorEvent` event with id "E1" for device 1 with a reading of 100.

This example statement looks for two `TemperatureSensorEvent` events from the same device, directly following each other, that indicate a jump in temperature of 10 or more between the two events:

```
select * from TemperatureSensorEvent
match_recognize (
  partition by device
  measures A.id as a_id, B.id as b_id, A.temp as a_temp, B.temp as b_temp
  pattern (A B)
  define
    B as Math.abs(B.temp - A.temp) >= 10
)
```

The `partition by` ensures that the regular expression applies to sequences of events from the same device.

The `measures` clause provides a list of properties or expressions to be selected from matching events. Each property name must be prefixed by the variable name.

In the `pattern` component the statement declares two variables: `A` and `B`. As a matter of convention, variable names are uppercase characters.

The `define` clause specifies no condition for variable A. This means that A defaults to true and any event matches A. Therefore, the pattern can start at any event.

The pattern `A B` indicates to look for a pattern in which an event that fulfills the condition for variable A is immediately followed by an event that fulfills the condition for variable B. A pattern thus defines the sequence (or sequences) of conditions that must be met for the pattern to fire.

Below table is an example sequence of events and output of the pattern:

Table 7.2. Example

Arrival Time	Tuple	Output Event (if any)
1000	id=E1, device=1, temp=50	
2000	id=E2, device=1, temp=55	
3000	id=E3, device=1, temp=60	
4000	id=E4, device=1, temp=70	a_id = E3, b_id = E4, a_temp = 60, b_temp = 70
5000	id=E5, device=1, temp=85	
6000	id=E6, device=1, temp=85	
7000	id=E7, device=2, temp=100	

At time 4000 when event with id `E4` (or event E4 or just E4 for short) arrives the pattern matches and produces an output event. Matching then skips past the last event of the current match (E4) and begins at event E5 (the default skip clause is past last row). Therefore events E4 and E5 do not constitute a match.

At time 3000, events E1 and E3 do not constitute a match as E3 does not immediately follow E, since there is E2 in between.

At time 7000, event E7 does not constitute a match as it is from device 2 and thereby not in the same partition as prior events.

7.4. Pattern and Pattern Operators

The `pattern` specifies the pattern to be recognized in the ordered sequence of events in a partition using regular expression syntax. Each variable name in a pattern corresponds to a boolean condition, which is specified later using the `define` component of the syntax. Thus the `pattern` can be regarded as implicitly declaring one or more variable names; the definition of those variable names appears later in the syntax. If a variable is not defined the variable defaults to true.

It is permitted for a variable name to occur more than once in a pattern, for example `pattern (A B A)`.

7.4.1. Operator Precedence

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

Table 7.3. Match Recognize Regular Expression Operator Precedence

Precedence	Operator	Description	Example
1	Grouping	`()`	`(A B)`
2	Single-character duplication	`* + ?`	`A* B+ C?`
3	Concatenation	(no operator)	`A B`
4	Alternation	`|`	`A | B`

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

7.4.2. Concatenation

The concatenation is indicated by the absence of any operator sign between two successive items in a pattern.

In below pattern the two items A and B have no operator between them. The pattern matches for any event immediately followed by an event from the same device that indicates a jump in temperature over 10:

```
select * from TemperatureSensorEvent
match_recognize (
  partition by device
  measures A.id as a_id, B.id as b_id, A.temp as a_temp, B.temp as b_temp
  pattern (A B)
  define
    B as Math.abs(B.temp - A.temp) >= 10
)
```

Please see the *Section 7.3.1, “Syntax Example”* for a sample event sequence.

7.4.3. Alternation

The alternation operator is a vertical bar (`|`).

The alternation operator has the lowest precedence of all operators. It tells the engine to match either everything to the left of the vertical bar, or everything to the right of the vertical bar. If you want to limit the reach of the alternation, you will need to use round brackets for grouping.

This example pattern looks for a sequence of an event with a temperature over 50 followed immediately by either an event with a temperature less then 45 or an event that indicates the temperature jumped by 10 (all for the same device):

```
select * from TemperatureSensorEvent
match_recognize (
  partition by device
  measures A.id as a_id, B.id as b_id, C.id as c.id
  pattern (A (B | C))
  define
    A as A.temp >= 50,
    B as B.temp <= 45,
    C as Math.abs(C.temp - A.temp) >= 10)
```

Below table is an example sequence of events and output of the pattern:

Table 7.4. Example

Arrival Time	Tuple	Output Event (if any)
1000	id=E1, device=1, temp=50	
2000	id=E2, device=1, temp=45	a_id=E1, b_id=E2, c_id=null
3000	id=E3, device=1, temp=46	
4000	id=E4, device=1, temp=48	
5000	id=E5, device=1, temp=50	
6000	id=E6, device=1, temp=60	a_id = E5, b_id = null, c_id=E6

7.4.4. Quantifiers Overview

Quantifiers are postfix operators with the following choices:

Table 7.5. Quantifiers

Quantifier	Meaning
*	Zero or more matches (greedy).
+	One or more matches (greedy).
?	Zero or one match (greedy).
*?	Zero or more matches (reluctant).
+?	One or more matches (reluctant).
??	Zero or one match (reluctant).

7.4.5. Variables Can be Singleton or Group

A *singleton variable* is a variable in a pattern that does not have a quantifier or has a zero-or-one quantifier (`?` or `??`) and occurs only once in the pattern (except with alteration). In the `measures` clause a singleton variable can be selected as:

```
variableName.propertyName
```

Variables with a zero-or-more or one-or-more quantifier, or variables that occur multiple places in a pattern (except when using alteration), may match multiple events and are *group variables*. In the `measures` clause a group variable must be selected either by providing an index or via any of the aggregation functions, such as `first`, `last`, `count` and `sum`:

```
variableName[index].propertyName
```

```
last(variableName.propertyName)
```

Please find examples of singleton and group variables and example `measures` clauses below.

7.4.5.1. Additional Aggregation Functions

For group variables all existing aggregation functions can be used and in addition the following aggregation functions may be used:

Table 7.6. Syntax and results of aggregate functions

Aggregate Function	Result
first([all|distinct] *expression*)	Returns the first value.
last([all|distinct] *expression*)	Returns the last value.

7.4.6. Eliminating Duplicate Matches

The execution of match recognize is continuous and real-time by default. This means that every arriving event, or batch of events if using batching, evaluates against the pattern and matches are immediately indicated. Elimination of duplicate matches occurs between all matches of the arriving events (or batch of events) at a given time.

As an alternative, and if your application does not require continuous pattern evaluation, you may use the `iterator` API to perform on-demand matching of the pattern. For the purpose of indicating to the engine to not generate continuous results, specify the `@Hint('iterate_only')` hint.

When using one-or-more, zero-or-more or zero-or-one quantifiers (`?, +, *, ??, +?, *?`), the output of the real-time continuous query can differ from the output of the on-demand `iterator` execution: The continuous query will output a match (or multiple matches) as soon as matches are detected at a given time upon arrival of events (not knowing what further events may arrive). The on-demand execution, since it knows all possible events in advance, can determine the longest match(es). Thus elimination of duplicate matches can lead to different results between real-time and on-demand use.

If the `all matches` keywords are specified, then all matches are returned as the result and no elimination of duplicate matches as below occurs.

Otherwise matches to a pattern in a partition are ordered by preferment. Preferment is given to matches based on the following priorities:

1. A match that begins at an earlier row is preferred over a match that begins at a later row.
2. Of two matches matching a greedy quantifier, the longer match is preferred.
3. Of two matches matching a reluctant quantifier, the shorter match is preferred.

After ranking matches by preferment, matches are chosen as follows:

1. The first match by preferment is taken.
2. The pool of matches is reduced as follows based on the SKIP TO clause: If SKIP PAST LAST ROW is specified, all matches that overlap the first match are discarded from the pool. If SKIP TO NEXT ROW is specified, then all matches that overlap the first row of the first match are discarded. If SKIP TO CURRENT ROW is specified, then no matches are discarded.
3. The first match by preferment of the ones remaining is taken.
4. Step 2 is repeated to remove more matches from the pool.
5. Steps 3 and 4 are repeated until there are no remaining matches in the pool.

7.4.7. Greedy Or Reluctant

Reluctant quantifiers are indicated by an additional question mark (`*?, +?, ??,`). Reluctant quantifiers try to match as few rows as possible, whereas non-reluctant quantifiers are greedy and try to match as many rows as possible.

Greedy and reluctant come into play only for match selection among multiple possible matches. When specifying `all matches` there is no difference between greedy and reluctant quantifiers.

Consider the below example. The conditions may overlap: an event with a temperature reading of 105 and over matches both A and B conditions:

```
select * from TemperatureSensorEvent
match_recognize (
  partition by device
  measures A.id as a_id, B.id as b_id
  pattern (A?? B?)
  define
    A as A.temp >= 100
    B as B.temp >= 105)
```

A sample sequence of events and pattern matches:

Table 7.7. Example

Arrival Time	Tuple	Output Event (if any)
1000	id=E1, device=1, temp=99	
2000	id=E2, device=2, temp=106	a_id=null, b_id=E2
3000	id=E3, device=1, temp=100	a_id=E3, b_id=null

As the `?` qualifier on condition B is greedy, event E2 matches the pattern and is indicated as a B event by the `measure` clause (and not as an A event therefore `a_id` is null).

7.4.8. Quantifier - One Or More (+ and +?)

The one-or-more quantifier (+) must be matched one or more times by events. The operator is greedy and the reluctant version is `+?`.

In the below example with `pattern (A+ B+)` the pattern consists of two variable names, A and B, each of which is quantified with `+`, indicating that they must be matched one or more times.

The pattern looks for one or more events in which the temperature is over 100 followed by one or more events indicating a higher temperature:

```
select * from TemperatureSensorEvent
match_recognize (
  partition by device
   measures first(A.id) as first_a, last(A.id) as last_a, B[0].id as b0_id,
 B[1].id as b1_id
  pattern (A+ B+)
  define
 A as A.temp >= 100,
 B as B.temp > A.temp)
```

An example sequence of events that matches the pattern above is:

Table 7.8. Example

Arrival Time	Tuple	Output Event (if any)
1000	id=E1, device=1, temp=99	
2000	id=E2, device=1, temp=100	
3000	id=E3, device=1, temp=100	
4000	id=E4, device=1, temp=101	first_a = E2, last_a = E3, b0_id = E4, b1_id = null
5000	id=E5, device=1, temp=102	

Note that for continuous queries, there is no match that includes event E5 since after the pattern matches for E4 the pattern skips to start fresh at E5 (by default skip clause). When performing on-demand matching via `iterator`, event E5 gets included in the match and the output is `first_a = E2, last_a = E3, b0_id = E4, b1_id = E5`.

7.4.9. Quantifier - Zero Or More (* and *?)

The zero-or-more quantifier (*) must be matched zero or more times by events. The operator is greedy and the reluctant version is `*?`.

The pattern looks for a sequence of events in which the temperature starts out below 50 and then stays between 50 and 60 and finally comes over 60:

```
select * from TemperatureSensorEvent
match_recognize (
  partition by device
  measures A.id as a_id, count(B.id) as count_b, C.id as c_id
  pattern (A B* C)
  define
 A as A.temp < 50,
 B as B.temp between 50 and 60,
 C as C.temp > 60)
```

An example sequence of events that matches the pattern above is:

Table 7.9. Example

Arrival Time	Tuple	Output Event (if any)
1000	id=E1, device=1, temp=55	

Arrival Time	Tuple	Output Event (if any)
2000	id=E2, device=1, temp=52	
3000	id=E3, device=1, temp=49	
4000	id=E4, device=1, temp=51	
5000	id=E5, device=1, temp=55	
6000	id=E5, device=1, temp=61	a_id=E3, count_b=2, c_id=E6

7.4.10. Quantifier - Zero Or One (? and ??)

The zero-or-one quantifier (?) must be matched zero or one time by events. The operator is greedy and the reluctant version is `??`.

The pattern looks for a sequence of events in which the temperature is below 50 and then dips to over 50 and then to under 50 before indicating a value over 55:

```
select * from TemperatureSensorEvent
match_recognize (
  partition by device
  measures A.id as a_id, B.id as b_id, C.id as c_id, D.id as d_id
  pattern (A B? C? D)
  define
 A as A.temp < 50,
 B as B.temp > 50,
 C as C.temp < 50,
 D as D.temp > 55)
```

An example sequence of events that matches the pattern above is:

Table 7.10. Example

Arrival Time	Tuple	Output Event (if any)
1000	id=E1, device=1, temp=44	
2000	id=E2, device=1, temp=49	
3000	id=E3, device=1, temp=51	
4000	id=E4, device=1, temp=49	
5000	id=E5, device=1, temp=56	a_id=E2, b_id=E3, c_id=E4, d_id=E5
6000	id=E5, device=1, temp=61	

7.5. `Define` Clause

Within `define` are listed the boolean conditions that defines a variable name that is declared in the pattern.

A variable name does not require a definition and if there is no definition, the default is a predicate that is always true. Such a variable name can be used to match any row.

The definitions of variable names may reference the same or other variable names as prior examples have shown.

If a variable in your condition expression is a singleton variable, then only individual columns may be referenced. If the variable is not matched by an event, a `null` value is returned.

If a variable in your condition expression is a group variable, then only indexed columns may be referenced. If the variable is not matched by an event, a `null` value is returned.

Aggregation functions are not allowed within expressions of the `define` clause.

7.5.1. The `Prev` Operator

The `prev` function may be used in a `define` expression to access columns of the previous row of a variable name. If there is no previous row, the null value is returned.

The `prev` function can accept an optional non-negative integer argument indicating the offset to the previous rows. That argument must be a constant. In this case, the engine returns the property from the N-th row preceding the current row, and if the row doesn't exist, it returns `null`.

This function can access variables currently defined, for example:

```
Y as Y.price < prev(Y.price, 2)
```

It is not legal to use `prev` with another variable then the one being defined:

```
// not allowed
Y as Y.price < prev(X.price, 2)
```

The `prev` function returns properties of events in the same partition. Also, it returns properties of events according to event order-of-arrival. When using data windows or deleting events from a named window, the remove stream does not remove events from the `prev` function.

The pattern looks for an event in which the temperature is greater or equal 100 and that, relative to that event, has an event preceding it by 2 events that also had a temperature greater or equal 100:

```
select * from TemperatureSensorEvent
match_recognize (
  partition by device
  measures A.id as a_id
  pattern (A)
  define
```

```
 A as A.temp > 100 and prev(A.temp, 2) > 100)
```

An example sequence of events that matches the pattern above is:

Table 7.11. Example

Arrival Time	Tuple	Output Event (if any)
1000	id=E1, device=1, temp=98	
2000	id=E2, device=1, temp=101	
3000	id=E3, device=1, temp=101	
4000	id=E4, device=1, temp=99	
5000	id=E5, device=1, temp=101	a_id=E5

7.6. `Measure` Clause

The `measures` clause defines exported columns that contain expressions over the pattern variables. The expressions can reference partition columns, singleton variables and any aggregation functions including `last` and `first` on the group variables.

Expressions in the `measures` clause must use the `as` keyword to assign a column name.

If a variable is a singleton variable then only individual columns may be referenced, not aggregates. If the variable is not matched by an event, a `null` value is returned.

If a variable is a group variable and used in an aggregate, then the aggregate is performed over all rows that have matched the variable. If a group variable is not in an aggregate function, its variable name must be post-fixed with an index. See *Section 7.4.5, “Variables Can be Singleton or Group”* for more information.

7.7. Datawindow-Bound

When using match recognize with a named window or stream bound by a data window, all events removed from the named window or data window also removed the match-in-progress that includes the event(s) removed.

The next example looks for four sensor events from the same device immediately following each other and indicating a rising temperature, but only events that arrived in the last 10 seconds are considered:

```
select * from TemperatureSensorEvent.win:time(10 sec)
match_recognize (
partition by device
measures A.id as a_id
pattern (A B C D)
define
B as B.temp > A.temp,
C as C.temp > B.temp,
D as D.temp > C.temp)
```

An example sequence of events that matches the pattern above is:

Table 7.12. Example

Arrival Time	Tuple	Output Event (if any)
1000	id=E1, device=1, temp=80	
2000	id=E2, device=1, temp=81	
3000	id=E3, device=1, temp=82	
4000	id=E4, device=1, temp=81	
7000	id=E5, device=1, temp=82	
9000	id=E6, device=1, temp=83	
13000	id=E7, device=1, temp=84	a_id=E4, a_id=E5, a_id=E6, a_id=E7
15000	id=E8, device=1, temp=84	
20000	id=E9, device=1, temp=85	
21000	id=E10, device=1, temp=86	
26000	id=E11, device=1, temp=87	

Note that E8, E9, E10 and E11 doe not constitute a match since E8 leaves the data window at 25000.

7.8. Interval

With the optional `interval` keyword and time period you can control how long the engine should wait for further events to arrive that may be part of a matching event sequence, before indicating a match (or matches). This is not applicable to on-demand pattern matching.

The interval timer starts are the arrival of the first event matching a sequence for a partition. When the time interval passes and an event sequence matches, duplicate matches are eliminated and output occurs.

The next example looks for sensor events indicating a temperature of over 100 waiting for any number of additional events with a temperature of over 100 for 10 seconds before indicating a match:

```
select * from TemperatureSensorEvent
match_recognize (
partition by device
measures A.id as a_id, count(B.id) as count_b, first(B.id) as first_b, last(B.id)
 as last_b
pattern (A B*)
interval 5 seconds
define
  A as A.temp > 100,
  B as B.temp > 100)
```

An example sequence of events that matches the pattern above is:

Table 7.13. Example

Arrival Time	Tuple	Output Event (if any)
1000	id=E1, device=1, temp=98	
2000	id=E2, device=1, temp=101	
3000	id=E3, device=1, temp=102	
4000	id=E4, device=1, temp=104	
5000	id=E5, device=1, temp=104	
7000		a_id=E2, count_b=3, first_b=E3, last_b=E5

Notice that the engine waits 5 seconds (5000 milliseconds) after the arrival time of the first event E2 of the match at 2000, to indicate the match at 7000.

7.9. Use with Different Event Types

You may match different types of events using match-recognize by following any of these strategies:

1. Declare a variant stream.
2. Declare a supertype for your event types in the `create schema` syntax.
3. Have you event classes implement a common interface or extend a common base class.

A short example that demonstrates variant streams and match-recognize is listed below:

```
// Declare one sample type
create schema S0 as (col string)
```

```
// Declare second sample type
create schema S1 as (col string)
```

```
// Declare variant stream holding either type
```

```
create variant schema MyVariantStream as S0, S1
```

```
// Populate variant stream
insert into MyVariantStream select * from S0
```

```
// Populate variant stream
insert into MyVariantStream select * from S1
```

```
// Simple pattern to match S0 S1 pairs
select * from MyVariantType.win:time(1 min)
match_recognize (
  measures A.id? as a, B.id? as b
  pattern (A B)
  define
    A as typeof(A) = 'S0',
    B as typeof(B) = 'S1'
)
```

7.10. Limitations

Please note the following limitations:

1. Subqueries are not allowed in expressions within `match_recognize`.
2. Joins and outer joins are not allowed in the same statement as `match_recognize`.
3. `match_recognize` may not be used within `on-select` or `on-insert` statements.
4. When using `match_recognize` on unbound streams (no data window provided) the `iterator` pull API returns no rows.
5. A Statement Object Model API for `match_recognize` is not yet available.

Chapter 8. EPL Reference: Operators

Esper arithmetic and logical operator precedence follows Java standard arithmetic and logical operator precedence.

8.1. Arithmetic Operators

The below table outlines the arithmetic operators available.

Table 8.1. 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.

8.2. Logical And Comparison Operators

The below table outlines the logical and comparison operators available.

Table 8.2. 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.
=, !=, <, > <=, >=, is, is not	Comparison.

8.2.1. Null-Value Comparison Operators

The `null` value is a special value, see *http://en.wikipedia.org/wiki/Null_(SQL)* [http://en.wikipedia.org/wiki/Null_%28SQL%29] (source:Wikipedia) for more information.

Thereby the following expressions all return `null`:

```
2 != null
```

```
null = null
```

```
2 != null or 1 = 2
```

```
2 != null and 2 = 2
```

Use the `is` and `is not` operators for comparing values that can be null.

The following expressions all return `true`:

```
2 is not null
```

```
null is not 2
```

```
null is null
```

```
2 is 2
```

The engine allows `is` and `is not` with any expression, not only in connection with the `null` constant.

8.3. Concatenation Operators

The below table outlines the concatenation operators available.

Table 8.3. Syntax and results of concatenation operators

Operator	Description
||	Concatenates character strings

8.4. Binary Operators

The below table outlines the binary operators available.

Table 8.4. 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).

8.5. 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})
```

8.6. Dot Operator

You can use the dot operator to invoke a method on the result of an expression. The dot operator uses the dot (`.`) or period character.

The dot-operator is relevant with enumeration methods: Enumeration methods perform tasks such as transformation, filtering, aggregation, sequence-matching, sorting and others on subquery results, named windows, event properties or inputs that are or can be projected to a collection of events, scalar values or objects. See *Chapter 10, EPL Reference: Enumeration Methods*

Further the dot-operator is relevant to date-time methods. Date-time methods work on date-time values to add or subtract time periods, set or round calendar fields or query fields, among other tasks. See *Chapter 11, EPL Reference: Date-Time Methods*.

This section only describes the dot-operator in relation to property instance methods, the special `get` and `size` indexed-property methods and duck typing.

The synopsis for the dot operator is as follows

```
expression.method([parameter [,...]])[.method(...)][...]
```

The expression to evaluate by the dot operator is in parenthesis. After the dot character follows the method name and method parameters in parenthesis.

You may use the dot operator when your expression returns an object that you want to invoke a method on. The dot operator allows duck typing and convenient array and collection access methods.

This example statement invokes the `getZones` method of the RFID event class by referring to the stream name assigned in the `from`-clause:

```
select rfid.getZones() from RFIDEvent as rfid
```

The `size()` method can be used to return the array length or collection size. Use the `get` method to return the value at a given index for an array or collection.

The next statement selects array size and returns the last array element:

```
select arrayproperty.size() as arraySize,
  arrayproperty.get((arrayproperty).size - 1) as lastInArray
  from ProductEvent
```

8.6.1. Duck Typing

Duck typing is when the engine checks at runtime for the existence of a method regardless of object class inheritance hierarchies. This can be useful, for example, when a dynamic property returns an object which may or may not provide a method to return the desired value.

Duck typing is disabled in the default configuration to consistently enforce strong typing. Please enable duck typing via engine expression settings as described in *Section 15.4.22, “Engine Settings related to Expression Evaluation”*.

The statement below selects a dynamic property by name `productDesc` and invokes the `getCounter()` method if that method exists on the property value, or returns the null value if the method does not exist for the dynamic property value of if the dynamic property value itself is null:

```
select (productDesc?).getCounter() as arraySize from ProductEvent
```

8.7. 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`.

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'
```

Expression may also return an array, a `java.util.Collection` or a `java.util.Map`. Thus event properties that are lists, sets or maps may provide values to compare against *test_expression*.

All expressions must be of the same type or a compatible type to *test_expression*. The `in` keyword may coerce number values to compatible types. If *expression* returns an array, then the component type of the array must be compatible, unless the component type of the array is `Object`.

If *expression* returns an array of component type `Object`, the operation compares each element of the array, applying `equals` semantics.

If *expression* returns a `Collection`, the operation determines if the collection contains the value returned by *test_expression*, applying `contains` semantics.

If *expression* returns a `Map`, the operation determines if the map contains the key value returned by *test_expression*, applying `containsKey` semantics.

Constants, arrays, `Collection` and `Map` expressions or event properties can be used combined.

For example, and assuming a property named 'mySpecialCmdList' exists that contains a list of command strings:

```
select * from RFIDEvent where command in ( 'OBSERVATION', 'SIGNAL',
 mySpecialCmdList)
```

When using prepared statements and substitution parameters with the `in` keyword, make sure to retain the parenthesis. Substitution values may also be arrays, `Collection` and `Map` values:

```
test_expression [not] in (? [,?...] )
```

Note that if there are no successes and at least one right-hand row yields null for the operator's result, the result of the any construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

8.7.1. '`in`' for Range Selection

The `in` keyword can be used to specify ranges, including open, half-closed, half-open and inverted ranges.

Ranges come in the following 4 varieties. The round `()` or square `[]` bracket indicate 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 following statement two statements are equivalent: Both statements select orders where the price is in the range of zero and 10000 (endpoints inclusive):

```
select * from OrderEvent where price in [0:10000]
select * from OrderEvent where price between 0 and 10000
```

The next statement selects order events where the price is greater then 100 and less-or-equal to 2000:

```
select * from OrderEvent where price in (100:2000]
```

Use the `not in` keywords to specify an inverted range.

The following statement selects an inverted range by selecting all order events where the price is less then zero or the price is greater or equal to 10000:

```
select * from OrderEvent where price not in (0:10000]
```

In case the value of low endpoint is less then the value of high endpoint the `in` operator reverses the range.

The following two statements are also equivalent:

```
select * from OrderEvent where price in [10000:0]
```

```
select * from OrderEvent where price >= 0 and price <= 1000
```

8.8. 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 (endpoints 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
```

While the `between` keyword always includes the endpoints of the range, the `in` operator allows finer control of endpoint inclusion.

In case the value of *begin_expression* is less then the value of *end_expression* the `between` operator reverses the range.

The following two statements are also equivalent:

```
select * from StockTickEvent where price between 60 and 55
```

```
select * from StockTickEvent where price >= 55 and price <= 60
```

8.9. 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 '!'
```

8.10. 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.*'
```

The `rexexp` function matches the entire region against the pattern via `java.util.regex.Matcher.matches()` method. Please consult the Java API documentation for more information or refer to *Regular Expression Flavors* [http://www.regular-expressions.info/refflavors.html].

8.11. The '`any`' and '`some`' Keywords

The `any` operator is true if the expression returns true for one or more of the values returned by a list of expressions including array, `Collection` and `Map` values.

The synopsis for the `any` keyword is as follows:

```
expression operator any (expression [,expression...] )
```

The left-hand expression is evaluated and compared to each expression result using the given operator, which must yield a Boolean result. The result of `any` is "true" if any true result is obtained.

The result is "false" if no true result is found (including the special case where the expressions are collections that return no rows).

The *operator* can be any of the following values: `=, !=, <>, <, <=, >, >=`.

The `some` keyword is a synonym for `any`. The `in` construct is equivalent to `= any`.

Expression may also return an array, a `java.util.Collection` or a `java.util.Map`. Thus event properties that are lists, sets or maps may provide values to compare against.

All expressions must be of the same type or a compatible type. The `any` keyword coerces number values to compatible types. If *expression* returns an array, then the component type of the array must be compatible, unless the component type of the array is `Object`.

If *expression* returns an array, the operation compares each element of the array.

If *expression* returns a `Collection`, the operation determines if the collection contains the value returned by the left-hand expression, applying `contains` semantics. When using relationship operators `<, <=, >, >=` the operator applies to each element in the collection, and non-numeric elements are ignored.

If *expression* returns a `Map`, the operation determines if the map contains the key value returned by the left-hand expression, applying `containsKey` semantics. When using relationship operators `<, <=, >, >=` the operator applies to each key in the map, and non-numeric map keys are ignored.

Constants, arrays, `Collection` and `Map` expressions or event properties can be used combined.

The next statement demonstrates the use of the `any` operator:

```
select * from ProductOrder where category != any (categoryArray)
```

The above query selects ProductOrder event that have a category field and a category array, and returns only those events in which the category value is not in the array.

Note that if there are no successes and at least one right-hand row yields null for the operator's result, the result of the `any` construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

8.12. The '`all`' Keyword

The `all` operator is true if the expression returns true for all of the values returned by a list of expressions including array, `Collection` and `Map` values.

The synopsis for the `all` keyword is as follows:

```
expression operator all (expression [,expression...] )
```

The left-hand expression is evaluated and compared to each expression result using the given operator, which must yield a Boolean result. The result of `all` is "true" if all rows yield true (including the special case where the expressions are collections that returns no rows). The result is "false" if any false result is found. The result is `null` if the comparison does not return false for any row, and it returns `null` for at least one row.

The *operator* can be any of the following values: `=, !=, <>, <, <=, >, >=`.

The `not in` construct is equivalent to `!= all`.

Expression may also return an array, a `java.util.Collection` or a `java.util.Map`. Thus event properties that are lists, sets or maps may provide values to compare against.

All expressions must be of the same type or a compatible type. The `all` keyword coerces number values to compatible types. If *expression* returns an array, then the component type of the array must be compatible, unless the component type of the array is `Object`.

If *expression* returns an array, the operation compares each element of the array.

If *expression* returns a `Collection`, the operation determines if the collection contains the value returned by the left-hand expression, applying `contains` semantics. When using relationship operators `<, <=, >, >=` the operator applies to each element in the collection, and non-numeric elements are ignored.

If *expression* returns a `Map`, the operation determines if the map contains the key value returned by the left-hand expression, applying `containsKey` semantics. When using relationship operators `<, <=, >, >=` the operator applies to each key in the map, and non-numeric map keys are ignored.

Constants, arrays, `Collection` and `Map` expressions or event properties can be used combined.

The next statement demonstrates the use of the `all` operator:

```
select * from ProductOrder where category = all (categoryArray)
```

The above query selects ProductOrder event that have a category field and a category array, and returns only those events in which the category value matches all values in the array.

8.13. The '`new`' Keyword

The `new` operator populates a new data structure by evaluating column names and assignment expressions. This is useful when an expression should return multiple results, for performing a transformation or inside enumeration method lambda expressions.

The synopsis for the `new` keyword is as follows:

```
new { column_name = [assignment_expression] [,column_name...] }
```

The result of the new-operator is a map data structure that contains *column_name* keys and values. If an assignment expression is provided for a column, the operator evaluates the expression and assigns the result to the column name. If no assignment expression is provided, the column name is assumed to be an event property name and the value is the event property value.

The next statement demonstrates the use of the `new` operator:

```
select new {category, price = 2*price} as priceInfo from ProductOrder
```

The above query returns a single property `priceInfo` for each arriving ProductOrder event. The property value is itself a map that contains two entries: For the key name `category` the value of the category property and for the key name `price` the value of the price property multiplied by two.

The next EPL is an example of the `new` operator within an expression definition and a `case`-statement (one EPL statement not multiple):

```
expression calcPrice {
  productOrder => case
    when category = 'fish' then new { sterialize = 'XRAY', priceFactor = 1.01 }
    when category = 'meat' then new { sterialize = 'UVL', priceFactor = 1 }
  end
}

select calcPrice(po) as priceDetail from ProductOrder po
```

In above example the expression `calcPrice` returns both a `sterialize` string value and a `priceFactor` double value. The expression is evaluated as part of the `select`-clause and the map-type result placed in the `priceDetail` property pf output events.

When used within the `case` operator, the operator validates that the data structure is compatible between each case-when result in terms of column names and types. The default value for `else` in `case` is `null`.

Chapter 9. EPL Reference: Functions

9.1. 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.

You may also register your own single-row function name with the engine so that your EPL statements become less cluttered. This is described in detail in *Section 17.3, “Single-Row Function”*. Single-row functions that return an object can be chained.

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 15.4.6, “Class and package imports”*.

The below table outlines the built-in single-row functions available.

Table 9.1. 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`
	Returns the `result` for the first condition that is true.

Single-row Function	Result
`[when condition then result ...] [else result] end`	
`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` millisecond 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.
`istream()`	Returns true if the event is part of the insert stream and false if the event is part of the remove stream.
`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 or all properties of a previous event, relative to the event order within a data window, or according to an optional index parameter (N) the positional Nth-from-last value.
`prevtail(expression, event_property)`	Returns a property value or all properties of the first event in a data window relative to the event order within a data window, or according to an optional index parameter (N) the positional Nth-from-first value.
`prevwindow(event_property)`	Returns a single property value of all events or all properties of all events in

Single-row Function	Result
	a data window in the order that reflects the sort order of the data window.
`prevcount(event_property)`	Returns the count of events (number of data points) in a data window.
`prior(integer, event_property)`	Returns a property value of a prior event, relative to the natural order of arrival of events
`typeof(expression)`	If expression is a stream name, returns the event type name of the evaluated event, often used with variant streams. If expression is a property name or expression, returns the name of the expression result type.

9.1.1. 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 `case` expression is sometimes used with the `new` operator to return multiple results, see *Section 8.13, “The 'new' Keyword”*.

The example below shows the first version of a `case` statement. It has a `String` return type and returns the value 'one'.

```
select case myexpression 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 ...
```

9.1.2. 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, BigInteger, BigDecimal`, where `string` is a short notation for `java.lang.String` and `BigInteger` as well as `BigDecimal` are the classes in `java.math`. 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 .

9.1.3. 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 ...
```

9.1.4. 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
```

9.1.5. 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 its 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
```

9.1.6. 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.

9.1.7. The `Istream` Function

The `istream` function returns a boolean value indicating whether within the context of expression evaluation the current event or set of events (joins) are part of the insert stream (true) or part of the remove stream (false). The function takes no parameters.

Use the `istream` function with data windows and `select irstream` and `insert irstream into`.

In the following example the `istream` function always returns boolean true since no data window is declared:

```
select irstream *, istream() from OrderEvent
```

The next example declares a data window. For newly arriving events the function returns boolean true, for events that expire after 10 seconds the function returns boolean false:

```
select irstream *, istream() from OrderEvent.win:time(10 sec)
```

The `istream` function returns true for all cases where insert or remove stream does not apply, such as when used in parameter expressions to data windows or in stream filter expressions.

9.1.8. 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
```

9.1.9. The `Previous` Function

The `prev` function returns the property value or all event properties of a previous event. For data windows that introduce a sort order other then the order of arrival, such as the sorted data window and the time order data window, the function returns the event at the specified position.

The `prev` function is not an aggregation function and therefore does not return results per group when used with `group by`. Please consider the `last`, `lastever` or `first` aggregation functions instead as described in *Section 9.2.2, “Event Aggregation Functions”*. You must use an aggregation function instead of `prev` when querying a named window.

The first parameter to the `prev` function is an index parameter and denotes the i-th previous event, in the order established by the data window. If no index is provided, the default index is 1 and the function returns the previous event. The second parameter is a property name or stream name. If specifying a property name, the function returns the value for the previous event property value. If specifying a stream name, the function returns the previous event underlying object.

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)
```

By using the stream alias in the `previous` function, the next example selects the trade event itself that is immediately previous to the current Trade event

```
select prev(1, trade) from Trade.win:length(10) as trade
```

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(3, volume desc)
```

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(10,    volume    desc),
 RankSelectionEvent unidirectional
```

Use the `prev` function in combination with a grouped data window to access a previous event per grouping criteria.

The example below returns the price of the previous Trade event for the same symbol, or `null` if for that symbol there is no previous Trade event:

```
select prev(1, price) from Trade.std:groupwin(symbol).win:length(2)
```

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)
```

An alternative form of the `prev` function allows the index to not appear or appear after the property name if the index value is a constant and not an expression:

```
select prev(1, symbol) from Trade
// ... equivalent to ...
select prev(symbol) from Trade
// ... and ...
select prev(symbol, 1) from Trade
```

The combination of the `prev` function and `std:groupwin` view returns the property value for a previous event in the given data window group.

The following example returns for each event the current smallest price per symbol:

```
select symbol, prev(0, price) as topPricePerSymbol
from Trade.std:groupwin(symbol).ext:sort(1, price asc)
```

9.1.9.1. 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 `std:groupwin` and data window view, without any additional sub-views. See *Section 12.2, "Data Window Views"* for built-in data window views.

9.1.9.2. 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:groupwin` 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.

9.1.10. The `Previous-Tail` Function

The `prevtail` function returns the property value or all event properties of the positional-first event in a data window. For data windows that introduce a sort order other then the order of arrival, such as the sorted data window and the time order data window, the function returns the first event at the specified position.

The `prevtail` function is not an aggregation function and therefore does not return results per group when used with `group by`. Please consider the `first`, `firstever` or `window` aggregation functions instead as described in *Section 9.2.2, “Event Aggregation Functions”*. You must use an aggregation function instead of `prevtail` when querying a named window.

The first parameter is an index parameter and denotes the i-th from-first event in the order established by the data window. If no index is provided the default is zero and the function returns the first event in the data window. The second parameter is a property name or stream name. If specifying a property name, the function returns the value for the previous event property value. If specifying a stream name, the function returns the previous event underlying object.

This example selects the value of the `price` property of the first (oldest) event held in the length window:

```
select prevtail(price) from Trade.win:length(10)
```

By using the stream alias in the `prevtail` function, the next example selects the trade event itself that is the second event held in the length window:

```
select prevtail(1, trade) from Trade.win:length(10) as trade
```

Since the `prevtail` 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 smallest, second-smallest and third-smallest volume.

```
select prevtail(0, symbol), prevtail(1, symbol), prevtail(2, symbol)
  from Trade.ext:sort(3, volume asc)
```

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  prevtail(rank,  symbol)  from  Trade.ext:sort(10,  volume  asc),
 RankSelectionEvent unidirectional
```

The `prev` function returns a `null` value if the data window does not currently holds positional-first or the Nth-from-first event. For batch data windows the value returned is relative to the current batch.

The following example returns the first and second symbol value in the batch:

```
select prevtail(0, symbol), prevtail(1, symbol) from Trade.win:time_batch(1 min)
```

An alternative form of the `prevtail` function allows the index to not appear or appear after the property name if the index value is a constant and not an expression:

```
select prevtail(1, symbol) from Trade
// ... equivalent to ...
select prevtail(symbol) from Trade
// ... and ...
select prevtail(symbol, 1) from Trade
```

The combination of the `prevtail` function and `std:groupwin` view returns the property value for a positional first event in the given data window group.

Let's look at an example. This statement outputs the oldest price per symbol retaining the last 10 prices per symbol:

```
select symbol, prevtail(0, price) as oldestPrice
from Trade.std:groupwin(symbol).win:length(10)
```

9.1.10.1. 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 `std:groupwin` and data window view, without any additional sub-views. See *Section 12.2, “Data Window Views”* for built-in data window views.

9.1.11. The `Previous-Window` Function

The `prevwindow` function returns property values or all event properties for all events in a data window. For data windows that introduce a sort order other then the order of arrival, such as the sorted data window and the time order data window, the function returns the event data sorted in that order, otherwise it returns the events sorted by order of arrival with the newest arriving event first.

The `prevwindow` function is not an aggregation function and therefore does not return results per group when used with `group by`. Please consider the `window` aggregation function instead as described in *Section 9.2.2, “Event Aggregation Functions”*. You must use an aggregation function instead of `prevwindow` when querying a named window.

The single parameter is a property name or stream name. If specifying a property name, the function returns the value of the event property for all events held by the data window. If specifying a stream name, the function returns the event underlying object for all events held by the data window.

This example selects the value of the `price` property of all events held in the length window:

```
select prevwindow(price) from Trade.win:length(10)
```

By using the stream alias in the `prevwindow` function, the next example selects all trade events held in the length window:

```
select prevwindow(trade) from Trade.win:length(10) as trade
```

When used with a data window that introduces a certain sort order, the `prevwindow` function returns events sorted according to that sort order.

The next statement outputs for every arriving event the current 10 underying trade event objects that have the largest volume:

```
select prevwindow(trade) from Trade.ext:sort(10, volume desc) as trade
```

The `prevwindow` function returns a `null` value if the data window does not currently hold any events.

The combination of the `prevwindow` function and `std:groupwin` view returns the property value(s) for all events in the given data window group.

This example statement outputs all prices per symbol retaining the last 10 prices per symbol:

```
select symbol, prevwindow(price) from Trade.std:groupwin(symbol).win:length(10)
```

9.1.11.1. 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 `std:groupwin` and data window view, without any additional sub-views. See *Section 12.2, “Data Window Views”* for built-in data window views.

9.1.12. The `Previous-Count` Function

The `prevcount` function returns the number of events held in a data window.

The `prevcount` function is not an aggregation function and therefore does not return results per group when used with `group by`. Please consider the `count(*)` aggregation function instead as described in *Section 9.2, “Aggregation Functions”*. You must use an aggregation function instead of `prevcount` when querying a named window.

The single parameter is a property name or stream name of the data window to return the count for.

This example selects the number of data points for the `price` property held in the length window:

```
select prevcount(price) from Trade.win:length(10)
```

By using the stream alias in the `prevcount` function the next example selects the count of trade events held in the length window:

```
select prevcount(trade) from Trade.win:length(10) as trade
```

The combination of the `prevcount` function and `std:groupwin` view returns the count of events in the given data window group.

This example statement outputs the number of events retaining the last 10 events per symbol:

```
select symbol, prevcount(price) from Trade.std:groupwin(symbol).win:length(10)
```

9.1.12.1. 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 `std:groupwin` and data window view, without any additional sub-views. See *Section 12.2, “Data Window Views”* for built-in data window views.

9.1.13. 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. The second parameter is a property name or stream name. If specifying a property name, the function returns the property value for the prior event. If specifying a stream name, the function returns the prior event underlying object.

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
```

By using the stream alias in the `prior` function, the next example selects the trade event itself that is immediately prior to the current Trade event

```
select prior(1, trade) from Trade as 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)
```

9.1.14. The `Type-Of` Function

The `typeof` function, when parameterized by a stream name, returns the event type name of the evaluated event which can be useful with variant streams. When parameterized by an expression or property name, the function returns the type name of the expression result or `null` if the expression result is null.

In summary, the function determines the return type of the expression at runtime by evaluating the expression and returns the type name of the expression result.

The `typeof` function is often used in conjunction with variant streams. A variant stream is a predefined stream into which events of multiple disparate event types can be inserted. The `typeof` function, when passed a stream name alias, returns the name of the event type of each event in the stream.

The following example elaborates on the use of variant streams with `typeof`. The first statement declares a variant stream `SequencePatternStream`:

```
create variant schema SequencePatternStream as *
```

The next statement inserts all order events and is followed by a statement to insert all product events:

```
insert into SequencePatternStream select * from OrderEvent;
```

```
insert into SequencePatternStream select * from PriceEvent;
```

This example statement returns the event type name for each event in the variant stream:

```
select typeof(sps) from SequencePatternStream as sps
```

The next example statement detects a pattern by utilizing the `typeof` function to find pairs of order event immediately followed by product event:

```
select * from SequencePatternStream match_recognize(
  measures A as a, B as b
  pattern (A B)
  define A as typeof(A) = "OrderEvent",
         B as typeof(B) = "ProductEvent"
  )
```

When passing a property name to the `typeof` function, the function evaluates whether the property type is event type (a fragment event type). If the property type is event type, the function returns the type name of the event in the property value or `null` if not provided. If the property type is not event type, the function returns the simple class name of the property value.

When passing an expression to the `typeof` function, the function evaluates the expression and returns the simple class name of the expression result value or `null` if the expression result value is null.

This example statement returns the simple class name of the value of the dynamic property `prop` of events in stream `MyStream`, or a `null` value if the property is not found for an event or the property value itself is `null`:

```
select typeof(prop?) from MyStream
```

When using subclasses or interface implementations as event classes or when using Map-event type inheritance, the function returns the event type name provided when the class or Map-type event was registered, or if the event type was not registered, the function returns the fully-qualified class name.

9.2. Aggregation Functions

Aggregation functions are stateful and consider sets of events or value points. The `group by` clause is often used in conjunction with aggregation functions to group the result-set by one or more columns.

The EPL language extends the standard SQL aggregation functions by allowing filters and by further useful aggregation functions that can track a data window or compute event rates, for example. Your application may also add its own aggregation function as *Section 17.5, "Aggregation Function"* describes.

Aggregation values are always computed incrementally: Insert and remove streams result in aggregation value changes. The exceptions are on-demand queries and joins when using the `unidirectional` keyword. Aggregation functions are optimized to retain the minimal information necessary to compute the aggregated result, and to share aggregation state between eligible other aggregation functions in the same statement so that same-kind aggregation state is never held multiple times unless required.

Most aggregation functions can also be used with unbound streams when no data window is specified. A few aggregation functions require a data window or named window as documented below.

9.2.1. SQL-Standard Functions

The SQL-standard aggregation functions are shown in below table.

Table 9.2. Syntax and results of SQL-standard aggregation functions

Aggregate Function	Result
avedev([all|distinct] *expression* [, *filter_expr*])	Mean deviation of the (distinct) values in the expression, returning a value of `double` type. The optional filter expression limits the values considered for computing the mean deviation.
avg([all|distinct] *expression* [, *filter_expr*])	Average of the (distinct) values in the expression, returning a value of `double` type. The optional filter expression limits the values considered for computing the average.
count([all|distinct] *expression* [, *filter_expr*])	Number of the (distinct) non-null values in the expression, returning a value of `long` type.

Aggregate Function	Result
	The optional filter expression limits the values considered for the count.
count(* [, *filter_expr*])	Number of events, returning a value of `long` type. The optional filter expression limits the values considered for the count.

max([all|distinct] *expression*)
 	
fmax([all|distinct] *expression*, *filter_expr*)
	Highest (distinct) value in the expression, returning a value of the same type as the expression itself returns.
	
	Use `fmax` to provide a filter expression that limits the values considered for computing the maximum.
	
	Consider using `maxby` instead if return values must include additional properties.

median([all|distinct] *expression* [, *filter_expr*])	Median (distinct) value in the expression, returning a value of `double` type. Double Not-a-Number (NaN) values are ignored in the median computation. The optional filter expression limits the values considered for computing the median.

min([all|distinct] *expression*)
 	
fmin([all|distinct] *expression*, *filter_expr*)
 	
Consider using `minby` instead if return values must include additional properties.
	Lowest (distinct) value in the expression, returning a value of the same type as the expression itself returns.
	
	Use `fmin` to provide a filter expression that limits the values considered for computing the maximum.

stddev([all|distinct] *expression* [, *filter_expr*])	Standard deviation of the (distinct) values in the expression, returning a value of `double` type. The optional filter expression limits the values considered for computing the standard deviation.
sum([all|distinct] *expression* [, *filter_expr*])	Totals the (distinct) values in the expression, returning a value of `long, double, float or integer` type depending on the expression. The optional filter expression limits the values considered for computing the total.

If your application provides double-type values to an aggregation function, avoid using Not-a-Number (NaN) and infinity. Also when using double-type values, round-off errors (or rounding errors) may occur due to double-type precision. Consider rounding your result value to the desired precision.

Each of the aggregation functions above takes an optional filter expression as a parameter. The filter expression must return a boolean-type value and applies to the events considered for the aggregation. If a filter expression is provided, then only if the filter expression returns a value of true does the engine update the aggregation for that event or combination of events.

Consider the following example, which computes the quantity fraction of buy orders among all orders:

```
select sum(quantity, side='buy') / sum(quantity) as buy_fraction from Orders
```

Use the `fmin` and `fmax` aggregation functions instead of the `min` and `max` aggregation functions when providing a filter expression (the `min` and `max` functions are also single-row functions).

The next example computes the minimum quantity for buy orders and a separate minimum quantity for sell orders:

```
select fmin(quantity, side='buy'), fmin(quantity, side = 'sell') from Orders
```

9.2.2. Event Aggregation Functions

The event aggregation functions return one or more events or event properties. When used with `group by` the event aggregation functions return one or more events or event properties per group.

The `sorted` and the `window` event aggregation functions require that a data window or named window is declared for the applicable stream. They cannot be used on unbound streams.

The below table summarizes the event aggregation functions available:

Table 9.3. Event Aggregation Functions

Function	Result
first(...)	Returns the first event or an event property value of the first event. *Section 9.2.2.1, “First Aggregation Function”*.
last(...)	Returns the last event or an event property value of the last event. *Section 9.2.2.2, “Last Aggregation Function”*.
maxby(criteria)	Returns the event with the highest sorted value according to criteria expressions. *Section 9.2.2.3, “Maxby Aggregation Function”*.

Function	Result
maxbyever(criteria)	Returns the event with the highest sorted value, ever, according to criteria expressions. *Section 9.2.2.4, "Maxbyever Aggregation Function"*.
minby(criteria)	Returns the event with the lowest sorted value according to criteria expressions. *Section 9.2.2.5, "Minby Aggregation Function"*.
minbyever(criteria)	Returns the event with the lowest sorted value, ever, according to criteria expressions. *Section 9.2.2.6, "Minbyever Aggregation Function"*.
sorted(criteria)	Returns events sorted according to criteria expressions. *Section 9.2.2.7, "Sorted Aggregation Function"*.
window(...)	Returns all events or all event's property values. *Section 9.2.2.8, "Window Aggregation Function"*.

In connection with named windows, event aggregation functions can also be used in `on-select`, selects with named window in the `from` clause, subqueries against named windows and on-demand fire-and-forget queries.

The event aggregation functions are often useful in connection with enumeration methods and they can provide input events for enumeration. Please see *Chapter 10, EPL Reference: Enumeration Methods* for more information.

When comparing the `last` aggregation function to the `prev` function, the differences are as follows. The `prev` function is not an aggregation function and thereby not sensitive to the presence of `group by`. The `prev` function accesses data window contents directly and respects the sort order of the data window. The `last` aggregation function returns results based on arrival order and tracks data window contents in a separate shared data structure.

When comparing the `first` aggregation function to the `prevtail` function, the differences are as follows. The `prevtail` function is not an aggregation function and thereby not sensitive to the presence of `group by`. The `prevtail` function accesses data window contents directly and respects the sort order of the data window. The `first` aggregation function returns results based on arrival order and tracks data window contents in a separate shared data structure.

When comparing the `window` aggregation function to the `prevwindow` function, the differences are as follows. The `prevwindow` function is not an aggregation function and thereby not sensitive to the presence of `group by`. The `prevwindow` function accesses data window contents directly and respects the sort order of the data window. The `window` aggregation function returns results based on arrival order and tracks data window contents in a separate shared data structure.

When comparing the `count` aggregation function to the `prevcount` function, the differences are as follows. The `prevcount` function is not an aggregation function and thereby not sensitive to the presence of `group by`.

When comparing the `last` aggregation function to the `nth` aggregation function, the differences are as follows. The `nth` aggregation function does not consider out-of-order deletes (for example with on-delete and sorted windows) and does not revert to the prior expression value when the last event or nth-event was deleted from a data window. The `last` aggregation function tracks the data window and reflects out-of-order deletes.

From an implementation perspective, the `first`, `last` and `window` aggregation functions share a common data structure for each stream. The `sorted`, `minby` and `maxby` aggregation functions share a common data structure for each stream.

9.2.2.1. `First` Aggregation Function

The synopsis for the `first` aggregation function is:

```
first(*|stream.*|value_expression [, index_expression])
```

The `first` aggregation function returns properties of the very first event. When used with `group by`, it returns properties of the first event for each group. When specifying an index expression, the function returns properties of the Nth-subsequent event to the first event, all according to order of arrival.

The first parameter to the function is required and defines the event properties or expression result to return. The second parameter is an optional *index_expression* that must return an integer value used as an index to evaluate the Nth-subsequent event to the first event.

You may specify the wildcard (`*`) character in which case the function returns the underlying event of the single selected stream. When selecting a single stream you may specify no parameter instead of wildcard. For joins and subqueries you must use the stream wildcard syntax below.

You may specify the stream name and wildcard (`*`) character in the *stream*`.*` syntax. This returns the underlying event for the specified stream.

You may specify a *value_expression* to evaluate for the first event. The value expression may not select properties from multiple streams.

The *index_expression* is optional. If no index expression is provided, the function returns the first event. If present, the function evaluates the index expression to determine the value for N, and evaluates the Nth-subsequent event to the first event. A value of zero returns the first event and a value of 1 returns the event subsequent to the first event. You may not specify event properties in the index expression.

The function returns `null` if there are no events or when the index is larger than the number of events held. When used with `group by`, it returns `null` if there are no events for that group or when the index is larger than the number of events held for that group.

To explain, consider the statement below which selects the underlying event of the first sensor event held by the length window of 2 events.

```
select first(*) from SensorEvent.win:length(2)
```

Assume event E1, event E2 and event E3 are of type SensorEvent. When event E1 arrives the statement outputs the underlying event E1. When event E2 arrives the statement again outputs the underlying event E1. When event E3 arrives the statement outputs the underlying event E2, since event E1 has left the data window.

The stream wildcard syntax is useful for joins and subqueries. This example demonstrates a subquery that returns the first SensorEvent when a DoorEvent arrives:

```
select (select first(se.*) from SensorEvent.win:length(2) as se) from DoorEvent
```

The following example shows the use of an index expression. The output value for `f1` is the temperature property value of the first event, the value for `f2` is the temperature property value of the second event:

```
select first(temperature, 0) as f1, first(temperature, 1) as f2
from SensorEvent.win:time(10 sec)
```

You may use dot-syntax to invoke a method on the first event. You may also append a property name using dot-syntax.

9.2.2.2. `Last` Aggregation Function

The synopsis for the `last` aggregation function is:

```
last(*|stream.*|value_expression [, index_expression])
```

The `last` aggregation function returns properties of the very last event. When used with `group by`, it returns properties of the last event for each group. When specifying an index expression, the function returns properties of the Nth-prior event to the last event, all according to order of arrival.

Similar to the `first` aggregation function described above, you may specify the wildcard (`*`) character, no parameter or stream name and wildcard (`*`) character or a *value_expression* to evaluate for the last event.

The *index_expression* is optional. If no index expression is provided, the function returns the last event. If present, the function evaluates the index expression to determine the value for N, and evaluates the Nth-prior event to the last event. A value of zero returns the last event and a value of 1 returns the event prior to the last event. You may not specify event properties in the index expression.

The function returns `null` if there are no events or when the index is larger than the number of events held. When used with `group by`, it returns `null` if there are no events for that group or when the index is larger than the number of events held for that group.

The next statement selects the underlying event of the first and last sensor event held by the time window of 10 seconds:

```
select first(*), last(*) from SensorEvent.win:time(10 sec)
```

The statement shown next selects the last temperature (`f1`) and the prior-to-last temperature (`f1`) of sensor events in the last 10 seconds:

```
select last(temperature, 0) as f1, select last(temperature, 1) as f2
from SensorEvent.win:time(10 sec)
```

9.2.2.3. `Maxby` Aggregation Function

The synopsis for the `maxby` aggregation function is:

```
maxby(sort_criteria_expression [asc/desc][, sort_criteria_expression [asc/
desc]...])
```

The `maxby` aggregation function returns the greatest of all events, compared by using criteria expressions. When used with `group by`, it returns the greatest of all events per group.

This example statement returns the sensor id and the temperature of the sensor event that had the highest temperature among all sensor events:

```
select maxby(temperature).sensorId, maxby(temperature).temperature from
 SensorEvent
```

The next EPL returns the sensor event that had the highest temperature and the sensor event that had the lowest temperature, per zone, among the last 10 seconds of sensor events:

```
select maxby(temperature), minby(temperature) from SensorEvent.win:time(10 sec)
 group by zone
```

Your EPL may specify multiple criteria expressions. If the sort criteria expression is descending please append the `desc` keyword.

The following EPL returns the sensor event with the highest temperature and if there are multiple sensor events with the highest temperature the query returns the sensor event that has the newest timestamp value:

```
select maxby(temperature asc, timestamp desc) from SensorEvent
```

Event properties that are listed in criteria expressions must refer to the same event stream and cannot originate from different event streams.

If your query does not define a data window and does not refer to a named window, the semantics of `maxby` are the same as `maxbyever`.

9.2.2.4. `Maxbyever` Aggregation Function

The synopsis for the `maxbyever` aggregation function is:

```
maxbyever(sort_criteria_expression [asc/desc][, sort_criteria_expression
 [asc/desc]...])
```

The `maxbyever` aggregation function returns the greatest of all events that ever occurred, compared by using criteria expressions. When used with `group by`, it returns the greatest of all events that ever occurred per group.

Compared to the `maxby` aggregation function the `maxbyever` does not consider the data window or named window contents and instead considers all arriving events.

The next EPL computes the difference, per zone, between the maximum temperature considering all events and the maximum temperature considering only the events in the last 10 seconds:

```
select maxby(temperature).temperature - maxbyever(temperature).temperature
from SensorEvent.win:time(10) group by zone
```

9.2.2.5. `Minby` Aggregation Function

The synopsis for the `minby` aggregation function is:

```
minby(sort_criteria_expression [asc/desc][, sort_criteria_expression [asc/
desc]...])
```

Similar to the `maxby` aggregation function, the `minby` aggregation function returns the lowest of all events, compared by using criteria expressions. When used with `group by`, it returns the lowest of all events per group.

Please review the section on `maxby` for more information.

9.2.2.6. `Minbyever` Aggregation Function

Similar to the `maxbyever` aggregation function, the `minbyever` aggregation function returns the lowest of all events that ever occurred, compared by using criteria expressions. When used with `group by`, it returns the lowest of all events per group that ever occured.

Please review the section on `maxbyever` for more information.

9.2.2.7. `Sorted` Aggregation Function

The synopsis for the `sorted` aggregation function is:

```
sorted(sort_criteria_expression [asc/desc][, sort_criteria_expression [asc/
desc]...])
```

The `sorted` aggregation function maintains a list of events sorted according to criteria expressions. When used with `group by`, it maintains a list of events sorted according to criteria expressions per group.

The sample EPL listed next returns events sorted according to temperature ascending for the same zone:

```
select sorted(temperature) from SensorEvent group by zone
```

Your EPL may specify multiple criteria expressions. If the sort criteria expression is descending please append the `desc` keyword.

Enumeration methods can be useful in connection with `sorted` as the function provides the sorted events as input.

This EPL statement finds the sensor event that when sorted according to temperature is the first sensor event for a Friday timestamp among sensor events for the same zone:

```
select sorted(temperature).first(v => timestamp.getDayOfWeek()=6)
from SensorEvent
```

Event properties that are listed in criteria expressions must refer to the same event stream and cannot originate from different event streams.

The use of `sorted` requires that your EPL defines a data window for the stream or utilizes a named window.

9.2.2.8. `Window` Aggregation Function

The synopsis for the `window` aggregation function is:

```
window(*|stream.*|value_expression)
```

The `window` aggregation function returns properties of all events in the data window or named window. When used with `group by`, it returns properties of all events in the data window or named window for each group.

Similar to the `first` aggregation function described above, you may specify the wildcard (`*`) character or stream name and wildcard (`*`) character or a *value_expression* to evaluate for all events.

The function returns `null` if there are no events. When used with `group by`, it returns `null` if there are no events for that group.

The next statement selects the underlying event of all events held by the time window of 10 seconds:

```
select window(*) from SensorEvent.win:time(10 sec)
```

The `window` aggregation function requires that your stream is bound by a data window or a named window. You may not use the `window` aggregation function on unbound streams with the exception of on-demand queries.

This example statement assumes that the `OrderWindow` named window exists. For each event entering or leaving the `OrderWindow` named window it outputs the total amount removing negative amounts:

```
select window(*).where(v => v.amount > 0).aggregate(0d, (r, v) => r + v.amount)
 from OrderWindow
```

9.2.3. Additional Aggregation Functions

Esper provides the following additional aggregation functions beyond those in the SQL standard:

Table 9.4. Syntax and results of EPL aggregation functions

Aggregate Function	Result
firstever(*expression* [, *filter_expr*])	The `firstever` aggregation function returns the very first value ever. When used with `group by` it returns the first value ever for that group. When used with a data window, the result of the function does not change as data points leave a data window. Use the `first` or `prevtail` function to return values relative to a data window.

Aggregate Function	Result
	The optional filter expression limits the values considered for retaining the first-ever value. The next example statement outputs the first price ever for sell orders: `select firstever(price, side='sell') from Order`

lastever(*expression* [, *filter_expr*])	Returns the last value or last value per group, when used with `group by`.
	
	This sample statement outputs the total price, the first price and the last price per symbol for the last 30 seconds of events and every 5 seconds:
	
	`select symbol, sum(price), lastever(price), firstever(price)`
	`from StockTickEvent.win:time(30 sec)`
	`group by symbol`
	`output every 5 sec`
	
	When used with a data window, the result of the function does not change as data points leave a data window (for example when all data points leave the data window). Use the `last` or `prev` function to return values relative to a data window.
	
	The optional filter expression limits the values considered for retaining the last-ever value.
	
	The next example statement outputs the last price (ever) for sell orders:
	
	`select lastever(price, side='sell') from Order`

leaving()	Returns true when any remove stream data has passed, for use in the `having` clause to output only when a data window has filled. The `leaving` aggregation function is useful when you want to trigger output after a data window has a remove stream data point. Use the `output after` syntax as an alternative to output after a time interval.

Aggregate Function	Result

	This sample statement uses `leaving()` to output after the first data point leaves the data window, ignoring the first datapoint:
	
	`select symbol, sum(price)`
	`from StockTickEvent.win:time(30 sec)`
	`having leaving()`

nth(*expression*, *N_index*)	Returns the Nth oldest element; If N=0 returns the most recent value. If N=1 returns the value before the most recent value. If N is larger than the events held in the data window for this group, returns null.
	
	A maximum N historical values are stored, so it can be safely used to compare recent values in large views without incurring excessive overhead.
	
	As compared to the `prev` row function, this aggregation function works within the current `group by` group, see *Section 3.7.2, “Output for Aggregation and Group-By”*.
	
	This statement outputs every 2 seconds the groups that have new data and their last price and the previous-to-last price:
	
	`select symbol, nth(price, 1), last(price)`
	`from StockTickEvent`
	`group by symbol`
	`output last every 2 sec`

rate(*number_of_seconds*)	Returns an event arrival rate per second over the provided number of seconds, computed based on engine time.
	
	Returns null until events fill the number of seconds. Useful with `output snapshot` to output a current rate. This function footprint is for use without a data window onto the stream(s).
	
	A sample statement to output, every 2 seconds, the arrival rate per second considering the last 10 seconds of events is shown here:
	
	`select rate(10) from StockTickEvent`
	`output snapshot every 2 sec`

Aggregate Function	Result
	The aggregation function retains an engine timestamp value for each arriving event.

rate(*timestamp_property*[, *accumulator*])	Returns an event arrival rate over the data window including the last remove stream event. The *timestamp_property* is the name of a long-type property of the event that provides a timestamp value.
	
	The first parameter is a property name or expression providing millisecond timestamp values.
	
	The optional second parameter is a property or expression for computing an accumulation rate: If a value is provided as a second parameter then the accumulation rate for that quantity is returned (e.g. turnover in dollars per second).
	
	This footprint is designed for use with a data window and requires a data window declared onto the stream. Returns null until events start leaving the window.
	
	This sample statement outputs event rate for each group (symbol) with fixed sample size of four events (and considering the last event that left). The `timestamp` event property must be part of the event for this to work.
	
	`select colour, rate(timestamp) as rate`
	`from`
	` StockTickEvent.std:groupwin(symbol).win:length(4)`
	`group by symbol`

Built-in aggregation functions can be disabled via configuration (see *Section 15.4.22.4, "Extended Built-in Aggregation Functions"*). A custom aggregation function of the same name as a built-on function may be registered to override the built-in function.

9.3. User-Defined Functions

A user-defined function (UDF) is a single-row function that 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.

You may register your own function name for the user-defined function. Please see the instructions in *Section 17.3, "Single-Row Function"* for registering a function name for a user-defined single-row function.

A single-row function that has been registered with a function name can simply be referenced as *function_name*(*parameters*) thus EPL statements can be less cluttered as no class name is required. The engine also optimizes evaluation of such registered single-row functions when used in filter predicate expressions as described in *Section 17.3.4, “Single-Row Functions in Filter Predicate Expressions”*.

An example EPL statement that utilizes the `discount` function is shown next (assuming that function has been registered).

```
select discount(quantity, price) from OrderEvent
```

When selecting from a single stream, use the wildcard (`*`) character to pass the underlying event:

```
select discount(*) from OrderEvent
```

Alternatively use the stream alias or EPL pattern tag to pass an event:

```
select discount(oe) from OrderEvent as oe
```

User-defined functions can be also be invoked on instances of an event: Please see *Section 5.4.5, “Using the Stream Name”* to invoke event instance methods on a named stream.

Note that user-defined functions (not single-row functions) are candidate for caching their return result if the parameters passed are constants and they are not used chained. Please see below for details and configuration.

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 8.5, “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 15.4.6, “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]
```

User-defined functions may also be chained: If a user-defined function returns an object then the object can itself be the target of the next function call and so on.

Assume that there is a `calculator` function in the `MyLib` class that returns a class which provides the `search` method taking two parameters. The EPL that takes the result of the `calculator` function and that calls the `search` method on the result and returns its return value is shown below:

```
select MyLib.calculator().search(zonevariable, zone) from RFIDEvent]
```

A user-defined function should be implemented thread-safe.

9.3.1. Event Type Conversion via User-Defined Function

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, the static method should have the following footprint:

```
public static MyEvent convert(MyOtherEvent otherEvent)
```

9.3.2. User-Defined Function Result Cache

For user-defined functions that take no parameters or only constants as parameters the engine automatically caches the return result of the function, and invokes the function only once. This is beneficial to performance if your function indeed returns the same result for the same input parameters.

You may disable caching of return values of user-defined functions via configuration as described in *Section 15.4.22.3, “User-Defined Function or Static Method Cache”*.

9.3.3. Parameter Matching

EPL follows Java standards in terms of widening, performing widening automatically in cases where widening type conversion is allowed without loss of precision, for both boxed and primitive types.

When user-defined functions are overloaded, the function with the best match is selected based on how well the arguments to a function can match up with the parameters, giving preference to the function that requires the least number of widening conversions.

Boxing and unboxing of arrays is not supported in UDF as it is not supported in Java. For example, an array of `Integer` and an array of `int` are not compatible types.

When passing the event or underlying event to your method, either declare the parameter to take `EventBean` (i.e. `myfunc(EventBean event)`) or as the underlying event type (i.e. `myfunc(OrderEvent event)`).

When using `{}` array syntax in EPL, the resulting type is always a boxed type: "`{1, 2}`" is an array of `Integer` (and not `int` since it may contain null values), "`{1.0, 2d}`" is an array of `Double` and "`{'A', "B"}`" is an array of `String`, while "`{1, "B", 2.0}`" is an array of `Object` (`Object[]`).

9.3.4. Receiving a Context Object

Esper can pass an object containing contextual information such as statement name, function name, engine URI and context partition id to your method. The container for this information is `EPLMethodInvocationContext` in package `com.espertech.esper.client.hook`. Please declare your method to take `EPLMethodInvocationContext` as the last parameter. The engine then passes the information along.

A sample method footprint and EPL are shown below:

```
public static double computeSomething(double number, EPLMethodInvocationContext
 context) {...}
```

```
select MyLib.computeSomething(10) from MyEvent
```

9.4. Select-Clause `transpose` Function

The `transpose` function is only valid in the select-clause and indicates that the result of the parameter expression should become the underlying event object of the output event.

The `transpose` function takes a single expression as a parameter. The result object of the parameter expression is subject to transposing as described below.

The function can be useful with `insert into` to allow an object returned by an expression to become the event itself in the output stream.

Any expression returning a Java object can be used with the `transpose` function. Typical examples for expressions are a static method invocation, the result of an enumeration method, a plug-in single row function or a subquery.

The examples herein assume that a single-row function by name `makeEvent` returns an `OrderEvent` instance (a POJO object, not shown).

The following EPL takes the result object of the invocation of the `makeEvent` method (assumed to be an OrderEvent instance) and returns the OrderEvent instance as the underlying event of the output event:

```
select transpose(makeEvent(oi)) from OrderIndication oi
```

Your select-clause can select additional properties or expressions. In this case the output event underlying object is a pair of the expression result object and the additional properties.

The next EPL also selects the `origin` property of the order indication event. The output event is a pair of the OrderEvent instance and a map containing the property name and value of origin:

```
select origin, transpose(makeEvent(oi)) from OrderIndication oi
```

If the `transpose` function is not a top-level function, i.e. if it occurs within another expression or within any other clause then the select-clause, the function simply returns the expression result of the parameter expression.

9.4.1. Transpose with Insert-Into

You may insert transposed output events into another stream.

If the stream name in the insert-into clause is already associated to an event type, the engine checks whether the event type associated to the stream name provided in the insert-into clause matches the event type associated to the object returned by the expression. If the stream name in the insert-into clause is not already associated to an existing event type the engine associates a new event type using the stream name provided in the insert-into clause.

For example, the next statement associates the stream name `OrderEvent` with the class. Alternatively this association can be achieved via static or runtime configuration API:

```
create schema OrderEvent as com.mycompany.OrderEvent
```

An EPL statement can insert into the `OrderEvent` stream the `OrderEvent` instance returned by the `makeEvent` method, as follows:

```
insert into OrderEvent select transpose(makeEvent(oi)) from OrderIndication oi
```

It is not valid to select additional properties or expressions in this case, as they would not be part of the output event. The following is not valid:

```
// not valid
insert   into   OrderEvent   select   origin,   transpose(makeEvent(oi))   from
 OrderIndication oi
```

Chapter 10. EPL Reference: Enumeration Methods

10.1. Overview

EPL provides enumeration methods that work with lambda expressions to perform common tasks on subquery results, named windows, event properties or inputs that are or can be projected to a collection of events, scalar values or objects.

A lambda expression is an anonymous expression. Lambda expressions are useful for encapsulating user-defined expressions that are applied to each element in a collection. This section discusses built-in enumeration methods and their lambda expression parameters.

Lambda expressions use the lambda operator =>, which is read as "goes to". The left side of the lambda operator specifies the lambda expression input parameter(s) (if any) and the right side holds the expression. The lambda expression x => x * x is read "x goes to x times x.". Lambda expressions are also used for expression declaration as discussed in *Section 5.2.8, “Expression Declaration”*.

When writing lambdas, you do not have to specify a type for the input parameter(s) or output result(s) because the engine can infer all types based on the input and the expression body. So if you are querying an RFIDEvent, for example, then the input variable is inferred to be an RFIDEvent event, which means you have access to its properties and methods.

The term *element* in respect to enumeration methods means a single event, scalar value or object in a collection that is the input to an enumeraton method. The term *collection* means a sequence or group of elements.

The below table summarizes the built-in enumeration methods available:

Table 10.1. Enumeration Methods

Method	Result
aggregate(seed, accumulator lambda)	Aggregate elements by using seed as an initial accumulator value and applying an accumulator expression. *Section 10.6.1, “Aggregate”*.
allof(predicate lambda)	Return true when all elements satisfy a condition. *Section 10.6.2, “AllOf”*.
anyof(predicate lambda)	Return true when any element satisfies a condition. *Section 10.6.3, “AnyOf”*.
average()	Computes the average of values obtained from numeric elements.

Method	Result
	Section 10.6.4, “Average”.
average(projection lambda)	Computes the average of values obtained from elements by invoking a projection expression on each element. *Section 10.6.4, “Average”*.
countof()	Returns the number of elements. *Section 10.6.5, “CountOf”*.
countof(predicate lambda)	Returns the number of elements that satisfy a condition. *Section 10.6.5, “CountOf”*.
except(source)	Produces the set difference of the two collections. *Section 10.6.6, “Except”*.
firstof()	Returns the first element. *Section 10.6.7, “FirstOf”*.
firstof(predicate lambda)	Returns the first element that satisfies a condition. *Section 10.6.7, “FirstOf”*.
groupby(key-selector lambda)	Groups the elements according to a specified key-selector expression. *Section 10.6.8, “GroupBy”*.
groupby(key-selector lambda, value-selector lambda)	Groups the elements according to a key-selector expression mapping each element to a value according to a value-selector. *Section 10.6.8, “GroupBy”*.
intersect(source)	Produces the set intersection of the two collections. *Section 10.6.9, “Intersect”*.
lastof()	Returns the last element. *Section 10.6.10, “LastOf”*.
lastof(predicate lambda)	Returns the last element that satisfies a condition. *Section 10.6.10, “LastOf”*.
leastFrequent()	Returns the least frequent value among a collection of values. *Section 10.6.11, “LeastFrequent”*.
leastFrequent(transform lambda)	Returns the least frequent value returned by the transform expression when applied to each element. *Section 10.6.11, “LeastFrequent”*.

Method	Result
max()	Returns the maximum value among a collection of elements. *Section 10.6.12, “Max”*.
max(value-selector lambda)	Returns the maximum value returned by the value-selector expression when applied to each element. *Section 10.6.12, “Max”*.
maxby(value-selector lambda)	Returns the element that provides the maximum value returned by the value-selector expression when applied to each element. *Section 10.6.13, “MaxBy”*.
min()	Returns the minimum value among a collection of elements. *Section 10.6.12, “Max”*.
min(value-selector lambda)	Returns the minimum value returned by the value-selector expression when applied to each element. *Section 10.6.14, “Min”*.
minby(value-selector lambda)	Returns the element that provides the minimum value returned by the value-selector expression when applied to each element.. *Section 10.6.15, “MinBy”*.
mostFrequent()	Returns the most frequent value among a collection of values. *Section 10.6.16, “MostFrequent”*.
mostFrequent(transform lambda)	Returns the most frequent value returned by the transform expression when applied to each element. *Section 10.6.16, “MostFrequent”*.
orderBy()	Sorts the elements in ascending order. *Section 10.6.17, “OrderBy and OrderByDesc”*.
orderBy(key-selector lambda)	Sorts the elements in ascending order according to a key. *Section 10.6.17, “OrderBy and OrderByDesc”*.
orderByDesc()	Sorts the elements in descending order. *Section 10.6.17, “OrderBy and OrderByDesc”*.
orderByDesc(key-selector lambda)	Sorts the elements in descending order according to a key. *Section 10.6.17, “OrderBy and OrderByDesc”*.
reverse	Reverses the order of elements.

Method	Result
	Section 10.6.18, “Reverse”.
selectFrom(transform lambda)	Transforms each element resulting in a collection of transformed elements. *Section 10.6.19, “SelectFrom”*.
sequenceEqual(second)	Determines whether two collections are equal by comparing each element (`equals` semantics apply). *Section 10.6.20, “SequenceEqual”*.
sumOf()	Computes the sum from a collection of numeric elements. *Section 10.6.21, “SumOf”*.
sumOf(projection lambda)	Computes the sum by invoking a projection expression on each element. *Section 10.6.21, “SumOf”*.
take(numElements)	Returns a specified number of contiguous elements from the start. *Section 10.6.22, “Take”*.
takeLast(numElements)	Returns a specified number of contiguous elements from the end. *Section 10.6.23, “TakeLast”*.
takeWhile(predicate lambda)	Returns elements from the start as long as a specified condition is true. *Section 10.6.24, “TakeWhile”*.
takeWhile((predicate, index) lambda)	Returns elements from the start as long as a specified condition is true, allowing each element's index to be used in the logic of the predicate expression. *Section 10.6.24, “TakeWhile”*.
takeWhileLast(predicate)	Returns elements from the end as long as a specified condition is true. *Section 10.6.25, “TakeWhileLast”*.
takeWhileLast((predicate,index) lambda)	Returns elements from the end as long as a specified condition is true, allowing each element's index to be used in the logic of the predicate expression. *Section 10.6.25, “TakeWhileLast”*.

Method	Result
toMap(key-selector lambda, value-selector lambda)	Returns a Map according to specified key selector and value-selector expressions. *Section 10.6.26, “ToMap”*.
union(source)	Forms a union of the input elements with source elements. *Section 10.6.27, “Union”*.
where(predicate lambda)	Filters elements based on a predicate. *Section 10.6.28, “Where”*.
where((predicate,index) lambda)	Filters elements based on a predicate, allowing each element's index to be used in the logic of the predicate expression. *Section 10.6.28, “Where”*.

10.2. Example Events

The examples in this section come out of the domain of location report (aka. RFID, asset tracking etc.) processing:

1. The `Item` event is a report of the location of a certain item. An item can be either a piece of luggage or a passenger.
2. The `LocationReport` event contains a list of `Item` items for which it reports location.
3. The `Zone` event describes areas that items may move through.

The examples use example single-row functions for computing the distance (`distance`) and for determining if a location falls within a rectangle (`inrect`) that are not provided by the EPL language. These example UDF functions are not enumeration methods and are used in EPL statements to provide a sensible example.

The `Item` event contains an `assetId` id, a (x,y) `location`, a `luggage` flag to indicate whether the item represents a luggage (true) or passenger (false), and the `assetIdPassenger` that holds the asset id of the associated passenger when the item is a piece of luggage.

The `Item` event is defined as follows (access methods not shown for brevity):

```
public class Item {
  String assetId;              // passenger or luggage asset id
  Location location;           // (x,y) location
  boolean luggage;             // true if this item is a luggage piece
   String assetIdPassenger;      // if the item is luggage, contains passenger
 associated
...
```

The `LocationReport` event contains a list of `Item` items for which it reports events.

The `LocationReport` event is defined as follows:

```
public class LocationReport {
  List<Item> items;
...
```

The `Zone` event contains a zone `name` and (x1, y1, x2, y2) `rectangle`.

The `Zone` event is defined as follows:

```
public class Zone {
  String name;
  Rectangle rectangle;
...
```

The `Location` object is a nested object to `Item` and provides the current (x,y) location:

```
public class Location {
  int x;
  int y;
...
```

The `Rectangle` object is a nested object to `Zone` and provides a zone rectangle(x1,y1,x2,y2):

```
public class Rectangle {
  int x1;
  int y1;
  int x2;
  int y2;
...
```

10.3. How to Use

10.3.1. Syntax

The syntax for enumeration methods is the same syntax as for any chained invocation:

```
input_coll.enum_method_name( [method_parameter [, method_parameter [,...]]])
   .[ [enum_method_name(...) [...]] | property_name]
```

Following the *input_coll* input collection (options outlined below), is the `.` (dot) operator and the *enum_method_name* enumeration method name. It follows in parenthesis a comma-separated list of method parameter expressions. Additional enumeration methods can be chained thereafter. An event property name can follow for those enumeration methods returning an event-typed (non-scalar) element.

If the method parameter is a lambda expression with a single lambda-parameter, specify the lambda-parameter name followed by the => lambda operator and followed by the expression. The synopsis for use with a single lambda-parameter is:

```
method_parameter: lamda_param => lamda_expression
```

If the method parameter is a lambda expression with two or more lambda-parameters, specify the lambda parameter names in parenthesis followed by the => lambda operator followed by the expression. The synopsis for use with multiple lambda-parameters is:

```
method_parameter: (lamda_param [,lamda_param [,...]]) => lamda_expression
```

Generally for lambda expressions, the engine applies the lambda expression to each element in the input collection. The expression yields a result that, depending on the particular enumeration method, is used for aggregation, as a filter or for output, for example.

10.3.2. Introductory Examples

Let's look at an EPL statement that employs the `where` enumeration method and a lambda expression. This example returns items that have a (x, y) location of (0, 0):

```
select items.where(i => i.location.x = 0 and i.location.y = 0) as zeroloc
from LocationReport
```

As enumeration methods can be chained, this selection is equivalent:

```
select items.where(i => i.location.x = 0).where(i => i.location.y = 0) as zeroloc
from LocationReport
```

According to above statement the engine outputs in field `zeroloc` a collection of `Item` objects matching the condition.

The `where` enumeration method has a second version that has two lambda-parameters. The second parameter is the name of the index property which represents the current index of the element within the collection.

This sample query returns a collection that consists of the first 3 items. This sample query does not use the `item` lambda parameter:

```
select items.where( (item, indexElement) => indexElement < 3) as firstThreeItems
from LocationReport
```

10.3.3. Input, Output and Limitations

It is not necessary to use classes for event representation. The example above applies the same to Object-array, Map or XML underlying events.

For most enumeration methods the input can be any collection of events, scalar values or objects. For some enumeration methods limitations apply that are documented below. For example, the `sumOf` enumeration method requires a collection of numeric scalar values if used without parameters. If the input to `sumOf` is a collection of events or scalar values the enumeration method requires a lambda expression as parameter that yields the numeric value to use to compute the sum.

Many examples of this section operate on the collection returned by the event property `items` in the `LocationReport` event class. There are many other inputs yielding collections as listed below. Most examples herein use an event property as a input simply because the example can thus be brief and does not need to refer to a subquery or named window or other concept.

For enumeration methods that return a collection, for example `where` and `orderBy`, the engine outputs an implementation of the `Collection` interface that contains the selected value(s). The collection returned must be considered read-only. As Java does not allow resettable iterators, the `Collection` interface allows more flexibility to query size and navigate among collection elements. We recommend against down-casting a collection returned by the engine to a more specific subclass of the `Collection` interface.

For enumeration methods that return an element, for example `firstOf`, `lastOf`, `minBy` and `maxBy` the engine outputs the scalar value or the underlying event if operating on events. You may add an event property name after the enumeration method to return a property value.

Enumeration methods generally retain the order of elements provided by the collection.

The following restrictions apply to enumeration methods:

1. Enumeration methods returning a collection return a read-only implementation of the `Collection` interface. You may not use any of the write-methods such as `add` or `remove` on a result collection.

10.4. Inputs

The input of data for built-in enumeration methods is a collection of scalar values, events or other objects. Input can originate from any of the following:

1. A subquery.
2. A named window.

3. A property of an event that is itself a collection of events or classes, i.e. an indexed property.
4. Any of the event aggregation functions (`window`, `first`, `last`, `sorted`, `maxby`, `minby`, `maxbyever`, `minbyever`).
5. The special `prevwindow`, `prev` and `prevtail` single-row functions.
6. A plug-in single-row function, a user-defined function or an enum type.
7. A declared expression.
8. Another enumeration method that returns a collection.
9. An array returned by the `{}` array operator.
10A collection or array returned by a method call on an event.
11A variable. Usually variables declared as an array.

10.4.1. Subquery Results

Subqueries can return the rows of another stream's data window or rows from a named window. By providing a where-clause the rows returned by a subquery can be correlated to data provided by stream(s) in the from-clause. See *Section 5.11, "Subqueries"*.

A subquery that selects `(*)` wildcard provides a collection of events as input. A subquery that selects a single value expression provides a collection of scalar values as input. Subqueries that selects multiple value expressions are not allowed as input to enumeration methods.

The following example uses a subquery to retrieve all zones for each location report item where the location falls within the rectangle of the zone. Please see a description of example events and functions above.

```
select assetId,
  (select * from Zone.std:unique(name)).where(z => inrect(z.rectangle, location))
 as zones
from Item
```

You may place the subquery in an expression declaration to reuse the subquery in multiple places of the same EPL statement.

This sample EPL declares the same query as above in an expression declaration:

```
expression myquery {itm =>
     (select  *  from  Zone.std:unique(name)).where(z  =>  inrect(z.rectangle,
 itm.location))
}
select assetId, myquery(item) as subq,
    myquery(item).where(z => z.zone = 'Z01') as assetItem
from Item as item
```

The above query also demonstrates how an enumeration method, in the example the `where`-method, can be run across the results returned by a subquery in an expression declaration.

Place a single column in the subquery select-clause to provide a collection of scalar values as input.

The next example selects all names of zones and orders the names returning an order collection of string names every 30 seconds:

```
select (select name from Zone.std:unique(name)).orderBy() as orderedZones
from pattern[every timer:interval(30)]
```

Note that the engine can cache intermediate results thereby is not forced to re-evaluate the subquery for each occurrence in the `select`-clause.

10.4.2. Named Window

Named windows are globally-visible data windows. See *Section 5.15, "Creating and Using Named Windows"*.

You may specify the named window name as input for an enumeration method and can optionally provide a correlation where-clause. The syntax is equivalent to a sub-query against a named window but much shorter.

Synopsis:

```
named-window-name[(correlation-expression)].enum-method-name(...)
```

When selecting all events in a named window you do not need the *correlation-expression*. To select a subset of data in the named window, specify a *correlation-expression*. From the perspective of best runtime performance, a correlation expression is preferred to reduce the number of rows returned.

The following example first declares a named window to hold the last zone event per zone name:

```
create window ZoneWindow.std:unique(name) as Zone
```

Then we create a statement to insert zone events that arrive to the named window:

```
insert into ZoneWindow select * from Zone
```

Finally this statement queries the named window to retrieve all zones for each location report item where the location falls within the rectangle of the zone:

```
select ZoneWindow.where(z => inrect(z.rectangle, location)) as zones from Item
```

If you have a filter or correlation expression, append the expression to the named window name and place in parenthesis.

This slightly modified query is the example above except that it adds a filter expression such that only zones with name Z1, Z2 or Z3 are considered:

```
select ZoneWindow(name in ('Z1', 'Z2', 'Z3')).where(z => inrect(z.rectangle,
 location)) as zones
from Item
```

You may prefix property names provided by the named window with the name to disambiguate property names.

This sample query prefixed the `name` property and returns the count of matching zones:

```
select ZoneWindow(ZoneWindow.name in ('Z1', 'Z2', 'Z3')).countof()) as zoneCount
from Item
```

The engine internally interprets the shortcut syntax and creates a subquery from it. Thus all indexing and query planning for subqueries against named windows apply here as well.

10.4.3. Event Property

Indexed event properties are event properties that are a collection, array or iterable of scalar values or objects.

The `LocationReport` event from the example contains a list of `Item` events. Any indexed property (list, array, collection, iterable) is eligible for use as input to an enumeration method. If the indexed property contains non-scalar objects the objects are treated as events and can be used as input to enumeration methods as a collection of events.

The next sample query returns items that are less then 20 units away from the center, taking the `items` event property provided by each `LocationReport` event as input:

```
select items.where(p => distance(0, 0, p.location.x, p.location.y) < 20) as
 centeritems
from LocationReport
```

The following example consists of two statements: The first statement declares an a new event type and the second statement invokes the `sequenceEqual` method to compare sequences contained in two properties of the same event:

```
create schema MyEvent (seqone String[], seqtwo String[])
```

```
select seqone.sequenceEqual(seqtwo) from MyEvent
```

10.4.4. Event Aggregation Function

Event aggregation functions return an event or multiple events. They are aggregation functions and as such sensitive to the presence of `group by`. See *Section 9.2.2, “Event Aggregation Functions”*.

You can use `window`, `first` or `last` event aggregation functions as input to an enumeration method. Specify the `*` wildcard as the parameter to the event aggregation function to provide a collection of events as input. Or specify a property name as the parameter to event aggregation function to provide a collection of scalar values as input.

You can use the `sorted`, `maxby`, `minby`, `maxbyever` or `minbyever` event aggregation functions as input to an enumeration method. Specify one or more criteria expressions that provide the sort order as parameters to the event aggregation function.

In this example query the `window(*)` aggregation function returns the last 10 seconds of item location reports for the same asset id as the incoming event. Among that last 10 seconds of events for the same asset id, the enumeration method returns those item location reports where the distance to center is less then 20, for each arriving Item event.

Sample query:

```
select window(*).where(p => distance(0, 0, p.location.x, p.location.y) < 20) as
 centeritems
from Item(type='P').win:time(10) group by assetId
```

The next sample query instead selects the asset id property of all events and returns an ordered collection:

```
select window(assetId).orderBy() as orderedAssetIds
from Item.win:time(10) group by assetId
```

The following example outputs the 5 highest prices per symbol among the last 10 seconds of stock ticks:

```
select sorted(price desc).take(5) as highest5PricesPerSymbol
```

```
from StockTick.win:time(10) group by symbol
```

10.4.5. `prev`, `prevwindow` and `prevtail` Single-Row Functions as Input

The `prev`, `prevwindow` and `prevtail` single-row functions allow access into a stream's data window however are not aggregation functions and and as such not sensitive to the presence of `group by`. See *Section 9.1.11, “The Previous-Window Function”*.

When using any of the `prev` single-row functions as input to a built-in enumeration method you can specify the stream name as a parameter to the function or an event property. The input to the enumeration method is a collection of events if you specify the stream name, or a collection of scalar value if you specify an event property.

In this example query the `prevwindow(stream)` single-row function returns the last 10 seconds of item location reports, among which the enumeration method filters those item location reports where the distance to center is less then 20, for each Item event that arrived in the last 10 seconds considering passenger-type Item events only (see filter type = 'P').

Sample query:

```
select prevwindow(items)
    .where(p => distance(0, 0, p.location.x, p.location.y) < 20) as centeritems
from Item(type='P').win:time(10) as items
```

This sample query demonstrates the use of the `prevwindow` function to return a collection of scalar values (collection of asset id) as input to `orderby`:

```
select prevwindow(assetId).orderBy() as orderedAssetIds
from Item.win:time(10) as items
```

10.4.6. Single-Row Function, User-Defined Function and Enum Types

Your single-row or user-defined function can return either an array or any collection that implements either the `Collection` or `Iterable` interface. For arrays, the array component type and for collections, the collection or iterable generic type should be the class providing event properties.

As an example, assume a `ZoneFactory` class exists and a static method `getZones()` returns a list of zones to filter items, for example:

```
public class ZoneFactory {
  public static Iterable<Zone> getZones() {
    List<Zone> zones = new ArrayList<Zone>();
    zones.add(new Zone("Z1", new Rectangle(0, 0, 20, 20)));
    return zones;
  }
}
```

Import the class through runtime or static configuration, or add the method above as a plug-in single-row function.

The following query returns for each Item event all zones that the item belongs to:

```
select ZoneFactory.getZones().where(z => inrect(z.rectangle, item.location)) as
 zones
from Item as item
```

If the class and method were registered as a plug-in single-row function, you can leave the class name off, for example:

```
select getZones().where(z => inrect(z.rectangle, item.location)) as zones
from Item as item
```

Your single-row or user-defined function can also return an array, collection or iterable or scalar values.

For example, the static method `getZoneNames()` returns a list of zone names:

```
public static String[] getZoneNames() {
  return new String[] { "Z1", "Z2"};
}
```

The following query returns zone names every 30 seconds and excludes zone Z1:

```
select getZoneNames().where(z => z != "Z1")
from pattern[every timer:interval(30)]
```

An enum type can also be a useful source for enumerable values.

The following sample Java declares an enum type `EnumOfZones`:

```
public enum EnumOfZones {
  ZONES_OUTSIDE(new String[] {"z1", "z2"}),
  ZONES_INSIDE(new String[] {"z3", "z4"})

  private final String[] zones;

  private EnumOfZones(String[] zones) {
   this.zones = zones;
  }

  public String[] getZones() {
    return zones;
  }
}
```

A sample statement that utilizes the enum type is shown next:

```
select EnumOfZones.ZONES_OUTSIDE.getZones().anyOf(v => v = zone) from Item
```

10.4.7. Declared Expression

A declared expression may return input data for an enumeration method.

The below query declares an expression that returns all passenger location reports among the items in the location report event in a column named `passengerCollection`. The query uses the result returned by the declared expression a second time to filter through the list returning the passenger location report where the asset id is a given value in a column named `passengerP01`.

Sample query:

```
expression passengers {
  lr => lr.items.where(l => l.type='P')
}
select passengers(lr) as passengerCollection,
  passengers(lr).where(x => assetId = 'P01') as passengerP01
from LocationReport lr
```

The engine applies caching techniques to avoid re-evaluating the declared expression multiple times.

10.4.8. Variables

A variable may provide input data for an enumeration method.

This constant of array type carries a list of invalid zones:

```
create constant variable string[] invalid_zones = { 'Z1', 'Z2' };
```

Sample query:

```
select invalid_zones.anyOf(v => v = name) as flagged from Zone
```

10.5. Example

Following the RFID asset tracking example as introduced earlier, this section introduces two use cases solved by enumeration methods.

The first use case requires us to find any luggage that is more then 20 units away from the passenger that the luggage belongs to. The declared expression `lostLuggage` solves this question.

The second question to answer is: For each of such lost luggage what single other passenger is nearest to that luggage. The declared expression `nearestOwner` which uses `lostLuggage` answers this question.

Below is the complete EPL statement (one statement not multiple):

```
// expression to return a collection of lost luggage
expression lostLuggage {
  lr => lr.items.where(l => l.type='L' and
    lr.items.some(p => p.type='P' and p.assetId=l.assetIdPassenger
     and LRUtil.distance(l.location.x, l.location.y, p.location.x, p.location.y)
 > 20))
}

// expression to return all passengers
expression passengers {
  lr => lr.items.where(l => l.type='P')
}

// expression to find the nearest owner
expression nearestOwner {
  lr => lostLuggage(lr).toMap(key => key.assetId,
    value => passengers(lr).minBy(
         p => LRUtil.distance(value.location.x, value.location.y, p.location.x,
 p.location.y))
    )
}

select lostLuggage(lr) as val1, nearestOwner(lr) as val2 from LocationReport lr
```

10.6. Reference

10.6.1. Aggregate

The `aggregate` enumeration method takes an expression providing the initialization value (seed) and an accumulator lambda expression. The return value is the final accumulator value.

Via the `aggregate` method you may perform a calculation over elements. The method initializes the aggregated value by evaluating the expression provided in the first parameter. The method then calls the lambda expression of the second parameter once for each element in the input. The lambda expression receives the last aggregated value and the element from the input. The result of the expression replaces the previous aggregated value and returns the final result after completing all elements.

An expression example with scalar values:

```
{1, 2, 3}.aggregate(0, (result, value) => result + value)  // Returns 6
```

The example below aggregates price of each OrderEvent in the last 10 seconds computing a total price:

```
// Initialization value is zero.
// Aggregate by adding up the price.
select  window(*).aggregate(0,  (result,  order)  =>  result  +  order.price)  as
 totalPrice
from OrderEvent.win:time(10)
```

In the query above, the initialization value is zero, `result` is used for the last aggregated value and `order` denotes the element that the expression adds the value of the price property.

This example aggregation builds a comma-separated list of all asset ids of all items:

```
select items.aggregate('',
   (result, item) => result || (case when result='' then '' else ',' end) ||
 item.assetId) as assets
from LocationReport
```

In above query, the empty string `''` represents the initialization value. The name `result` is used for the last aggregated value and the name `item` is used to denote the element.

The type value returned by the initialization expression must match to the type of value returned by the accumulator lambda expression.

If the input is null the method returns null. If the input is empty the method returns the initialization value.

10.6.2. AllOf

The `allof` enumeration method determines whether all elements satisfy the predicate condition.

The method takes a single parameter: The predicate lambda expression that must yield a Boolean result. The enumeration method applies the lambda expression to each element and if the expression returns true for all elements, the method returns true.

An expression example with scalar values:

```
{1, 2, 3}.allOf(v => v > 0)    // Returns true as all values are > 0
{1, 2, 3}.allOf(v => v > 1)    // Returns false
```

The EPL statement below returns true when all items are within 1000 unit distance of center:

```
select items.allof(i => distance(i.location.x, i.location.y, 0, 0) < 1000) as
 centered
from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns true.

10.6.3. AnyOf

The `anyof` enumeration method determines whether any element satisfies the predicate condition.

The only parameter is the predicate lambda expression that must yield a Boolean result. The enumeration method applies the lambda expression to each element and if the expression returns true for all elements, the method returns true.

An expression example with scalar values:

```
{1, 2, 3}.anyOf(v => v > 0)    // Returns true
{1, 2, 3}.anyOf(v => v > 1)    // Returns true
{1, 2, 3}.anyOf(v => v > 3)    // Returns false
```

The EPL statement below return true when any of the items are within 10 unit distance of center:

```
select items.anyof(i => distance(i.location.x, i.location.y, 0, 0) < 10) as
 centered
```

```
from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns false.

10.6.4. Average

The `average` enumeration method computes the average of scalar values. If passing a projection lambda expression the method computes the average obtained by invoking the projection lambda expression on each element.

The method takes a projection lambda expression yielding a numeric value as a parameter. It applies the lambda expression to each element and computes the average of the result, returning a Double value. A BigDecimal is returned for expressions returning BigInteger or BigDecimal.

An expression example with scalar values:

```
{1, 2, 3}.average()    // Returns 2
```

The EPL statement as shown next computes the average distance from center among all items in the location report event:

```
select  items.average(i  =>  distance(i.location.x,  i.location.y,  0,  0))  as
 avgdistance
from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns double zero or BigDecimal zero. For BigDecimal precision and rounding, please see *Section 15.4.22.6, “Math Context”*.

10.6.5. CountOf

The `countof` enumeration method returns the number of elements, or the number of elements that satisfy a condition.

The enumeration method has two versions: The first version takes no parameters and computes the number of elements. The second version takes a predicate lambda expression that must yield Boolean true or false, and computes the number of elements that satisfy the condition.

An expression example with scalar values:

```
{1, 2, 3}.countOf()    // Returns 3
{1, 2, 3}.countOf(v => v < 2)    // Returns 1
```

The next sample statement counts the number of items:

```
select items.countOf() as cnt from LocationReport
```

This example statement counts the number of items that have a distance to center that is less then 20 units:

```
select items.countOf(i => distance(i.location.x, i.location.y, 0, 0) < 20) as
 cntcenter
from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns integer zero.

10.6.6. Except

The `except` enumeration method forms a set difference of the input elements with the elements that the parameter expression yields.

The enumeration method takes a single parameter that must itself return a collection of events, objects or scalar values. The method returns the elements of the first collection that do not appear in the second collection.

An expression example with scalar values:

```
{1, 2, 3}.except({1})   // Returns {2, 3}
```

The following statement compares the items of the last location report against all items in the previous 10 location reports, and reports for each combination only those items in the current item report that are not also in the location report compared to:

```
select za.items.except(zb.items) as itemsCompared
from LocationReport as za unidirectional, LocationReport.win:length(10) as zb
```

If the input is null the method returns null. For scalar values and objects equals-semantics apply.

10.6.7. FirstOf

The `firstOf` enumeration method returns the first element or the first element that satisfies a condition.

The method has two versions: The first version takes no parameters and returns the first element. The second version takes a predicate lambda expression yielding true or false. It applies the lambda expression to each element and returns the first element for which the expression returns true. The return type is the element itself and not a collection. You may append a property name to return the property value for the first element.

An expression example with scalar values:

```
{1, 2, 3}.firstOf()   // Returns 1
{1, 2, 3}.firstOf(v => v / 2 > 1)   // Returns 3
```

In the following EPL sample the query returns the first item that has a distance to center that is less then 20 units:

```
select items.firstof(i => distance(i.location.x, i.location.y, 0, 0) < 20) as
 firstcenter
from LocationReport
```

The next sample EPL returns the first item's asset id:

```
select items.firstof().assetId as firstAssetId from LocationReport
```

If the input is null, empty or if none of the elements match the condition the method returns null.

10.6.8. GroupBy

The `groupby` enumeration method groups the elements according to a specified key-selector lambda expression. There are two version of the `groupby` method.

The first version of the method takes a key-selector lambda expression and returns a Map of key with each value a list of objects, one for each distinct key that was encountered. The result is a `Map<Object, Collection<Object>>` wherein object is the event underlying object.

The second version of the method takes a key-selector lambda expression and value-selector lambda expression and returns a Map of key with each value a list of values, one for each distinct key that was encountered. The result is a `Map<Object, Collection<Object>>` wherein object is the result of applying the value-selector expression.

The next query filters out all luggage items using a `where` method and then groups by the luggage's passenger asset id. It returns a map of passenger asset id and the collection of luggage items for each passenger:

```
select items.where(type='L').groupby(i => assetIdPassenger) as luggagePerPerson
from LocationReport
```

The query shown below generates a map of item asset id and distance to center:

```
select items.groupby(
       k  =>  assetId,  v  =>  distance(v.location.x,  v.location.y,  0,  0))  as
 distancePerItem
from LocationReport
```

If the input is null the method returns null. Null values as key and value are allowed.

10.6.9. Intersect

The `intersect` enumeration method forms a set intersection of the input elements with the elements that the parameter expression yields.

The enumeration method takes a single parameter that must itself return a collection of events, objects or scalar values. The method returns the elements of the first collection that also appear in the second collection.

An expression example with scalar values:

```
{1, 2, 3}.intersect({2, 3})   // Returns {2, 3}
```

The following statement compares the items of the last location report against all items in the previous 10 location reports, and reports for each combination all items in the current item report that also occur in the other location report:

```
select za.items.intersect(zb.items) as itemsCompared
from LocationReport as za unidirectional, LocationReport.win:length(10) as zb
```

If the input is null the method returns null. For scalar values and objects equals-semantics apply.

10.6.10. LastOf

The `lastOf` enumeration method returns the last element or the last element that satisfies a condition.

The method has two versions: The first version takes no parameters and returns the last element. The second version takes a predicate lambda expression yielding true or false. It applies the lambda expression to each element and returns the last element for which the expression returns true. The return type is the element itself and not a collection. You may append a property name to return the property value for the last element.

An expression example with scalar values:

```
{1, 2, 3}.lastOf()   // Returns 3
{1, 2, 3}.lastOf(v => v < 3)   // Returns 2
```

In the following EPL sample the query returns the last item that has a distance to center that is less then 20 units:

```
select items.lastof(i => distance(i.location.x, i.location.y, 0, 0) < 20) as
 lastcenter
from LocationReport
```

The next sample EPL returns the last item's asset id:

```
select items.lastof().assetId as lastAssetId from LocationReport
```

If the input is null, empty or if none of the elements match the condition the method returns null.

10.6.11. LeastFrequent

The `leastFrequent` enumeration method returns the least frequent value among a collection of values, or the least frequent value after applying a transform expression to each element.

The method has two versions: The first version takes no parameters and returns the least frequent value. The second version takes a transform lambda expression yielding the value to count occurrences for. The method applies the lambda expression to each element and returns the expression result value with the least number of occurrences. The return type is the type of value in the collection or the type of value returned by the transform lambda expression if one was provided.

An expression example with scalar values:

```
{1, 2, 3, 2, 1}.leastFrequent()   // Returns 3
```

The example EPL below returns the least frequent item type, counting the distinct item types among all items for the current LocationReport event:

```
select items.leastFrequent(i => type) as leastFreqType from LocationReport
```

If the input is null or empty the method returns null. The transform expression may also yield null. A null value can be returned as the most frequent value if the most frequent value is null. If multiple values have the same number of occurrences the method returns the first value with the least number of occurrences considering the ordering of the collection.

10.6.12. Max

The `max` enumeration method returns the maximum value among a collection of values.

If no value-selector lambda expression is provided, the method finds the maximum.

If a value-selector lambda expression is provided, the enumeration method invokes a value-selector lambda expression on each element and returns the maximum value. The type of value returned follows the return type of the lambda expression that was provided as parameter.

An expression example with scalar values:

```
{1, 2, 3, 2, 1}.max()   // Returns 3
```

The next query returns the maximum distance of any item from center:

```
select items.max(i => distance(i.location.x, i.location.y, 0, 0)) as maxcenter
from LocationReport
```

The value-selector lambda expression must return a comparable type: Any primitive or boxed type or `Comparable` type is permitted.

If the input is null, empty or if none of the elements when transformed return a non-null value the method returns null.

10.6.13. MaxBy

The `maxBy` enumeration method returns the element that provides the maximum value returned by the value-selector lambda expression when applied to each element.

The enumeration method returns the element itself. You may append an event property name to return a property value of the element.

The next query returns the first item with the maximum distance to center:

```
select  items.maxBy(i  =>  distance(i.location.x,  i.location.y,  0,  0))  as
 maxItemCenter
from LocationReport
```

The next sample returns the type of the item with the largest asset id (string comparison) among all items:

```
select items.maxBy(i => assetId).type as minAssetId from LocationReport
```

The transform expression must return a comparable type: Any primitive or boxed type or `Comparable` type is permitted.

If the input is null, empty or if none of the elements when transformed return a non-null value the method returns null.

10.6.14. Min

The `min` enumeration method returns the minimum value among a collection of values.

If no value-selector lambda expression is provided, the method finds the minimum.

If a value-selector lambda expression is provided, the enumeration method invokes a value-selector lambda expression on each element and returns the minimum value. The type of value returned follows the return type of the lambda expression that was provided as parameter.

An expression example with scalar values:

```
{1, 2, 3, 2, 1}.min()   // Returns 1
```

The next query returns the minimum distance of any item to center:

```
select items.min(i => distance(i.location.x, i.location.y, 0, 0)) as mincenter
from LocationReport
```

The transform expression must return a comparable type: Any primitive or boxed type or `Comparable` type is permitted.

If the input is null, empty or if none of the elements when transformed return a non-null value the method returns null.

10.6.15. MinBy

The `minBy` enumeration method returns the element that provides the minimum value returned by the value-selector lambda expression when applied to each element.

The enumeration method returns the element itself. You may append an event property name to return a property value of the element.

The next query returns the first item with the minimum distance to center:

```
select  items.minBy(i  =>  distance(i.location.x,  i.location.y,  0,  0))  as
 minItemCenter
from LocationReport
```

The next sample returns the type of the item with the smallest asset id (string comparison) among all items:

```
select items.minBy(i => assetId).type as minAssetId from LocationReport
```

The transform expression must return a comparable type: Any primitive or boxed or `Comparable` type is permitted.

If the input is null, empty or if none of the elements when transformed return a non-null value the method returns null.

10.6.16. MostFrequent

The `mostFrequent` enumeration method returns the most frequent value among a collection of values, or the most frequent value after applying a transform expression to each element.

The method has two versions: The first version takes no parameters and returns the most frequent value. The second version takes a transform lambda expression yielding the value to count occurrences for. The method applies the lambda expression to each element and returns the expression result value with the most number of occurrences. The return type is the type of value in the collection or the type of value returned by the transform lambda expression if one was provided.

An expression example with scalar values:

```
{1, 2, 3, 2, 1, 2}.mostFrequent()   // Returns 2
```

The example EPL below returns the least frequent item type, counting the distinct item types among all items for the current LocationReport event:

```
select items.leastFrequent(i => type) as leastFreqType from LocationReport
```

If the input is null or empty the method returns null. The transform expression may also yield null. A null value can be returned as the most frequent value if the most frequent value is null. If multiple values have the same number of occurrences the method returns the first value with the most number of occurrences considering the ordering of the collection.

10.6.17. OrderBy and OrderByDesc

The `orderBy` enumeration method sorts elements in ascending order according to a key. The `orderByDesc` enumeration method sorts elements in descending order according to a key.

The enumeration method takes a single key-selector lambda expression as parameter and orders elements according to the key yielded by the expression. For same-value keys, it maintains the existing order.

An expression example with scalar values:

```
{2, 3, 2, 1}.orderBy()   // Returns {1, 2, 2, 3}
```

This example orders all items from a location report according to their distance from center:

```
select  items.orderBy(i  =>  distance(i.location.x,  i.location.y,  0,  0))  as
 itemsNearFirst,
    items.orderByDesc(i  =>  distance(i.location.x,  i.location.y,  0,  0))  as
 itemsFarFirst
from LocationReport
```

The key-selector lambda expression must return a comparable type: Any primitive or boxed or `Comparable` type is permitted.

If the input is null the method returns null. If the input is empty the method returns an empty collection.

10.6.18. Reverse

The `reverse` enumeration method simply reverses the order of elements returning a collection.

An expression example with scalar values:

```
{2, 3, 2, 1}.reverse()   // Returns {1, 2, 3, 2}
```

The following EPL reverses the items:

```
select items.reverse() as reversedItems from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns an empty collection.

10.6.19. SelectFrom

The `selectFrom` enumeration method transforms each element resulting in a collection of transformed elements.

The enumeration method applies a transformation lambda expression to each element and returns the result of each transformation as a collection. Use the `new` operator to yield multiple values for each element, see *Section 8.13, “The 'new' Keyword”*.

The next EPL query returns a collection of asset ids:

```
select items.selectFrom(i => assetId) as itemAssetIds from LocationReport
```

This sample EPL query evaluates each item and returns the asset id as well as the distance from center for each item:

```
select items.selectFrom(i =>
  new {
    assetId,
    distanceCenter = distance(i.location.x, i.location.y, 0, 0)
  } ) as itemInfo from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns an empty collection.

10.6.20. SequenceEqual

The `sequenceEqual` enumeration method determines whether two collections are equal by comparing each element.

The method enumerates the two source collections in parallel and compares corresponding elements by using the `equals` method to compare. The method takes a single parameter expression that must return a collection containing elements of the same type as the input. The method returns true if the two source sequences are of equal length and their corresponding elements are equal.

An expression example with scalar values:

```
{1, 2, 3}.sequenceEqual({1})   // Returns false
{1, 2, 3}.sequenceEqual({1, 2, 3})   // Returns true
```

The following example compares the asset id of all items to the asset ids returned by a method `ItemUtil.redListed()` which is assumed to return a list of asset id of string type:

```
select items.selectFrom(i => assetId).sequenceEquals(ItemUtil.redListed()) from
 LocationReport
```

If the input is null the method returns null.

10.6.21. SumOf

The `sumOf` enumeration method computes the sum. If a projection lambda expression is provided, the method invokes the projection lambda expression on each element and computes the sum on each returned value.

The projection lambda expression should yield a numeric value, BigDecimal or BigInteger value. Depending on the type returned by the projection lambda expression the method returns either Integer, Long, Double, BigDecimal or BigInteger.

An expression example with scalar values:

```
{1, 2, 3}.sumOf()   // Returns 6
```

The following example computes the sum of the distance of each item to center:

```
select items.sum(i => distance(i.location.x, i.location.y, 0, 0) as
 totalAllDistances
from LocationReport
```

If the input is null or empty the method returns null.

10.6.22. Take

The `take` enumeration method returns a specified number of contiguous elements from the start.

The enumeration method takes a single size (non-lambda) expression that returns an Integer value.

An expression example with scalar values:

```
{1, 2, 3}.take(2)   // Returns {1, 2}
```

The following example returns the first 5 items:

```
select items.take(5) as first5Items from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns an empty collection.

10.6.23. TakeLast

The `takeLast` enumeration method returns a specified number of contiguous elements from the end.

The enumeration method takes a single size (non-lambda) expression that returns an Integer value.

An expression example with scalar values:

```
{1, 2, 3}.takeLast(2)   // Returns {2, 3}
```

The following example returns the last 5 items:

```
select items.takeLast(5) as last5Items from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns an empty collection.

10.6.24. TakeWhile

The `takeWhile` enumeration method returns elements from the start as long as a specified condition is true.

The enumeration method has two versions. The first version takes a predicate lambda expression and the second version takes a predicate lambda expression and index for use within the predicate expression. Both versions return elements from the start as long as the specified condition is true.

An expression example with scalar values:

```
{1, 2, 3}.takeWhile(v => v < 3)   // Returns {1, 2}
{1, 2, 3}.takeWhile((v,ind) => ind > 2)   // Returns {1, 2}
{1, 2, -1, 4, 5, 6}.takeWhile((v,ind) => ind < 5 and v > 0)  // Returns {1,
 2} (Take while index<5 amd value>0)
```

This example selects all items from a location report in the order provided until the first item that has a distance to center greater then 20 units:

```
select items.takeWhile(i => distance(i.location.x, i.location.y, 0, 0) < 20)
from LocationReport
```

The second version of the `where` represents the index of the input element starting at zero for the first element.

The next example is similar to the query above but also limits the result to the first 10 items:

```
select items.takeWhile((i, ind) => distance(i.location.x, i.location.y, 0, 0)
 < 20) and ind < 10)
from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns an empty collection.

10.6.25. TakeWhileLast

The `takeWhileLast` enumeration method returns elements from the end as long as a specified condition is true.

The enumeration method has two versions. The first version takes a predicate lambda expression and the second version takes a predicate lambda expression and index for use within the predicate expression. Both versions return elements from the end as long as the specified condition is true.

An expression example with scalar values:

```
{1, 2, 3}.takeWhileLast(v => v < 3)   // Returns {} (empty collection)
{1, 2, 3}.takeWhileLast(v => v > 1)   // Returns {2, 3}
{1, 2, 3}.takeWhileLast((v,ind) => ind > 2)   // Returns {2, 3}
{1, 2, -1, 4, 5, 6}.takeWhileLast((v,ind) => ind < 5 and v > 0)  // Returns {4,
 5, 6} (Take while index<5 amd value>0)
```

This example selects all items from a location report, starting from the last element and proceeding backwards, until the first item that has a distance to center greater then 20 units:

```
select items.takeWhile(i => distance(i.location.x, i.location.y, 0, 0) < 20)
from LocationReport
```

The second version provides the index of the input element starting at zero for the last element (reverse index).

The next example is similar to the query above but also limits the result to the last 10 items:

```
select items.takeWhile((i, ind) => distance(i.location.x, i.location.y, 0, 0)
 < 20) and ind < 10)
from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns an empty collection.

10.6.26. ToMap

The `toMap` enumeration method returns a Map according to specified key-selector lambda expression and value-selector lambda expression.

The enumeration method takes a key-selector expression and a value-selector expression. For each element the method applies the key-selector expression to determine the map key and the value-selector expression to determine the map value. If the key already exists in the map the value is overwritten.

The next example EPL outputs a map of item asset id and distance to center for each item:

```
select items.toMap(k => k.assetId, v => distance(v.location.x, v.location.y, 0,
 0)) as assetDistance
from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns an empty map.

10.6.27. Union

The `union` enumeration method forms a union of the input elements with the elements that the parameter expression yields.

The enumeration method takes a single parameter that must itself return a collection of events (input), objects or scalar values. It appends the collection to the input elements and returns the appended collection of elements.

An expression example with scalar values:

```
{1, 2, 3}.union({4, 5})   // Returns {1, 2, 3, 4, 5}
```

This example selects a union of all items that have an asset id of L001 or that are of type passenger:

```
select items.where(i => i.assetId = 'L001')
    .union(items.where(i => i.type = 'P')) as itemsUnion
from LocationReport
```

If the input is null the method returns null.

10.6.28. Where

The `where` enumeration method filters elements based on a predicate.

The enumeration method has two versions. The first version takes a predicate lambda expression and the second version takes a predicate lambda expression and index for use within the predicate expression. Both version returns all elements for which the predicate expression is true.

An expression example with scalar values:

```
{1, 2, 3}.where(v => v != 2)   // Returns {1, 3}
```

This example selects all items from a location report that are passenger-type:

```
select items.where(p => p.type = 'P') from LocationReport
```

The second version of the `where` represents the index of the input element starting at zero for the first element.

The example below selects all items from a location report that are passenger-type but ignores the first 3 elements:

```
select items.where((p, ind) => p.type = 'P' and ind > 2) from LocationReport
```

If the input is null the method returns null. If the input is empty the method returns an empty collection.

Chapter 11. EPL Reference: Date-Time Methods

11.1. Overview

EPL date-time methods work on date-time values to perform common tasks such as comparing times and time periods, adding or subtracting time periods, setting or rounding calendar fields and querying fields.

Date-time methods operate on:

1. Any expression or event property that returns either a long-type millisecond value, a `java.util.Calendar` or `java.util.Date` including subclasses. Consider the built-in single-row function `current_timestamp` for use with date-time methods.
2. Any event for which the event type declares a start timestamp property name and optionally also an end timestamp property name. Date-time methods operate on events by means of the *stream-alias*.*method-name* syntax.

The below table summarizes the built-in date-time methods available:

Table 11.1. Date-Time Methods

Method	Result
after(event or timestamp)	Returns true if an event happens after another event, or a timestamp is after another timestamp. *Section 11.4.5, "After"*.
before(event or timestamp)	Returns true if an event happens before another event, or a timestamp is before another timestamp. *Section 11.4.6, "Before"*.
between(timestamp, timestamp, boolean, boolean)	Returns true if a timestamp is between two timestamps. *Section 11.3.1, "Between"*.
coincides(event or timestamp)	Returns true if an event and another event happen at the same time, or two timestamps are the same value. *Section 11.4.7, "Coincides"*.
during(event or timestamp)	Returns true if an event happens during the occurrence of another event, or when a timestamps falls within the occurrence of an event. *Section 11.4.8, "During"*.

Method	Result
finishes(event or timestamp)	Returns true if an event starts after another event starts and the event ends at the same time as the other event. *Section 11.4.9, “Finishes”*.
finishedBy(event or timestamp)	Returns true if an event starts before another event starts and ends at the same time as the other event. *Section 11.4.10, “Finished By”*.
format()	Formats the date-time returning a string. *Section 11.3.2, “Format”*.
get(field)	Returns the value of the given date-time value field. *Section 11.3.3, “Get (By Field)”*.

getMillisOfSecond()
 	
getSecondOfMinute()
 	
getMinuteOfHour()
 	
getHourOfDay()
 	
getDayOfWeek()
 	
getDayOfMonth()
 	
getDayOfYear()
 	
getWeekyear()
 	
getMonthOfYear()
 	
getYear()
 	
getEra()
	Returns the value of the given date-time value field.
	
	Section 11.3.4, “Get (By Name) ”.

includes(event or timestamp)	Returns true if the parameter event happens during the occurrence of the event, or when a timestamps falls within the occurrence of an event. *Section 11.4.11, “Includes”*.
meets(event or timestamp)	Returns true if the event's end time is the same as another event's start time. *Section 11.4.12, “Meets”*.
metBy(event or timestamp)	Returns true if the event's start time is the same as another event's end time.

Method	Result
	Section 11.4.13, "Met By".
minus(duration-millis)	Returns a date-time with the specified duration in long-type milliseconds taken away. Section 11.3.5, "Minus".
minus(time-period)	Returns a date-time with the specified duration in time-period syntax taken away. Section 11.3.5, "Minus".
overlaps(event or timestamp)	Returns true if the event starts before another event starts and finishes after the other event starts, but before the other event finishes (events have an overlapping period of time). Section 11.4.14, "Overlaps".
overlappedBy(event or timestamp)	Returns true if the parameter event starts before the input event starts and the parameter event finishes after the input event starts, but before the input event finishes (events have an overlapping period of time). Section 11.4.15, "Overlapped By".
plus(duration-millis)	Returns a date-time with the specified duration in long-type milliseconds added. Section 11.3.6, "Plus".
plus(time-period)	Returns a date-time with the specified duration in time-period syntax added. Section 11.3.6, "Plus".
roundCeiling(field)	Returns a date-time rounded to the highest whole unit of the date-time field. Section 11.3.7, "RoundCeiling".
roundFloor(field)	Returns a date-time rounded to the lowest whole unit of the date-time field. Section 11.3.8, "RoundFloor".
roundHalf(field)	Returns a date-time rounded to the nearest whole unit of the date-time field. Section 11.3.9, "RoundHalf".
set(field, value)	Returns a date-time with the specified field set to the value returned by a value expression.

Method	Result
	Section 11.3.10, “Set (By Field)”.
starts(event or timestamp)	Returns true if an event and another event start at the same time and the event's end happens before the other event's end. *Section 11.4.16, “Starts”*.
startedBy(event or timestamp)	Returns true if an event and another event start at the same time and the other event's end happens before the input event's end. *Section 11.4.17, “Started By”*.
withDate(year,month,day)	Returns a date-time with the specified date, retaining the time fields. *Section 11.3.11, “WithDate”*.
withMax(field)	Returns a date-time with the field set to the maximum value for the field. *Section 11.3.12, “WithMax”*.
withMin(field)	Returns a date-time with the field set to the minimum value for the field. *Section 11.3.13, “WithMin”*.
withTime(hour,minute,sec,msec)	Returns a date-time with the specified time, retaining the date fields. *Section 11.3.14, “WithTime”*.
toCalendar()	Returns the `Calendar` object for this date-time value. *Section 11.3.15, “ToCalendar”*.
toDate()	Returns the `Date` object for this date-time value. *Section 11.3.16, “ToDate”*.
toMillisec()	Returns the long-type milliseconds value for this date-time value. *Section 11.3.17, “ToMillisec”*.

11.2. How to Use

11.2.1. Syntax

The syntax for date-time methods is the same syntax as for any chained invocation:

```
input_val.datetime_method_name( [method_parameter [, method_parameter
 [,...]]])
   .[ datetime_method_name(...) [...]]
```

Following the *input_val* input value is the . (dot) operator and the *datetime_method_name* date-time method name. It follows in parenthesis a comma-separated list of method parameter expressions. Additional date-time methods can be chained thereafter.

The input value can be any expression or event property that returns a value of type long or `java.util.Calendar` or `java.util.Date`. If the input value is null, the expression result is also null.

The input value can also be an event. In this case the event type of the event must have the start timestamp property name defined and optionally also the end timestamp property name.

The following example EPL statement employs the `withTime` date-time method. This example returns the current engine time with the time-part set to 1 am:

```
select current_timestamp.withTime(1, 0, 0, 0) as time1am from MyEvent
```

As date-time methods can be chained, this EPL is equivalent:

```
select current_timestamp.set('hour', 1).set('min', 0).set('sec', 0).set('msec',
 0) as time1am
from MyEvent
```

The statement above outputs in field `time1am` a long-type millisecond-value reflecting 1am on the same date as engine time. Since the input value is provided by the built-in `current_timestamp` function which returns current engine date as a long-type millisecond value the output is also a long-type millisecond value.

You may apply a date-time method to an event property.

Assume that the `RFIDEvent` event type has a `Date`-type property by name `timeTaken`. The following query rounds each time-taken value down to the nearest minute and outputs a `Date`-type value in column `timeTakenRounded`:

```
select timeTaken.roundFloor('min') as timeTakenRounded from RFIDEvent
```

You may apply a date-time method to events. This example assumes that the RFIDEvent and WifiEvent event types both have a timestamp property defined. The EPL compares the timestamps of the RFIDEvent and the WifiEvent:

```
select rfid.after(wifi) as isAfter
from RFIDEvent.std:lastevent() rfid, WifiEvent.std:lastevent() wifi
```

For comparing date-time values and considering event duration (event start and end timestamps) we recommend any of the interval algebra methods. You may also compare millisecond values using the `between` or `in` ranges and inverted ranges or relational operators (`> , <, >=, <=`).

From a performance perspective, the date-time method evaluation ensures that for each unique chain of date-time methods only a single calendar objects is copied or created when necessary.

11.3. Calendar and Formatting Reference

11.3.1. Between

The `between` date-time method compares the input date-time value to the two date-time values passed in and returns true if the input value falls between the two parameter values.

The synopsis is:

```
input_val.between(range_start, range_end [, include_start, include_end])
```

The method takes either 2 or 4 parameters. The first two parameters *range_start* and *range_end* are expressions or properties that yield either a long-typed, Date-typed or Calendar-typed range start and end value.

The next two parameters *include_start* and *include_end* are optional. If not specified, the range start value and range end value are included in the range i.e. specify a closed range where both endpoints are included. If specified, the expressions must return a boolean-value indicating whether to include the range start value and range end value in the range.

The example below outputs true when the time-taken property value of the RFID event falls between the time-start property value and the time-end property value (closed range includes endpoints):

```
select timeTaken.between(timeStart, timeEnd) from RFIDEvent
```

The example below performs the same test as above but does not include endpoints (open range includes neither endpoint):

```
select timeTaken.between(timeStart, timeEnd, false, false) from RFIDEvent
```

If the range end value is less then the range start value, the algorithm reverses the range start and end value.

If the input date-time value or any of the parameter values evaluate to null the method returns a null result value.

11.3.2. Format

The `format` date-time method formats the date-time returning a string.

The method takes no parameters. It returns the date-time value formatted using the default locale format obtained from `new SimpleDateFormat()`.

The example below outputs the time-taken property value of the RFID event:

```
select timeTaken.format() as timeTakenStr from RFIDEvent
```

11.3.3. Get (By Field)

The `get` date-time method returns the value of the given date-time value field.

The method takes a single string-constant field name as parameter. Please see *Section 5.2.1, “Specifying Time Periods”* for a list of recognized keywords (not case-sensitive).

The method returns the numeric value of the field within the date-time value. The value returned adheres to `Calendar`-class semantics: For example, the value for `month` starts at zero and has a maximum of 11.

The example below outputs the month value of the time-taken property value of the RFID event:

```
select timeTaken.get('month') as timeTakenMonth from RFIDEvent
```

11.3.4. Get (By Name)

The following list of getter-methods are available: `getMillisOfSecond()`, `getSecondOfMinute()`, `getMinuteOfHour()`, `getHourOfDay()`, `getDayOfWeek()`, `getDayOfMonth()`, `getDayOfYear()`, `getWeekYear()`, `getMonthOfYear()`, `getYear()` and `getEra()`.

All get-methods take no parameter and return the numeric value of the field within the date-time value. The value returned adheres to `Calendar`-class semantics: For example, the value for `month` starts at zero and has a maximum of 11.

The example below outputs the month value of the time-taken property value of the RFID event:

```
select timeTaken.getMonthOfYear() as timeTakenMonth from RFIDEvent
```

11.3.5. Minus

The `minus` date-time method returns a date-time with the specified duration taken away.

The method has two versions: The first version takes the duration as a long-type millisecond value. The second version takes the duration as a time-period expression, see *Section 5.2.1, "Specifying Time Periods"*.

The example below demonstrates the time-period parameter to subtract two minutes from the time-taken property value of the RFID event:

```
select timeTaken.minus(2 minutes) as timeTakenMinus2Min from RFIDEvent
```

The next example is equivalent but passes a millisecond-value instead:

```
select timeTaken.minus(2*60*1000) as timeTakenMinus2Min from RFIDEvent
```

11.3.6. Plus

The `plus` date-time method returns a date-time with the specified duration added.

The method has two versions: The first version takes the duration as a long-type millisecond value. The second version takes the duration as a time-period expression, see *Section 5.2.1, "Specifying Time Periods"*.

The next example adds two minutes to the time-taken property value of the RFID event:

```
select timeTaken.plus(2 minutes) as timeTakenPlus2Min from RFIDEvent
```

The next example is equivalent but passes a millisecond-value instead:

```
select timeTaken.plus(2*60*1000) as timeTakenPlus2Min from RFIDEvent
```

11.3.7. RoundCeiling

The `roundCeiling` date-time method rounds to the highest whole unit of the date-time field.

The method takes a single string-constant field name as parameter. Please see *Section 5.2.1, "Specifying Time Periods"* for a list of recognized keywords (not case-sensitive).

The next example rounds-to-ceiling the minutes of the time-taken property value of the RFID event:

```
select timeTaken.roundCeiling('min') as timeTakenRounded from RFIDEvent
```

If the input time is `2002-05-30 09:01:23.050`, for example, the output is `2002-05-30 09:02:00.000` (example timestamps are in format `yyyy-MM-dd HH:mm:ss.SSS`).

11.3.8. RoundFloor

The `roundFloor` date-time method rounds to the lowest whole unit of the date-time field.

The method takes a single string-constant field name as parameter. Please see *Section 5.2.1, “Specifying Time Periods”* for a list of recognized keywords (not case-sensitive).

The next example rounds-to-floor the minutes of the time-taken property value of the RFID event:

```
select timeTaken.roundFloor('min') as timeTakenRounded from RFIDEvent
```

If the input time is `2002-05-30 09:01:23.050`, for example, the output is `2002-05-30 09:01:00.000` (example timestamps are in format `yyyy-MM-dd HH:mm:ss.SSS`).

11.3.9. RoundHalf

The `roundFloor` date-time method rounds to the nearest whole unit of the date-time field.

The method takes a single string-constant field name as parameter. Please see *Section 5.2.1, “Specifying Time Periods”* for a list of recognized keywords (not case-sensitive).

The next example rounds the minutes of the time-taken property value of the RFID event:

```
select timeTaken.roundHalf('min') as timeTakenRounded from RFIDEvent
```

The following table provides a few examples of the rounding (example timestamps are in format `yyyy-MM-dd HH:mm:ss.SSS`):

Table 11.2. RoundHalf Examples

Input	Output
2002-05-30 09:01:23.050	2002-05-30 09:01:00.000
2002-05-30 09:01:29.999	2002-05-30 09:01:00.000
2002-05-30 09:01:30.000	2002-05-30 09:02:00.000

11.3.10. Set (By Field)

The `set` date-time method returns a date-time with the specified field set to the value returned by an expression.

The method takes a string-constant field name and an expression returning an integer-value as parameters. Please see *Section 5.2.1, “Specifying Time Periods”* for a list of recognized keywords (not case-sensitive).

The method returns the new date-time value with the field set to the provided value. Note that value adheres to `Calendar`-class semantics: For example, the value for `month` starts at zero and has a maximum of 11.

The example below outputs the time-taken with the value for month set to April:

```
select timeTaken.set('month', 3) as timeTakenMonth from RFIDEvent
```

11.3.11. WithDate

The `withDate` date-time method returns a date-time with the specified date, retaining the time fields.

The method takes three expressions as parameters: An expression for year, month and day.

The method returns the new date-time value with the date fields set to the provided values. For expressions returning null the method ignores the field for which null is returned. Note the `Calendar`-class semantics: For example, the value for `month` starts at zero and has a maximum of 11.

The example below outputs the time-taken with the date set to May 30, 2002:

```
select timeTaken.withDate(2002, 4, 30) as timeTakenDated from RFIDEvent
```

11.3.12. WithMax

The `withMax` date-time method returns a date-time with the field set to the maximum value for the field.

The method takes a string-constant field name as parameter. Please see *Section 5.2.1, “Specifying Time Periods”* for a list of recognized keywords (not case-sensitive).

The method returns the new date-time value with the specific date field set to the maximum value.

The example below outputs the time-taken property value with the second-part as 59 seconds:

```
select timeTaken.withMax('sec') as timeTakenMaxSec from RFIDEvent
```

11.3.13. WithMin

The `withMin` date-time method returns a date-time with the field set to the minimum value for the field.

The method takes a string-constant field name as parameter. Please see *Section 5.2.1, “Specifying Time Periods”* for a list of recognized keywords (not case-sensitive).

The method returns the new date-time value with the specific date field set to the minimum value.

The example below outputs the time-taken property value with the second-part as 0 seconds:

```
select timeTaken.withMin('sec') as timeTakenMaxSec from RFIDEvent
```

11.3.14. WithTime

The `withTime` date-time method returns a date-time with the specified time, retaining the date fields.

The method takes four expressions as parameters: An expression for hour, minute, second and millisecond.

The method returns the new date-time value with the time fields set to the provided values. For expressions returning null the method ignores the field for which null is returned.

The example below outputs the time-taken with the time set to 9am:

```
select timeTaken.withTime(9, 0, 0, 0) as timeTakenDated from RFIDEvent
```

11.3.15. ToCalendar

The `toCalendar` date-time method returns the `Calendar` object for this date-time value.

The method takes no parameters.

The example below outputs the time-taken as a `Calendar` object:

```
select timeTaken.toCalendar() as timeTakenCal from RFIDEvent
```

11.3.16. ToDate

The `toDate` date-time method returns the `Date` object for this date-time value.

The method takes no parameters.

The example below outputs the time-taken as a `Date` object:

```
select timeTaken.toDate() as timeTakenDate from RFIDEvent
```

11.3.17. ToMillisec

The `toMillisec` date-time method returns the long-typed millisecond value for this date-time value.

The method takes no parameters.

The example below outputs the time-taken as a long-typed `millisecond` value:

```
select timeTaken.toMillisec() as timeTakenLong from RFIDEvent
```

11.4. Interval Algebra Reference

Interval algebra methods compare start and end timestamps of events or timestamps in general.

When the expression input is only a timestamp value, such as a long-type value or a `Date` or `Calendar` object, the start and end timestamp represented by that value are the same timestamp value.

When expression input is an event stream alias, the engine determine the event type for the stream. If the event type declares a start timestamp property name, the engine uses that start timestamp property to determine the start timestamp for the event. If the event type also declares an end timestamp property name, the engine uses that end timestamp property to determine the end timestamp for the event (i.e. an event with duration). If an end timestamp property name is not declared, the start and end timestamp for each event is the same value and the event is considered to have zero duration (i.e. a point-in-time event).

Interval algebra methods all return `Boolean`-type value. When the input value start timestamp is null, or the end timestamp (if declared for the event type) is null or any of the start timestamp and end timestamp (if declared for the event type) values of the first parameter is null, the result value is null.

11.4.1. Examples

The examples in this section simply use `A` and `B` as event type names. The alias `a` is used to represent `A`-type events and respectively the alias `b` represents `B`-type events.

The `create-schema` for types `A` and `B` is shown next. The two types are declared the same. The example declares the property providing start timestamp values as `startts` and the property providing end timestamp values as `endts`:

```
create schema A as (startts long, endts long) starttimestamp 'startts'
 endtimestamp 'endts'
```

```
create schema B as (startts long, endts long) starttimestamp 'startts'
 endtimestamp 'endts'
```

The sample EPL below joins the last A and the last B event. It detects A-B event combinations for which, when comparing timestamps, the last A event that occurs before the last B event. The example employs the `before` method:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.before(b)
```

For simplicity, the examples in this section refer to `A` and the alias `a` as the input event. The examples refer to `B` and the alias `b` as the parameter event.

11.4.2. Interval Algebra Parameters

The first parameter of each interval algebra methods is the event or timestamp to compare to.

All remaining parameters to interval algebra methods are intervals and can be any of the following:

1. A constant, an event property or more generally any expression returning a numeric value that is the number of seconds. For example, in the expression `a.before(b, 2)` the parameter 2 is interpreted to mean 2 seconds. The expression `a.before(b, myIntervalProperty)` is interpreted to mean `myIntervalProperty` seconds.
2. A time period expression as described in *Section 11.4.11, "Includes"*. For example: `a.before(b, 1 hour 2 minutes)`.

When an interval parameter is provided and is null, the method result value is null.

11.4.3. Performance

The engine analyzes interval algebra methods as well as the `between` date-time method in the where-clause and builds a query plan for execution of joins and subqueries. The query plan can include hash and btree index lookups using the start and end timestamps as computed by expressions or provided by events as applicable. Consider turning on query plan logging to obtain information on the query plan used.

The query planning is generally most effective when no additional thresholds or ranges are provided to interval algebra methods, as the query planner may not consider an interval algebra method that it cannot plan.

The query planner may also not optimally plan the query execution if events or expressions return different types of date representation. Query planning works best if all date representations use the same long, Date or Calendar types.

11.4.4. Limitations

Date-time method that change date or time fields, such as `withTime`, `withDate`, `set` or `round` methods set the end timestamp to the start timestamp.

For example, in the following expression the parameter to the `after` method has a zero duration, and not the end timestamp that the event B `endts` property provides.

```
a.after(b.withTime(9, 0, 0, 0))
```

11.4.5. After

The `after` date-time method returns true if an event happens after another event, or a timestamp is after another timestamp.

The method compares the input value's start timestamp (a.startTimestamp) to the first parameter's end timestamp (b.endTimestamp) to determine whether A happens after B.

If used with one parameter, for example in `a.after(b)`, the method returns true if A starts after B ends.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.after(b)
// Above matches when:
//   a.startTimestamp - b.endTimestamp > 0
```

If providing two parameters, for example in `a.after(b, 5 sec)`, the method returns true if A starts at least 5 seconds after B ends.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.after(b,
 5 sec)
// Above matches when:
//   a.startTimestamp - b.endTimestamp >= 5 seconds
```

If providing three parameters, for example in `a.after(b, 5 sec, 10 sec)`, the method returns true if A starts at least 5 seconds but no more then 10 seconds after B ends.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.after(b,
 5 sec, 10 sec)
// Above matches when:
//   5 seconds <= a.startTimestamp - b.endTimestamp <= 10 seconds
```

Negative values for the range are allowed. For example in `a.after(b, -5 sec, -10 sec)`, the method returns true if A starts at least 5 seconds but no more then 10 seconds before B ends.

If the range low endpoint is greater than the range high endpoint, the engine automatically reverses them. Thus `a.after(b, 10 sec, 5 sec)` is the same semantics as `a.after(b, 5 sec, 10 sec)`.

11.4.6. Before

The `before` date-time method returns true if an event happens before another event, or a timestamp is before another timestamp.

The method compares the input value's end timestamp (a.endTimestamp) and the first parameter's start timestamp (b.startTimestamp) to determine whether A happens before B.

If used with one parameter, for example in `a.before(b)`, the method returns true if A ends before B starts.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.before(b)
// Above matches when:
//   b.startTimestamp - a.endTimestamp > 0
```

If providing two parameters, for example in `a.before(b, 5 sec)`, the method returns true if A ends at least 5 seconds before B starts.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.before(b,
 5 sec)
// Above matches when:
//   b.startTimestamp - a.endTimestamp >= 5 seconds
```

If providing three parameters, for example in `a.before(b, 5 sec, 10 sec)`, the method returns true if A ends at least 5 seconds but no more then 10 seconds before B starts.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.before(b,
 5 sec, 10 sec)
// Above matches when:
//   5 seconds <= b.startTimestamp - a.endTimestamp <= 10 seconds
```

Negative values for the range are allowed. For example in `a.before(b, -5 sec, -10 sec)`, the method returns true if A starts at least 5 seconds but no more then 10 seconds after B starts.

If the range low endpoint is greater than the range high endpoint, the engine automatically reverses them. Thus `a.before(b, 10 sec, 5 sec)` is the same semantics as `a.before(b, 5 sec, 10 sec)`.

11.4.7. Coincides

The `coincides` date-time method returns true if an event and another event happen at the same time, or two timestamps are the same value.

The method compares the input value's start and end timestamp with the first parameter's start and end timestamp and determines if they equal.

If used with one parameter, for example in `a.coincides(b)`, the method returns true if the start timestamp of A and B are the same and the end timestamps of A and B are also the same.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.coincides(b)
// Above matches when:
//   a.startTimestamp = b.startTimestamp and a.endTimestamp = b.endTimestamp
```

If providing two parameters, for example in `a.coincides(b, 5 sec)`, the method returns true if the difference between the start timestamps of A and B is equal to or less then 5 seconds and the difference between the end timestamps of A and B is also equal to or less then 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.coincides(b,
 5 sec)
// Above matches when:
//   abs(a.startTimestamp - b.startTimestamp) <= 5 sec and
//   abs(a.endTimestamp - b.endTimestamp) <= 5 sec
```

If providing three parameters, for example in `a.coincides(b, 5 sec, 10 sec)`, the method returns true if the difference between the start timestamps of A and B is equal to or less then 5 seconds and the difference between the end timestamps of A and B is equal to or less then 10 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.coincides(b,
 5 sec, 10 sec)
// Above matches when:
//   abs(a.startTimestamp - b.startTimestamp) <= 5 seconds and
//   abs(a.endTimestamp - b.endTimestamp) <= 10 seconds
```

A negative value for interval parameters is not allowed. If your interval parameter is itself an expression that returns a negative value the engine logs a warning message and returns null.

11.4.8. During

The `during` date-time method returns true if an event happens during the occurrence of another event, or when a timestamps falls within the occurrence of an event..

The method determines whether the input value's start and end timestamp are during the first parameter's start and end timestamp. The symmetrical opposite is *Section 11.4.11, “Includes”*.

If used with one parameter, for example in `a.during(b)`, the method returns true if the start timestamp of A is after the start timestamp of B and the end timestamp of A is before the end timestamp of B.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.during(b)
// Above matches when:
//   b.startTimestamp < a.startTimestamp <= a.endTimestamp < b.endTimestamp
```

If providing two parameters, for example in `a.during(b, 5 sec)`, the method returns true if the difference between the start timestamps of A and B is equal to or less then 5 seconds and the difference between the end timestamps of A and B is also equal to or less then 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.during(b,
 5 sec)
// Above matches when:
//   0 < a.startTimestamp - b.startTimestamp <= 5 sec and
//   0 < a.endTimestamp - b.endTimestamp <= 5 sec
```

If providing three parameters, for example in `a.during(b, 5 sec, 10 sec)`, the method returns true if the difference between the start timestamps of A and B and the difference between the end timestamps of A and B is between 5 and 10 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.during(b,
 5 sec, 10 sec)
// Above matches when:
//   5 seconds <= a.startTimestamp - b.startTimestamp <= 10 seconds and
//   5 seconds <= a.endTimestamp - b.endTimestamp <= 10 seconds
```

If providing five parameters, for example in `a.during(b, 5 sec, 10 sec, 20 sec, 30 sec)`, the method returns true if the difference between the start timestamps of A and B is between 5 seconds and 10 seconds and the difference between the end timestamps of A and B is between 20 seconds and 30 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b
  where a.during(b, 5 sec, 10 sec, 20 sec, 30 sec)
// Above matches when:
//   5 seconds <= a.startTimestamp - b.startTimestamp <= 10 seconds and
//   20 seconds < a.endTimestamp - b.endTimestamp <= 30 seconds
```

11.4.9. Finishes

The `finishes` date-time method returns true if an event starts after another event starts and the event ends at the same time as the other event.

The method determines whether the input value's start timestamp is after the first parameter's start timestamp and the end timestamp of the input value and the first parameter are the same. The symmetrical opposite is *Section 11.4.10, “Finished By”*.

If used with one parameter, for example in `a.finishes(b)`, the method returns true if the start timestamp of A is after the start timestamp of B and the end timestamp of A and B are the same.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.finishes(b)
// Above matches when:
//   b.startTimestamp < a.startTimestamp and a.endTimestamp = b.endTimestamp
```

If providing two parameters, for example in `a.finishes(b, 5 sec)`, the method returns true if the start timestamp of A is after the start timestamp of B and the difference between the end timestamps of A and B is equal to or less then 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.finishes(b,
 5 sec)
// Above matches when:
//   b.startTimestamp < a.startTimestamp and
//   abs(a.endTimestamp - b.endTimestamp ) <= 5 seconds
```

A negative value for interval parameters is not allowed. If your interval parameter is itself an expression that returns a negative value the engine logs a warning message and returns null.

11.4.10. Finished By

The `finishedBy` date-time method returns true if an event starts before another event starts and the event ends at the same time as the other event.

The method determines whether the input value's start timestamp happens before the first parameter's start timestamp and the end timestamp of the input value and the first parameter are the same. The symmetrical opposite is *Section 11.4.9, “Finishes”*.

If used with one parameter, for example in `a.finishedBy(b)`, the method returns true if the start timestamp of A is before the start timestamp of B and the end timestamp of A and B are the same.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.finishedBy(b)
// Above matches when:
//   a.startTimestamp < b.startTimestamp and a.endTimestamp = b.endTimestamp
```

If providing two parameters, for example in `a.finishedBy(b, 5 sec)`, the method returns true if the start timestamp of A is before the start timestamp of B and the difference between the end timestamps of A and B is equal to or less then 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.finishedBy(b,
 5 sec)
// Above matches when:
//   a.startTimestamp < b.startTimestamp and
//   abs(a.endTimestamp - b.endTimestamp ) <= 5 seconds
```

11.4.11. Includes

The `includes` date-time method returns true if the parameter event happens during the occurrence of the input event, or when a timestamps falls within the occurrence of an event.

The method determines whether the first parameter's start and end timestamp are during the input value's start and end timestamp. The symmetrical opposite is *Section 11.4.8, “During”*.

If used with one parameter, for example in `a.includes(b)`, the method returns true if the start timestamp of B is after the start timestamp of A and the end timestamp of B is before the end timestamp of A.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.includes(b)
// Above matches when:
//   a.startTimestamp < b.startTimestamp <= b.endTimestamp < a.endTimestamp
```

If providing two parameters, for example in `a.includes(b, 5 sec)`, the method returns true if the difference between the start timestamps of A and B is equal to or less then 5 seconds and the difference between the end timestamps of A and B is also equal to or less then 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.includes(b,
 5 sec)
// Above matches when:
//   0 < b.startTimestamp - a.startTimestamp <= 5 sec and
//   0 < a.endTimestamp - b.endTimestamp <= 5 sec
```

If providing three parameters, for example in `a.includes(b, 5 sec, 10 sec)`, the method returns true if the difference between the start timestamps of A and B and the difference between the end timestamps of A and B is between 5 and 10 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.includes(b,
 5 sec, 10 sec)
// Above matches when:
//   5 seconds <= a.startTimestamp - b.startTimestamp <= 10 seconds and
//   5 seconds <= a.endTimestamp - b.endTimestamp <= 10 seconds
```

If providing five parameters, for example in `a.includes(b, 5 sec, 10 sec, 20 sec, 30 sec)`, the method returns true if the difference between the start timestamps of A and B is between 5 seconds and 10 seconds and the difference between the end timestamps of A and B is between 20 seconds and 30 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b
  where a.includes(b, 5 sec, 10 sec, 20 sec, 30 sec)
// Above matches when:
//   5 seconds <= a.startTimestamp - b.startTimestamp <= 10 seconds and
//   20 seconds <= a.endTimestamp - b.endTimestamp <= 30 seconds
```

11.4.12. Meets

The `meets` date-time method returns true if the event's end time is the same as another event's start time.

The method compares the input value's end timestamp and the first parameter's start timestamp and determines whether they equal.

If used with one parameter, for example in `a.meets(b)`, the method returns true if the end timestamp of A is the same as the start timestamp of B.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.meets(b)
// Above matches when:
//   a.endTimestamp = b.startTimestamp
```

If providing two parameters, for example in `a.meets(b, 5 sec)`, the method returns true if the difference between the end timestamp of A and the start timestamp of B is equal to or less then 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.meets(b,
 5 sec)
// Above matches when:
//   abs(b.startTimestamp - a.endTimestamp) <= 5 seconds
```

A negative value for the interval parameter is not allowed. If your interval parameter is itself an expression that returns a negative value the engine logs a warning message and returns null.

11.4.13. Met By

The `metBy` date-time method returns true if the event's start time is the same as another event's end time.

The method compares the input value's start timestamp and the first parameter's end timestamp and determines whether they equal.

If used with one parameter, for example in `a.metBy(b)`, the method returns true if the start timestamp of A is the same as the end timestamp of B.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.metBy(b)
// Above matches when:
//   a.startTimestamp = b.endTimestamp
```

If providing two parameters, for example in `a.metBy(b, 5 sec)`, the method returns true if the difference between the end timestamps of B and the start timestamp of A is equal to or less then 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.metBy(b,
 5 sec)
// Above matches when:
//   abs(a.startTimestamp - b.endTimestamp) <= 5 seconds
```

A negative value for the interval parameter is not allowed. If your interval parameter is itself an expression that returns a negative value the engine logs a warning message and returns null.

11.4.14. Overlaps

The `overlaps` date-time method returns true if the event starts before another event starts and finishes after the other event starts, but before the other event finishes (events have an overlapping period of time).

The method determines whether the input value's start and end timestamp indicate an overlap with the first parameter's start and end timestamp, such that A starts before B starts and A ends after B started but before B ends.

If used with one parameter, for example in `a.overlaps(b)`, the method returns true if the start timestamp of A is before the start timestamp of B and the end timestamp of A and is before the end timestamp of B.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.overlaps(b)
// Above matches when:
//   a.startTimestamp < b.startTimestamp < a.endTimestamp < b.endTimestamp
```

If providing two parameters, for example in `a.overlaps(b, 5 sec)`, the method returns true if, in addition, the difference between the end timestamp of A and the start timestamp of B is equal to or less then 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.overlaps(b,
 5 sec)
// Above matches when:
//   a.startTimestamp < b.startTimestamp < a.endTimestamp < b.endTimestamp and
//   0 <= a.endTimestamp - b.startTimestamp <= 5 seconds
```

If providing three parameters, for example in `a.overlaps(b, 5 sec, 10 sec)`, the method returns true if, in addition, the difference between the end timestamp of A and the start timestamp of B is between 5 and 10 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.overlaps(b,
 5 sec, 10 sec)
// Above matches when:
//   a.startTimestamp < b.startTimestamp < a.endTimestamp < b.endTimestamp and
//   5 seconds <= a.endTimestamp - b.startTimestamp <= 10 seconds
```

11.4.15. Overlapped By

The `overlappedBy` date-time method returns true if the parameter event starts before the input event starts and the parameter event finishes after the input event starts, but before the input event finishes (events have an overlapping period of time).

The method determines whether the input value's start and end timestamp indicate an overlap with the first parameter's start and end timestamp, such that B starts before A starts and B ends after A started but before A ends.

If used with one parameter, for example in `a.overlappedBy(b)`, the method returns true if the start timestamp of B is before the start timestamp of A and the end timestamp of B and is before the end timestamp of A.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where
 a.overlappedBy(b)
// Above matches when:
//   b.startTimestamp < a.startTimestamp < b.endTimestamp < a.endTimestamp
```

If providing two parameters, for example in `a.overlappedBy(b, 5 sec)`, the method returns true if, in addition, the difference between the end timestamp of B and the start timestamp of A is equal to or less then 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where
 a.overlappedBy(b, 5 sec)
// Above matches when:
//   b.startTimestamp < a.startTimestamp < b.endTimestamp < a.endTimestamp and
//   0 <= b.endTimestamp - a.startTimestamp <= 5 seconds
```

If providing three parameters, for example in `a.overlappedBy(b, 5 sec, 10 sec)`, the method returns true if, in addition, the difference between the end timestamp of B and the start timestamp of A is between 5 and 10 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where
 a.overlappedBy(b, 5 sec, 10 sec)
// Above matches when:
//   b.startTimestamp < a.startTimestamp < b.endTimestamp < a.endTimestamp and
//   5 seconds <= b.endTimestamp - a.startTimestamp <= 10 seconds
```

11.4.16. Starts

The `starts` date-time method returns true if an event and another event start at the same time and the event's end happens before the other event's end.

The method determines whether the start timestamps of the input value and the first parameter are the same and the end timestamp of the input value is before the end timestamp of the first parameter.

If used with one parameter, for example in `a.starts(b)`, the method returns true if the start timestamp of A and B are the same and the end timestamp of A is before the end timestamp of B.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.starts(b)
// Above matches when:
//   a.startTimestamp = b.startTimestamp and a.endTimestamp < b.endTimestamp
```

If providing two parameters, for example in `a.starts(b, 5 sec)`, the method returns true if the difference between the start timestamps of A and B is between is equal to or less then 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.starts(b,
 5 sec)
// Above matches when:
//   abs(a.startTimestamp - b.startTimestamp) <= 5 seconds and
//   a.endTimestamp < b.endTimestamp
```

A negative value for the interval parameter is not allowed. If your interval parameter is itself an expression that returns a negative value the engine logs a warning message and returns null.

11.4.17. Started By

The `startedBy` date-time method returns true if an event and another event start at the same time and the other event's end happens before the input event's end.

The method determines whether the start timestamp of the input value and the first parameter are the same and the end timestamp of the first parameter is before the end timestamp of the input value.

If used with one parameter, for example in `a.startedBy(b)`, the method returns true if the start timestamp of A and B are the same and the end timestamp of B is before the end timestamp of A.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.startedBy(b)
// Above matches when:
//   a.startTimestamp = b.startTimestamp and b.endTimestamp < a.endTimestamp
```

If providing two parameters, for example in `a.startedBy(b, 5 sec)`, the method returns true if the difference between the start timestamps of A and B is between is equal to or less then 5 seconds.

Sample EPL:

```
select * from A.std:lastevent() as a, B.std:lastevent() as b where a.startedBy(b,
 5 sec)
```

```
// Above matches when:
//   abs(a.startTimestamp - b.startTimestamp) <= 5 seconds and
//    b.endTimestamp < a.endTimestamp
```

A negative value for the interval parameter is not allowed. If your interval parameter is itself an expression that returns a negative value the engine logs a warning message and returns null.

Chapter 12. 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. Please also see *Section 5.4.3, "Specifying Views"* for the use of views in the `from` clause with streams, patterns and named windows.

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. Derived-value views derive a new value from event streams and post the result as events of a new type.

Two or more data window views can be combined. This allows a sets of events retained by one data window to be placed into a union or an intersection with the set of events retained by one or more other data windows. Please see *Section 5.4.4, "Multiple Data Window Views"* for more detail.

The keep-all data window counts as a data window but has no expiry policy: it retains all events received. The grouped-window declaration allocates a new data window per grouping criteria and thereby counts as a data window, but cannot appear alone.

The next table summarizes data window views:

Table 12.1. 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 expression*, *time period*)	Sliding time window, based on the millisecond time value supplied by an expression.

View	Syntax	Description
Time Batch Window	win:time_batch(*time period*[,*optional reference point*] [, *flow control*])	Tumbling window that batches events and releases them every specified time interval, with flow control options.
Externally-timed Batch Window	win:ext_timed_batch(*timestamp expression*, *time period*[,*optional reference point*])	Tumbling window that batches events and releases them every specified time interval based on the millisecond value supplied by an expression.
Time-Length Combination Batch Window	win:time_length_batch(*time period, size [, flow control]*)	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(*size, sort criteria*)	Sorts by values returned by sort criteria expressions and keeps only the top events up to the given size.
Ranked Window	ext:rank(*unique criteria(s), size, sort criteria(s)*)	Retains only the most recent among events having the same value for the criteria expression(s) sorted by sort criteria expressions and keeps only the top events up to the given size.
Time-Order Window	ext:time_order(*timestamp expression, time period*)	Orders events that arrive out-of-order, using an expression providing timestamps to be ordered.
Unique Window	std:unique(*unique criteria(s)*)	Retains only the most recent among events having the same value for the criteria expression(s). Acts as a length window of size 1 for each distinct expression value.
Grouped Data Window	std:groupwin(*grouping criteria(s)*)	Groups events into sub-views by the value of the specified

View	Syntax	Description
		expression(s), generally used to provide a separate data window per group.
Last Event Window	std:lastevent()	Retains the last event, acts as a length window of size 1.
First Event Window	std:firstevent()	Retains the very first arriving event, disregarding all subsequent events.
First Unique Window	std:firstunique(*unique criteria(s)*)	Retains only the very first among events having the same value for the criteria expression(s), disregarding all subsequent events for same value(s).
First Length Window	win:firstlength(*size*)	Retains the first *size* events, disregarding all subsequent events.
First Time Window	win:firsttime(*time period*)	Retains the events arriving until the time interval has passed, disregarding all subsequent events.
Expiry Expression Window	win:expr(*expiry expression*)	Expire events based on the result of an expiry expression passed as a parameter.
Expiry Expression Batch Window	win:expr_batch(*expiry expression*)	Tumbling window that batches events and releases them based on the result of an expiry expression passed as a parameter.

The table below summarizes views that derive information from received events and present the derived information as an insert and remove stream of events that are typed specifically to carry the result of the computations:

Table 12.2. Built-in Derived-Value Views

View	Syntax	Description
Size	std:size([*expression*, ...])	Derives a count of the number of events in a data window, or in an insert stream if used without a data window, and optionally provides additional

View	Syntax	Description
		event properties as listed in parameters.
Univariate statistics	stat:uni(*value expression* [,*expression*, ...])	Calculates univariate statistics on the values returned by the expression.
Regression	stat:linest(*value expression, value expression* [,*expression*, ...])	Calculates regression on the values returned by two expressions.
Correlation	stat:correl(*value expression, value expression* [,*expression*, ...])	Calculates the correlation value on the values returned by two expressions.
Weighted average	stat:weighted_avg(*value expression, value expression* [,*expression*, ...])	Calculates weighted average given a weight expression and an expression to compute the average for.

12.1. A Note on View Parameters

The syntax for view specifications starts with the namespace name and the name and is followed by optional view parameter expressions in parenthesis:

```
namespace:name(view_parameters)
```

This example specifies a time window of 5 seconds:

```
select * from StockTickEvent.win:time(5 sec)
```

All expressions are allowed as parameters to views, including expressions that contain variables or substitution parameters for prepared statements. Subqueries, the special `prior` and `prev` functions and aggregations (with the exception of the expression window and expression batch window) are not allowed as view parameters.

For example, assuming a variable by name `VAR_WINDOW_SIZE` is defined:

```
select * from StockTickEvent.win:time(VAR_WINDOW_SIZE)
```

Expression parameters for views are evaluated at the time the view is first created with the exception of the expression window (`win:expr`) and expression batch window (`win:expr_batch`). Also consider multiple data windows in intersection or union (keywords `retain-intersection` and `retain-union`). Consider writing a custom plug-in view if your application requires behavior that is not yet provided by any of the built-in views.

If a view takes no parameters, use empty parenthesis `()`.

12.2. Data Window Views

All the views explained below are data window views, as are `std:unique`, `std:firstunique`, `std:lastevent` and `std:firstevent`.

12.2.1. 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 expression as a parameter providing a numeric size value that defines the window size:

```
win:length(size_expression)
```

The below example sums the price for the last 5 stock ticks for symbol GE.

```
select sum(price) from StockTickEvent(symbol='GE').win:length(5)
```

The next example keeps a length window of 10 events of stock trade events, with a separate window for each symbol. The sum of price is calculated only for the last 10 events for each symbol and aggregates per symbol:

```
select sum(price) from StockTickEvent.std:groupwin(symbol).win:length(10) group
 by symbol
```

A length window of 1 is equivalent to the last event window `std:lastevent`. The `std:lastevent` data window is the preferred notation:

```
select * from StockTickEvent.std:lastevent() // Prefer this
// ... equivalent to ...
select * from StockTickEvent.win:length(1)
```

12.2.2. 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. Provide an expression defining the number of events to batch as a parameter:

```
win:length_batch(size_expression)
```

The next statement buffers events until a minimum of 10 events have collected. Listeners to updates posted by this view receive updated information only when 10 or more events have collected.

```
select * from StockTickEvent.win:length_batch(10)
```

12.2.3. 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. Provide a time period (see *Section 5.2.1, “Specifying Time Periods”*) or an expression defining the number of seconds as a parameter:

```
win:time(time period)
```

```
win:time(seconds_interval_expression)
```

For the GE stock tick events in the last 1 second, calculate a sum of price.

```
select sum(price) from StockTickEvent(symbol='GE').win:time(1 sec)
```

The following time windows are equivalent specifications:

```
win:time(2 minutes 5 seconds)
win:time(125 sec)
win:time(125)
win:time(MYINTERVAL)  // MYINTERVAL defined as a variable
```

12.2.4. 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 a timestamp expression. The view takes two parameters: the expression to return long-typed timestamp values, and a time period or expression that provides a number of seconds:

```
win:ext_timed(timestamp_expression, time_period)
```

```
win:ext_timed(timestamp_expression, seconds_interval_expression)
```

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 result of the timestamp expression when evaluated against the events entering the window.

The algorithm underlying the view compares the timestamp value returned by the expression when the oldest event arrived with the timestamp value returned by the expression for the newest arriving event on event arrival. If the time interval between the timestamp values 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 (or other expression value returned) 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 strict ordering of the timestamp values returned by the timestamp expression. The view is not useful for ordering events in time order, please use the time-order view instead.

On a related subject, engine time itself can be entirely under control of the application as described in *Section 14.8, “Controlling Time-Keeping”*, allowing control over all time-based aspects of processing in one place.

12.2.5. 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 an expression providing a number of seconds as a parameter, plus optional parameters described next.

```
win:time_batch(time_period [,optional_reference_point] [,flow_control])
```

```
win:time_batch(seconds_interval_expression [,optional_reference_point]
 [,flow_control])
```

The time batch window takes a second, optional parameter that serves as a reference point to batch flush times. If not specified, the arrival of the first event into the batch window sets the reference point. Therefore if the reference point is not specified and the first event arrives at time t_1, then the batch flushes at time t_1 plus *time_period* and every *time_period* thereafter.

Note that using this view means that the engine keeps events in memory until the time is up: Consider your event arrival rate and determine if this is the behavior you want. Use context declaration or output rate limiting such as `output snapshot` as an alternative.

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)
```

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. The view allows overriding this default behavior via flow control keywords.

The synopsis with flow control parameters is:

```
win:time_batch(time_period or seconds_interval_expr
 [,optional_reference_point]
    [, "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. When using this keyword the `irstream` keyword should be used in the `select` clause to ensure the remove stream is also output. Note that FORCE_UPDATE is for use with listeners to the same statement and not for use with named windows. Consider output rate limiting instead.

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. 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_batch(10 sec, "FORCE_UPDATE, START_EAGER")
```

The optional reference point is provided as a long-value of milliseconds relative to January 1, 1970 and time 00:00:00.

The following example statement sets the reference point to 5 seconds and the batch size to 1 hour, so that each batch output is 5 seconds after each hour:

```
select * from OrderSummaryEvent.win:time_batch(1 hour, 5000L)
```

12.2.6. Externally-timed batch window (`win:ext_timed_batch`)

Similar to the time batch window, this view buffers events (tumbling) and releases them every specified time interval in one update, but based on the millisecond time value supplied by a timestamp expression. The view has two required parameters taking an expression that returns long-typed timestamp values and a time period or constant-value expression that provides a number of seconds:

```
win:ext_timed_batch(timestamp_expression, time_period
 [,optional_reference_point])
```

```
win:ext_timed_batch(timestamp_expression, seconds_interval_expression
 [,optional_reference_point])
```

The externally-timed batch window takes a third, optional parameter that serves as a reference point to batch flush times. If not specified, the arrival of the first event into the batch window sets the reference point. Therefore if the reference point is not specified and the first event arrives at time t_1, then the batch flushes at time t_1 plus *time_period* and every *time_period* thereafter.

The key difference comparing the externally-timed batch window to the regular time batch window is that the window tumbles not based on the engine time, but strictly based on the result of the timestamp expression when evaluated against the events entering the window.

The algorithm underlying the view compares the timestamp value returned by the expression when the oldest event arrived with the timestamp value returned by the expression for the newest arriving event on event arrival. If the time interval between the timestamp values is larger then the timer period parameter, then the algorithm posts the current batch of events. The window therefore posts batches only when events arrive and only considers each event's timestamp property (or other expression value returned) and not engine time.

Note that using this view means that the engine keeps events in memory until the time is up: Consider your event arrival rate and determine if this is the behavior you want. Use context declaration or output rate limiting such as `output snapshot` as an alternative.

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 when event arrive with timestamps that indicate the start of a new batch:

```
select * from StockTickEvent.win:ext_timed_batch(timestamp, 5 sec)
```

The optional reference point is provided as a long-value of milliseconds relative to January 1, 1970 and time 00:00:00.

The following example statement sets the reference point to 5 seconds and the batch size to 1 hour, so that each batch output is 5 seconds after each hour:

```
select * from OrderSummaryEvent.win:ext_timed_batch(timestamp, 1 hour, 5000L)
```

The externally-timed data window expects strict ordering of the timestamp values returned by the timestamp expression. The view is not useful for ordering events in time order, please use the timeorder view instead.

On a related subject, engine time itself can be entirely under control of the application as described in *Section 14.8, “Controlling Time-Keeping”*, allowing control over all time-based aspects of processing in one place.

12.2.7. 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 an expression providing a number of seconds, and an expression for the number of events:

```
win:time_length_batch(time_period, number_of_events_expression)
```

```
win:time_length_batch(seconds_interval_expression, number_of_events_expression)
```

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
 seconds_interval_expression, number_of_events_expression,
    "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. When using this keyword the irstream keyword should be used in the select clause to ensure the remove stream is also output.

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")
```

12.2.8. 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 seconds-expression 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)
```

```
win:time_accum(seconds_interval_expression)
```

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.

12.2.9. 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.

12.2.10. First Length (`win:firstlength`)

The `firstlength` view retains the very first *size_expression* events.

The synopsis is:

```
win:firstlength(size_expression)
```

If used within a named window and an `on-delete` clause deletes events, the view accepts further arriving events until the number of retained events reaches the size of *size_expression*.

The below example creates a view that retains only the first 10 events:

```
select * from MyEvent.win:firstlength(10)
```

12.2.11. First Time (`win:firsttime`)

The `firsttime` view retains all events arriving within a given time interval after statement start.

The synopsis is:

```
win:firsttime(time_period)
```

```
win:firsttime(seconds_interval_expression)
```

The below example creates a view that retains only those events arriving within 1 minute and 10 seconds of statement start:

```
select * from MyEvent.win:firsttime(1 minute 10 seconds)
```

12.2.12. Expiry Expression (`win:expr`)

The `expr` view applies an expiry expression and removes events from the data window when the expression returns false.

Use this view to implement rolling and dynamically shrinking or expanding time, length or other windows. Rolling can, for example, be controlled based on event properties of arriving events, based on aggregation values or based on the return result of user-defined functions. Use this view to accumulate events until a value changes or other condition occurs based on arriving events or change of a variable value.

The synopsis is:

```
win:expr(expiry_expression)
```

The expiry expression can be any expression including expressions on event properties, variables, aggregation functions or user-defined functions. The view applies this expression to the oldest event(s) currently in the view, as described next.

When a new event arrives or when a variable value referenced by the expiry expression changes then the view applies the expiry expression starting from the oldest event in the data window. If the expiry expression returns false for the oldest event, the view removes the event from the data window. The view then applies the expression to the next oldest event. If the expiry expression returns true for the oldest event, no further evaluation takes place and the view indicates any new and expired events through insert and remove stream.

By using variables in the expiry expression it is possible to change the behavior of the view dynamically at runtime. When one or more variables used in the expression are updated the view evaluates the expiry expression starting from the oldest event.

Aggregation functions, if present in the expiry expression, are continuously updated as events enter and leave the data window. Use the grouped data window with this window to compute aggregations per group.

The engine makes the following built-in properties available to the expiry expression:

Table 12.3. Built-in Properties of the Expiry Expression Data Window View

Name	Type	Description
current_count	int	The number of events in the data window including the currently-arriving event.
expired_count	int	The number of events expired during this evaluation.
newest_event	(same event type as arriving events)	The last-arriving event itself.
newest_timestamp	long	The engine timestamp associated with the last-arriving event.
oldest_event	(same event type as arriving events)	The currently-evaluated event itself.
oldest_timestamp	long	The engine timestamp associated with the currently-evaluated event.
view_reference	Object	The object handle to this view.

This EPL declares an expiry expression that retains the last 2 events:

```
select * from MyEvent.win:expr(current_count <= 2)
```

The following example implements a dynamically-sized length window by means of a SIZE variable. As the SIZE variable value changes the view retains the number of events according to the current value of SIZE:

```
create variable int SIZE = 1000
```

```
select * from MyEvent.win:expr(current_count <= SIZE)
```

The next EPL retains the last 2 seconds of events:

```
select * from MyEvent.win:expr(oldest_timestamp > newest_timestamp - 2000)
```

The following example implements a dynamically-sized time window. As the SIZE millisecond variable value changes the view retains a time interval accordingly:

```
create variable long SIZE = 1000
```

```
select * from MyEvent.win:expr(newest_timestamp - oldest_timestamp < SIZE)
```

The following example declares a KEEP variable and flushes all events from the data window when the variable turns false:

```
create variable boolean KEEP = true
```

```
select * from MyEvent.win:expr(KEEP)
```

The next example specifies a rolling window that removes the oldest events from the window until the total price of all events in the window is less then 1000:

```
select * from MyEvent.win:expr(sum(price) < 1000)
```

This example retains all events that have the same value of the `flag` event property. When the `flag` value changes, the data window expires all events with the old `flag` value and retains only the most recent event of the new `flag` value:

```
select * from MyEvent.win:expr(newest_event.flag = oldest_event.flag)
```

12.2.12.1. Limitations

You may not use subqueries or the `prev` and `prior` functions as part of the expiry expression. Consider using a named window and `on-delete` or `on-merge` instead.

When using variables in the expiry expression, the thread that updates the variable does not evaluate the view. The thread that updates the variable instead schedules a reevaluation and view evaluates by timer execution.

12.2.13. Expiry Expression Batch (`win:expr_batch`)

The `expr_batch` view buffers events (tumbling window) and releases them when a given expiry expression returns true.

Use this view to implement dynamic or custom batching behavior, such as for dynamically shrinking or growing time, length or other batches, for batching based on event properties of arriving events, aggregation values or for batching based on a user-defined function.

The synopsis is:

```
win:expr_batch(expiry_expression, [include_triggering_event])
```

The expiry expression can be any expression including expressions on event properties, variables, aggregation functions or user-defined functions. The view applies this expression to arriving event(s), as described next.

The optional second parameter *include_triggering_event* defines whether to include the event that triggers the batch in the current batch (`true`, the default) or in the next batch (`false`).

When a new event arrives or when a variable value referenced by the expiry expression changes or when events get removed from the data window then the view applies the expiry expression. If the expiry expression returns true the data window posts the collected events as the insert stream and the last batch of events as remove stream.

By using variables in the expiry expression it is possible to change the behavior of the view dynamically at runtime. When one or more variables used in the expression are updated the view evaluates the expiry expression as well.

Aggregation functions, if present in the expiry expression, are continuously updated as events enter the data window and reset when the engine posts a batch of events. Use the grouped data window with this window to compute aggregations per group.

The engine makes the following built-in properties available to the expiry expression:

Table 12.4. Built-in Properties of the Expiry Expression Data Window View

Name	Type	Description
current_count	int	The number of events in the data window including the currently-arriving event.
newest_event	(same event type as arriving events)	The last-arriving event itself.
newest_timestamp	long	The engine timestamp associated with the last-arriving event.
oldest_event	(same event type as arriving events)	The currently-evaluated event itself.
oldest_timestamp	long	The engine timestamp associated with the currently-evaluated event.
view_reference	Object	The object handle to this view.

This EPL declares an expiry expression that posts event batches consisting of 2 events:

```
select * from MyEvent.win:expr_batch(current_count >= 2)
```

The following example implements a dynamically-sized length batch window by means of a SIZE variable. As the SIZE variable value changes the view accomulates and posts the number of events according to the current value of SIZE:

```
create variable int SIZE = 1000
```

```
select * from MyEvent.win:expr_batch(current_count >= SIZE)
```

The following example accumulates events until an event arrives that has a value of `postme` for property `myvalue`:

```
select * from MyEvent.win:expr_batch(myvalue = 'postme')
```

The following example declares a POST variable and posts a batch of events when the variable turns true:

```
create variable boolean POST = false
```

```
select * from MyEvent.win:expr_batch(POST)
```

The next example specifies a tumbling window that posts a batch of events when the total price of all events in the window is greater then 1000:

```
select * from MyEvent.win:expr_batch(sum(price) > 1000)
```

Specify the second parameter as `false` when you want the triggering event not included in the current batch.

This example batches all events that have the same value of the `flag` event property. When the `flag` value changes, the data window releases the batch of events collected for the old `flag` value. The data window collects the most recent event and the future arriving events of the same new `flag` value:

```
select * from MyEvent.win:expr_batch(newest_event.flag != oldest_event.flag,
 false)
```

12.2.13.1. Limitations

You may not use subqueries or the `prev` and `prior` functions as part of the expiry expression. Consider using a named window and `on-delete` or `on-merge` instead.

When using variables in the expiry expression, the thread that updates the variable does not evaluate the view. The thread that updates the variable instead schedules a reevaluation and view evaluates by timer execution.

12.3. Standard view set

12.3.1. Unique (`std:unique`)

The `unique` view is a view that includes only the most recent among events having the same value(s) for the result of the specified expression or list of expressions.

The synopsis is:

```
std:unique(unique_expression [, unique_expression ...])
```

The view acts as a length window of size 1 for each distinct value returned by an expression, or combination of values returned by multiple expressions. It thus posts as old events the prior event of the same value(s), if any.

An expression may return a `null` value. The engine treats a `null` value as any other value. An expression can also return a custom application object, whereby the application class should implement the `hashCode` and `equals` methods.

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)
```

When using `unique` the engine plans queries applying an implicit unique index, where applicable. Specify `@Hint('DISABLE_UNIQUE_IMPLICIT_IDX')` to force the engine to plan queries using a non-unique index.

12.3.2. Grouped Data Window (`std:groupwin`)

This view groups events into sub-views by the value returned by the specified expression or the combination of values returned by a list of expressions. The view takes a single expression to supply the group criteria values, or a list of expressions as parameters, as the synopsis shows:

```
std:groupwin(grouping_expression [, grouping_expression ...])
```

The *grouping_expression* expression(s) return one or more group keys, by which the view creates sub-views for each distinct group key. Note that the expression should not return an unlimited number of values: the grouping expression should not return a time value or otherwise unlimited key.

An expression may return a `null` value. The engine treats a `null` value as any other value. An expression can also return a custom application object, whereby the application class should implement the `hashCode` and `equals` methods.

Use `group by` instead of the grouped data window to control how aggregations are grouped.

A grouped data window with a length window of 1 is equivalent to the unique data window `std:unique`. The `std:unique` data window is the preferred notation:

```
select * from StockTickEvent.std:unique(symbol) // Prefer this
// ... equivalent to ...
select * from StockTickEvent.std:groupwin(symbol).win:length(1)
```

This example computes the total price for the last 5 events considering the last 5 events per each symbol, aggregating the price across all symbols (since no `group by` clause is specified the aggregation is across all symbols):

```
select symbol, sum(price) from StockTickEvent.std:groupwin(symbol).win:length(5)
```

The @Hint("`reclaim_group_aged=`*age_in_seconds*") hint instructs the engine to discard grouped data window state that has not been updated for *age_in_seconds* seconds. The optional @Hint("`reclaim_group_freq=`*sweep_frequency_in_seconds*") can be specified in addition to control the frequency at which the engine sweeps data window state. If the hint is not specified, the frequency defaults to the same value as *age_in_seconds*. Use the hints when your group criteria returns a changing or unlimited number of values. By default and without hints the view does not reclaim or remove data windows for group criteria values.

The updated sample statement with both hints:

```
// Remove data window views for symbols not updated for 10 seconds or more and
 sweep every 30 seconds
```

```
@Hint('reclaim_group_aged=10,reclaim_group_freq=30')
select symbol, sum(price) from StockTickEvent.std:groupwin(symbol).win:length(5)
```

Reclaim executes when an event arrives and not in the timer thread. In the example above reclaim can occur up to 40 seconds of engine time after the newest event arrives. Reclaim may affect iteration order for the statement and iteration order becomes indeterministic with reclaim.

To compute the total price for the last 5 events considering the last 5 events per each symbol and outputting a price per symbol, add the `group by` clause:

```
select symbol, sum(price) from StockTickEvent.std:groupwin(symbol).win:length(5)
 group by symbol
```

The `std:groupwin` grouped-window view can also take multiple expressions that provide values to group by. This example computes the total price for each symbol and feed for the last 10 events per symbol and feed combination:

```
select sum(price) from StockTickEvent.std:groupwin(symbol, feed).win:length(10)
```

The order in which the `std:groupwin` grouped-window 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.

Without the `std:groupwin` declaration query the same query returns the total price per symbol for only the last 10 events across all symbols. Here the engine allocates only one length window for all events:

```
select sum(price) from StockTickEvent.win:length(10)
```

We have learned that by placing the `std:groupwin` grouped-window 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:groupwin(symbol).win:length(10)` the engine allocates a new length window for each distinct symbol. However in `win:length(10)` alone the engine maintains a single length window.

The `std:groupwin` can be used with multiple data window views to achieve a grouped intersection or union policy.

The next query retains the last 4 events per symbol and only those events that are also not older then 10 seconds:

```
select * from StockTickEvent.std:groupwin(symbol).win:length(4).win:time(10)
```

Last, we consider a grouped data window for two group criteria. Here, the query results are total price per symbol and feed for the last 100 events per symbol and feed.

```
select sum(price) from StockTickEvent.std:groupwin(symbol, feed).win:length(100)
```

Note

A note on grouped time windows: When using grouped-window with time windows, note that whether the engine retains 5 minutes of events or retains 5 minutes of events per group, the result is the same from the perspective of retaining events as both policies retain, considering all groups, the same set of events. Therefore please specify the time window alone (ungrouped).

For example:

```
// Use this:
select sum(price) from StockTickEvent.win:time(1 minute)

// is equivalent to (don't use this):
//              select              sum(price)              from
 StockTickEvent.std:groupwin(symbol).win:time(1 minute)

// Use the group-by clause for grouping aggregation by symbol.
```

For advanced users: There is an optional view that can control how the `std:groupwin` grouped-window view gets evaluated and that view is the `std:merge` view. The merge view can only occur after a `std:groupwin` grouped-window view in a view chain and controls at what point in the view chain the merge of the data stream occurs from view-instance-per-criteria to single view.

Compare the following statements:

```
select * from Market.std:groupwin(ticker).win:length(1000000)
    .stat:weighted_avg(price, volume).std:merge(ticker)
// ... and ...
select * from Market.std:groupwin(ticker).win:length(1000000).std:merge(ticker)
    .stat:weighted_avg(price, volume)
```

If your statement does not specify the optional `std:merge` view, the semantics are the same as the first statement.

The first statement, in which the merge-view is added to the end (same as no merge view), computes weighted average per ticker, considering, per-ticker, the last 1M Market events for each ticker. The second statement, in which the merge view is added to the middle, computes weighted average considering, per-ticker, the last 1M Market events, computing the weighted average for all such events using a single view rather then multiple view instances with one view per ticker.

12.3.3. Size (`std:size`)

This view posts the number of events received from a stream or view plus any additional event properties or expression values listed as parameters. The synopsis is:

```
std:size([expression, ...] [ * ])
```

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). The view only posts output events when the `size` count changes and does not stay the same.

As optional parameters the view takes a list of expressions that the view evaluates against the last arriving event and provides along the `size` field. You may also provide the `*` wildcard selector to have the view output all event properties.

An alternative to receiving a data window event count is the `prevcount` function. Compared to the `std:size` view the `prevcount` function requires a data window while the `std:size` view does not. The related `count(...)` aggregation function provides a count per group when used with `group by`.

When combined with a data window view, the size view reports the current number of events in the data window in the insert stream and the prior number of events in the data window as the remove stream. This example reports the number of tick events within the last 1 minute:

```
select size from StockTickEvent.win:time(1 min).std:size()
```

To select additional event properties you may add each event property to output as a parameter to the view.

The next example selects the symbol and feed event properties in addition to the `size` property:

```
select size, symbol, feed from StockTickEvent.win:time(1 min).std:size(symbol,
 feed)
```

This example selects all event properties in addition to the `size` property:

```
select * from StockTickEvent.win:time(1 min).std:size(*)
```

The size view is also useful in conjunction with a `std:groupwin` grouped-window view to count the number of events per group. The EPL below returns the number of events per symbol.

```
select size from StockTickEvent.std:groupwin(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 occurrences of the pattern.

```
select       size       from       pattern[every       s=StockTickEvent       ->
 TradeEvent(symbol=s.symbol)].std:size()
```

12.3.4. 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 the last stock tick event for the symbol GE.

```
select * from StockTickEvent(symbol='GE').std:lastevent()
```

If you want to output the last event within a sliding window, please see *Section 9.1.9, “The Previous Function”*. That function accepts a relative (count) or absolute index and returns event properties or an event in the context of the specified data window.

12.3.5. First Event (`std:firstevent`)

This view retains only the first arriving event:

```
std:firstevent()
```

All events arriving after the first event are discarded.

If used within a named window and an `on-delete` clause deletes the first event, the view resets and will retain the next arriving event.

An example of a statement that retains the first `ReferenceData` event arriving is:

```
select * from ReferenceData.std:firstevent()
```

If you want to output the first event within a sliding window, please see *Section 9.1.9, “The Previous Function”*. That function accepts a relative (count) or absolute index and returns event properties or an event in the context of the specified data window.

12.3.6. First Unique (`std:firstunique`)

The `firstunique` view retains only the very first among events having the same value for the specified expression or list of expressions.

The synopsis is:

```
std:firstunique(unique_expression [, unique_expression ...])
```

If used within a named window and an `on-delete` clause deletes events, the view resets and will retain the next arriving event for the expression result value(s) of the deleted events.

The below example creates a view that retains only the first event per category:

```
select * from ReferenceData.std:firstunique(category)
```

When using `firstunique` the engine plans queries applying an implicit unique index, where applicable. Specify `@Hint('DISABLE_UNIQUE_IMPLICIT_IDX')` to force the engine to plan queries using a non-unique index.

12.4. Statistics views

The statistics views can be used combined with data window views or alone. Very similar to aggregation functions, these views aggregate or derive information from an event stream. As compared to aggregation functions, statistics views can post multiple derived fields including properties from the last event that was received. The derived fields and event properties are available for querying in the `where`-clause and are often compared to prior values using the `prior` function.

Statistics views accept one or more primary value expressions and any number of optional additional expressions that return values based on the last event received.

12.4.1. Univariate statistics (`stat:uni`)

This view calculates univariate statistics on a numeric expression. The view takes a single value expression as a parameter plus any number of optional additional expressions to return properties of the last event. The value expression must return a numeric value:

```
stat:uni(value_expression [,expression, ...] [ * ])
```

After the value expression you may optionally list additional expressions or event properties to evaluate for the stream and return their value based on the last arriving event. You may also provide the `*` wildcard selector to have the view output all event properties.

Table 12.5. 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
`stddev`	Sample standard deviation (square root of variance)
`stddevpa`	Population standard deviation

The below example selects the standard deviation on price for stock tick events for the last 10 events.

```
select stddev from StockTickEvent.win:length(10).stat:uni(price)
```

To add properties from the event stream you may simply add all additional properties as parameters to the view.

This example selects all of the derived values, based on the price property, plus the values of the symbol and feed event properties:

```
select * from StockTickEvent.win:length(10).stat:uni(price, symbol, feed)
```

The following example selects all of the derived values plus all event properties:

```
select * from StockTickEvent.win:length(10).stat:uni(price, symbol, *)
```

12.4.2. Regression (`stat:linest`)

This view calculates regression and related intermediate results on the values returned by two expressions. The view takes two value expressions as parameters plus any number of optional additional expressions to return properties of the last event. The value expressions must return a numeric value:

```
stat:linest(value_expression, value_expression [,expression, ...] [ * ])
```

After the two value expressions you may optionally list additional expressions or event properties to evaluate for the stream and return their value based on the last arriving event. You may also provide the `*` wildcard selector to have the view output all event properties.

Table 12.6. Regression derived properties

Property Name	Description
`slope`	Slope.
`YIntercept`	Y intercept.
`XAverage`	X average.
`XStandardDeviationPop`	X standard deviation population.
`XStandardDeviationSample`	X standard deviation sample.
`XSum`	X sum.
`XVariance`	X variance.
`YAverage`	X average.
`YStandardDeviationPop`	Y standard deviation population.
`YStandardDeviationSample`	Y standard deviation sample.
`YSum`	Y sum.
`YVariance`	Y variance.
`dataPoints`	Number of data points.
`n`	Number of data points.
`sumX`	Sum of X (same as X Sum).
`sumXSq`	Sum of X squared.
`sumXY`	Sum of X times Y.
`sumY`	Sum of Y (same as Y Sum).
`sumYSq`	Sum of Y squared.

The next example calculates regression and returns the 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)
```

To add properties from the event stream you may simply add all additional properties as parameters to the view.

This example selects all of the derived values, based on the price and offer properties, plus the values of the symbol and feed event properties:

```
select * from StockTickEvent.win:time(10 seconds).stat:linest(price, offer,
 symbol, feed)
```

The following example selects all of the derived values plus all event properties:

```
select * from StockTickEvent.win:time(10 seconds).stat:linest(price, offer, *)
```

12.4.3. Correlation (`stat:correl`)

This view calculates the correlation value on the value returned by two expressions. The view takes two value expressions as parameters plus any number of optional additional expressions to return properties of the last event. The value expressions must be return a numeric value:

```
stat:correl(value_expression, value_expression [,expression, ...] [ * ])
```

After the two value expressions you may optionally list additional expressions or event properties to evaluate for the stream and return their value based on the last arriving event. You may also provide the `*` wildcard selector to have the view output all event properties.

Table 12.7. Correlation derived properties

Property Name	Description
`correlation`	Correlation between two event properties

The next example calculates correlation on price and offer over all stock tick events for GE:

```
select correlation from StockTickEvent(symbol='GE').stat:correl(price, offer)
```

To add properties from the event stream you may simply add all additional properties as parameters to the view.

This example selects all of the derived values, based on the price and offer property, plus the values of the feed event property:

```
select * from StockTickEvent(symbol='GE').stat:correl(price, offer, feed)
```

The next example selects all of the derived values plus all event properties:

```
select * from StockTickEvent(symbol='GE').stat:correl(price, offer, *)
```

12.4.4. Weighted average (`stat:weighted_avg`)

This view returns the weighted average given an expression returning values to compute the average for and an expression returning weight. The view takes two value expressions as parameters plus any number of optional additional expressions to return properties of the last event. The value expressions must return numeric values:

```
stat:weighted_avg(value_expression_field, value_expression_weight
 [,expression, ...] [ * ])
```

After the value expression you may optionally list additional expressions or event properties to evaluate for the stream and return their value based on the last arriving event. You may also provide the `*` wildcard selector to have the view output all event properties.

Table 12.8. Weighted average derived properties

Property Name	Description
`average`	Weighted average

A statement that derives the volume-weighted average price for the last 3 seconds for a given symbol is shown below:

```
select average
from  StockTickEvent(symbol='GE').win:time(3  seconds).stat:weighted_avg(price,
 volume)
```

To add properties from the event stream you may simply add all additional properties as parameters to the view.

This example selects all of the derived values, based on the price and volume properties, plus the values of the symbol and feed event properties:

```
select *
```

```
from StockTickEvent.win:time(3 seconds).stat:weighted_avg(price, volume, symbol,
 feed)
```

The next example selects all of the derived values plus the values of all event properties:

```
select *
from StockTickEvent.win:time(3 seconds).stat:weighted_avg(price, volume, *)
```

Aggregation functions could instead be used to compute the weighted average as well. The next example also posts weighted average per symbol considering the last 3 seconds of stock tick data:

```
select symbol, sum(price*volume)/sum(volume)
from StockTickEvent.win:time(3 seconds) group by symbol
```

The following example computes weighted average keeping a separate data window per symbol considering the last 5 events of each symbol:

```
select symbol, average
from StockTickEvent.std:groupwin(symbol).win:length(5).stat:weighted_avg(price,
 volume)
```

12.5. Extension View Set

The views in this set are data windows that order events according to a criteria.

12.5.1. Sorted Window View (`ext:sort`)

This view sorts by values returned by the specified expression or list of expressions and keeps only the top (or bottom) events up to the given size.

This view retains all events in the stream that fall into the sort range. Use the ranked window as described next to retain events per unique key(s) and sorted.

The syntax is as follows:

```
ext:sort(size_expression,
    sort_criteria_expression [asc/desc][, sort_criteria_expression [asc/
desc]...])
```

An expression may be followed by the optional `asc` or `desc` keywords to indicate that the values returned by that expression are sorted in ascending or descending sort order.

The view below retains only those events that have the highest 10 prices considering all events (and not only the last event per symbol, see rank below) and reports a total price:

```
select sum(price) from StockTickEvent.ext:sort(10, price desc)
```

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(10, price desc, symbol asc)
```

The sorted window is often used with the `prev`, `prevwindow` or `prevtail` single-row functions to output properties of events at a certain position or to output the complete data window according to sort order.

Use the grouped window to retain a separate sort window for each group. For example, the views `std:groupwin(market).ext:sort(10, price desc)` instruct the engine to retain, per market, the highest 10 prices.

12.5.2. Ranked Window View (`ext:rank`)

This view retains only the most recent among events having the same value for the criteria expression(s), sorted by sort criteria expressions and keeps only the top events up to the given size.

This view is similar to the sorted window in that it keeps only the top (or bottom) events up to the given size, however the view also retains only the most recent among events having the same value(s) for the specified uniqueness expression(s).

The syntax is as follows:

```
ext:rank(unique_expression [, unique_expression ...],
    size_expression,
    sort_criteria_expression [asc/desc][, sort_criteria_expression [asc/
desc]...])
```

Specify the expressions returning unique key values first. Then specify a constant value that is the size of the ranked window. Then specify the expressions returning sort criteria values. The sort criteria expressions may be followed by the optional `asc` or `desc` keywords to indicate that the values returned by that expression are sorted in ascending or descending sort order.

The view below retains only those events that have the highest 10 prices considering only the last event per symbol and reports a total price:

```
select sum(price) from StockTickEvent.ext:rank(symbol, 10, price desc)
```

The following example retains, for the last event per market and symbol, those events that sort by price and quantity ascending into the first 10 ranks:

```
select * from StockTickEvent.ext:rank(market, symbol, 10, price, quantity)
```

The ranked window is often used with the `prev`, `prevwindow` or `prevtail` single-row functions to output properties of events at a certain position or to output the complete data window according to sort order.

This example outputs every 5 seconds the top 10 events according to price descending and considering only the last event per symbol:

```
select prevwindow(*) from StockTickEvent.ext:rank(symbol, 10, price desc)
  output snapshot every 5 seconds limit 1  // need only 1 row
```

Use the grouped window to retain a separate rank for each group. For example, the views `std:groupwin(market).ext:rank(symbol, 10, price desc)` instruct the engine to retain, per market, the highest 10 prices considering the last event per symbol.

12.5.3. Time-Order View (`ext:time_order`)

This view orders events that arrive out-of-order, using timestamp-values provided by an expression, and by comparing that timestamp value to engine system time.

The syntax for this view is as follows.

```
ext:time_order(timestamp_expression, time_period)
```

```
ext:time_order(timestamp_expression, seconds_interval_expression)
```

The first parameter to the view is the expression that supplies timestamp values. The timestamp is expected to be a long-typed millisecond value that denotes an event's time of consideration by the view (or other expression). This is typically the time of arrival. The second parameter is a number-of-seconds expression 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 timestamp values to engine time, the view requires that the 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 rstream 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 in both the `insert into` clause and the `select` clause to select ordered events only. It 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 or use a sensible expression that returns timestamp values for consideration by the view. The view also works well if you use externally-provided time via timer events.

Chapter 13. EPL Reference: Data Flow

13.1. Introduction

Data flows in Esper EPL have the following purposes:

1. Support for data flow programming and flow-based programming.
2. Declarative and runtime manageable integration of Esper input and output adapters that may be provided by EsperIO or by an application.
3. Remove the need to use an event bus achieving dataflow-only visibility of events and event types for performance gains.

Data flow operators communicate via streams of either underlying event objects or wrapped events. Underlying event objects are POJO, Map, Object-array or DOM/XML. Wrapped events are represented by `EventBean` instances that associate type information to underlying event objects.

For more information on data flow programming or flow-based programming please consult the *Wikipedia FBP Article* [http://en.wikipedia.org/wiki/Flow-based_programming].

Esper offers a number of useful built-in operators that can be combined in a graph to program a data flow. In addition EsperIO offers prebuilt operators that act as sources or sinks of events. An application can easily create and use its own data flow operators.

Using data flows an application can provide events to the data flow operators directly without using an engine's event bus. Not using an event bus (as represented by `EPRuntime.sendEvent`) can achieve performance gains as the engine does not need to match events to statements and the engine does not need to wrap underlying event objects in `EventBean` instances.

Data flows also allow for finer-grained control over threading, synchronous and asynchronous operation.

Note

Data flows are new in release 4.6 and may be subject to evolutionary change.

13.2. Usage

13.2.1. Overview

Your application declares a data flow using `create dataflow` *dataflow-name*. Declaring the data flow causes the EPL compiler to validate the syntax and some aspects of the data flow graph of operators. Declaring the data flow does not actually instantiate or execute a data flow. Resolving event types and instantiating operators (as required) takes place at time of data flow instantiation.

After your application has declared a data flow, it can instantiate the data flow and execute it. A data flow can be instantiated as many times as needed and each data flow instance can only be executed once.

The example EPL below creates a data flow that, upon execution, outputs the text `Hello World` to console and then ends.

```
create dataflow HelloWorldDataFlow
  BeaconSource -> helloworld.stream { text: 'hello world' , iterations: 1}
  LogSink(helloworld.stream) {}
```

The sample data flow above declares a `BeaconSource` operator parameterized by the "hello world" text and 1 iteration. The `->` keyword reads as *produces streams*. The `BeaconSource` operator produces a single stream named `helloworld.stream`. The `LogSink` operator receives this stream and prints it unformatted.

The next program code snippet declares the data flow to the engine:

```
String epl = "create dataflow HelloWorldDataFlow\n" +
  "BeaconSource -> helloworldStream { text: 'hello world' , iterations: 1}\n" +
  "LogSink(helloworldStream) {}";
epService.getEPAdministrator().createEPL(epl);
```

After declaring a data flow to an engine, your application can then instantiate and execute the data flow.

The following program code snippet instantiates the data flow:

```
EPDataFlowInstance instance =

 epService.getEPRuntime().getDataFlowRuntime().instantiate("HelloWorldDataFlow");
```

A data flow instance is represented by an `EPDataFlowInstance` object.

The next code snippet executes the data flow instance:

```
instance.run();
```

By using the `run` method of `EPDataFlowInstance` the engine executes the data flow using the same thread (blocking execute) and returns when the data flow completes. A data flow completes when all operators receive final markers.

The hello world data flow simply prints an unformatted `Hello World` string to console. Please check the built-in operator reference for `BeaconSource` and `LogSink` for more options.

13.2.2. Syntax

The synopsis for declaring a data flow is:

```
create dataflow name
 [schema_declarations]
 [operator_declarations]
```

After `create dataflow` follows the data flow name and a mixed list of event type (schema) declarations and operator declarations.

Schema declarations define an event type. Specify any number of `create schema` clauses as part of the data flow declaration followed by a comma character to end each schema declaration. The syntax for `create schema` is described in *Section 5.16, "Declaring an Event Type: Create Schema"*.

All event types that are defined as part of a data flow are private to the data flow and not available to other EPL statements. To define event types that are available across data flows and other EPL statements, use a `create schema` EPL statement, runtime or static configuration.

Annotations as well as expression declarations and scripts can also be pre-pended to the data flow declaration.

13.2.2.1. Operator Declaration

For each operator, declare the operator name, input streams, output streams and operator parameters.

The syntax for declaring a data flow operator is:

```
operator_name [(input_streams)]  [-> output_streams] {
  [parameter_name : parameter_value_expr] [, ...]
}
```

The operator name is an identifier that identifies an operator.

If the operator accepts input streams then those may be listed in parenthesis after the operator name, see *Section 13.2.2.2, "Declaring Input Streams"*.

If the operator can produce output streams then specify `->` followed by a list of output stream names and types. See *Section 13.2.2.3, "Declaring Output Streams"*.

Following the input and output stream declaration provide curly brackets (`{}`) containing operator parameters. See *Section 13.2.2.4, "Declaring Operator Parameters"*.

An operator that receives no input streams, produces no output streams and has no parameters assigned to it is shown in this EPL example data flow:

```
create dataflow MyDataFlow
  MyOperatorSimple {}
```

The next EPL shows a data flow that consists of an operator `MyOperator` that receives a single input stream `myInStream` and produces a single output stream `myOutStream` holding `MyEvent` events. The EPL configures the operator parameter `myParameter` with a value of 10:

```
create dataflow MyDataFlow
  create schema MyEvent as (id string, price double),
  MyOperator(myInStream) -> myOutStream<MyEvent> {
    myParameter : 10
  }
```

The next sections outline input stream, output stream and parameter assignment in greater detail.

13.2.2.2. Declaring Input Streams

In case the operator receives input streams, list the input stream names within parenthesis following the operator name. As part of the input stream declaration you may use the `as` keyword to assign an alias short name to one or multiple input streams.

The EPL shown next declares `myInStream` and assigns the alias `mis`:

```
create dataflow MyDataFlow
  MyOperator(myInStream as mis) {}
```

Multiple input streams can be listed separated by comma. We use the term *input port* to mean the ordinal number of the input stream in the order the input streams are listed.

The EPL as below declares two input streams and assigns an alias to each. The engine assigns `streamOne` to input port 0 (zero) and `streamTwo` to port 1.

```
create dataflow MyDataFlow
  MyOperator(streamOne as one, streamTwo as two) {}
```

You may assign multiple input streams to the same port and alias by placing the stream names into parenthesis. All input streams for the same port must have the same event type associated.

The next EPL statement declares an operator that receives input streams `streamA` and `streamB` both assigned to port 0 (zero) and alias `streamsAB`:

```
create dataflow MyDataFlow
  MyOperator( (streamA, streamB) as streamsAB) {}
```

Input and output stream names can have the dot-character in their name.

The following is also valid EPL:

```
create dataflow MyDataFlow
  MyOperator(my.in.stream) -> my.out.stream {}
```

Note

Reserved keywords may not appear in the stream name.

13.2.2.3. Declaring Output Streams

In case the operator produces output streams, list the output streams after the `->` keyword. Multiple output streams can be listed separated by comma. We use the term *output port* to mean the ordinal number of the output stream in the order the output streams are listed.

The sample EPL below declares an operator that produces two output streams `my.out.one` and `my.out.two`.

```
create dataflow MyDataFlow
  MyOperator -> my.out.one, my.out.two {}
```

Each output stream can be assigned optional type information within less/greater-then (<>). Type information is required if the operator cannot deduce the output type from the input type and the operator does not declare explicit output type(s). The event type name can either be an event type defined within the same data flow or an event type defined in the engine.

This EPL example declares an `RFIDSchema` event type based on an object-array event representation and associates the output stream `rfid.stream` with the `RFIDSchema` type. The stream `rfid.stream` therefore carries object-array (`Object[]`) typed objects according to schema `RFIDSchema`:

```
create dataflow MyDataFlow
  create objectarray schema RFIDSchema (tagId string, locX double, locY double),
  MyOperator -> rfid.stream<RFIDSchema> {}
```

The keyword `eventbean` is reserved: Use `eventbean`<*type-name*> to indicate that a stream carries `EventBean` instances of the given type instead of the underlying event object.

This EPL example declares an `RFIDSchema` event type based on an object-array event representation and associates the output stream `rfid.stream` with the event type, such that the stream `rfid.stream` carries `EventBean` objects:

```
create dataflow MyDataFlow
  create objectarray schema RFIDSchema (tagId string, locX double, locy double),
  MyOperator -> rfid.stream<eventbean<RFIDSchema>> {}
```

Use questionmark (`?`) to indicate that the type of events is not known in advance.

In the next EPL the stream `my.stream` carries `EventBean` instances of any type:

```
create dataflow MyDataFlow
  MyOperator -> my.stream<eventbean<?>> {}
```

13.2.2.4. Declaring Operator Parameters

Operators can receive constants, objects, EPL expressions and complete EPL statements as parameters. All parameters are listed within curly brackets (`{}`) after input and output stream declarations. Curly brackets are required as a separator even if the operator has no parameters.

The syntax for parameters is:

```
name : value_expr [,...]
```

The parameter name is an identifier that is followed by the colon (`:`) or equals (`=`) character and a value expression. A value expression can be any expression, system property, JSON notation object or EPL statement. Parameters are separated by comma character.

The next EPL demonstrates operator parameters that are scalar values:

```
create dataflow MyDataFlow
  MyOperator {
    stringParam : 'sample',
    secondString : "double-quotes are fine",
    intParam : 10
  }
```

Operator parameters can be any EPL expression including expressions that use variables. Subqueries, aggregations and the `prev` and `prior` functions cannot be applied here.

The EPL shown below lists operator parameters that are expressions:

```
create dataflow MyDataFlow
  MyOperator {
    intParam : 24*60*60,
    threshold : var_threshold // a variable defined in the engine
  }
```

To obtain the value of a system property, the special `systemProperties` property name is reserved for access to system properties.

The following EPL sets operator parameters to a value obtained from a system property:

```
create dataflow MyDataFlow
  MyOperator {
    someSystemProperty : systemProperties('mySystemProperty')
  }
```

Any JSON value can also be used as a value. Use square brackets `[]` for JSON arrays. Use curly brackets `{}` to hold nested Map or other object values. Provide the special `class` property to instantiate a given instance by class name. The engine populates the respective array, Map or Object as specified in the JSON parameter value.

The below EPL demonstrates operator parameters that are JSON values:

```
create dataflow MyDataFlow
  MyOperator {
    myStringArray: ['a', "b"],
    myMapOrObject: {
      a : 10,
      b : 'xyz',
    },
    myInstance: {
      class: 'com.myorg.myapp.MyImplementation',
      myValue : 'sample'
    }
  }
```

The special parameter name `select` is reserved for use with EPL select statements. Please see the `Select` built-in operator for an example.

13.3. Built-in Operators

The below table summarizes the built-in data flow operators (Esper only) available:

Table 13.1. Esper Built-in Operators

Operator	Description
BeaconSource	Utility source that generates events. See *Section 13.3.1, “BeaconSource”*.
Emitter	Special operator for injecting events into a stream. See *Section 13.4.5, “Start Captive”*.
EPStatementSource	One or more EPL statements act as event sources. See *Section 13.3.2, “EPStatementSource”*.
EventBusSink	The event bus is the sink: Sends events from the data flow into the event bus. See *Section 13.3.3, “EventBusSink”*.
EventBusSource	The event bus is the source: Receives events from the event bus into the data flow. See *Section 13.3.4, “EventBusSource”*.
Filter	Filters an input stream and produces an output stream containing the events passing the filter criteria. See *Section 13.3.5, “Filter”*.
LogSink	Utility sink that outputs events to console or log. See *Section 13.3.6, “LogSink”*.
Select	An EPL select statement that executes on the input stream events. See *Section 13.3.7, “Select”*.

The below table summarizes the built-in EsperIO data flow operators. Please see the EsperIO documentation and source for more information.

Table 13.2. EsperIO Built-in Operators

Operator	Description
AMQPSource	Attaches to AMQP broker to receive messages to process.
AMQPSink	Attaches to AMQP broker to send messages.
FileSource	Reads one or more files and produces events from file data.
FileSink	Write one or more files from events received.

13.3.1. BeaconSource

The BeaconSource operator generates events and populates event properties.

The BeaconSource operator does not accept any input streams and has no input ports.

The BeaconSource operator must have a single output stream. When the BeaconSource operator completed generating events according to the number of iterations provided or when it is cancelled it outputs a final marker to the output stream.

Parameters for the BeaconSource operator are all optional parameters:

Table 13.3. BeaconSource Parameters

Name	Description
initialDelay	Specifies the number of seconds delay before producing events.
interval	Time interval between events. Takes a integer or double-typed value for the number of seconds. The interval is zero when not provided.
iterations	Number of events produced. Takes an integer value. When not provided the operator produces tuples until the data flow instance gets cancelled.

Event properties to be populated can simply be added to the parameters.

If your declaration provides an event type for the output stream then BeaconSource will populate event properties of the underlying events. If no event type is specified, BeaconSource creates an anonymous object-array event type to carry the event properties that are generated and associates this type with its output stream.

Examples are:

```
create dataflow MyDataFlow
  create schema SampleSchema(tagId string, locX double), // sample type

  // BeaconSource that produces empty object-array events without delay
  // or interval until cancelled.
  BeaconSource -> stream.one {}

  // BeaconSource that produces one RFIDSchema event populating event properties
  // from a user-defined function "generateTagId" and the provided values.
  BeaconSource -> stream.two<SampleSchema> {
    iterations : 1,
    tagId : generateTagId(),
    locX : 10
  }

  // BeaconSource that produces 10 object-array events populating
  // the price property with a random value.
  BeaconSource -> stream.three {
    iterations : 1,
    interval : 10, // every 10 seconds
    initialDelay : 5, // start after 5 seconds
    price : Math.random() * 100
  }
```

13.3.2. EPStatementSource

The EPStatementSource operator maintains a subscription to the results of one or more EPL statements. The operator produces the statement output events.

The EPStatementSource operator does not accept any input streams and has no input ports.

The EPStatementSource operator must have a single output stream. It does not generate a final or other marker.

Either the statement name or the statement filter parameter is required:

Table 13.4. EPStatementSource Parameters

Name	Description
collector	Optional parameter, used to transform statement output events to submitted events.
statementName	Name of the statement that produces events. The statement does not need to exist at the time of data flow instantiation.
statementFilter	Implementation of the `EPDataFlowEPStatementFilter` that returns true for each statement that produces events. Statements do not need to exist at the time of data flow instantiation.

If a statement name is provided, the operator subscribes to output events of the statement if the statement exists or when it gets created at a later point in time.

If a statement filter is provided instead, the operator subscribes to output events of all statements that currently exist and pass the filter `pass` method or that get created at a later point in time and pass the filter `pass` method.

The `collector` can be specified to transform output events. If no collector is specified the operator submits the underlying events of the insert stream received from the statement. The collector object must implement the interface `EPDataFlowIRStreamCollector`.

Examples are:

```
create dataflow MyDataFlow
  create schema SampleSchema(tagId string, locX double), // sample type

  // Consider only the statement named MySelectStatement when it exists.
  // No transformation.
  EPStatementSource -> stream.one<eventbean<?>> {
    statementName : 'MySelectStatement'
  }

  // Consider all statements that match the filter object provided.
  // No transformation.
  EPStatementSource -> stream.two<eventbean<?>> {
    statementFilter : {
      class : 'com.mycompany.filters.MyStatementFilter'
    }
  }

  // Consider all statements that match the filter object provided.
  // With collector that performs transformation.
  EPStatementSource -> stream.two<SampleSchema> {
    collector : {
      class : 'com.mycompany.filters.MyCollector'
    },
    statementFilter : {
      class : 'com.mycompany.filters.MyStatementFilter'
    }
  }
```

13.3.3. EventBusSink

The EventBusSink operator send events received from a data flow into the event bus. Any statement that looks for any of the events gets triggered, equivalent to `EPRuntime.sendEvent` or the `insert into` clause.

The EventBusSink operator accepts any number of input streams. The operator forwards all events arriving on any input ports to the event bus, equivalent to `EPRuntime.sendEvent`.

The EventBusSink operator cannot declare any output streams.

Parameters for the EventBusSink operator are all optional parameters:

Table 13.5. EventBusSink Parameters

Name	Description
collector	Optional parameter, used to transform data flow events to event bus events.

The `collector` can be specified to transform data flow events to event bus events. If no collector is specified the operator submits the events directly to the event bus. The collector object must implement the interface `EPDataFlowEventCollector`.

Examples are:

```
create dataflow MyDataFlow
  BeaconSource -> instream<SampleSchema> {}  // produces a sample stream

  // Send SampleSchema events produced by beacon to the event bus.
  EventBusSink(instream) {}

  // Send SampleSchema events produced by beacon to the event bus.
  // With collector that performs transformation.
  EventBusSink(instream) {
    collector : {
      class : 'com.mycompany.filters.MyCollector'
    }
  }
```

13.3.4. EventBusSource

The EventBusSource operator receives events from the event bus and produces an output stream of the events received. With the term event bus we mean any event visible to the engine either because the application send the event via `EPRuntime.sendEvent` or because statements populated streams as a result of `insert into`.

The EventBusSource operator does not accept any input streams and has no input ports.

The EventBusSource operator must have a single output stream. It does not generate a final or other marker. The event type declared for the output stream is the event type of events received from the event bus.

All parameters to EventBusSource are optional:

Table 13.6. EventBusSource Parameters

Name	Description
collector	Optional parameter and used to transform event bus events to submitted events.
filter	Filter expression for event bus matching.

The `collector` can be specified to transform output events. If no collector is specified the operator submits the underlying events of the stream received from the event bus. The collector object must implement the interface `EPDataFlowEventBeanCollector`.

The `filter` is an expression that the event bus compiles and efficiently matches even in the presence of a large number of event bus sources. The filter expression must return a boolean-typed value, returning true for those events that the event bus passes to the operator.

Examples are:

```
create dataflow MyDataFlow

  // Receive all SampleSchema events from the event bus.
  // No transformation.
  EventBusSource -> stream.one<SampleSchema> {}

  // Receive all SampleSchema events with tag id '001' from the event bus.
  // No transformation.
  EventBusSource -> stream.one<SampleSchema> {
    filter : tagId = '001'
  }

  // Receive all SampleSchema events from the event bus.
  // With collector that performs transformation.
  EventBusSource -> stream.two<SampleSchema> {
    collector : {
      class : 'com.mycompany.filters.MyCollector'
    },
  }
```

13.3.5. Filter

The Filter operator filters an input stream and produces an output stream containing the events passing the filter criteria. If a second output stream is provided, the operator sends events not passing filter criteria to that output stream.

The Filter operator accepts a single input stream.

The Filter operator requires one or two output streams. The event type of the input and output stream(s) must be the same. The first output stream receives the matching events according to the filter expression. If declaring two output streams, the second stream receives non-matching events.

The Filter operator has a single required parameter:

Table 13.7. Filter Parameters

Name	Description
filter	The filter criteria expression.

Examples are:

```
create dataflow MyDataFlow
  create schema SampleSchema(tagId string, locX double), // sample type
  BeaconSource -> samplestream<SampleSchema> {}  // sample source

  // Filter all events that have a tag id of '001'
  Filter(samplestream) -> tags_001 {
    filter : tagId = '001'
  }

  // Filter all events that have a tag id of '001',
  // putting all other events into the second stream
  Filter(samplestream) -> tags_001, tags_other {
    filter : tagId = '001'
  }
```

13.3.6. LogSink

The LogSink operator outputs events to console or log file in either a JSON, XML or built-in format (the default).

The LogSink operator accepts any number of input streams. All events arriving on any input ports are logged.

The LogSink operator cannot declare any output streams.

Parameters for the LogSink operator are all optional parameters:

Table 13.8. LogSink Parameters

Name	Description
format	Specify format as a string value: `json` for JSON-formatted output, `xml` for XML-formatted output and `summary` (default) for a built-in format.
layout	Pattern string according to which output is formatted. Place `%df` for data flow name, `%p` for port number, `%i` for data flow instance id, `%t` for title, `%e` for event data.
log	Boolean true (default) for log output, false for console output.
linefeed	Boolean true (default) for line feed, false for no line feed.
title	String title text pre-pended to output.

Examples are:

```
create dataflow MyDataFlow
  BeaconSource -> instream {}  // produces sample stream to use below

  // Output textual event to log using defaults.
  LogSink(instream) {}

  // Output JSON-formatted to console.
  LogSink(instream) {
    format : 'json',
    layout : '%t [%e]',
    log : false,
    linefeed : true,
    title : 'My Custom Title:'
  }
```

13.3.7. Select

The Select operator is configured with an EPL select statement. It applies events from input streams to the select statement and outputs results either continuously or when the final marker arrives.

The Select operator accepts one or more input streams.

The Select operator requires a single output stream.

The Select operator requires the `select` parameter, all other parameters are optional:

Table 13.9. Select Operator Parameters

Name	Description
iterate	Boolean indicator whether results should be output continuously or only upon arrival of the final marker.
select	EPL `select` statement in parenthesis.

Set the optional `iterate` flag to false (the default) to have the operator output results continuously. Set the `iterate` flag to true to indicate that the operator outputs results only when the final marker arrives. If `iterate` is true then output rate limiting clauses are not supported.

The `select` parameter is required and provides an EPL select statement within parenthesis. For each input port the statement should list the input stream name or the alias name in the `from` clause. Only filter-based streams are allowed in the `from` clause and patterns or named windows are not supported. Also not allowed are the `insert into` clause, the `irstream` keyword and subselects.

The Select operator determines the event type of output events based on the `select` clause. It is not necessary to declare an event type for the output stream.

Examples are:

```
create dataflow MyDataFlow
  create schema SampleSchema(tagId string, locX double), // sample type
  BeaconSource -> instream<SampleSchema> {}  // sample stream
  BeaconSource -> secondstream<SampleSchema> {}  // sample stream

  // Simple continuous count of events
  Select(instream) -> outstream {
    select: (select count(*) from instream)
  }

  // Demonstrate use of alias
  Select(instream as myalias) -> outstream {
    select: (select count(*) from myalias)
  }

  // Output only when the final marker arrives
  Select(instream as myalias) -> outstream {
    select: (select count(*) from myalias),
    iterate: true
  }
  // Same input port for the two sample streams
  Select( (instream, secondstream) as myalias) -> outstream {
    select: (select count(*) from myalias)
  }

  // A join with multiple input streams,
  // joining the last event per stream forming pairs
  Select(instream, secondstream) -> outstream {
    select: (select a.tagId, b.tagId
        from instream.std:lastevent() as a, secondstream.std:lastevent() as b)
  }

  // A join with multiple input streams and using aliases.
  Select(instream as S1, secondstream as S2) -> outstream {
    select: (select a.tagId, b.tagId
        from S1.std:lastevent() as a, S2.std:lastevent() as b)
  }
```

13.4. API

This section outlines the steps to declare, instantiate, execute and cancel or complete data flows.

13.4.1. Declaring a Data Flow

Use the `createEPL` and related `create` methods on `EPAdministrator` to declare a data flow or the deployment admin API. The `EPStatementObjectModel` statement object model can also be used to declare a data flow.

Annotations that are listed at the top of the EPL text are applied to all EPL statements and operators in the data flow. Annotations listed for a specific operator apply to that operator only.

The next program code snippet declares a data flow to the engine:

```
String epl = "@Name('MyStatementName') create dataflow HelloWorldDataFlow\n" +
  "BeaconSource -> helloworldStream { text: 'hello world' , iterations: 1}\n" +
  "LogSink(helloworldStream) {}";
EPStatement stmt = epService.getEPAdministrator().createEPL(epl);
```

The statement name that can be assigned to the statement is used only for statement management. Your application may stop and/or destroy the statement declaring the data flow thereby making the data flow unavailable for instantiation. Existing instances of the data flow are not affected by a stop or destroy of the statement that declares the data flow (example: `stmt.destroy()`).

Listeners or the subscriber to the statement declaring a data flow receive no events or other output. The statement declaring a data flow returns no rows when iterated.

13.4.2. Instantiating a Data Flow

The `com.espertech.esper.client.dataflow.EPDataFlowRuntime` available via `getDataFlowRuntime` on `EPRuntime` manages declared data flows.

Use the `instantiate` method on `EPDataFlowRuntime` to instantiate a data flow after it has been declared. Pass the data flow name and optional instantiation options to the method. A data flow can be instantiated any number of times.

A data flow instance is represented by an instance of `EPDataFlowInstance`. Each instance has a state as well as methods to start, run, join and cancel as well as methods to obtain execution statistics.

Various optional arguments including operator parameters can be passed to `instantiate` via the `EPDataFlowInstantiationOptions` object as explained in more detail below.

The following code snippet instantiates the data flow:

```
EPDataFlowInstance instance =

 epService.getEPRuntime().getDataFlowRuntime().instantiate("HelloWorldDataFlow");
```

The engine does not track or otherwise retain data flow instances in memory. It is up to your application to retain data flow instances as needed.

Each data flow instance associates to a state. The start state is `EPDataFlowState.INSTANTIATED`. The end state is either `COMPLETED` or `CANCELLED`.

The following table outlines all states:

Table 13.10. Data Flow Instance States

State	Description
INSTANTIATED	Start state, applies when a data flow instance has been instantiated and has not executed.
RUNNING	A data flow instance transitions from instantiated to running when any of the `start`, `run` or `startCaptive` methods are invoked.
COMPLETED	A data flow instance transitions from running to completed when all final markers have been processed by all operators.
CANCELLED	A data flow instance transitions from running to cancelled when your application invokes the `cancel` method on the data flow instance.

13.4.3. Executing a Data Flow

After your application instantiated a data flow instance it can execute the data flow instance using either the `start`, `run` or `startCaptive` methods.

Use the `start` method to have the engine allocate a thread for each source operator. Execution is non-blocking. Use the `join` method to have one or more threads join a data flow instance execution.

Use the `run` method to have the engine use the current thread to execute the single source operator. Multiple source operators are not allowed when using `run`.

Use the `startCaptive` method to have the engine return all `Runnable` instances and emitters, for the purpose of having complete control over execution. The engine allocates no threads and does not perform any logic for the data flow unless your application employs the `Runnable` instances and emitters returned by the method.

The next code snippet executes the data flow instance as a blocking call:

```
instance.run();
```

By using the `run` method of `EPDataFlowInstance` the engine executes the data flow instance using the same thread (blocking execute) and returns when the data flow instance completes. A data flow instance completes when all operators receive final markers.

The hello world data flow simply prints an unformatted `Hello World` string to console. The `BeaconSource` operator generates a final marker when it finishes the 1 iteration. The data flow instance thus transitions to complete after the `LogSink` operator receives the final marker, and the thread invoking the `run` method returns.

The next code snippet executes the data flow instance as a non-blocking call:

```
instance.start();
```

Use the `cancel` method to cancel execution of a running data flow instance:

```
instance.cancel();
```

Use the `join` method to join execution of a running data flow instance, causing the joining thread to block until the data flow instance either completes or is cancelled:

```
instance.join();
```

13.4.4. Instantiation Options

The `EPDataFlowInstantiationOptions` object that can be passed to the `instantiate` method may be used to customize the operator graph, operator parameters and execution of the data flow instance.

Passing runtime parameters to data flow operators is easiest using the `addParameterURI` method. The first parameter is the data flow operator name and the operator parameter name separated by the slash character. The second parameter is the value object.

For example, in order to pass the file name to the `FileSource` operator at runtime, use the following code:

```
EPDataFlowInstantiationOptions options = new EPDataFlowInstantiationOptions();
options.addParameterURI("FileSource/file", filename);
EPDataFlowInstance instance = epService.getEPRuntime().getDataFlowRuntime()
    .instantiate("MyFileReaderDataFlow",options);
instance.run();
```

The optional `operatorProvider` member takes an implementation of the `EPDataFlowOperatorProvider` interface. The engine invokes this provider to obtain operator instances.

The optional `parameterProvider` member takes an implementation of the `EPDataFlowOperatorParameterProvider` interface. The engine invokes this provider to obtain operator parameter values. The values override the values provided via parameter URI above.

The optional `exceptionHandler` member takes an implementation of the `EPDataFlowExceptionHandler` interface. The engine invokes this provider to when exceptions occur.

The optional `dataFlowInstanceId` can be assigned any string value for the purpose of identifying the data flow instance.

The optional `dataFlowInstanceUserObject` can be assigned any object value for the purpose of associating a user object to the data flow instance.

Set the `operatorStatistics` flag to true to obtain statistics for operator execution.

Set the `cpuStatistics` flag to true to obtain CPU statistics for operator execution.

13.4.5. Start Captive

Use the `startCaptive` method on a `EPDataFlowInstance` data flow instance when your application requires full control over threading. This method returns an `EPDataFlowInstanceCaptive` instance that contains a list of `java.lang.Runnable` instances that represent each source operator.

The special `Emitter` operator can occur in a data flow. This emitter can be used to inject events into the data flow without writing a new operator. Emitter takes a single `name` parameter that provides the name of the emitter and that is returned in a map of emitters by `EPDataFlowInstanceCaptive`.

The example EPL below creates a data flow that uses emitter.

```
create dataflow HelloWorldDataFlow
  create objectarray schema SampleSchema(text string), // sample type

  Emitter -> helloworld.stream<SampleSchema> { name: 'myemitter' }
  LogSink(helloworld.stream) {}
```

Your application may obtain the Emitter instance and sends events directly into the output stream. This feature is only supported in relationship with `startCaptive` since the engine does not allocate any threads or run source operators.

The example code snippet below obtains the emitter instance and send events directly into the data flow instance:

```
EPDataFlowInstance instance =



 options);
EPDataFlowInstanceCaptive captiveStart = instance.startCaptive();
Emitter emitter = captiveStart.getEmitters().get("myemitter");
emitter.submit(new Object[] {"this is some text"});
```

When emitting DOM XML events please emit the root element obtained from `document.getDocumentElement()`.

13.4.6. Data Flow Punctuation with Markers

When your application executes a data flow instance by means of the `start` (non-blocking) or `run` (blocking) methods, the data flow instance stays running until either completed or cancelled. While cancellation is always via the `cancel` method, completion occurs when all source operators provide final markers.

The final marker is an object that implements the `EPDataFlowSignalFinalMarker` interface. Some operators may also provide or process data window markers which implement the `EPDataFlowSignalWindowMarker` interface. All such signals implement the `EPDataFlowSignal` interface.

Some source operators such as `EventBusSource` and `EPStatementSource` do not generate final markers as they act continuously.

13.4.7. Exception Handling

All exceptions during the execution of a data flow are logged and reported to the `EPDataFlowExceptionHandler` instance if one was provided.

If no exception handler is provided or the provided exception handler re-throws or generates a new runtime exception, the source operator handles the exception and completes (ends). When all source operators complete then the data flow instance transitions to complete.

13.5. Examples

The following example is a rolling top words count implemented as a data flow, over a 30 second time window and providing the top 3 words every 2 seconds:

```
create dataflow RollingTopWords
  create objectarray schema WordEvent (word string),

  Emitter -> wordstream<WordEvent> {name:'a'} {} // Produces word stream

  Select(wordstream) -> wordcount { // Sliding time window count per word
    select: (select word, count(*) as wordcount
          from wordstream.win:time(30) group by word)
  }

  Select(wordcount) -> wordranks { // Rank of words
    select: (select window(*) as rankedWords
          from wordcount.ext:sort(3, wordcount desc)
          output snapshot every 2 seconds)
  }

  LogSink(wordranks) {}
```

The next example implements a bargain index computation that separates a mixed trade and quote event stream into a trade and a quote stream, computes a vwap and joins the two streams to compute an index:

```
create dataflow VWAPSample
  create objectarray schema TradeQuoteType as (type string, ticker string, price
 double, volume long, askprice double, asksize long),

  MyObjectArrayGraphSource -> TradeQuoteStream<TradeQuoteType> {}

  Filter(TradeQuoteStream) -> TradeStream {
    filter: type = "trade"
  }
  Filter(TradeQuoteStream) -> QuoteStream {
    filter: type = "quote"
  }

  Select(TradeStream) -> VwapTrades {
    select: (select ticker, sum(price * volume) / sum(volume) as vwap,
          min(price) as minprice
          from TradeStream.std:groupwin(ticker).win:length(4) group by ticker)
  }

  Select(VwapTrades as T, QuoteStream as Q) -> BargainIndex {
    select:
     (select case when vwap > askprice then asksize * (Math.exp(vwap - askprice))
 else 0.0d end as index
      from T.std:unique(ticker) as t, Q.std:lastevent() as q
      where t.ticker = q.ticker)
  }

  LogSink(BargainIndex) {}
```

The final example is a word count data flow, in which three custom operators tokenize, word count and aggregate. The custom operators in this example are discussed next.

```
create dataflow WordCount
  MyLineFeedSource -> LineOfTextStream {}
  MyTokenizerCounter(LineOfTextStream) -> SingleLineCountStream {}
  MyWordCountAggregator(SingleLineCountStream) -> WordCountStream {}
  LogSink(WordCountStream) {}
```

13.6. Operator Implementation

This section discusses how to implement classes that serve as operators in a data flow. The section employs the example data flow as shown earlier.

This example data flow has operators `MyLineFeedSource`, `MyTokenizerCounter` and `MyWordCountAggregator` that are application provided operators:

```
create dataflow WordCount
  MyLineFeedSource -> LineOfTextStream {}
  MyTokenizerCounter(LineOfTextStream) -> SingleLineCountStream {}
  MyWordCountAggregator(SingleLineCountStream) -> WordCountStream {}
  LogSink(WordCountStream) {}
```

In order to resolve application operators, add the package or operator class to imports:

```
// Sample code adds 'package.*' to simply import the package.
epService.getEPAdministrator().getConfiguration()
  .addImport(MyTokenizerCounter.class.getPackage().getName() + ".*");
```

13.6.1. Sample Operator Acting as Source

The implementation class must implement the `DataFlowSourceOperator` interface.

The implementation for the sample `MyLineFeedSource` with comments is:

```
// The OutputTypes annotation can be used to specify the type of events
// that are output by the operator.
// If provided, it is not necessary to declare output types in the data flow.
// The event representation is object-array.
@OutputTypes(value = {
    @OutputType(name = "line", typeName = "String")
    })

// Provide the DataFlowOpProvideSignal annotation to indicate that
// the source operator provides a final marker.
@DataFlowOpProvideSignal

public class MyLineFeedSource implements DataFlowSourceOperator {

     // Use the DataFlowContext annotation to indicate the field that receives
 the emitter.
     // The engine provides the emitter.
    @DataFlowContext
    private EPDataFlowEmitter dataFlowEmitter;

    // Mark a parameter using the DataFlowOpParameter annotation
    @DataFlowOpParameter
    private String myStringParameter;

    private final Iterator<String> lines;

    public MyLineFeedSource(Iterator<String> lines) {
        this.lines = lines;
    }

    // Invoked by the engine at time of data flow instantiation.
     public DataFlowOpInitializeResult initialize(DataFlowOpInitializateContext
 context) throws Exception {
        return null; // can return type information here instead
    }
    // Invoked by the engine at time of data flow instante execution.
    public void open(DataFlowOpOpenContext openContext) {
      // attach to input
    }

    // Invoked by the engine in a tight loop.
    // Submits the events which contain lines of text.
    public void next() {
        // read and submit events
        if (lines.hasNext()) {
            dataFlowEmitter.submit(new Object[] {lines.next()});
        }
        else {
            dataFlowEmitter.submitSignal(new EPDataFlowSignalFinalMarker() {});
        }
    }

    // Invoked by the engine at time of cancellation or completion.
    public void close(DataFlowOpCloseContext openContext) {
      // detach from input
    }
}
```

13.6.2. Sample Tokenizer Operator

The implementation for the sample `MyTokenizerCounter` with comments is:

```
// Annotate with DataFlowOperator so the engine knows its a data flow operator
@DataFlowOperator

@OutputTypes({
    @OutputType(name = "line", type = int.class),
    @OutputType(name = "wordCount", type = int.class),
    @OutputType(name = "charCount", type = int.class)
    })

public class MyTokenizerCounter {

    @DataFlowContext
    private EPDataFlowEmitter dataFlowEmitter;

    // Name the method that receives data onInput(...)
    public void onInput(String line) {
        // tokenize
        StringTokenizer tokenizer = new StringTokenizer(line, " \t");
        int wordCount = tokenizer.countTokens();
        int charCount = 0;
        while(tokenizer.hasMoreTokens()) {
            String token = tokenizer.nextToken();
            charCount += token.length();
        }

        // submit count of line, words and characters
        dataFlowEmitter.submit(new Object[] {1, wordCount, charCount});
    }
}
```

13.6.3. Sample Aggregator Operator

The implementation for the sample `MyWordCountAggregator` with comments is:

```
@DataFlowOperator

@OutputTypes(value = {
    @OutputType(name = "stats", type = MyWordCountStats.class)
    })

public class MyWordCountAggregator {

    @DataFlowContext
    private EPDataFlowEmitter dataFlowEmitter;

    private final MyWordCountStats aggregate = new MyWordCountStats();

    public void onInput(int lines, int words, int chars) {
        aggregate.add(lines, words, chars);
    }

    // Name the method that receives a marker onSignal
    public void onSignal(EPDataFlowSignal signal) {
        // Received puntuation, submit aggregated totals
        dataFlowEmitter.submit(aggregate);
    }
}
```

Chapter 14. API Reference

14.1. API Overview

Esper has the following primary interfaces:

- The event and event type interfaces are described in *Section 14.6, “Event and Event Type”*.
- The administrative interface to create and manage EPL and pattern statements, and set runtime configurations, is described in *Section 14.3, “The Administrative Interface”*.
- The runtime interface to send events into the engine, set and get variable values and execute on-demand queries, is described in *Section 14.4, “The Runtime Interface”*.

For EPL introductory information please see *Section 5.1, “EPL Introduction”* and patterns are described at *Section 6.1, “Event Pattern Overview”*.

The JavaDoc documentation is also a great source for API information.

14.2. The Service Provider Interface

The `EPServiceProvider` interface represents an engine instance. 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 providerURI)` methods return an instance of the Esper engine. The latter can be used to obtain multiple instances of the engine for different provider URI values. The `EPServiceProviderManager` determines if the provider URI matches all prior provider URI values and returns the same engine instance for the same provider URI value. If the provider 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 the provider URI `RFIDProcessor1`. Subsequent calls to get an engine with the same provider URI return the same instance.

```
EPServiceProvider                              epService                              =
 EPServiceProviderManager.getProvider("RFIDProcessor1");
```

Since the getProvider methods return the same cached engine instance for each URI, there is no need to statically cache an engine instance in your application.

An existing Esper engine instance can be reset via the initialize method on the EPServiceProvider instance. This operation stops and removes all statements and resets the engine to the configuration provided when the engine instance for that URI was obtained. If no configuration is provided, an empty (default) configuration applies.

After initialize your application must obtain new administrative and runtime services. Any administrative and runtime services obtained before the initialize are invalid and have undefined behavior.

The next code snippet outlines a typical sequence of use:

```
// Configure the engine, this is optional
Configuration config = new Configuration();
config.configure("configuration.xml"); // load a configuration from file
config.set....(...);    // make additional configuration settings

// Obtain an engine instance
EPServiceProvider                          epService                           =
 EPServiceProviderManager.getDefaultProvider(config);

// Optionally, use initialize if the same engine instance has been used before
 to start clean
epService.initialize();

// Optionally, make runtime configuration changes
epService.getEPAdministrator().getConfiguration().add...(...);

// Destroy the engine instance when no longer needed, frees up resources
epService.destroy();
```

An existing Esper engine instance can be destroyed via the destroy method on the EPServiceProvider instance. This stops and removes all statements as well as frees all resources held by the instance. After a destroy the engine can no longer be used.

The EPServiceStateListener interface may be implemented by your application to receive callbacks when an engine instance is about to be destroyed and after an engine instance has been initialized. Listeners are registered via the addServiceStateListener method. The EPStatementStateListener interface is used to receive callbacks when a new statement gets created and when a statement gets started, stopped or destroyed. Listeners are registered via the addStatementStateListener method.

When destroying an engine instance your application must make sure that threads that are sending events into the engine have completed their work. More generally, the engine should not be currently in use during or after the destroy operation.

As engine instances are completely independent, your application may not send `EventBean` instances obtained from one engine instance into a second engine instance since the event type space between two engine instances is not shared.

14.3. The Administrative Interface

14.3.1. 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 5.4.2, “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();
```

The `create` methods on `EPAdministrator` also accept a user object. The user object is associated with a statement at time of statement creation and is a single, unnamed field that is stored with every statement. Applications may put arbitrary objects in this field. Use the `getUserObject` method on `EPStatement` to obtain the user object of a statement and `StatementAwareUpdateListener` for listeners.

Your application may create new statements or stop and destroy existing statements using any thread and also within listener or subscriber code. If using POJO events, your application may not create or manage statements in the event object itself while the same event is currently being processed by a statement.

14.3.2. 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 14.1. 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.
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 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.

Name	Methods on `EPStatement`	Description
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.

If your application attaches both a subscriber and one or more listeners then the subscriber receives the result first before any of the listeners.

If your application attaches more then one listener then the `UpdateListener` listeners receive results first in the order they were added to the statement, and `StatementAwareUpdateListener` listeners receive results next in the order they were added to the statement. To change the order of delivery among listeners your application can add and remove listeners at runtime.

If you have configured outbound threading, it means a thread from the outbound thread pool delivers results to the subscriber and listeners instead of the processing or event-sending thread.

If outbound threading is turned on, we recommend turning off the engine setting preserving the order of events delivered to listeners as described in *Section 15.4.10.1, “Preserving the order of events delivered to listeners”*. If outbound threading is turned on statement execution is not blocked for the configured time in the case a subscriber or listener takes too much time.

14.3.3. 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 does not need to implement an interface or extend a superclass. Only one subscriber object may be set for a statement.

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.

14.3.3.1. 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

In the case that your subscriber class offers multiple `update` method footprints, the engine selects the closest-matching footprint by comparing the output types and method parameter types. The engine prefers the update method that is an exact match of types, followed by an update method that requires boxing or unboxing, followed by an update method that requires widening and finally any other allowable update method.

14.3.3.1.1. 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) {...}
```

14.3.3.1.2. 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 name if supplied, or the expression string itself for columns without a name.

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) {...}
```

14.3.3.1.3. 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) {...}
```

14.3.3.1.4. 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 updateStart and updateEnd.

The updateStart 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 updateStart method immediately prior to delivering events to the update and updateRStream methods.

The updateEnd method must take no parameters. The engine invokes the updateEnd 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 updateStart(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 updateEnd() {...}
```

14.3.3.2. 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. This is applicable when an EPL delivers multiple output rows for a given input event or time advancing, for example when multiple pattern matches occur for the same incoming event, for a join producing multiple output rows or with output rate limiting, for example.

The event delivery follow the scheme as described earlier in *Section 14.3.3.1.2, “Row Delivery as Map and Object Array ”*. The subscriber class must provide one of the following methods:

Table 14.2. 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

14.3.3.2.1. 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) {...}
```

14.3.3.3. No-Parameter Update Method

In the case that your subscriber object wishes to receive no data from a statement please follow the instructions here.

You EPL statement must select a single `null` value.

For example, your statement may be:

```
select null from OrderEvent(price > 100)
```

Your subscriber class exposes the method:

```
public void update() {...}
```

14.3.4. 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.

Important

`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)`

14.3.4.1. Subscription Snapshot and Atomic Delivery

The `addListenerWithReplay` method provided by `EPStatement` makes it possible to send a snapshot of current statement results to a listener when the listener is added.

When using the `addListenerWithReplay` method to register a listener, the listener receives current statement results as the first call to the update method of the listener, passing in the newEvents parameter the current statement results as an array of zero or more events. Subsequent calls to the update method of the listener are statement results.

Current statement results are the events returned by the `iterator` or `safeIterator` methods.

Delivery is atomic: Events occurring during delivery of current results to the listener are guaranteed to be delivered in a separate call and not lost. The listener implementation should thus minimize long-running or blocking operations to reduce lock times held on statement-level resources.

14.3.5. 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 engine employs a read-write lock per context partition and obtains a read lock for iteration. Thus safe iterator guarantees correct results even as events are being processed by other threads and other context partitions. The cost is that the iterator obtains and holds zero, one or multiple context partition locks for that statement 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.

When statements are used with contexts and context partitions, the APIs to identify, filter and select context partitions for statement iteration are described in *Section 14.19, “Context Partition Selection”*.

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.

Consider using the `on-select` clause and a named window if your application requires iterating over a partial result set or requires indexed access for fast iteration; Note that `on-select` requires that you sent a trigger event, which may contain the key values for indexed access.

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 context partitions stay locked and event processing for statement context partitions 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 in statements without grouping or aggregation. For statements with grouping or aggregation, the iterator in combintion with an output clause returns last output group and aggregation results. Use a separate statement and the `insert into` clause to control the output rate for iteration, if so required.
4. When iterating a statement that operates on an unbound stream (no data window declared), please note the following:
   - When iterating a statement that groups and aggregates values from an unbound stream and that specifies `output snapshot`, the engine retains groups and aggregations for output as iteration results or upon the output snapshot condition .
   - When iterating a statement that groups and aggregates values from an unbound stream and that does not specify `output snapshot`, the engine only retains the last aggregation values and the iterated result contains only the last updated group.
   - When iterating a statement that operates on an unbound stream and does not group and aggregate, the iterator returns the last event.

14.3.6. 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

14.3.7. Runtime 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 a `ConfigurationOperations` object.

Please consult the JavaDoc of `ConfigurationOperations` for further information. The section *Section 15.6, “Runtime Configuration”* provides a summary of available configurations.

In summary, the configuration operations available on a running engine instance are as follows:

- Add new event types for all event representations, check if an event type exists, update an existing event type, remove an event type, query a list of types and obtain a type by name.
- Add and remove variables (get and set variable values is done via the runtime API).

- Add a variant stream.
- Add a revision event type.
- Add event types for all event classes in a given Java package, using the simple class name as the event name.
- Add import for user-defined functions.
- Add a plug-in aggregation function, plug-in single row function, plug-in event type, plug-in event type resolution URIs.
- Control metrics reporting.
- Additional items please see the `ConfigurationOperations` interface.

For examples of above runtime configuration API functions please consider the Configuration chapter, which applies to both static configuration and runtime configuration as the `ConfigurationOperations` interface is the same.

14.4. The Runtime Interface

The `EPRuntime` interface is used to send events for processing into an Esper engine, set and get variable values and execute on-demand queries.

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));
```

Tip

Events, in theoretical terms, are observations of a state change that occurred in the past. Since one cannot change an event that happened in the past, events are best modelled as immutable objects.

Caution

The 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.

Important

Another important method in the runtime interface is the `route` method. This method is designed for use by `UpdateListener` and subscriber implementations as well as engine extensions that need to send events into an engine instance to avoid the possibility of a stack overflow due to nested calls to `sendEvent` and to ensure correct processing of the current and routed event.

14.4.1. Event Sender

The `EventSender` interface processes event objects that are of a known type. This facility can reduce the overhead of event object reflection and type lookup as an event sender is always associated to a single concrete event type.

Use the method `getEventSender(String eventTypeName)` to obtain an event sender for processing events of the named type:

```
EventSender sender = epService.getEPRuntime().getEventSender("MyEvent");
sender.sendEvent(myEvent);
```

For events backed by a Java class (JavaBean events), the event sender ensures that the event object equals the underlying class, or implements or extends the underlying class for the given event type name.

For events backed by a `java.util.Map` (Map events), the event sender does not perform any checking other then checking that the event object implements Map.

For events backed by a `Object[]` (Object-array events), the event sender does not perform any checking other then checking that the event object implements Object[]. The array elements must be in the exact same order of properties as declared and array length must always be at least the number of properties declared.

For events backed by a org.w3c.Node (XML DOM events), the event sender checks that the root element name equals the root element name for the event type.

A second method to obtain an event sender is the method `getEventSender(URI[])`, which takes an array of URIs. This method is for use with plug-in event representations. The event sender returned by this method processes event objects that are of one of the types of one or more plug-in event representations. Please consult *Section 17.8, “Event Type And Event Object”* for more information.

14.4.2. 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.

14.5. On-Demand Fire-And-Forget Query Execution

As your application may not require streaming results and may not know each query in advance, the on-demand query facility provides for ad-hoc execution of an EPL expression.

On-demand queries are not continuous in nature: The query engine executes the query once and returns all result rows to the application. On-demand query execution is very lightweight as the engine performs no statement creation and the query leaves no traces within the engine.

Esper provides the facility to explicitly index named windows to speed up on-demand and continuous queries. Please consult *Section 5.15.13, “Explicitly Indexing Named Windows”* for more information.

When named windows are used with contexts and context partitions, the APIs to identify, filter and select context partitions for on-demand queries can be found in *Section 14.19, “Context Partition Selection”*.

The `EPRuntime` interface provides three ways to run on-demand queries:

1. Use the `executeQuery` method to executes a given on-demand query exactly once, see *Section 14.5.1, “On-Demand Query Single Execution”*.
2. Use the `prepareQuery` method to prepare a given on-demand query such that the same query can be executed multiple times without repeated parsing, see *Section 14.5.2, “On-Demand Query Prepared Unparameterized Execution”*.
3. Use the `prepareQueryWithParameters` method to prepare a given on-demand query that may have substitution parameters such that the same query can be parameterized and executed multiple times without repeated parsing, see *Section 14.5.3, “On-Demand Query Prepared Parameterized Execution”*

If your application must execute the same EPL on-demand query multiple times with different parameters use `prepareQueryWithParameters`.

If your application must execute the same EPL on-demand query multiple times without use either `prepareQuery` or `prepareQueryWithParameters` and specify no substitution parameters.

By using any of the `prepare...` methods the engine can compile an EPL query string or object model once and reuse the object and thereby speed up repeated execution.

The following limitations apply:

- An on-demand EPL expression only evaluates against the named windows that your application creates. On-demand queries may not specify any other streams or application event types.
- The following clauses are not allowed in on-demand EPL: `insert into` and `output`.
- Views and patterns are not allowed to appear in on-demand queries.
- On-demand EPL may not perform subqueries.
- The `previous` and `prior` functions may not be used.

14.5.1. On-Demand Query Single Execution

Use the `executeQuery` method for executing an on-demand query once. For repeated execution, please consider any of the `prepare...` methods instead.

The next program listing runs an on-demand query against a named window `MyNamedWindow` and prints a column of each row result of the query:

```
String query = "select * from MyNamedWindow";
EPOnDemandQueryResult result = epRuntime.executeQuery(query);
for (EventBean row : result.getArray()) {
  System.out.println("name=" + row.get("name"));
}
```

14.5.2. On-Demand Query Prepared Unparameterized Execution

Prepared on-demand queries are designed for repeated execution and may perform better then the dynamic single-execution method if running the same query multiple times. For use with parameter placeholders please see *Section 14.5.3, “On-Demand Query Prepared Parameterized Execution”*.

The next code snippet demonstrates prepared on-demand queries without parameter placeholder:

```
String query = "select * from MyNamedWindow where orderId = '123'"
EPOnDemandPreparedQuery prepared = epRuntime.prepareQuery(query);
EPOnDemandQueryResult result = prepared.execute();

// ...later on execute once more ...
prepared.execute(); // execute a second time
```

14.5.3. On-Demand Query Prepared Parameterized Execution

Substitution parameters are inserted into an on-demand query 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.

All substitution parameters must be replaced by actual values before an on-demand query with substitution parameters can be executed. 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 the query executed more than 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.

The next program listing runs a prepared and parameterized on-demand query against a named window `MyNamedWindow`:

```
String query = "select * from MyNamedWindow where orderId = ?";
EPOnDemandPreparedQueryParameterized                prepared                 =
 epRuntime.prepareQueryWithParameters(query);

// Set the required parameter values before each execution
prepared.setObject(1, "123");
result = epRuntime.executeQuery(prepared);
```

```
// ...execute a second time with new parameter values...
prepared.setObject(1, "456");
result = epRuntime.executeQuery(prepared);
```

This second example uses the `in` operator and has multiple parameters:

```
String query = "select * from MyNamedWindow where orderId in (?) and price > ?";
EPOnDemandPreparedQueryParameterized                prepared                 =
 epRuntime.prepareQueryWithParameters(query);
prepared.setObject(1, new String[] {"123", "456"});
prepared.setObject(2, 1000.0});
```

14.6. Event and Event Type

An `EventBean` object represents a row (event) in your continuous query's result set. Each `EventBean` object has an associated `EventType` object providing event metadata.

An `UpdateListener` implementation receives one or more `EventBean` events with each invocation. Via the `iterator` method on `EPStatement` your application can poll or read data out of statements. Statement iterators also return `EventBean` instances.

Each statement provides the event type of the events it produces, available via the `getEventType` method on `EPStatement`.

14.6.1. Event Type Metadata

An `EventType` object encapulates all the metadata about a certain type of events. As Esper supports an inheritance hierarchy for event types, it also provides information about super-types to an event type.

An `EventType` object provides the following information:

- For each event property, it lists the property name and type as well as flags for indexed or mapped properties and whether a property is a fragment.
- The direct and indirect super-types to the event type.
- Value getters for property expressions.
- Underlying class of the event representation.

For each property of an event type, there is an `EventPropertyDescriptor` object that describes the property. The `EventPropertyDescriptor` contains flags that indicate whether a property is an indexed (array) or a mapped property and whether access to property values require an integer index value (indexed properties only) or string key value (mapped properties only). The descriptor also contains a fragment flag that indicates whether a property value is available as a fragment.

The term *fragment* means an event property value that is itself an event, or a property value that can be represented as an event. The `getFragmentType` on `EventType` may be used to determine a fragment's event type in advance.

A fragment event type and thereby fragment events allow navigation over a statement's results even if the statement result contains nested events or a graph of events. There is no need to use the Java reflection API to navigate events, since fragments allow the querying of nested event properties or array values, including nested Java classes.

When using the Map or Object-array event representation, any named Map type or Object-array type nested within a Map or Object-array as a simple or array property is also available as a fragment. When using Java objects either directly or within Map or Object-array events, any object that is neither a primitive or boxed built-in type, and that is not an enumeration and does not implement the Map interface is also available as a fragment.

The nested, indexed and mapped property syntax can be combined to a property expression that may query an event property graph. Most of the methods on the `EventType` interface allow a property expression to be passed.

Your application may use an `EventType` object to obtain special getter-objects. A getter-object is a fast accessor to a property value of an event of a given type. All getter objects implement the `EventPropertyGetter` interface. Getter-objects work only for events of the same type or sub-types as the `EventType` that provides the `EventPropertyGetter`. The performance section provides additional information and samples on using getter-objects.

14.6.2. Event Object

An event object is an `EventBean` that provides:

- The property value for a property given a property name or property expression that may include nested, indexed or mapped properties in any combination.
- The event type of the event.
- Access to the underlying event object.
- The `EventBean` fragment or array of `EventBean` fragments given a property name or property expression.

The `getFragment` method on `EventBean` and `EventPropertyGetter` return the fragment `EventBean` or array of `EventBean`, if the property is itself an event or can be represented as an event. Your application may use `EventPropertyDescriptor` to determine which properties are also available as fragments.

The underlying event object of an `EventBean` can be obtained via the `getUnderlying` method. Please see *Chapter 2, Event Representations* for more information on different event representations.

From a threading perspective, it is safe to retain and query `EventBean` and `EventType` objects in multiple threads.

14.6.3. Query Example

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 14.3. 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 either of type `Object[]` (object-array) or of type `java.util.Map`.

Use `statement.getEventType().getUnderlyingType()` to inspect the underlying type for all events delivered to listeners. Whether the engine delivers Map or Object-array events to listeners can be specified as follows. If the statement provides the `@EventRepresentation(array=true)` annotation the engine delivers the output events as object array. If the statement provides the `@EventRepresentation(array=false)` annotation the engine delivers output events as a Map. If neither annotation is provided, the engine delivers the configured default event representation as discussed in *Section 15.4.11.1, “Default Event Representation”*.

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 14.4. 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();
```

14.6.4. Pattern Example

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"));
    System.out.println("b event=" + newData[0].get("b"));
  }
}
```

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:

```
select * from pattern[
  every a=A -> (timer:interval(60 sec) and not B(id=a.id))]
```

A code snippet out of a possible `UpdateListener` implementation to this statement that retrives the events as fragments may look as below:

```
EventBean a = (EventBean) newEvents[0].getFragment("a");
// ... or using a nested property expression to get a value out of A event...
double value = (Double) newEvent[0].get("a.value");
```

Some pattern objects return an array of events. An example is the unbound repeat operator. Here is a sample pattern that collects all A events until a B event arrives:

```
select * from pattern [a=A until b=B]
```

A possible code to retrieve different fragments or property values:

```
EventBean[] a = (EventBean[]) newEvents[0].getFragment("a");
// ... or using a nested property expression to get a value out of A event...
double value = (Double) newEvent[0].get("a[0].value");
```

14.7. 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 application-managed threads or 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.

Esper does not prescribe a specific threading model. 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.

In the default configuration 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). If your application relies on external timer events instead of the internal timer then there need not be any Esper-managed internal threads.

The fact that event processing can take place within your application thread's 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.

In the default configuration, 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 context partition level to protect context partition resources. For stateless EPL select-statements the engine does not use a context-partition lock and operates lock-free for the context partition. For stateful statements, the maximum (theoretical) degree of parallelism is 2^31-1 (2,147,483,647) parallel threads working to process a single EPL statement under a hash segmented context.

You may turn off context partition locking engine-wide (also read the caution notice) as described in *Section 15.4.23.3, "Disable Locking"*. You may disable context partition locking for a given statement by providing the `@NoLock` annotation as part of your EPL. Note, we provide the `@NoLock` annotation for the purpose of identifying locking overhead, or when your application is single-threaded, or when using an external mechanism for concurrency control or for example with virtual data windows or plug-in data windows to allow customizing concurrency for a given statement or named window. Using this annotation may have unpredictable results unless your application is taking concurrency under consideration.

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, see *Section 15.4.10.1, “Preserving the order of events delivered to listeners”*. For example, assume thread T1 processes an event applied to statement S producing output event O1. Assume thread T2 processes another event applied to statement S and produces output event O2. The engine employs a configurable latch system to ensure that listeners to statement S receive and may complete processing of O1 before receiving O2. When using outbound threading (advanced threading options) or changing the configuration this guarantee is weakened or removed.

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, see , see *Section 15.4.10.2, “Preserving the order of events for insert-into streams”*. For example, assume thread T1 processes an event applied to statement S and thread T2 processes another event applied to statement S. Assume statement S inserts into into stream ST. T1 produces an output event O1 for processing by consumers of ST1 and T2 produces an output event O2 for processing by consumers of ST. The engine employs a configurable latch system such that O1 is processed before O2 by consumers of ST. When using route execution threading (advanced threading options) or changing the configuration this guarantee is weakened or removed.

We generally recommended that listener implementations block minimally or do not block at all. By implementing listener code as non-blocking code execution threads can often achieve higher levels of concurrency.

We recommended that, when using a single listener or subscriber instance to receive output from multiple statements, that the listener or subscriber code is multithread-safe. If your application has shared state between listener or subscriber instances then such shared state should be thread-safe.

14.7.1. Advanced Threading

In the default configuration the same application thread that invokes any of the `sendEvent` methods will process the event fully and also deliver output events to listeners and subscribers. By default the single internal timer thread based on system time performs time-based processing and delivery of time-based results.

This default configuration reduces the processing overhead associated with thread context switching, is lightweight and fast and works well in many environments such as J2EE, server or client. Latency and throughput requirements are largely use case dependant, and Esper provides engine-level facilities for controlling concurrency that are described next.

Inbound Threading queues all incoming events: A pool of engine-managed threads performs the event processing. The application thread that sends an event via any of the `sendEvent` methods returns without blocking.

Outbound Threading queues events for delivery to listeners and subscribers, such that slow or blocking listeners or subscribers do not block event processing.

Timer Execution Threading means time-based event processing is performed by a pool of engine-managed threads. With this option the internal timer thread (or external timer event) serves only as a metronome, providing units-of-work to the engine-managed threads in the timer execution pool, pushing threading to the level of each statement for time-based execution.

Route Execution Threading means that the thread sending in an event via any of the `sendEvent` methods (or the inbound threading pooled thread if inbound threading is enabled) only identifies and pre-processes an event, and a pool of engine-managed threads handles the actual processing of the event for each statement, pushing threading to the level of each statement for event-arrival-based execution.

The engine starts engine-managed threads as daemon threads when the engine instance is first obtained. The engine stops engine-managed threads when the engine instance is destroyed via the `destroy` method. When the engine is initialized via the `initialize` method the existing engine-managed threads are stopped and new threads are created. When shutting down your application, use the `destroy` method to stop engine-managed threads.

Note that the options discussed herein may introduce additional processing overhead into your system, as each option involves work queue management and thread context switching.

If your use cases require ordered processing of events or do not tolerate disorder, the threading options described herein may not be the right choice. For enforcing a processing order within a given criteria, your application must enforce such processing order using Java or .NET code. Esper will not enforce order of processing if you enable inbound or route threading. Your application code could, for example, utilize a thread per group of criteria keys, a latch per criteria key, or a queue per criteria key, or use Java's completion service, all depending on your ordering requirements.

If your use cases require loss-less processing of events, wherein the threading options mean that events are held in an in-memory queue, the threading options described herein may not be the right choice.

Care should be taken to consider arrival rates and queue depth. Threading options utilize unbound queues or capacity-bound queues with blocking-put, depending on your configuration, and may therefore introduce an overload or blocking situation to your application. You may use the service provider interface as outlined below to manage queue sizes, if required, and to help tune the engine to your application needs. Consider throttling down the event send rate when the API (see below) indicates that events are getting queued.

All threading options are on the level of an engine. If you require different threading behavior for certain statements then consider using multiple engine instances, consider using the `route` method or consider using application threads instead.

Please consult *Section 15.4.10, “Engine Settings related to Concurrency and Threading”* for instructions on how to configure threading options. Threading options take effect at engine initialization time.

14.7.1.1. Inbound Threading

With inbound threading an engine places inbound events in a queue for processing by one or more engine-managed threads other then the delivering application threads.

The delivering application thread uses one of the `sendEvent` methods on `EPRuntime` to deliver events or may also use the `sendEvent` method on a `EventSender`. The engine receives the event and places the event into a queue, allowing the delivering thread to continue and not block while the event is being processed and results are delivered.

Events that are sent into the engine via one of the `route` methods are not placed into queue but processed by the same thread invoking the `route` operation.

14.7.1.2. Outbound Threading

With outbound threading an engine places outbound events in a queue for delivery by one or more engine-managed threads other then the processing thread originating the result.

With outbound threading your listener or subscriber class receives statement results from one of the engine-managed threads in the outbound pool of threads. This is useful when you expect your listener or subscriber code to perform significantly blocking operations and you do not want to hold up event processing.

14.7.1.3. Timer Execution Threading

With timer execution threading an engine places time-based work units into a queue for processing by one or more engine-managed threads other then the internal timer thread or the application thread that sends an external timer event.

Using timer execution threading the internal timer thread (or thread delivering an external timer event) serves to evaluate which time-based work units must be processed. A pool of engine-managed threads performs the actual processing of time-based work units and thereby offloads the work from the internal timer thread (or thread delivering an external timer event).

Enable this option as a tuning parameter when your statements utilize time-based patterns or data windows. Timer execution threading is fine grained and works on the level of a time-based schedule in combination with a statement.

14.7.1.4. Route Execution Threading

With route execution threading an engine identifies event-processing work units based on the event and statement combination. It places such work units into a queue for processing by one or more engine-managed threads other then the thread that originated the event.

While inbound threading works on the level of an event, route execution threading is fine grained and works on the level of an event in combination with a statement.

14.7.1.5. Threading Service Provider Interface

The service-provider interface `EPServiceProviderSPI` is an extension API that allows to manage engine-level queues and thread pools .

The service-provider interface `EPServiceProviderSPI` is considered an extension API and subject to change between release versions.

The following code snippet shows how to obtain the `BlockingQueue<Runnable>` and the `ThreadPoolExecutor` for the managing the queue and thread pool responsible for inbound threading:

```
EPServiceProviderSPI spi = (EPServiceProviderSPI) epService;
int queueSize = spi.getThreadingService().getInboundQueue().size();
ThreadPoolExecutor                         threadpool                         =
 spi.getThreadingService().getInboundThreadPool();
```

14.7.2. Processing Order

14.7.2.1. Competing Statements

This section discusses the order in which N competing statements that all react to the same arriving event execute.

The engine, by default, does not guarantee to execute competing statements in any particular order unless using @Priority. We therefore recommend that an application does not rely on the order of execution of statements by the engine, since that best shields the behavior of an application from changes in the order that statements may get created by your application or by threading configurations that your application may change at will.

If your application requires a defined order of execution of competing statements, use the @Priority EPL syntax to make the order of execution between statements well-defined (requires that you set the prioritized-execution configuration setting). And the @Drop can make a statement preempt all other lowered priority ones that then won't get executed for any matching events.

14.7.2.2. Competing Events in a Work Queue

This section discusses the order of event evaluation when multiple events must be processed, for example when multiple statements use insert-into to generate further events upon arrival of an event.

The engine processes an arriving event completely before indicating output events to listeners and subscribers, and before considering output events generated by insert-into or routed events inserted by listeners or subscribers.

For example, assume three statements (1) select * from MyEvent and (2) insert into ABCStream select * from MyEvent. (3) select * from ABCStream. When a MyEvent event arrives then the listeners to statements (1) and (2) execute first (default threading model). Listeners to statement (3) which receive the inserted-into stream events are always executed after delivery of the triggering event.

Among all events generated by insert-into of statements and the events routed into the engine via the `route` method, all events that insert-into a named window are processed first in the order generated. All other events are processed thereafter in the order they were generated.

When enabling timer or route execution threading as explained under advanced threading options then the engine does not make any guarantee to the processing order except that is will prioritize events inserted into a named window.

14.8. Controlling Time-Keeping

There are two modes for an engine to keep track of time: The internal timer based on JVM system time (the default), and externally-controlled time giving your application full control over the concept of time within an engine or isolated service.

An isolated service is an execution environment separate from the main engine runtime, allowing full control over the concept of time for a group of statements, as further described in *Section 14.10, “Service Isolation”*.

By default the internal timer provides time and evaluates schedules. External clocking can be used to supply time ticks to the engine instead. The latter is useful for testing time-based event sequences or for synchronizing the engine with an external time source.

The internal timer relies on the `java.util.concurrent.ScheduledThreadPoolExecutor` class for time tick events. The next section describes timer resolution for the internal timer, by default set to 100 milliseconds but is configurable via the threading options. When using externally-controlled time the timer resolution is in your control.

To disable the internal timer and use externally-provided time instead, there are two options. The first option is to use the configuration API at engine initialization time. The second option toggles on and off the internal timer at runtime, via special timer control events that are sent into the engine like any other event.

If using a timer execution thread pool as discussed above, the internal timer or external time event provide the schedule evaluation however do not actually perform the time-based processing. The time-based processing is performed by the threads in the timer execution thread pool.

This code snippet shows the use of the configuration API to disable the internal timer and thereby turn on externally-provided time (see the Configuration section for configuration via XML file):

```
Configuration config = new Configuration();
config.getEngineDefaults().getThreading().setInternalTimerEnabled(false);
```

```
EPServiceProvider                          epService                          =
 EPServiceProviderManager.getDefaultProvider(config);
```

After disabling the internal timer, it is wise to set a defined time so that any statements created thereafter start relative to the time defined. Use the `CurrentTimeEvent` class to indicate current time to the engine and to move time forward for the engine (a.k.a application-time model).

This code snippet obtains the current time and sends a timer event in:

```
long timeInMillis = System.currentTimeMillis();
CurrentTimeEvent timeEvent = new CurrentTimeEvent(timeInMillis);
epService.getEPRuntime().sendEvent(timeEvent);
```

Alternatively, you can use special timer control events to enable or disable the internal timer. Use the `TimerControlEvent` class to control timer operation at runtime.

The next code snippet demonstrates toggling to external timer at runtime, by sending in a `TimerControlEvent` event:

```
EPServiceProvider epService = EPServiceProviderManager.getDefaultProvider();
EPRuntime runtime = epService.getEPRuntime();
// switch to external clocking
runtime.sendEvent(new
 TimerControlEvent(TimerControlEvent.ClockType.CLOCK_EXTERNAL));
```

Your application sends a `CurrentTimeEvent` event when it desires to move the time forward. All aspects of Esper engine time related to EPL statements and patterns are driven by the time provided by the `CurrentTimeEvent` that your application sends in.

The next example sequence of instructions sets time to zero, then creates a statement, then moves time forward to 1 seconds later and then 6 seconds later:

```
// Set start time at zero.
runtime.sendEvent(new CurrentTimeEvent(0));

// create a statement here
epAdministrator.createEPL("select * from MyEvent output every 5 seconds");

// move time forward 1 second
runtime.sendEvent(new CurrentTimeEvent(1000));

// move time forward 5 seconds
runtime.sendEvent(new CurrentTimeEvent(6000));
```

When sending external timer events, your application should make sure that `long`-type time values are ascending. That is, each long-type value should be either the same value or a larger value then the prior value provided by a `CurrentTimeEvent`. The engine outputs a warning if time events move back in time.

Your application may use the `getNextScheduledTime` method in `EPRuntime` to determine the earliest time a schedule for any statement requires evaluation.

The following code snippet sets the current time, creates a statement and prints the next scheduled time which is 1 minute later then the current time:

```
// Set start time to the current time.
runtime.sendEvent(new CurrentTimeEvent(System.currentTimeMillis()));

// Create a statement.
epService.getEPAdministrator().createEPL("select             *             from
 pattern[timer:interval(1 minute)]");

// Print next schedule time
System.out.println("Next        schedule        at        "        +        new
 Date(runtime.getNextScheduledTime());
```

14.8.1. Controlling Time Using Time Span Events

With `CurrentTimeEvent`, as described above, your application can advance engine time to a given point in time. In addition, the `getNextScheduledTime` method in `EPRuntime` returns the next scheduled time according to started statements. You would typically use `CurrentTimeEvent` to advance time at a relatively high resolution.

To advance time for a span of time without sending individual `CurrentTimeEvent` events to the engine, the API provides the class `CurrentTimeSpanEvent`. You may use `CurrentTimeSpanEvent` with or without a resolution.

If your application only provides the target end time of time span to `CurrentTimeSpanEvent` and no resolution, the engine advances time up to the target time by stepping through all relevant times according to started statements.

If your application provides the target end time of time span and in addition a `long`-typed resolution, the engine advances time up to the target time by incrementing time according to the resolution (regardless of next scheduled time according to started statements).

Consider the following example:

```
// Set start time to Jan.1, 2010, 00:00 am for this example
SimpleDateFormat format = new SimpleDateFormat("yyyy MM dd HH:mm:ss SSS");
Date startTime = format.parse("2010 01 01 00:00:00 000");
runtime.sendEvent(new CurrentTimeEvent(startTime.getTime()));
```

```
// Create a statement.
EPStatement     stmt     =     epService.getEPAdministrator().createEPL("select
 current_timestamp() as ct " +
  "from pattern[every timer:interval(1 minute)]");
stmt.addListener(...); // add a listener

// Advance time to 10 minutes after start time
runtime.sendEvent(new CurrentTimeSpanEvent(startTime.getTime() + 10*60*1000));
```

The above example advances time to 10 minutes after the time set using `CurrentTimeSpanEvent`. As the example does not pass a resolution, the engine advances time according to statement schedules. Upon sending the `CurrentTimeSpanEvent` the listener sees 10 invocations for minute 1 to minute 10.

To advance time according to a given resolution, you may provide the resolution as shown below:

```
// Advance time to 10 minutes after start time at 100 msec resolution
runtime.sendEvent(new  CurrentTimeSpanEvent(startTime.getTime()  +  10*60*1000,
 100));
```

14.8.2. Additional Time-Related APIs

Consider using the service-provider interface `EPRuntimeSPI` `EPRuntimeIsolatedSPI`. The two interfaces are service-provider interfaces that expose additional function to manage statement schedules. However the SPI interfaces should be considered an extension API and are subject to change between release versions.

Additional engine-internal SPI interfaces can be obtained by downcasting `EPServiceProvider` to `EPServiceProviderSPI`. For example the `SchedulingServiceSPI` exposes schedule information per statement (downcast from `SchedulingService`). Engine-internal SPI are subject to change between versions.

14.9. 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 (configurable).

The internal timer thread, by default, 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 thread can be configured to use nano-second time as returned by `System.nanoTime()`. If configured for nano-second time, the engine computes an offset of the nano-second ticks to wall clock time upon startup to present back an accurate millisecond wall clock time. Please see section *Section 15.4.18, “Engine Settings related to Time Source”* to configure the internal timer thread to use `System.nanoTime()`.

The internal timer is based on `java.util.concurrent.ScheduledThreadPoolExecutor` (`java.util.Timer` does not support 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.

14.10. Service Isolation

14.10.1. Overview

An *isolated service* allows an application to control event visibility and the concept of time as desired on a statement level: Events sent into an isolated service are visible only to those statements that currently reside in the isolated service and are not visible to statements outside of that isolated service. Within an isolated service an application can control time independently, start time at a point in time and advance time at the resolution and pace suitable for the statements added to that isolated service.

As discussed before, a single Java Virtual Machine may hold multiple Esper engine instances unique by engine URI. Within an Esper engine instance the default execution environment for statements is the `EPRuntime` engine runtime, which coordinates all statement's reaction to incoming events and to time passing (via internal or external timer).

Subordinate to an Esper engine instance, your application can additionally allocate multiple isolated services (or execution environments), uniquely identified by a name and represented by the `EPServiceProviderIsolated` interface. In the isolated service, time passes only when you application sends timer events to the `EPRuntimeIsolated` instance. Only events explicitly sent to the isolated service are visible to statements added.

Your application can create new statements that start in an isolated service. You can also move existing statements back and forth between the engine and an isolated service.

Isolation does not apply to variables: Variables are global in nature. Also, as named windows are globally visibly data windows, consumers to named windows see changes in named windows even though a consumer or the named window (through the create statement) may be in an isolated service.

An isolated service allows an application to:

1. Suspend a statement without loosing its statement state that may have accumulated for the statement.
2. Control the concept of time separately for a set of statements, for example to simulate, backtest, adjust arrival order or compute arrival time.
3. Initialize statement state by replaying events, without impacting already running statements, to catch-up statements from historical events for example.

While a statement resides in an isolated runtime it receives only those events explicitly sent to the isolated runtime, and performs time-based processing based on the timer events provided to that isolated runtime.

Use the `getEPServiceIsolated` method on `EPServiceProvider` passing a name to obtain an isolated runtime:

```
EPServiceProviderIsolated                   isolatedService                  =
 epServiceManager.getEPServiceIsolated("name");
```

Set the start time for your isolated runtime via the `CurrentTimeEvent` timer event:

```
// In this example start the time at the system time
long startInMillis = System.currentTimeMillis();
isolatedService.getEPRuntime().sendEvent(new CurrentTimeEvent(startInMillis));
```

Use the `addStatement` method on `EPAdministratorIsolated` to move an existing statement out of the engine runtime into the isolated runtime:

```
// look up the existing statement
EPStatement                              stmt                               =
 epServiceManager.getEPAdministrator().getStatement("MyStmt");

// move it to an isolated service
isolatedService.getEPAdministrator().addStatement(stmt);
```

To remove the statement from isolation and return the statement back to the engine runtime, use the `removeStatement` method on `EPAdministratorIsolated`:

```
isolatedService.getEPAdministrator().removeStatement(stmt);
```

To create a new statement in the isolated service, use the `createEPL` method on `EPAdministratorIsolated`:

```
isolatedService.getEPAdministrator().createEPL(
  "@Name('MyStmt') select * from Event", null, null);
// the example is passing the statement name in an annotation and no user object
```

The `destroy` method on `EPServiceProviderIsolated` moves all currently-isolated statements for that isolated service provider back to engine runtime.

When moving a statement between engine runtime and isolated service or back, the algorithm ensures that events are aged according to the time that passed and time schedules stay intact.

To use isolated services, your configuration must have view sharing disabled as described in *Section 15.4.12.1, “Sharing View Resources between Statements”*.

14.10.2. Example: Suspending a Statement

By adding an existing statement to an isolated service, the statement's processing effectively becomes suspended. Time does not pass for the statement and it will not process events, unless your application explicitly moves time forward or sends events into the isolated service.

First, let's create a statement and send events:

```
EPStatement  stmt  =  epServiceManager.getEPAdministrator().createEPL("select  *
 from TemperatureEvent.win:time(30)");
epServiceManager.getEPRuntime().send(new TemperatureEvent(...));
// send some more events over time
```

The steps to suspend the previously created statement are as follows:

```
EPServiceProviderIsolated                   isolatedService                   =
 epServiceManager.getEPServiceIsolated("suspendedStmts");
isolatedService.getEPAdministrator().addStatement(stmt);
```

To resume the statement, move the statement back to the engine:

```
isolatedService.getEPAdministrator().removeStatement(stmt);
```

If the statement employed a time window, the events in the time window did not age. If the statement employed patterns, the pattern's time-based schedule remains unchanged. This is because the example did not advance time in the isolated service.

14.10.3. Example: Catching up a Statement from Historical Data

This example creates a statement in the isolated service, replays some events and advances time, then merges back the statement to the engine to let it participate in incoming events and engine time processing.

First, allocate an isolated service and explicitly set it to a start time. Assuming that `myStartTime` is a long millisecond time value that marks the beginning of the data to replay, the sequence is as follows:

```
EPServiceProviderIsolated                   isolatedService                   =
 epServiceManager.getEPServiceIsolated("suspendedStmts");
isolatedService.getEPRuntime().sendEvent(new CurrentTimeEvent(myStartTime));
```

Next, create the statement. The sample statement is a pattern statement looking for temperature events following each other within 60 seconds:

```
EPStatement stmt = epAdmin.createEPL(
  "select * from pattern[every a=TemperatureEvent -> b=TemperatureEvent where
 timer:within(60)]");
```

For each historical event to be played, advance time and send an event. This code snippet assumes that `currentTime` is a time greater then `myStartTime` and reflects the time that the historical event should be processed at. It also assumes `historyEvent` is the historical event object.

```
isolatedService.getEPRuntime().sendEvent(new CurrentTimeEvent(currentTime));
isolatedService.getEPRuntime().send(historyEvent);
// repeat the above advancing time until no more events
```

Finally, when done replaying events, merge the statement back with the engine:

```
isolatedService.getEPAdministrator().removeStatement(stmt);
```

14.10.4. Isolation for Insert-Into

When isolating statements, events that are generated by `insert into` are visible within the isolated service that currently holds that `insert into` statement.

For example, assume the below two statements named A and B:

```
@Name('A') insert into MyStream select * from MyEvent
@Name('B') select * from MyStream
```

When adding statement A to an isolated service, and assuming a `MyEvent` is sent to either the engine runtime or the isolated service, a listener to statement B does not receive that event.

When adding statement B to an isolated service, and assuming a `MyEvent` is sent to either the engine runtime or the isolated service, a listener to statement B does not receive that event.

14.10.5. Isolation for Named Windows

When isolating named windows, the event visibility of events entering and leaving from a named window is not limited to the isolated service. This is because named windows are global data windows (a relation in essence).

For example, assume the below three statements named A, B and C:

```
@Name('A') create window MyNamedWindow.win:time(60) as select * from MyEvent
@Name('B') insert into MyNamedWindow select * from MyEvent
@Name('C') select * from MyNamedWindow
```

When adding statement A to an isolated service, and assuming a `MyEvent` is sent to either the engine runtime or the isolated service, a listener to statement A and C does not receive that event.

When adding statement B to an isolated service, and assuming a `MyEvent` is sent to either the engine runtime or the isolated service, a listener to statement A and C does not receive that event.

When adding statement C to an isolated service, and assuming a `MyEvent` is sent to the engine runtime, a listener to statement A and C does receive that event.

14.10.6. Runtime Considerations

Moving statements between an isolated service and the engine is an expensive operation and should not be performed with high frequency.

When using multiple threads to send events and at the same time moving a statement to an isolated service, it its undefined whether events will be delivered to a listener of the isolated statement until all threads completed sending events.

Metrics reporting is not available for statements in an isolated service. Advanced threading options are also not available in the isolated service, however it is thread-safe to send events including timer events from multiple threads to the same or different isolated service.

14.11. Exception Handling

You may register one or more exception handlers for the engine to invoke in the case it encounters an exception processing a continuously-executing statement. By default and without exception handlers the engine cancels execution of the current EPL statement that encountered the exception, logs the exception and continues to the next statement, if any. The configuration is described in *Section 15.4.24, “Engine Settings related to Exception Handling”*.

If your application registers exception handlers as part of engine configuration, the engine invokes the exception handlers in the order they are registered passing relevant exception information such as EPL statement name, expression and the exception itself.

Exception handlers receive any EPL statement unchecked exception such as internal exceptions or exceptions thrown by plug-in aggregation functions or plug-in views. The engine does not provide to exception handlers any exceptions thrown by static method invocations for function calls, method invocations in joins, methods on event classes and listeners or subscriber exceptions.

An exception handler can itself throw a runtime exception to cancel execution of the current event against any further statements.

For on-demand queries the API indicates any exception directly back to the caller without the exception handlers being invoked, as exception handlers apply to continuous queries only. The same applies to any API calls other then `sendEvent` and the `EventSender` methods.

As the configuration section describes, your application registers one or more classes that implement the `ExceptionHandlerFactory` interface in the engine configuration. Upon engine initialization the engine obtains a factory instance from the class name that then provides the exception handler instance. The exception handler class must implement the `ExceptionHandler` interface.

14.12. Condition Handling

You may register one or more condition handlers for the engine to invoke in the case it encounters certain conditions, as outlined below, when executing a statement. By default and without condition handlers the engine logs the condition at informational level and continues processing. The configuration is described in *Section 15.4.25, “Engine Settings related to Condition Handling”*.

If your application registers condition handlers as part of engine configuration, the engine invokes the condition handlers in the order they are registered passing relevant condition information such as EPL statement name, expression and the condition information itself.

Currently the only conditions indicated by this facility are raised by the pattern followed-by operator, see *Section 6.5.8.1, “Limiting Sub-Expression Count”* and see *Section 6.5.8.2, “Limiting Engine-wide Sub-Expression Count”*.

A condition handler may not itself throw a runtime exception or return any value.

As the configuration section describes, your application registers one or more classes that implement the `ConditionHandlerFactory` interface in the engine configuration. Upon engine initialization the engine obtains a factory instance from the class name that then provides the condition handler instance. The condition handler class must implement the `ConditionHandler` interface.

14.13. 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 statement object model is fully mutable. Mutating a any list such as returned by `getChildren()`, for example, is acceptable and supported.

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.

14.13.1. 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 14.5. 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 name for use 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.

14.13.2. 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))
  ));
```

14.13.3. 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)));
```

14.13.4. 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(OutputLimitSelector.DEFAULT,
 Expressions.timePeriod(null, null, null, 10.0, null)));
model.setOrderByClause(OrderByClause.create("line"));
```

14.13.5. 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);
```

14.13.6. 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 OrderEvent
```

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()
  .addWithName("symbol", "sym")
  .addWithName("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);
```

14.14. Prepared Statement and Substitution Parameters

The `prepare` method that is part of the administrative API pre-compiles an EPL statement and stores the precompiled 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, including view parameters and pattern observers and guards. Substitution parameters cannot be used where a numeric constant is required rather then an expression and in SQL statements.

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);
```

14.15. Engine and Statement Metrics Reporting

The engine can report key processing metrics through the JMX platform mbean server by setting a single configuration flag described in *Section 15.4.19, “Engine Settings related to JMX Metrics”*. For additional detailed reporting and metrics events, please read on.

Metrics reporting is a feature that allows an application to receive ongoing reports about key engine-level and statement-level metrics. Examples are the number of incoming events, the CPU time and wall time taken by statement executions or the number of output events per statement.

Metrics reporting is, by default, disabled. To enable reporting, please follow the steps as outlined in *Section 15.4.20, “Engine Settings related to Metrics Reporting”*. Metrics reporting must be enabled at engine initialization time. Reporting intervals can be controlled at runtime via the `ConfigurationOperations` interface available from the administrative API.

Your application can receive metrics at configurable intervals via EPL statement. A metric datapoint is simply a well-defined event. The events are `EngineMetric` and `StatementMetric` and the Java class representing the events can be found in the client API in package `com.espertech.esper.client.metric`.

Since metric events are processed by the engine the same as application events, your EPL may use any construct on such events. For example, your application may select, filter, aggregate properties, sort or insert into a stream or named window all metric events the same as application events.

This example statement selects all engine metric events:

```
select * from com.espertech.esper.client.metric.EngineMetric
```

The next statement selects all statement metric events:

```
select * from com.espertech.esper.client.metric.StatementMetric
```

Make sure to have metrics reporting enabled since only then do listeners or subscribers to a statement such as above receive metric events.

The engine provides metric events after the configured interval of time has passed. By default, only started statements that have activity within an interval (in the form of event or timer processing) are reported upon.

The default configuration performs the publishing of metric events in an Esper daemon thread under the control of the engine instance. Metrics reporting honors externally-supplied time, if using external timer events.

Via runtime configuration options provided by `ConfigurationOperations`, your application may enable and disable metrics reporting globally, provided that metrics reporting was enabled at initialization time. Your application may also enable and disable metrics reporting for individual statements by statement name.

Statement groups is a configuration feature that allows to assign reporting intervals to statements. Statement groups are described further in the *Section 15.4.20, “Engine Settings related to Metrics Reporting”* section. Statement groups cannot be added or removed at runtime.

The following limitations apply:

- If your Java VM version does not report current thread CPU time (most JVM do), then CPU time is reported as zero (use `ManagementFactory.getThreadMXBean().isCurrentThreadCpuTimeSupported()` to determine if your JVM supports this feature).

  Note: In some JVM the accuracy of CPU time returned is very low (in the order of 10 milliseconds off) which can impact the usefulness of CPU metrics returned. Consider measuring CPU time in your application thread after sending a number of events in the same thread, external to the engine as an alternative.
- Your Java VM may not provide high resolution time via `System.nanoTime`. In such case wall time may be inaccurate and inprecise.
- CPU time and wall time have nanosecond precision but not necessarily nanosecond accuracy, please check with your Java VM provider.
- There is a performance cost to collecting and reporting metrics.
- Not all statements may report metrics: The engine performs certain runtime optimizations sharing resources between similar statements, thereby not reporting on certain statements unless resource sharing is disabled through configuration.

14.15.1. Engine Metrics

Engine metrics are properties of `EngineMetric` events:

Table 14.6. EngineMetric Properties

Name	Description
engineURI	The URI of the engine instance.
timestamp	The current engine time.
inputCount	Cumulative number of input events since engine initialization time. Input events are defined as events send in via application threads as well as `insert into` events.
inputCountDelta	Number of input events since last reporting period.
scheduleDepth	Number of outstanding schedules.

14.15.2. Statement Metrics

Statement metrics are properties of `StatementMetric`. The properties are:

Table 14.7. StatementMetric Properties

Name	Description
engineURI	The URI of the engine instance.
timestamp	The current engine time.
statementName	Statement name, if provided at time of statement creation, otherwise a generated name.
cpuTime	Statement processing CPU time (system and user) in nanoseconds (if available by Java VM).
wallTime	Statement processing wall time in nanoseconds (based on `System.nanoTime`).
numInput	Number of input events to the statement.
numOutputIStream	Number of insert stream rows output to listeners or the subscriber, if any.
numOutputRStream	Number of remove stream rows output to listeners or the subscriber, if any.

The totals reported are cumulative relative to the last metric report.

14.16. Event Rendering to XML and JSON

Your application may use the built-in XML and JSON formatters to render output events into a readable textual format, such as for integration or debugging purposes. This section introduces the utility classes in the client `util` package for rendering events to strings. Further API information can be found in the JavaDocs.

The `EventRenderer` interface accessible from the runtime interface via the `getEventRenderer` method provides access to JSON and XML rendering. For repeated rendering of events of the same event type or subtypes, it is recommended to obtain a `JSONEventRenderer` or

`XMLEventRenderer` instance and use the `render` method provided by the interface. This allows the renderer implementations to cache event type metadata for fast rendering.

In this example we show how one may obtain a renderer for repeated rendering of events of the same type, assuming that `statement` is an instance of `EPStatement`:

```
JSONEventRenderer jsonRenderer = epService.getEPRuntime().
    getEventRenderer().getJSONRenderer(statement.getEventType());
```

Assuming that `event` is an instance of `EventBean`, this code snippet renders an event into the JSON format:

```
String jsonEventText = jsonRenderer.render("MyEvent", event);
```

The XML renderer works the same:

```
XMLEventRenderer xmlRenderer = epService.getEPRuntime().
    getEventRenderer().getXMLRenderer(statement.getEventType());
```

...and...

```
String xmlEventText = xmlRenderer.render("MyEvent", event);
```

If the event type is not known in advance or if you application does not want to obtain a renderer instance per event type for fast rendering, your application can use one of the following methods to render an event to a XML or JSON textual format:

```
String json = epService.getEPRuntime().getEventRenderer().renderJSON(event);
String xml = epService.getEPRuntime().getEventRenderer().renderXML(event);
```

Use the `JSONRenderingOptions` or `XMLRenderingOptions` classes to control how events are rendered. To render specific event properties using a custom event property renderer, specify an `EventPropertyRenderer` as part of the options that renders event property values to strings. Please see the JavaDoc documentation for more information.

14.16.1. JSON Event Rendering Conventions and Options

The JSON renderer produces JSON text according to the standard documented at `http://www.json.org`.

The renderer formats simple properties as well as nested properties and indexed properties according to the JSON string encoding, array encoding and nested object encoding requirements.

The renderer does render indexed properties, it does not render indexed properties that require an index, i.e. if your event representation is backed by POJO objects and your getter method is `getValue(int index)`, the indexed property values are not part of the JSON text. This is because the implementation has no way to determine how many index keys there are. A workaround is to have a method such as `Object[] getValue()` instead.

The same is true for mapped properties that the renderer also renders. If a property requires a Map key for access, i.e. your getter method is `getValue(String key)`, such property values are not part of the result text as there is no way for the implementation to determine the key set.

14.16.2. XML Event Rendering Conventions and Options

The XML renderer produces well-formed XML text according to the XML standard.

The renderer can be configured to format simple properties as attributes or as elements. Nested properties and indexed properties are always represented as XML sub-elements to the root or parent element.

The root element name provided to the XML renderer must be the element name of the root in the XML document and may include namespace instructions.

The renderer does render indexed properties, it does not render indexed properties that require an index, i.e. if your event representation is backed by POJO objects and your getter method is `getValue(int index)`, the indexed property values are not part of the XML text. This is because the implementation has no way to determine how many index keys there are. A workaround is to have a method such as `Object[] getValue()` instead.

The same is true for mapped properties that the renderer also renders. If a property requires a Map key for access, i.e. your getter method is `getValue(String key)`, such property values are not part of the result text as there is no way for the implementation to determine the key set.

14.17. Plug-in Loader

A plug-in loader is for general use with input adapters, output adapters or EPL code deployment or any other task that can benefits from being part of an Esper configuration file and that follows engine lifecycle.

A plug-in loader implements the `com.espertech.esper.plugin.PluginLoader` interface and can be listed in the configuration.

Each configured plug-in loader follows the engine instance lifecycle: When an engine instance initializes, it instantiates each `PluginLoader` implementation class listed in the configuration. The engine then invokes the lifecycle methods of the `PluginLoader` implementation class before and after the engine is fully initialized and before an engine instance is destroyed.

Declare a plug-in loader in your configuration XML as follows:

```
...
  <plugin-loader name="MyLoader" class-name="org.mypackage.MyLoader">
    <init-arg name="property1" value="val1"/>
  </plugin-loader>
...
```

Alternatively, add the plug-in loader via the configuration API:

```
Configuration config = new Configuration();
Properties props = new Properties();
props.put("property1", "value1");
config.addPluginLoader("MyLoader", "org.mypackage.MyLoader", props);
```

Implement the `init` method of your `PluginLoader` implementation to receive initialization parameters. The engine invokes this method before the engine is fully initialized, therefore your implementation should not yet rely on the engine instance within the method body:

```
public class MyPluginLoader implements PluginLoader {
  public void init(String loaderName, Properties properties, EPServiceProviderSPI
 epService) {
      // save the configuration for later, perform checking
  }
  ...
```

The engine calls the `postInitialize` method once the engine completed initialization and to indicate the engine is ready for traffic.

```
public void postInitialize() {
  // Start the actual interaction with external feeds or the engine here
}
...
```

The engine calls the `destroy` method once the engine is destroyed or initialized for a second time.

```
public void destroy() {
  // Destroy resources allocated as the engine instance is being destroyed
}
```

14.18. Interrogating EPL Annotations

As discussed in *Section 5.2.7, “Annotation”* an EPL annotation is an addition made to information in an EPL statement. The API and examples to interrogate annotations are described here.

You may use the `getAnnotations` method of `EPStatement` to obtain annotations specified for an EPL statement. Or when compiling an EPL expression to a `EPStatementObjectModel` statement object model you may also query, change or add annotations.

The following example code demonstrates iterating over an `EPStatement` statement's annotations and retrieving values:

```
String exampleEPL = "@Tag(name='direct-output', value='sink 1') select * from
 RootEvent";
EPStatement stmt = epService.getEPAdministrator().createEPL(exampleEPL);
for (Annotation annotation : stmt.getAnnotations()) {
  if (annotation instanceof Tag) {
    Tag tag = (Tag) annotation;
    System.out.println("Tag name " + tag.name() + " value " + tag.value());
  }
}
```

The output of the sample code shown above is `Tag name direct-output value sink 1`.

14.19. Context Partition Selection

This chapter discusses how to select context partitions. Contexts are discussed in *Chapter 4, Context and Context Partitions* and the reasons for context partition selection are introduced in *Section 4.7, “Operations on Specific Context Partitions”*.

The section is only relevant when you declare a context. It applies to all different types of hash, partitioned, category, overlapping or other temporal contexts. The section uses a category context for the purpose of illustration. The API discussed herein is general and handles all different types of contexts including nested contexts.

Consider a category context that separates bank transactions into small, medium and large:

```
// declare category context
create context TxnCategoryContext
  group by amount < 100 as small,
  group by amount between 100 and 1000 as medium,
  group by amount > 1000 as large from BankTxn
```

```
// retain 1 minute of events of each category separately
context TxnCategoryContext select * from BankTxn.win:time(1 minute)
```

In order for your application to iterate one or more specific categories it is necessary to identify which category, i.e. which context partition, to iterate. Similarly for on-demand queries, to execute on-demand queries against one or more specific categories, it is necessary to identify which context partition to execute the on-demand query against.

Your application may iterate one or more specific context partitions using either the `iterate` or `safeIterate` method of `EPStatement` by providing an implementation of the `ContextPartitionSelector` interface.

For example, assume your application must obtain all bank transactions for small amounts. It may use the API to identify the category and iterate the associated context partition:

```
ContextPartitionSelectorCategory          categorySmall          =          new
 ContextPartitionSelectorCategory() {
    public Set<String> getLabels() {
      return Collections.singleton("small");
    }
  };
Iterator<EventBean> it = stmt.iterator(categorySmall);
```

Your application may execute on-demand queries against one or more specific context partitions by using the `executeQuery` method on `EPRuntime` or the `execute` method on `EPOnDemandPreparedQuery` and by providing an implementation of `ContextPartitionSelector`.

On-demand queries execute against named windows, therefore below EPL statement creates a named window which the engine manages separately for small, medium and large transactions according to the context declared earlier:

```
// Named window per category
context TxnCategoryContext create window BankTxnWindow.win:time(1 min) as BankTxn
```

The following code demonstrates how to fire an on-demand query against the small and the medium category:

```
ContextPartitionSelectorCategory         categorySmallMed         =         new
 ContextPartitionSelectorCategory() {
    public Set<String> getLabels() {
      return new HashSet<String>(Arrays.asList("small", "medium"));
    }
  };
epService.getEPRuntime().executeQuery(
   "select count(*) from BankTxnWindow",
   new ContextPartitionSelector[] {categorySmallMed});
```

The following limitations apply:

- On-demand queries may not join named windows that declare a context.

14.19.1. Selectors

This section summarizes the selector interfaces that are available for use to identify and interrogate context partitions. Please refer to the JavaDoc documentation for package `com.espertech.esper.client.context` and classes therein for additional information.

Use an implementation of `ContextPartitionSelectorAll` or the `ContextPartitionSelectorAll.INSTANCE` object to instruct the engine to consider all context partitions.

Use an implementation of `ContextPartitionSelectorById` if your application knows the context partition ids to query. This selector instructs the engine to consider only those provided context partitions based on their integer id value. The engine outputs the context partition id in the built-in property `context.id`.

Use an implementation of `ContextPartitionSelectorFiltered` to receive and interrogate context partitions. Use the `filter` method that receives a `ContextPartitionIdentifier` to return a boolean indicator whether to include the context partition or not. The `ContextPartitionIdentifier` provides information about each context partition. Your application may not retain `ContextPartitionIdentifier` instances between `filter` method invocations as the engine reuses the same instance. This selector is not supported with nested contexts.

Use an implementation of `ContextPartitionSelectorCategory` with category contexts.

Use an implementation of `ContextPartitionSelectorSegmented` with keyed segmented contexts.

Use an implementation of `ContextPartitionSelectorHash` with hash segmented contexts.

Use an implementation of `ContextPartitionSelectorNested` in combination with the selectors described above with nested contexts.

14.20. Context Partition Administration

This chapter briefly discusses the API to manage context partitions. Contexts are discussed in *Chapter 4, Context and Context Partitions*.

The section is only relevant when you declare a context. It applies to all different types of hash, partitioned, category, overlapping or other temporal contexts.

The administrative API for context partitions is `EPContextPartitionAdmin`. Use the `getContextPartitionAdmin` method of the `EPAdministrator` interface to obtain said service.

The context partition admin API allows an application to:

- Start, stop and destroy individual context partitions.
- Interrogate the state and identifiers for existing context partitions.
- Determine statements associated to a context and context nesting level.

Stopping individual context partitions is useful to drop state, free memory and suspend a given context partition without stopping or destroying any associated statements. For example, assume a keyed segmented context per user id. To suspend and free the memory for a given user id your application can stop the user id's context partition. The engine does not allocate a context partition for this user id again, until your application destroys or starts the context partition.

Destroying individual context partitions is useful to drop state, free memory and deregister the given context partition without stopping or destroying any associated statements. For example, assume a keyed segmented context per user id. To deregister and free the memory for a given user id your application can destroy the user id's context partition. The engine can allocate a fresh context partition for this user id when events for this user id arrive.

Please see the JavaDoc documentation for more information.

14.21. Test and Assertion Support

Esper offers a listener and an assertions class to facilitate automated testing of EPL rules, for example when using a test framework such as `JUnit` or `TestNG`.

Esper does not require any specific test framework. If your application has the `JUnit` test framework in classpath Esper uses `junit.framework.AssertionFailedError` to indicate assertion errors, so as to integrate with continuous integration tools.

For detailed method-level information, please consult the JavaDoc of the package `com.espertech.esper.client.scopetest`.

The class `com.espertech.esper.client.scopetest.EPAssertionUtil` provides methods to assert or compare event property values as well as perform various array arthithmatic, sort events and convert events or iterators to arrays.

The class `com.espertech.esper.client.scopetest.SupportUpdateListener` provides an `UpdateListener` implementation that collects events and returns event data for assertion.

The class `com.espertech.esper.client.scopetest.SupportSubscriber` provides a subscriber implementation that collects events and returns event data for assertion. The `SupportSubscriberMRD` is a subscriber that accepts events multi-row delivery. The `SupportSubscriber` and `SupportSubscriberMRD` work similar to `SupportUpdateListener` that is introduced in more detail below.

14.21.1. `EPAssertionUtil` Summary

The below table only summarizes the most relevant assertion methods offered by `EPAssertionUtil`. Methods provide multiple footprints that are not listed in detail below. Please consult the JavaDoc for additional method-level information.

Table 14.8. Method Summary for EPAssertionUtil

Name	Description
`assertProps`	Methods that assert that property values of a single `EventBean`, POJO or Map matches compared to expected values.
`assertPropsPerRow`	Methods that assert that property values of multiple `EventBean`, POJOs or Maps match compared to expected values.
`assertPropsPerRowAnyOrder`	Same as above, but any row may match. Useful for unordered result sets.
`assertEqualsExactOrder`	Methods that compare arrays, allowing `null`. as parameters.
`assertEqualsAnyOrder`	Same as above, but any row may match. Useful for unordered result sets.

14.21.2. `SupportUpdateListener` Summary

The below table only summarizes the most relevant methods offered by `SupportUpdateListener`. Please consult the JavaDoc for additional information.

Table 14.9. Method Summary for SupportUpdateListener

Name	Description
`reset`	Initializes listener clearing current events and resetting the invoked flag.
`getAndClearIsInvoked`	Returns the "invoked" flag indicating the listener has been invoked, and clears the flag.
`getLastNewData`	Returns the last events received by the listener.
`getAndResetDataListsFlattened`	Returns all events received by the listener as a pair.
`assertOneGetNewAndReset`	Asserts that exactly one new event was received and no removed events, returns the event and resets the listener.
`assertOneGetNew`	Asserts that exactly one new event was received and returns the event.

14.21.3. Usage Example

The next code block is a short but complete programming example that asserts that the properties received from output events match expected value.

```
String epl = "select personName, count(*) as cnt from PersonEvent.win:length(3)
 group by personName";
```

```
EPStatement stmt = epService.getEPAdministrator().createEPL(epl);

SupportUpdateListener listener = new SupportUpdateListener();
stmt.addListener(listener);

epService.getEPRuntime().sendEvent(new PersonEvent("Joe"));
EPAssertionUtil.assertProps(listener.assertOneGetNewAndReset(),
 "personName,cnt".split(","),
    new Object[]{"Joe", 1L});
```

A few additional examples are shown below:

```
String[] fields = new String[] {"property"};
EPAssertionUtil.assertPropsPerRow(listener.getAndResetDataListsFlattened(),
 fields,
    new Object[][]{{"E2"}}, new Object[][]{{"E1"}});
```

```
EPAssertionUtil.assertPropsPerRow(listener.getAndResetLastNewData(), fields,
    new Object[][]{{"E1"}, {"E2"}, {"E3"}});
```

```
assertTrue(listener.getAndClearIsInvoked());
```

Please refer to the Esper codebase test sources for more examples using the assertion class and the listener class.

Chapter 15. 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.

The difference between using a `Configuration` object and the `ConfigurationOperations` interface is that for the latter, all configuration including event types added through that interface are considered runtime configurations. This means they will be discarded when calling the `initialize` method on an `EPServiceProvider` instance.

15.1. 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.addEventType("PriceLimit", PriceLimit.class.getName());
configuration.addEventType("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 as further described in *Section 14.3.7, "Runtime Configuration"*. Through this interface applications can, for example, add new event types at runtime and then create new statements that rely on the additional configuration. The `getConfiguration` method on `EPAdministrator` allows access to `ConfigurationOperations`.

15.2. 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");
```

15.3. 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-4-0.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.xsd">
                                  <event-type                 name="StockTick"
 class="com.espertech.esper.example.stockticker.event.StockTick"/>
                                 <event-type                 name="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.

15.4. Configuration Items

15.4.1. Events represented by Java Classes

15.4.1.1. 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-name package-name="com.mycompany.order.event"/>
```

The same configuration but using the `Configuration` class:

```
Configuration config = new Configuration();
config.addEventTypeAutoName("com.mycompany.order.event");
// ... or ...
config.addEventTypeAutoName(MyEvent.getPackage().getName());
```

15.4.1.2. Event type name to Java class mapping

This configuration item can be used to allow event pattern statements and EPL statements to use an event type name 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 name can also be defined for such classes.

The example pattern statement below first shows a pattern that uses the name `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 name can be listed in the XML configuration file as shown below. The `Configuration` API can also be used to programatically specify an event type name, as shown in an earlier code snippet.

```
<event-type                                                 name="StockTick"
 class="com.espertech.esper.example.stockticker.event.StockTick"/>
```

15.4.1.3. 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 15.1. 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                                              name="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.addEventType("MyLegacyEvent",       MyLegacyEventClass.class.getName(),
 legacyDef);

EPServiceProvider  epService  =  EPServiceProviderManager.getProvider("sample",
 configuration);
```

15.4.1.4. 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                                                  name="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.addEventType("StockTick", StockTickEvent.class.getName(), legacyDef);
```

15.4.1.5. 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                                              name="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.addEventType("MyLegacyEvent",       MyLegacyEventClass.class.getName(),
 legacyDef);
```

15.4.1.6. 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 15.2. 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                                              name="MyLegacyEvent"
 class="com.mycompany.package.MyLegacyEventClass">
  <legacy-type property-resolution-style="case_insensitive"/>
</event-type>
```

15.4.1.7. Factory and Copy Method

The `insert into` clause and directly instantiate and populate your event object. By default the engine invokes the default constructor to instantiate an event object. To change this behavior, you may configure a factory method. The factory method is a method name or a class name plus a method name (in the format class.method) that returns an instance of the class.

The `update` clause can change event properties on an event object. For the purpose of maintaining consistency, the engine may have to copy your event object via serialization (implement the `java.io.Serializable` interface). If instead you do not want any copy operations to occur, or your application needs to control the copy operation, you may configure a copy method. The copy method is the name of a method on the event object that copies the event object.

The sample below shows this option in XML configuration, however the setting can also be changed via `ConfigurationEventTypeLegacy`:

```
<event-type                                              name="MyLegacyEvent"
 class="com.mycompany.package.MyLegacyEventClass">
```

```
                                       <legacy-type                   factory-
method="com.mycompany.myapp.MySampleEventFactory.createMyLegacyTypeEvent" copy-
method="myCopyMethod"/>
</event-type>
```

The copy method should be a public method that takes no parameters and returns a new event object (it may not return `this`). The copy method may not be a static method and may not take parameters.

The `Beacon` data flow operator in connection with the Sun JVM can use `sun.reflect.ReflectionFactory` if the class has no default no-argument constructor.

15.4.1.8. Start and End Timestamp

For use with date-time interval methods, for example, you may let the engine know which property of your class carries the start and end timestamp value.

The sample below shows this option in XML configuration, however the setting can also be changed via API. The sample sets the name of the property providing the start timestamp to `startts` and the name of the property providing the end timestamp `endts`:

```
<event-type                                               name="MyLegacyEvent"
 class="com.mycompany.package.MyLegacyEventClass">
  <legacy-type start-timestamp-property-name="startts" end-timestamp-property-
name="endts"/>
</event-type>
```

15.4.2. Events represented by `java.util.Map`

The engine can process `java.util.Map` events via the `sendEvent(Map map, String eventTypeName)` 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 you can provide an event type 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 name="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 types for the `class` attribute are listed in *Section 15.5, "Type Names"*. In addition, any fully-qualified Java class name that can be resolved via `Class.forName` is allowed.

You can also use the configuration API to configure `Map` event types, as the short code snippet below demonstrates:

```
Map<String, Object> properties = new Map<String, Object>();
properties.put("carId", "int");
properties.put("carType", "string");
properties.put("assembly", Assembly.class.getName());

Configuration configuration = new Configuration();
configuration.addEventType("MyMapEvent", properties);
```

For strongly-typed nested maps (maps-within-maps), the configuration API method `addEventType` can also 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)
```

A Map event type may also become a subtype of one or more supertypes that must also be Map event types. The `java-util-map` element provides the optional attribute `supertype-names` that accepts a comma-separated list of names of Map event types that are supertypes to the type:

```
<event-type name="AccountUpdate">
<java-util-map supertype-names="BaseUpdate, AccountEvent">
...
```

For initialization time configuration, the `addMapSuperType` method can be used to add Map hierarchy information. For runtime configuration, pass the supertype names to the `addEventType` method in `ConfigurationOperations`.

A Map event type may declare a start and end timestamp property name. The XML shown next instructs the engine that the `startts` property carries the event start timestamp and the `endts` property carries the event end timestamp:

```
<event-type name="AccountUpdate">
<java-util-map start-timestamp-property-name="startts" end-timestamp-property-
name="endts">
...
```

15.4.3. Events represented by `Object[]` (Object-array)

The engine can process Object-array (`Object[]`) events via the `sendEvent(Object[] array, String eventTypeName)` method on the `EPRuntime` interface. Elements in the Object array represent event properties. Values can be of any type. JavaBean-style objects as values in an `Object[]` can be processed by the engine, and strongly-typed nested map or object-array event types are also supported. Please see the *Chapter 2, Event Representations* section for details on how to use `Object[]` events with the engine.

Via configuration you can provide an event type name for `Object[]` 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 `MyObjectArrayEvent`.

```
<event-type name="MyObjectArrayEvent">
  <objectarray>
    <objectarray-property name="carId" class="int"/>
    <objectarray-property name="carType" class="string"/>
    <objectarray-property name="assembly" class="com.mycompany.Assembly"/>
  </objectarray>
</event-type>
```

This configuration defines the `carId` property of `MyObjectArrayEvent` events to be of type `int` and in the object array first element (`[0]`). The `carType` property to be of type `java.util.String` is expected in the second array element (`[1]`) . The `assembly` property of the object array event will contain instances of `com.mycompany.Assembly` for the engine to query in element two (`[2]`).

Note that the engine does not verify the length and property values of object array events when your application sends object-array events into the engine. For the example above, the proper object array would look as follows: `new Object[] {carId, carType, assembly}`.

The valid types for the `class` attribute are listed in *Section 15.5, “Type Names”*. In addition, any fully-qualified Java class name that can be resolved via `Class.forName` is allowed.

You can also use the configuration API to configure `Object[]` event types, as the short code snippet below demonstrates:

```
String[] propertyNames = {"carId", "carType", "assembly"};
Object[] propertyTypes = {int.class, String.class, Assembly.class};

Configuration configuration = new Configuration();
configuration.addEventType("MyObjectArrayEvent", propertyNames, propertyTypes);
```

Finally, here is a sample EPL statement that uses the configured `MyObjectArrayEvent` object-array event. This statement uses the `chassisTag` and `numParts` properties of `Assembly` objects.

```
select    carType,    assembly.chassisTag,    count(assembly.numParts)    from
 MyObjectArrayEvent.win:time(60 sec)
```

An Object-array event type may also become a subtype of one supertype that must also be an Object-array event type. The `objectarray` element provides the optional attribute `supertype-names` that accepts a single name of an Object-array event type that is the supertype to the type:

```
<event-type name="AccountUpdate">
<objectarray supertype-names="BaseUpdate">
...
```

An Object-array event type may declare a start and end timestamp property name. The XML shown next instructs the engine that the `startts` property carries the event start timestamp and the `endts` property carries the event end timestamp:

```
<event-type name="AccountUpdate">
<objectarray  start-timestamp-property-name="startts"  end-timestamp-property-
name="endts">
...
```

15.4.4. 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 name in order to process `org.w3c.dom.Node` events:

```
<event-type name="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.addEventType("MyXMLNodeEvent", desc);
```

The next example presents configuration options in a sample configuration entry.

```
<event-type name="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. You may also use `schema-text` instead to provide the actual text of the schema.

Here is an example EPL statement using the configured event type named `AutoIdRFIDEvent`.

```
select ID, countTags from AutoIdRFIDEvent.win:time(30 sec)
```

15.4.4.1. 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. Use the `schema-text` attribute instead when it is more practical to provide the actual text of the schema.

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.

15.4.4.2. Explicit XPath Property

The `xpath-property` element adds explicitly-names 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
- node
- nodeset

In case you need your XPath expression to return a type other then the types listed above, an optional cast-to type can be specified. If specified, the operation firsts obtains the result of the XPath expression as the defined type (number, string, boolean) and then casts or parses the returned type to the specified cast-to-type. At runtime, a warning message is logged if the XPath expression returns a result object that cannot be casted or parsed.

The next line shows how to return a long-type property for an XPath expression that returns a string:

```
desc.addXPathProperty("name", "/element/sub", XPathConstants.STRING, "long");
```

The equivalent configuration XML is:

```
<xpath-property   property-name="name"     xpath="/element/sub"   type="string"
 cast="long"/>
```

See *Section 15.5, “Type Names”* for a list of cast-to type names.

15.4.4.3. 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.addEventType("GetQuote", desc);
```

15.4.4.4. XPath Variable and Function Resolver

If your XPath expressions require variables or functions, your application may provide the class name of an `XPathVariableResolver` and `XPathFunctionResolver`. At type initialization time the engine instantiates the resolver instances and provides these to the XPathFactory.

This example shows the API to set this configuration.

```
ConfigurationEventTypeXMLDOM desc = new ConfigurationEventTypeXMLDOM();
desc.setXPathFunctionResolver(MyXPathFunctionResolver.class.getName());
desc.setXPathVariableResolver(MyXPathVariableResolver.class.getName());
```

15.4.4.5. Auto Fragment

This option is for use when a XSD schema is provided and determines whether the engine automatically creates an event type when a property expression transposes a property that is a complex type according to the schema.

An example:

```
ConfigurationEventTypeXMLDOM desc = new ConfigurationEventTypeXMLDOM();
desc.setAutoFragment(false);
```

15.4.4.6. XPath Property Expression

By default Esper employs the built-in DOM walker implementation to evaluate XPath expressions, which is not namespace-aware.

This configuration setting, when set to true, instructs the engine to rewrite property expressions into XPath.

An example:

```
ConfigurationEventTypeXMLDOM desc = new ConfigurationEventTypeXMLDOM();
desc.setXPathPropertyExpr(true);
```

15.4.4.7. Event Sender Setting

By default an `EventSender` for a given XML event type validates the root element name for which the type has been declared against the one provided by the `org.w3c.Node` sent into the engine.

This configuration setting, when set to false, instructs an `EventSender` to not validate.

An example:

```
ConfigurationEventTypeXMLDOM desc = new ConfigurationEventTypeXMLDOM();
desc.setEventSenderValidatesRoot(false);
```

15.4.4.8. Start and End Timestamp

You may configure the name of the properties that provides the event start timestamp and the event end timestamp as part of the configuration.

An example that configures `startts` as the property name providing the start timestamp and `endts` as the property name providing the end timestamp:

```
ConfigurationEventTypeXMLDOM desc = new ConfigurationEventTypeXMLDOM();
desc.setStartTimestampPropertyName("startts");
desc.setEndTimestampPropertyName("endts");
```

15.4.5. Events represented by Plug-in Event Representations

As part of the extension API plug-in event representations allows an application to create new event types and event instances based on information available elsewhere. Please see *Section 17.8, “Event Type And Event Object”* for details.

The configuration examples shown next use the configuration API to select settings. All options are also configurable via XML, please refer to the sample configuration XML in file `esper.sample.cfg.xml`.

15.4.5.1. Enabling an Custom Event Representation

Use the method `addPlugInEventRepresentation` to enable a custom event representation, like this:

```
URI rootURI = new URI("type://mycompany/myproject/myname");
config.addPlugInEventRepresentation(rootURI,
    MyEventRepresentation.class.getName(), null);
```

The `type://` part of the URI is an optional convention for the scheme part of an URI.

If your event representation takes initialization parameters, these are passed in as the last parameter. Initialization parameters can also be stored in the configuration XML, in which case they are passed as an XML string to the plug-in class.

15.4.5.2. Adding Plug-in Event Types

To register event types that your plug-in event representation creates in advance of creating an EPL statement (either at runtime or at configuration time), use the method `addPlugInEventType`:

```
URI childURI = new URI("type://mycompany/myproject/myname/MyEvent");
configuration.addPlugInEventType("MyEvent", new URI[] {childURI}, null);
```

Your plug-in event type may take initialization parameters, these are passed in as the last parameter. Initialization parameters can also be stored in the configuration XML.

15.4.5.3. Setting Resolution URIs

The engine can invoke your plug-in event representation when an EPL statement is created with an event type name that the engine has not seen before. Plug-in event representations can resolve such names to an actual event type. In order to do this, you need to supply a list of resolution URIs. Use the method `setPlugInEventTypeNameResolutionURIs`, at runtime or at configuration time:

```
URI childURI = new URI("type://mycompany/myproject/myname");
configuration.setPlugInEventTypeNameResolutionURIs(new URI[] {childURI});
```

15.4.6. Class and package imports

Esper allows invocations of static Java library functions in expressions, as outlined in *Section 9.1, “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. Any additional imports that are specified in configuration files or through the API are added to the configuration in addition to the imports below.

- 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
```

15.4.7. Cache Settings for From-Clause 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 15.4.9.6.1, “LRU Cache”*. The expiry-time cache documentation can be found in *Section 15.4.9.6.2, “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>
```

15.4.8. 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, an optional initialization value and an optional boolean-type flag indicating whether the variable is a constant (false by default). 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. Declare each variable a constant to achieve the best performance.

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"/>
<variable name="test" type="int" constant="true"/>
```

Please find the list of valid values for the `type` attribute in *Section 15.5, “Type Names”*.

15.4.9. 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 5.13, “Accessing 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`, via `java.sql.DriverManager` and via data source factory. Either one of these methods to obtain 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 and close the connection when the lookup is done (pooled).
- 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`.

15.4.9.1. Connections obtained via DataSource

This configuration causes Esper to obtain a database connection from a `javax.sql.DataSource` available from your JNDI provider.

The setting is most useful when running within an application server or when a JNDI directory is otherwise present in your Java VM. If your application environment does not provide an available `DataSource`, the next section outlines how to use Apache DBCP as a `DataSource` implementation with connection pooling options and outlines how to use a custom factory for `DataSource` implementations.

If your `DataSource` provides connections out of a connection pool, your configuration should set the collection lifecycle setting to `pooled`.

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();
```

In order to plug-in your own implementation of the `DataSource` interface, your application may use an existing JNDI provider as provided by an application server if running in a J2EE environment.

In case your application does not have an existing JNDI implementation to register a `DataSource` to provide connections, you may set the `java.naming.factory.initial` property in the configuration to point to your application's own implementation of the `javax.naming.spi.InitialContextFactory` interface that can return your application `DataSource` though the `javax.naming.Context` provided by the factory implementation. Please see Java Naming and Directory Interface (JNDI) API documentation for further information.

15.4.9.2. Connections obtained via DataSource Factory

This section describes how to use *Apache Commons Database Connection Pooling (Apache DBCP)* *[http://commons.apache.org/dbcp]* with Esper. We also explain how to provide a custom application-specific `DataSource` factory if not using Apache DBCP.

If your `DataSource` provides connections out of a connection pool, your configuration should set the collection lifecycle setting to `pooled`.

Apache DBCP provides comprehensive means to test for dead connections or grow and shrik a connection pool. Configuration properties for Apache DBCP can be found at *Apache DBCP configuration* *[http://commons.apache.org/dbcp/configuration.html]*. The listed properties are passed to Apache DBCP via the properties list provided as part of the Esper configuration.

The snippet of XML below is an example that configures a database named `mydb3` to obtain connections via the pooling `DataSource` provided by Apache DBCP `BasicDataSourceFactory`.

The listed properties are passed to DBCP to instruct DBCP how to manage the connection pool. The settings below initialize the connection pool to 2 connections and provide the validation query `select 1 from dual` for DBCP to validate a connection before providing a connection from the pool to Esper:

```
<database-reference name="mydb3">
  <!-- For a complete list of properties see Apache DBCP. -->
                             <datasourcefactory-connection              class-
name="org.apache.commons.dbcp.BasicDataSourceFactory">
    <env-property name="username" value ="myusername"/>
    <env-property name="password" value ="mypassword"/>
    <env-property name="driverClassName" value ="com.mysql.jdbc.Driver"/>
    <env-property name="url" value ="jdbc:mysql://localhost/test"/>
    <env-property name="initialSize" value ="2"/>
    <env-property name="validationQuery" value ="select 1 from dual"/>
  </datasourcefactory-connection>
  <connection-lifecycle value="pooled"/>
</database-reference>
```

The same configuration options provided through the API:

```
Properties props = new Properties();
props.put("username", "myusername");
props.put("password", "mypassword");
props.put("driverClassName", "com.mysql.jdbc.Driver");
props.put("url", "jdbc:mysql://localhost/test");
props.put("initialSize", 2);
props.put("validationQuery", "select 1 from dual");
```

```
ConfigurationDBRef configDB = new ConfigurationDBRef();
// BasicDataSourceFactory is an Apache DBCP import
configDB.setDataSourceFactory(props, BasicDataSourceFactory.class.getName());
configDB.setConnectionLifecycleEnum(ConfigurationDBRef.ConnectionLifecycleEnum.POOLED);

Configuration configuration = new Configuration();;
configuration.addDatabaseReference("mydb3", configDB);
```

Apache Commons DBCP is a separate download and not provided as part of the Esper distribution. The Apache Commons DBCP jar file requires the Apache Commons Pool jar file.

Your application can provide its own factory implementation for `DataSource` instances: Set the class name to the name of the application class that provides a public static method named `createDataSource` which takes a single `Properties` object as parameter and returns a `DataSource` implementation. For example:

```
configDB.setDataSourceFactory(props, MyOwnDataSourceFactory.class.getName());
...
class MyOwnDataSourceFactory {
  public static DataSource createDataSource(Properties properties) {
    return new MyDataSourceImpl(properties);
  }
}
```

15.4.9.3. 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.

15.4.9.4. 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.

15.4.9.5. 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.

In the `pooled` setting, the engine obtains a database connection from the data source or driver manager for every query, and closes the connection when done, returning the database connection to the pool if using a pooling data source.

In the `retain` setting, the engine retains a separate dedicated database connection for each statement and does not close the connection between uses.

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>
```

15.4.9.6. 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.

15.4.9.6.1. 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>
```

15.4.9.6.2. 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.

15.4.9.7. 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"/>
```

15.4.9.8. 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` which is a constant value of `2` per JDBC API to the Java `int` type.

```
<sql-types-mapping sql-type="2" java-type="int" />
```

15.4.9.9. Metadata Origin

This setting controls how the engine retrieves SQL statement metadata from JDBC prepared statements.

Table 15.3. 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.

15.4.10. Engine Settings related to Concurrency and Threading

15.4.10.1. 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'.

15.4.10.2. 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>
```

15.4.10.3. 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.

15.4.10.4. Advanced Threading Options

The settings described herein are for enabling advanced threading options for inbound, outbound, timer and route executions.

Take the next snippet of XML configuration as an example. It configures all threading options to 2 threads, which may not be suitable to your application, however demonstrates the configuration:

```
<engine-settings>
  <defaults>
    <threading>
      <threadpool-inbound enabled="true" num-threads="2"/>
      <threadpool-outbound enabled="true" num-threads="2" capacity="1000"/>
      <threadpool-timerexec enabled="true" num-threads="2"/>
      <threadpool-routeexec enabled="true" num-threads="2"/>
    </threading>
  </defaults>
```

```
</engine-settings>
```

By default, queues are unbound and backed by `java.util.concurrent.LinkedBlockingQueue`. The optional `capacity` attribute can be set to instruct the threading option to configure a capacity-bound queue with a sender-wait (blocking put) policy, backed `ArrayBlockingQueue`.

This example uses the API for configuring inbound threading :

```
Configuration config = new Configuration();
config.getEngineDefaults().getThreading().setThreadPoolInbound(true);
config.getEngineDefaults().getThreading().setThreadPoolInboundNumThreads(2);
```

With a bounded work queue, the queue size and pool size should be tuned together. A large queue coupled with a small pool can help reduce memory usage, CPU usage, and context switching, at the cost of potentially constraining throughput.

15.4.10.5. Engine Fair Locking

By default Esper configures the engine-level lock without fair locking. The engine-level lock coordinates event processing threads (threads that send events) with threads that perform administrative functions (threads that start or destroy statements, for example). A fair lock is generally less performing that an unfair lock thus the default configuration is an unfair lock.

If your application is multi-threaded and multiple threads sends events without gaps and if the per-event processing time is significant, then configuring a fair lock can help prioritize administrative functions. Administrative functions exclude event-processing threads until the administrative function completed. You may need to set this flag to prevent lock starvation to perform an administrative function in the face of concurrent event processing. Please consult the Java API documentation under `ReentrantReadWriteLock` and *Fair Mode* for more information.

The XML configuration to enable fair locking, which is disabled by default, is as follows:

```
<engine-settings>
  <defaults>
    <threading engine-fairlock="true"/>
  </defaults>
</engine-settings>
```

The API to change the setting:

```
Configuration config = new Configuration();
config.getEngineDefaults().getThreading().setEngineFairlock(true);
```

15.4.11. Engine Settings related to Event Metadata

15.4.11.1. Default Event Representation

The default event representation is the Map event representation. In a future release the default event representation is likely to switch to Object-array.

The default event representation is relevant when your query outputs individual properties to a listener and it does not specify a specific event representation in an annotation. The default event representation is also relevant for `create schema` and `create window`.

Note that the engine may still use the Map representation for certain types of statements even when the default event representation is object array.

For example, consider the following query:

```
select propertyOne, propertyTwo from MyEvent
```

Listeners to the statement above currently receive a Map-type event. By setting the configuration flag to object-array as described herein, listeners to the statement receive an Object-array-type event instead.

The XML snippet below is an example of setting the default event representation to Object-array:

```
<esper-configuration
  <engine-settings>
    <defaults>
      <event-meta>
        <event-representation type="objectarray"/>
      </event-meta>
    </defaults>
  </engine-settings>
</esper-configuration>
```

The code snippet shown next sets the default event representation to Object-array in the configuration object:

```
configuration.getEngineDefaults().getEventMeta().

 setDefaultEventRepresentation(Configuration.EventRepresentation.OBJECTARRAY);
```

15.4.11.2. Java Class Property Names, Case Sensitivity and Accessor Style

The engine-wide settings discussed here are used when you want to control case sensitivity or accessor style for all event classes as a default. The two settings are found under `class-property-resolution` under `event-meta` in the XML configuration.

To control the case sensitivity as discussed in *Section 15.4.1.6, "Case Sensitivity and Property Names"*, add the `style` attribute in the XML configuration to set a default case sensitivity applicable to all event classes unless specifically overridden by class-specific configuration. The default case sensitivity is `case_sensitive` (case sensitivity turned on).

To control the accessor style as discussed in *Section 15.4.1.3, "Non-JavaBean and Legacy Java Event Classes"*, add the `accessor-style` attribute in the XML configuration to set a default accessor style applicable to all event classes unless specifically overridden by class-specific configuration. The default accessor style is `javabean` JavaBean accessor style.

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);
config.getEngineDefaults().getEventMeta().setDefaultAccessorStyle(
    ConfigurationEventTypeLegacy.AccessorStyle.PUBLIC);
```

15.4.11.3. Cache Size for Anonymous Event Types

By default the engine maintains a cache of the last 5 recently allocated anonymous event types. Anonymous event types are unnamed output event types associated to statements. The `anonymous-cache` element under the `event-meta` element in the XML configuration contols the cache size. The cache size can be set to zero to disable the cache.

The next code snippet shows how to control this setting via the API:

```
Configuration config = new Configuration();
config.getEngineDefaults().getEventMeta().setAnonymousCacheSize(5);
```

15.4.12. Engine Settings related to View Resources

15.4.12.1. 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);
```

15.4.12.2. Configuring Multi-Expiry Policy Defaults

By default, when combining multiple data window views, Esper applies an intersection of the data windows unless the `retain-union` keyword is provided which instructs to apply an union. The setting described herein may be used primarily for backward compatibility to instruct that intersection should not be the default.

Here is a sample statement that specifies multiple expiry policies:

```
select * from MyEvent.std:unique(price).std:unique(quantity)
```

By default Esper applies intersection as described in *Section 5.4.4, “Multiple Data Window Views”*.

Here is the setting to allow multiple data windows without the intersection default:

```
Configuration config = new Configuration();
config.getEngineDefaults().getViewResources().setAllowMultipleExpiryPolicies(true);
```

When setting this option to true, and when using multiple data window views for a given stream, the behavior is as follows: The top-most data window receives an insert stream of events. It passes each insert stream event to each further data window view in the chain. Each data window view may remove events according to its expiry policy. Such remove stream events are only passed to data window views further in the chain, and are not made available to data window views earlier in the chain.

It is recommended to leave the default setting at false.

15.4.13. Engine Settings related to Logging

15.4.13.1. 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.

Statement-level processing information can be output via the `@Audit` annotation, please see *Section 16.3.1, “@Audit Annotation”*.

When debug-level logging is enabled by setting the flag as below and by setting DEBUG in the Log4j configuration file, then the timer processing may produce extensive debug output that you may not want to have in the log file. The `timer-debug` setting in the XML or via API as below disables timer debug output which is enabled by default.

The API to use to enable debug logging and disable timer event output is shown here:

```
Configuration config = new Configuration();
config.getEngineDefaults().getLogging().setEnableExecutionDebug(true);
config.getEngineDefaults().getLogging().setEnableTimerDebug(false);
```

Note: this is a configuration option that applies to all engine instances of a given Java module or VM.

The XML snippet is:

```
<esper-configuration>
  <engine-settings>
    <defaults>
      <logging>
        <execution-path enabled="true"/>
        <timer-debug enabled="false"/>
    </logging>
    </defaults>
  </engine-settings>
</esper-configuration>
```

15.4.13.2. Query Plan Logging

By default, the engine does not produce query plan output unless logging at debug-level. To enable query plan logging, set this option in the configuration. When enabled, the engine reports, at INFO level, any query plans under the log name `com.espertech.esper.queryplan`.

The API to use to enable query plan logging is shown here:

```
Configuration config = new Configuration();
config.getEngineDefaults().getLogging().setEnableQueryPlan(true);
```

The XML snippet is:

```
<esper-configuration>
  <engine-settings>
    <defaults>
      <logging>
        <query-plan enabled="true"/>
    </logging>
    </defaults>
  </engine-settings>
</esper-configuration>
```

15.4.13.3. JDBC Logging

By default, the engine does not measure JDBC query execution times or report the number of rows returned from a JDBC query through logging. To enable JDBC logging, set this option in the configuration. When enabled, the engine reports, at INFO level, any JDBC query performance and number of rows returned under the log name `com.espertech.esper.jdbc`.

The API to use to enable query plan logging is shown here:

```
Configuration config = new Configuration();
config.getEngineDefaults().getLogging().setEnableJDBC(true);
```

The XML snippet is:

```
<esper-configuration>
  <engine-settings>
    <defaults>
      <logging>
        <jdbc enabled="true"/>
    </logging>
    </defaults>
  </engine-settings>
</esper-configuration>
```

15.4.13.4. Audit Logging

The settings herein control the output format of `@Audit` logs.

This setting applies to all engine instances in the same JVM. Please also see the API documentation for information on pattern conversion characters.

Table 15.4. Audit Log Conversion Characters

Character	Description
`m`	Audit message.
`s`	Statement name.
`u`	Engine URI.

The API to use to set am audit log format is shown here:

```
Configuration config = new Configuration();
config.getEngineDefaults().getLogging().setAuditPattern("[%u] [%s] %m");
```

The XML snippet is:

```
<esper-configuration>
  <engine-settings>
    <defaults>
      <logging>
        <audit pattern="[%u] [%s]%m"/>
    </logging>
    </defaults>
  </engine-settings>
</esper-configuration>
```

15.4.14. Engine Settings related to Variables

15.4.14.1. 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 5.18.3, "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>
```

15.4.15. Engine Settings related to Patterns

15.4.15.1. Followed-By Operator Maximum Subexpression Count

You may use this setting to limit the total engine-wide number of pattern sub-expressions that all followed-by operators may manage. When the limit is reached, a condition is raised by the engine through the condition callback API.

By default, when the limit is reached, the engine also prevents the start of new pattern sub-expressions, until pattern sub-expressions end and the limit is no longer reached. By setting the `prevent-start` flag to false you can instruct the engine to only raise a condition and continue to allow the start of new pattern sub-expressions.

The implications of the settings described herein are also detailed in *Section 6.5.8.2, "Limiting Engine-wide Sub-Expression Count"*.

A sample XML configuration for this setting is shown below:

```
<engine-settings>
  <defaults>
    <patterns>
      <max-subexpression value="100" prevent-start="false"/>
    </patterns>
  </defaults>
</engine-settings>
```

The limit can be changed and disabled or enabled at runtime via the runtime configuration API. Pass a null value as the limit to disable limit checking.

A sample code snippet that sets a new limit is:

```
epService.getEPAdministrator().getConfiguration().setPatternMaxSubexpressions(100L);
```

15.4.16. Engine Settings related to Scripts

You may configure a default script dialect as described herein. The default script dialect is `js` which stands for JavaScript, since most JVM ship with an integrated JavaScript engine.

A sample XML configuration for this setting is shown below:

```
<engine-settings>
```

```
<defaults>
  <scripts default-dialect="js"/>
</defaults>
</engine-settings>
```

A sample code snippet that sets a new script dialect is:

```
config.getEngineDefaults().getScripts().setDefaultDialect("js");
```

15.4.17. Engine Settings related to Stream Selection

15.4.17.1. 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);
```

15.4.18. Engine Settings related to Time Source

15.4.18.1. Default Time Source

This setting only applies if internal timer events control engine time (default). If external timer events provide engine clocking, the setting does not apply.

By default, the internal timer uses the call `System.currentTimeMillis()` to determine engine time in milliseconds. Via this setting the internal timer can be instructed to use `System.nanoTime()` instead. Please see *Section 14.9, “Time Resolution”* for more information.

Note: This is a Java VM global setting. If running multiple engine instances in a Java VM, the timer setting is global and applies to all engine instances in the same Java VM, for performance reasons.

A sample XML configuration for this setting is shown below, whereas the sample setting sets the time source to the nanosecond time provider:

```
<engine-settings>
  <defaults>
    <time-source>
      <time-source-type value="nano" />
    </time-source>
  </defaults>
</engine-settings>
```

The equivalent code snippet using the configuration API is here:

```
Configuration config = new Configuration();
config.getEngineDefaults().getTimeSource().
      setTimeSourceType(ConfigurationEngineDefaults.TimeSourceType.NANO);
```

15.4.19. Engine Settings related to JMX Metrics

Please set the flag as described herein to have the engine report key counters and other processing information through the JMX mbean platform server. By default JMX is not enabled.

A sample XML configuration is shown below:

```
<engine-settings>
  <defaults>
    <metrics-reporting jmx-engine-metrics="true"/>
  </defaults>
</engine-settings>
```

A sample code snippet to set this configuration via the API follows:

```
configuration.getEngineDefaults().getMetricsReporting().setJmxEngineMetrics(true);
```

15.4.20. Engine Settings related to Metrics Reporting

This section explains how to enable and configure metrics reporting, which is by default disabled. Please see *Section 14.15, “Engine and Statement Metrics Reporting”* for more information on the metrics data reported to your application.

The flag that enables metrics reporting is global to a Java virtual machine. If metrics reporting is enabled, the overhead incurred for reporting metrics is carried by all engine instances per Java VM.

Metrics reporting occurs by an engine-controlled separate daemon thread that each engine instance starts at engine initialization time, if metrics reporting and threading is enabled (threading enabled is the default).

Engine and statement metric intervals are in milliseconds. A negative or zero millisecond interval value may be provided to disable reporting.

To control statement metric reporting for individual statements or groups of statements, the engine provides a facility that groups statements by statement name. Each such statement group may have different reporting intervals configured, and intervals can be changed at runtime through runtime configuration. A statement group is assigned a group name at configuration time to identify the group.

Metrics reporting configuration is part of the engine default settings. All configuration options are also available via the `Configuration` API.

A sample XML configuration is shown below:

```
<engine-settings>
  <defaults>
        <metrics-reporting  enabled="true"  engine-interval="1000"  statement-
interval="1000"
        threading="true"/>
  </defaults>
</engine-settings>
```

The `engine-interval` setting (defaults to 10 seconds) determines the frequency in milliseconds at which the engine reports engine metrics, in this example every 1 second. The `statement-interval` is for statement metrics. The `threading` flag is true by default since reporting takes place by a dedicated engine thread and can be set to false to use the external or internal timer thread instead.

The next example XML declares a statement group: The statements that have statement names that fall within the group follow a different reporting frequency:

```
<metrics-reporting enabled="true" statement-interval="0">
   <stmtgroup  name="MyStmtGroup"  interval="2000"  default-include="true"  num-
stmts="100"
        report-inactive="true">
    <exclude-regex>.*test.*</exclude-regex>
  </stmtgroup>
</metrics-reporting>
```

The above example configuration sets the `statement-interval` to zero to disable reporting for all statements. It defines a statement group by name `MyStmtGroup` and specifies a 2-second interval. The example sets the `default-include` flag to true (by default false) to include all statements in the statement group. The example also sets `report-inactive` to true (by default false) to report inactive statements.

The `exclude-regex` element may be used to specify a regular expression that serves to exclude statements from the group. Any statement whose statement name matches the exclude regular expression is not included in the group. In the above example, all statements with the characters 'test' inside their statement name are excluded from the group.

Any statement not belonging to any of the statement groups follow the configured statement interval.

There are additional elements available to include and exclude statements: `include-regex`, `include-like` and `exclude-like`. The latter two apply SQL-like matching. All patterns are case-sensitive.

Here is a further example of a possible statement group definition, which includes statements whose statement name have the characters `@REPORT` or `@STREAM`, and excludes statements whose statement name have the characters `@IGNORE` or `@METRICS` inside.

```
<metrics-reporting enabled="true">
  <stmtgroup name="MyStmtGroup" interval="1000">
    <include-like>%@REPORT%</include-like>
    <include-regex>.*@STREAM.*</include-like>
    <exclude-like>%@IGNORE%</exclude-like>
    <exclude-regex>.*@METRICS.*</exclude-regex>
  </stmtgroup>
</metrics-reporting>
```

15.4.21. Engine Settings related to Language and Locale

Locale-dependence in Esper can be present in the sort order of string values by the `order by` clause and by the sort view.

By default, Esper sorts string values using the `compare` method that is not locale dependent. To enable local dependent sorting you must set the configuration flag as described below.

The XML configuration sets the locale dependent sorting as shown below:

```
<engine-settings>
  <defaults>
    <language sort-using-collator="true"/>
  </defaults>
</engine-settings>
```

The API to change the setting:

```
Configuration config = new Configuration();
config.getEngineDefaults().getLanguage().setSortUsingCollator(true);
```

15.4.22. Engine Settings related to Expression Evaluation

15.4.22.1. Integer Division and Division by Zero

By default Esper returns double-typed values for divisions regardless of operand types. Division by zero returns positive or negative double infinity.

To have Esper use Java-standard integer division instead, use this setting as described here. In Java integer division, when dividing integer types, the result is an integer type. This means that if you divide an integer unevenly by another integer, it returns the whole number part of the result, does not perform any rounding and the fraction part is dropped. If Java-standard integer division is enabled, when dividing an integer numerator by an integer denominator, the result is an integer number. Thus the expression `1 / 4` results in an integer zero. Your EPL must then convert at least one of the numbers to a double value before the division, for example by specifying `1.0 / 4` or by using `cast(myint, double)`.

When using Java integer division, division by zero for integer-typed operands always returns null. However division by zero for double-type operands still returns positive or negative double infinity. To also return null upon division by zero for double-type operands, set the flag to true as below (default is false).

The XML configuration is as follows:

```
<engine-settings>
  <defaults>
    <expression integer-division="false" division-by-zero-is-null="false"/>
  </defaults>
</engine-settings>
```

The API to change the setting:

```
Configuration config = new Configuration();
config.getEngineDefaults().getExpression().setIntegerDivision(true);
config.getEngineDefaults().getExpression().setDivisionByZeroReturnsNull(true);
```

15.4.22.2. Subselect Evaluation Order

By default Esper updates sub-selects with new events before evaluating the enclosing statement. This is relevant for statements that look for the same event in both the `from` clause and subselects.

To have Esper evaluate the enclosing clauses before updating the subselect in a subselect expression, set the flag as indicated herein.

The XML configuration as below sets the same as the default value:

```
<engine-settings>
  <defaults>
    <expression self-subselect-preeval="true"/>
  </defaults>
</engine-settings>
```

Here is a sample statement that utilitzes a sub-select against the same-events:

```
select * from MyEvent where prop not in (select prop from
 MyEvent.std:unique(otherProp))
```

By default the subselect data window updates first before the `where` clause is evaluated, thereby above statement never returns results.

Changing the setting described here causes the `where` clause to evaluate before the subselect data window updates, thereby the statement does post results.

15.4.22.3. User-Defined Function or Static Method Cache

By default Esper caches the result of an user-defined function if the parameter set to that function is empty or all parameters are constant values. Results of custom plug-in single-row functions are not cached according to the default configuration, unless the single-row function is explicitly configured with value cache enabled.

To have Esper evaluate the user-defined function regardless of constant parameters, set the flag to false as indicated herein.

The XML configuration as below sets the same as the default value:

```
<engine-settings>
  <defaults>
    <expression udf-cache="true"/>
  </defaults>
</engine-settings>
```

15.4.22.4. Extended Built-in Aggregation Functions

By default Esper provides a number of additional aggregation functions over the SQL standards. To have Esper only allow the standard SQL aggregation functions and not the additional ones, disable the setting as described here.

The XML configuration as below sets the same as the default value:

```
<engine-settings>
  <defaults>
    <expression extend-agg="true"/>
  </defaults>
</engine-settings>
```

15.4.22.5. Duck Typing

By default Esper validates method references when using the dot operator syntax at time of statement creation. With duck typing, the engine resolves method references at runtime.

The XML configuration as below sets the same as the default value:

```
<engine-settings>
  <defaults>
    <expression ducktyping="false"/>
  </defaults>
</engine-settings>
```

15.4.22.6. Math Context

By default, when computing the average of BigDecimal values, the engine does not pass a `java.math.MathContext`. Use the setting herein to specify a default math context.

The below XML configuration sets precision to 2 and rounding mode ceiling:

```
<engine-settings>
  <defaults>
    <expression math-context="precision=2 roundingMode=CEILING"/>
  </defaults>
</engine-settings>
```

An example API configuration is shown next:

```
config.getEngineDefaults().getExpression().setMathContext();
```

15.4.23. Engine Settings related to Execution of Statements

15.4.23.1. Prioritized Execution

By default Esper ignores @Priority and @Drop annotations and executes unprioritized, that is the engine does not attempt to interpret assigned priorities and reorder executions based on priority. Use this setting if your application requires prioritized execution.

By setting this configuration, the engine executes statements, when an event or schedule matches multiple statements, according to the assigned priority, starting from the highest priority value. See built-in EPL annotations in *Section 5.2.7.6, “@Priority”*.

By enabling this setting view sharing between statements as described in *Section 15.4.12.1, “Sharing View Resources between Statements”* is disabled.

The XML configuration to enable the flag, which is disabled by default, is as follows:

```
<engine-settings>
  <defaults>
    <execution prioritized="true"/>
  </defaults>
</engine-settings>
```

The API to change the setting:

```
Configuration config = new Configuration();
```

```
config.getEngineDefaults().getExecution().setPrioritized(true);
```

15.4.23.2. Context Partition Fair Locking

By default Esper configures context partition locks without fair locking. If your application is multi-threaded and performs very frequent reads via iterator or fire-and-forget queries, you may need to set this flag to prevent lock starvation in the face of concurrent reads and writes. Please consult the Java API documentation under `ReentrantReadWriteLock` and *Fair Mode* for more information.

The XML configuration to enable fair locking, which is disabled by default, is as follows:

```
<engine-settings>
  <defaults>
    <execution fairlock="true"/>
  </defaults>
</engine-settings>
```

The API to change the setting:

```
Configuration config = new Configuration();
config.getEngineDefaults().getExecution().setFairlock(true);
```

15.4.23.3. Disable Locking

By default Esper configures context partition locks as required after analyzing your EPL statements. You may disable context partition locks engine-wide using the setting described here. Use the `@NoLock` annotation instead to disable locking for a given statement or named window only.

CAUTION: We provide this setting for the purpose of identifying locking overhead, or when your application is single-threaded, or when using an external mechanism for concurrency control. Setting disable-locking to true may have unpredictable results unless your application is taking concurrency under consideration.

The XML configuration to disable context level locking is as follows:

```
<engine-settings>
  <defaults>
    <execution disable-locking="false"/>
  </defaults>
</engine-settings>
```

The API to change the setting:

```
Configuration config = new Configuration();
config.getEngineDefaults().getExecution().setDisableLocking(true);
```

15.4.23.4. Threading Profile

This setting is for performance tuning when using a large number of threads, such as 100 or more threads.

The configuration provides a setting that instructs the engine to reduce the use of thread-local variables. As the engine may use thread-local variables to reduce temporary object allocation, it can potentially use too much memory when a large number of threads are used with the engine. By setting the threading profile to large, the engine will allocate temporary objects instead of using thread-local variables for certain logic. Currently this setting applies exclusively to patterns and their filter expressions.

The XML configuration to set a large threading profile is as follows:

```
<engine-settings>
  <defaults>
    <execution threading-profile="large"/>
  </defaults>
</engine-settings>
```

The API to change the setting:

```
Configuration config = new Configuration();
config.getEngineDefaults().getExecution().setThreadingProfile(
    ConfigurationEngineDefaults.ThreadingProfile.LARGE);
```

15.4.24. Engine Settings related to Exception Handling

Use the settings as described here to register an exception handler factory class that provides an exception handler. The engine invokes exception handlers in the order they are listed to handle a continues-query unchecked exception, as further described in *Section 14.11, “Exception Handling”*.

Please provide the full-qualified class name of each class that implements the `com.espertech.esper.client.hook.ExceptionHandlerFactory` interface in the engine defaults configuration as below.

The XML configuration is as follows:

```
<engine-settings>
```

```
  <defaults>
    <exceptionHandling>
    <handlerFactory class="my.company.cep.MyCEPEngineExceptionHandlerFactory"/>
    </exceptionHandling>
  </defaults>
</engine-settings>
```

The API calls to register an exception handler factory are as follows:

```
Configuration config = new Configuration();
config.getEngineDefaults().getExceptionHandling().addClass(MyCEPEngineExceptionHandlerFactory.c
```

15.4.25. Engine Settings related to Condition Handling

Use the settings as described here to register a condition handler factory class that provides a condition handler. The engine invokes condition handlers in the order they are listed to indicate conditions, which is the term used for notification when certain predefined limits are reached, as further described in *Section 14.12, “Condition Handling”*.

Please provide the full-qualified class name of each class that implements the `com.espertech.esper.client.hook.ConditionHandlerFactory` interface in the engine defaults configuration as below.

The XML configuration is as follows:

```
<engine-settings>
  <defaults>
    <conditionHandling>
    <handlerFactory class="my.company.cep.MyCEPEngineConditionHandlerFactory"/>
    </conditionHandling>
  </defaults>
</engine-settings>
```

The API calls to register a condition handler factory are as follows:

```
Configuration config = new Configuration();
config.getEngineDefaults().getConditionHandling().addClass(MyCEPEngineConditionHandlerFactory.c
```

15.4.26. Revision Event Type

Revision event types reflect a versioning relationship between events of same or different event types. Please refer to *Section 2.10, “Updating, Merging and Versioning Events”* and *Section 5.15.14, “Versioning and Revision Event Type Use with Named Windows”*.

The configuration consists of the following:

- An name of an event type whose events are *base* events.
- Zero, one or more names of event types whose events are *delta* events.
- One or more property names that supply the key values that tie base and delta events to existing revision events. Properties must exist on the event type as simple properties. Nested, indexed or mapped properties are not allowed.
- Optionally, a strategy for overlaying or merging properties. The default strategy is *Overlay Declared* as described below.

The XML configuration for this setting is shown below:

```
<revision-event-type name="UserProfileRevisions">
  <base-event-type name="ProfileCreation"/>
  <delta-event-type name="ProfileUpdate"/>
  <key-property name="userid"/>
</revision-event-type>
```

If configuring via runtime or initialization-time API, this code snippet explains how:

```
Configuration config = new Configuration();
ConfigurationRevisionEventType configRev = new ConfigurationRevisionEventType();
configRev.setNameBaseEventType("ProfileCreation");
configRev.addNameDeltaEventType("ProfileUpdate");
configRev.setKeyPropertyNames(new String[] {"userid"});
config.addRevisionEventType("UserProfileRevisions", configRev);
```

As the configuration provides names of base and delta event types, such names must be configured for JavaBean, Map or XML events as the previous sections outline.

The next table outlines the available strategies:

Table 15.5. Property Revision Strategies

Name	Description
Overlay Declared (default)	A fast strategy for revising events that groups properties provided by base and delta events and overlays contributed properties to compute a revision.

Name	Description
	For use when there is a limited number of combinations of properties that change on an event, and such combinations are known in advance. The properties available on the output revision events are all properties of the base event type. Delta event types do not add any additional properties that are not present on the base event type. Any null values or non-existing property on a delta (or base) event results in a null values for the same property on the output revision event.
Merge Declared	A strategy for revising events by merging properties provided by base and delta events, considering null values and non-existing (dynamic) properties as well. For use when there is a limited number of combinations of properties that change on an event, and such combinations are known in advance. The properties available on the output revision events are all properties of the base event type plus all additional properties that any of the delta event types provide. Any null values or non-existing property on a delta (or base) event results in a null values for the same property on the output revision event.
Merge Non-null	A strategy for revising events by merging properties provided by base and delta events, considering only non-null values. For use when there is an unlimited number of combinations of properties that change on an event, or combinations are not known in advance. The properties available on the output revision events are all properties of the base event type plus all additional properties that any of the delta event types provide. Null values returned by delta (or base) event properties provide no value to output revision events, i.e. null values are not merged.
Merge Exists	A strategy for revising events by merging properties provided by base and delta events, considering only values supplied by event properties that exist. For use when there is an unlimited number of combinations of properties that change on an event, or combinations are not known in advance. The properties available on the output revision events are all properties of the base event type plus all additional properties that any of the delta event types provide.

Name	Description
	All properties are treated as dynamic properties: If an event property does not exist on a delta event (or base) event the property provides no value to output revision events, i.e. non-existing property values are not merged.

15.4.27. Variant Stream

A *variant stream* is a predefined stream into which events of multiple disparate event types can be inserted, and which can be selected from in patterns and the `from` clause.

The name of the variant stream and, optionally, the type of events that the stream may accept, are part of the stream definition. By default, the variant stream accepts only the predefined event types. The engine validates your `insert into` clause which inserts into the variant stream against the predefined types.

A variant stream can be set to accept any type of event, in which case all properties of the variant stream are effectively dynamic properties. Set the `type variance` flag to `ANY` to indicate the variant stream accepts any type of event.

The following XML configuration defines a variant stream by name `OrderStream` that carries only `PartsOrder` and `ServiceOrder` events:

```
<variant-stream name="OrderStream">
  <variant-event-type name="PartsOrder"/>
  <variant-event-type name="ServiceOrder"/>
</variant-stream>
```

This code snippet sets up a variant stream by name `OutgoingEvent`:

```
Configuration config = new Configuration();
ConfigurationVariantStream variant = new ConfigurationVariantStream();
variant.setTypeVariance(ConfigurationVariantStream.TypeVariance.ANY);
config.addVariantStream("OutgoingEvent", variant);
```

If specifying variant event type names, make sure such names have been configured for JavaBean, Map or XML events.

15.5. Type Names

Certain configuration values accept type names. Type names can occur in the configuration of variable types, Map-event property types as well as XPath cast types, for example. Types names are not case-sensitive.

The table below outlines all possible type names:

Table 15.6. Variable Type Names

Type Name	Type
`string`, `varchar`, `varchar2` or `java.lang.String`	A string value
`int`, `integer` or `java.lang.Integer`	An integer value
`long` or `java.lang.Long`	A long value
`bool`, `boolean` or `java.lang.Boolean`	A boolean value
`double` or `java.lang.Double`	A double value
`float` or `java.lang.Float`	A float value
`short` or `java.lang.Short`	A short value
`char`, `character` or `java.lang.Character`	A character value
`byte` or `java.lang.Byte`	A byte value

15.6. Runtime 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. Please consult the JavaDoc documentation for more detail.

15.7. Logging Configuration

Esper logs all messages to Apache commons logging under an appropriate log level. To output log messages you can add Log4j to classpath and configure Log4j as below.

Esper's only direct dependency for logging is the Apache Commons Logging interfaces. You may use Log4j as described here, or you may use SLF4J instead (for example) as described in http://www.slf4j.org/legacy.html. You can also redirect SLF4J to any backend you want - nop, logback, jul as needed.

Statement-level processing information can be output, please see *Section 16.3.1, “@Audit Annotation”*.

For performance reasons, Esper does not log any debug-level or informational-level messages for event execution unless explicitly configured via *Section 15.4.13.1, “Execution Path Debug Logging”*.

A callback API for receiving certain critical engine reports is available as described in *Section 14.11, “Exception Handling”*.

More information on configuring engine-level settings for logging are at *Section 15.4.13, “Engine Settings related to Logging”*.

Apache Commons Logging uses an automatic discovery mechanism to find a log framework that it will delegate to. If your application has another component using logback, please inspect you logback configuration and add `com.espertech`.

The next table explains the log levels:

Table 15.7. Log Levels

Log Level	Use
Debug	Displays detailed engine-internal information that may not be easy to understand for application developers but are useful for engine support.
Info	Used for a few critical engine-level log messages.
Warn	Certain important warning or informational messages are displayed under the warning level.
Error	Exceptions reported within the engine or by plug-in components are reported under the error level. When users enter invalid EPL statements such validation errors are not reported as error logs and are indicated via API exception instead.

15.7.1. Log4j Logging Configuration

Log4j is the default logging component. Please find additional information for Log4j configuration and extension in *http://logging.apache.org/log4j*.

The easiest way to configure Log4j is by providing a Log4J configuration file, similar to the `log4j.xml` file shipped in the `etc` folder of the distribution.

Add the `log4j.configuration` system property to the `java` command line and provide the file name of the Log4j configuration file, making sure your classpath also includes the directory of the file:

```
java -Dlog4j.configuration=log4j.xml ...
```

Chapter 16. Development Lifecycle

This chapter presents information related to the development lifecycle for developing an event processing application with EPL. It includes information on authoring, testing, debugging, packaging and deploying.

16.1. Authoring

Enterprise Edition includes authoring tools for EPL statements and modules by providing form-based dialogs, templates, an expression builder, simulation tool and other tools. Enterprise Edition also supports hot deployment and packaging options for EPL and related code.

EPL statements can be organized into modules as described below. Any text editor can edit EPL statement and module text. A text editor or IDE that highlights SQL syntax or keywords works.

For authoring configuration files please consult the XSD schema files as provided with the distribution.

For information on authoring event classes or event definitions in general please see *Chapter 2, Event Representations* or *Section 5.16, "Declaring an Event Type: Create Schema"*.

16.2. Testing

We recommend testing EPL statements using a test framework such as JUnit or TestNG. Please consult the Esper test suite for extensive examples, which can be downloaded from the distribution site.

Esper's API provides test framework classes to simplify automated testing of EPL statements. Please see *Section 14.21, "Test and Assertion Support"* for more information.

We recommend performing latency and throughput tests early in the development lifecycle. Please consider the performance tips in *Chapter 20, Performance* for optimal performance.

Consider engine and statement metrics reporting for identifying slow-performing statements, for example. See *Section 14.15, "Engine and Statement Metrics Reporting"*.

16.3. Debugging

One important tool for debugging is the parameterized `@Audit` annotation. This annotation allows to output, on statement-level, detailed information about many aspects of statement processing.

Another tool for logging engine-level detail is *Section 15.4.13.1, "Execution Path Debug Logging"*.

Please see *Section 15.7, "Logging Configuration"* for information on configuring logging in general.

16.3.1. @Audit Annotation

Use the `@Audit` annotation to have the engine output detailed information about statement processing. The engine reports, at INFO level, the information under log name `com.espertech.esper.audit`. You may define an output format for audit information via configuration.

You may provide a comma-separated list of category names to `@Audit` to output information related to specific categories only. The table below lists all available categories. If no parameter is provided, the engine outputs information for all categories. Category names are not case-sensitive.

For the next statement the engine produces detailed processing information (all categories) for the statement:

```
@Name('All Order Events') @Audit select * from OrderEvent
```

For the next statement the engine provides information about new events and also about event property values (2 categories are listed):

```
@Name('All Order Events') @Audit('stream,property') select price from OrderEvent
```

Here is a more complete example that uses the API to create the schema, create above statement and send an event:

```
epService.getEPAdministrator().createEPL("create    schema    OrderEvent(price
 double)");
String epl = "@Name('All-Order-Events') @Audit('stream,property') select price
 from OrderEvent";
epService.getEPAdministrator().createEPL(epl).addListener(listener);
epService.getEPRuntime().sendEvent(Collections.singletonMap("price",    100d),
 "OrderEvent");
```

The output is similar to the following:

```
INFO  [audit] Statement All-Order-Events stream OrderEvent inserted {price=100.0}
INFO  [audit] Statement All-Order-Events property price value 100.0
```

Table 16.1. @Audit Categories

Category	Description
Dataflow-Source	Each data flow source operator providing an event.
Dataflow-Op	Each data flow operator processing an event.

Category	Description
Dataflow-Transition	Each data flow instance state transition.
Exprdef	Each expression declaration name and return value.
Expression	Each top-level expression and its return value.
Expression-nested	Each expression including child or nested expressions and their return value.
Insert	Each event inserted via insert-into.
Pattern	Each pattern sub-expression and its change in truth-value.
Pattern-instances	Each pattern sub-expression and its count of active instances.
Property	Each property name and the event's property value.
Schedule	Each schedule modification and trigger received by a statement.
Stream	Each new event received by a statement.
View	Each view name and its insert and remove stream.

Note that the engine only evaluates select-clause expressions if either a listener or subscriber is attached to the statement or if used with insert-into.

16.4. Packaging and Deploying Overview

Please consider Esper Enterprise Edition as a target deployment platform. Esper alone does not ship with a server as it is designed as a core CEP engine.

To support packaging and deploying event-driven applications, Esper offers infrastructure as outlined herein:

- EPL modules to build a cohesive, easily-externalizable deployment unit out of related statements as described in *Section 16.5, “EPL Modules”*.
- The deployment administrative interface is described in *Section 16.6, “The Deployment Administrative Interface”*.
- Instructions and code for use when the deployment target is a J2EE web application server or servlet runtime, please see *Section 16.7, “J2EE Packaging and Deployment”*.

16.5. EPL Modules

An EPL module file is a plain text file in which EPL statements appear separated by the semicolon (;) character. It bundles EPL statements with optional deployment instructions. A service provider instance keeps track of the known and/or deployed EPL modules and makes it easy to add, remove, deploy and undeploy EPL modules.

The synopsis of an EPL module file is:

```
[module module_name;]
[uses module_name; | import import_name;] [uses module_name; |
 import import_name;] [...]
[epl_statement;] [epl_statement;] [...]
```

Use the `module` keyword followed a *module_name* identifier or a package (identifiers separated by dots) to declare the name of the module. The module name declaration must be at the beginning of the file, comments and whitespace excluded. The module name serves to check uses-dependences of other modules.

If a module file requires certain constructs that may be shared by other module files, such as named windows, variables, event types, variant streams or inserted-into streams required by statements, a module file may specify zero to many dependent modules with the `uses` keyword. At deployment time the engine checks the uses-dependencies and ensures that a module of that name is already deployed or will be deployed as part of the deployments. The deployment API supports ordering modules according to their uses-relationship.

If the EPL statements in the module require Java classes such as for underlying events or user-defined functions, use the `import` keyword followed by the fully-qualified class name or package name in the format `package.*`. The `uses` and `import` keywords are optional and must occur after the `module` declaration.

Following the optional deployment instructions are any number of *epl_statement* EPL statements that are separated by semicolon (`;`).

The following is a sample EPL module file explained in detail thereafter:

```
// Declare the name for the module
module org.myorganization.switchmonitor;

// Declare other module(s) that this module depends on
uses org.myorganization.common;

// Import any Java/.NET classes in an application package
import org.myorganization.events.*;

// Declare an event type based on a Java class in the package that was imported
 as above
create schema MySwitchEvent as MySwitchEventPOJO;

// Sample statement
@Name('Off-On-Detector')
insert into MyOffOnStream
select * from pattern[every-distinct(id) a=MySwitchEvent(status='off')
  -> b=MySwitchEvent(id=a.id, status='on')];
```

```
// Sample statement
@Name('Count-Switched-On')
@Description('Count per switch id of the number of Off-to-On switches in the
 last 1 hour')
select id, count(*) from MyOffOnStream.win:time(1 hour) group by id;
```

The example above declares a module name of `org.myorganization.switchmonitor`. As defined by the `uses` keyword, it ensures that the `org.myorganization.common` module is already deployed. The example demonstrates the `import` keyword to make a package name known to the engine for resolving POJO class names, as the example assumes that `MySwitchEventPOJO` is a POJO event class. In addition the example module contains two statements separated by semicolon characters.

Your application code may, after deployment, look up a statement and attach listeners as shown here:

```
epService.getEPAdministrator().getStatement("Count-Switched-
On").addListener(...);
```

16.6. The Deployment Administrative Interface

The `com.espertech.esper.client.deploy.EPDeploymentAdmin` service available from the `EPAdministrator` interface by method `getDeploymentAdmin` provides the functionality available to manage packaging and deployment. Please consult the JavaDoc documentation for more information.

The deployment API allows to read resources and parse text strings to obtain an object representation of the EPL module, the `Module`. A `Module` object can also be simply constructed.

After your application obtains a `Module` instance it may either use `deploy` to deploy the module directly, starting all statements of the module. Alternatively your application may add a module, making it known without starting statements for later deployment. In each case the module is assigned a deployment id, which acts as a unique primary key for all known modules. Your application may assign its own deployment id or may have the engine generate a deployment id (two footprints for `add` and `deploy` methods).

A module may be in two states: undeployed or deployed. When calling `add` to add a module, it starts life in the undeployed state. When calling `deploy` to deploy a module, it starts life in the deployed state. A module may be transitioned by providing the deployment id and by calling the `deploy` or `undeploy` methods.

Your code can remove a module in undeployed state using the `remove` method or the `undeployRemove` method. If the module is in deployed state, use `undeployRemove` to undeploy and remove the module.

The `DeploymentOptions` instance that can be passed to the `deploy` method when validating or deploying modules controls validation, fail-fast, rollback and the isolated service provider, if any, for the deployment. Also use `DeploymentOptions` to set a user object per statement or to set a statement name per statement.

We also provide additional sample code to read and deploy modules as part of the J2EE considerations below.

16.6.1. Reading Module Content

Read and parse module files via the `EPDeploymentAdmin` interface `read` and `parse` methods, which returns a `Module` instance to represent the module information.

This code snippet demonstrates reading and parsing a module given a file name:

```
EPServiceProvider epService = EPServiceProviderManager.getDefaultProvider();
EPDeploymentAdmin deployAdmin =
 epService.getEPAdministrator().getDeploymentAdmin();
Module module = deployAdmin.read(new File("switchmonitor.epl"));
```

The service provides additional read and parse methods to read from a URL, classpath, input stream or string.

16.6.2. Ordering Multiple Modules

Since modules may have inter-dependencies as discussed under the `uses` declaration, the deployment interface provides the `getDeploymentOrder` method to order a collection of modules before deployment.

Assuming your application reads multiple modules into a `mymodules` module list, this code snippet orders the modules for deployment and validates dependency declarations for each module:

```
List<Module> mymodules =  ... read modules...;
DeploymentOrder order = deployAdmin.getDeploymentOrder(mymodules, new
 DeploymentOrderOptions());
```

16.6.3. Deploying and Undeploying

The deployment interface returns a deployment id for each module made known by adding a module or by deploying a module. To undeploy the module your application must provide the deployment id. Your application can assign its own deployment id or obtain the module name from the `Module` and use that as the deployment id.

The `undeploy` operation removes all named windows, variables, event types or any other information associated to the statements within the module to be undeployed.

The next code snippet deploys modules, starting each modulle's EPL statements:

```
for (Module mymodule : order.getOrdered()) {
      DeploymentResult   deployResult   =   deployAdmin.deploy(mymodule,   new
 DeploymentOptions());
}
```

Undeploying a module destroys all started statements associated to the module.

To undeploy and at the same time remove the module from the list of known modules use the `undeployRemove` method and pass the deployment id:

```
deployAdmin.undeployRemove(deployResult.getDeploymentId());
```

16.6.4. Listing Deployments

The deployment interface returns all module information that allows your application to determine which modules are known and their current state.

To obtain a list of all known modules or information for a specific module, the calls are:

```
DeploymentInformation[] info = deployAdmin.getDeploymentInformation();

// Given a deployment id, return the deployment information
DeploymentInformation infoModule = deployAdmin.getDeploymen(deploymentId);
```

16.6.5. State Transitioning a Module

The following sample code adds a module, transitions the module to deployed, then undeploys and removes the module entirely;

```
// This sample uses the parse method to obtain a module
Module module = deployAdmin.parse("create schema MySchema (col1 int)";

// Make the module know; It now shows up in undeployed state
String moduleDeploymentId = deployAdmin.add(module);

// Start all statements, passing a null options object for default options
deployAdmin.deploy(moduleDeploymentId, null);

// Undeploy module, destroying all statements
deployAdmin.undeploy(moduleDeploymentId);
```

```
// Remove module; It will no longer be known
deployAdmin.remove(moduleDeploymentId);
```

16.6.6. Best Practices

Use the `@Name` annotation to assign a name to each statement that your application would like to attach a listener or subscriber, or look up the statement for iteration or management by the administrative API.

Use the `create schema` syntax and the `import` keyword to define event types. When sharing event types, named windows or variables between modules use the `uses` keyword to declare a separate module that holds the shared definitions.

To validate whether a set of statements is complete and can start without issues, set the following flags on a `DeploymentOptions` instance passed to the `deploy` method as the code snippet below shows:

```
DeploymentOptions options = new DeploymentOptions();
options.setIsolatedServiceProvider("validation"); // we isolate any statements
options.setValidateOnly(true); // validate leaving no started statements
options.setFailFast(false); // do not fail on first error
epService.getEPAdministrator().getDeploymentAdmin()
  .deploy(module, options);
```

16.7. J2EE Packaging and Deployment

Esper can well be deployed as part of a J2EE web or enterprise application archive to a web application server. When designing for deployment into a J2EE web application server, please consider the items discussed here.

We provide a sample servlet context listener in this section that uses the deployment API to deploy and undeploy modules as part of the servlet lifecycle.

The distribution provides a message-driven bean (MDB) example that you may find useful.

Esper does not have a dependency on any J2EE or Servlet APIs to allow the engine to run in any environment or container.

16.7.1. J2EE Deployment Considerations

As multiple web applications deployed to a J2EE web application server typically have a separate classloader per application, you should consider whether engine instances need to be shared between applications or can remain separate engine instances. Consider the `EPServiceProviderManager` a Singleton. When deploying multiple web applications, your J2EE container classloader may provide a separate instance of the Singleton

`EPServiceProviderManager` to each web application resulting in multiple independent engine instances.

To share `EPServiceProvider` instances between web applications, one approach is to add the Esper jar files to the system classpath. A second approach can be to have multiple web applications share the same servet context and have your application place the `EPServiceProvider` instance into a servlet context attribute for sharing. Architecturally you may also consider a single archived application (such as an message-driven bean) that all your web applications communicate to via the JMS broker provided by your application server or an external JMS broker.

As per J2EE standards there are restrictions in regards to starting new threads in J2EE application code. Esper adheres to these restrictions: It allows to be driven entirely by external events. To remove all Esper threads, set the internal timer off and leave the advanced threading options turned off. To provide timer events when the internal timer is turned off, you should check with your J2EE application container for support of the Java system timer or for support of batch or work loading to send timer events to an engine instance.

As per J2EE standards there are restrictions in regards to input and output by J2EE application code. Esper adheres to these restrictions: By itself it does not start socket listeners or performs any file IO.

16.7.2. Servlet Context Listener

When deploying a J2EE archive that contains EPL modules files we provide sample code to read and deploy EPL modules files packaged with the enterprise or web application archive when the servlet initializes. The sample undeploys EPL modules when the servlet context gets destroyed.

A sample `web.xml` configuration extract is:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  <listener>
    <listener-class>SampleServletListener</listener-class>
  </listener>
  <context-param>
    <param-name>eplmodules</param-name>
    <param-value>switchmonitor.epl</param-value>
</context-param>
</web-app>
```

A servet listener that deploys EPL module files packaged into the archive on context initialization and that undeploys when the application server destroys the context is shown here:

```
public class SampleServletListener implements ServletContextListener {
  private List<String> deploymentIds = new ArrayList<String>();

  public void contextInitialized(ServletContextEvent servletContextEvent) {
    try {
                              EPServiceProvider     epServiceProvider     =
 EPServiceProviderManager.getDefaultProvider();
                                           String       modulesList       =
 servletContextEvent.getServletContext().getInitParameter("eplmodules");
      List<Module> modules = new ArrayList<Module>();
      if (modulesList != null) {
        String[] split = modulesList.split(",");
        for (int i = 0; i < split.length; i++) {
          String resourceName = split[i].trim();
          if (resourceName.length() == 0) {
            continue;
          }
                                                   String     realPath     =
 servletContextEvent.getServletContext().getRealPath(resourceName);
    Module module = epServiceProvider.getEPAdministrator()
            .getDeploymentAdmin().read(new File(realPath));
          modules.add(module);
        }
      }

      // Determine deployment order
      DeploymentOrder order = epServiceProvider.getEPAdministrator()
                  .getDeploymentAdmin().getDeploymentOrder(modules, null);

      // Deploy
      for (Module module : order.getOrdered()) {
        DeploymentResult result = epServiceProvider.getEPAdministrator()
                 .getDeploymentAdmin().deploy(module, new DeploymentOptions());
        deploymentIds.add(result.getDeploymentId());
      }
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  public void contextDestroyed(ServletContextEvent servletContextEvent) {
                           EPServiceProvider       epServiceProvider       =
 EPServiceProviderManager.getDefaultProvider();
    for (String deploymentId : deploymentIds) {

 epServiceProvider.getEPAdministrator().getDeploymentAdmin().undeployRemove(deploymentId);
    }
  }
```

```
}
```

16.8. Monitoring and JMX

The engine can report key processing metrics through the JMX platform mbean server by setting a single configuration flag described in *Section 15.4.19, “Engine Settings related to JMX Metrics”*.

Engine and statement-level metrics reporting is described in *Section 14.15, “Engine and Statement Metrics Reporting”*.

Chapter 17. Integration and Extension

17.1. Overview

This chapter summarizes integration and describes in detail each of the extension APIs that allow integrating external data and/or extend engine functionality.

For information on input and output adapters that connect to an event transport and perform event transformation for incoming and outgoing on-the-wire event data, for use with streaming data, please see the EsperIO reference documentation. The data flow instances as described in *Chapter 13, EPL Reference: Data Flow* are an easy way to plug in operators that perform input and output. Data flows allow providing parameters and managing individual flows independent of engine lifecycle. Also consider using the Plug-in Loader API for creating a new adapter that starts or stops as part of the CEP engine initialization and destroy lifecycle, see *Section 14.17, "Plug-in Loader"*.

To join data that resides in a relational database and that is accessible via JDBC driver and SQL statement the engine offers a syntax for using SQL within EPL, see *Section 5.13, "Accessing Relational Data via SQL"*. A relational database input and output adapter for streaming input from and output to a relational database also exists (EsperIO).

To join data that resides in a non-relational store the engine offers a two means: First, the virtual data window, as described below, for transparently integrating the external store as a named window. The second mechanism is a special join syntax based on static method invocation, see *Section 5.14, "Accessing Non-Relational Data via Method Invocation"*.

Tip

The best way to test that your extension code works correctly is to write unit tests against an EPL statement that utilizes the extension code. Samples can be obtained from Esper regression test code base.

Note

For all extension code and similar to listeners and subscribers, to send events into the engine from extension code the `route` method should be used (and not `sendEvent`) to avoid the possibility of stack overflow due to event-callback looping and ensure correct processing of the current and routed event .

Note

For all extension code it is not safe to administrate the engine within the extension code. For example, it is not safe to implement a data window view that creates a new statement or destroys an existing statement.

17.2. Virtual Data Window

Use a virtual data window if you have a (large) external data store that you want to access as a named window. The access is transparent: There is no need to use special syntax or join syntax. All regular queries including subqueries, joins, on-merge, on-select, on-insert, on-delete, on-update and fire-and-forget are supported with virtual data windows.

There is no need to keep any data or events in memory with virtual data windows. The only requirement for virtual data windows is that all data rows returned are `EventBean` instances.

When implementing a virtual data window it is not necessary to send any events into the engine or to use insert-into. The event content is simply assumed to exist and accessible to the engine via the API implementation you provide.

The distribution ships with a sample virtual data window in the examples folder under the name `virtualdw`. The code snippets below are extracts from the example.

We use the term *store* here to mean a source set of data that is managed by the virtual data window. We use the term *store row* or just *row* to mean a single data item provided by the store. We use the term *lookup* to mean a read operation against the store returning zero, one or many rows.

Virtual data windows allow high-performance low-latency lookup by exposing all relevant EPL query access path information. This makes it possible for the virtual data window to choose the desired access method into its store.

The following steps are required to develop and use a virtual data window:

1. Implement the interface `com.espertech.esper.client.hook.VirtualDataWindowFactory`.
2. Implement the interface `com.espertech.esper.client.hook.VirtualDataWindow`.
3. Implement the interface `com.espertech.esper.client.hook.VirtualDataWindowLookup`.
4. Register the factory class in the engine configuration.

Once you have completed above steps, the virtual data window is ready to use in EPL statements.

From a threading perspective, virtual data window implementation classes must be thread-safe if objects are shared between multiple named windows. If no objects are shared between multiple different named windows, thereby each object is only used for the same named window and other named windows receive a separate instance, it is no necessary that the implementation classes are thread-safe.

17.2.1. How to Use

Your application must first register the virtual data window factory as part of engine configuration:

```
Configuration config = new Configuration();
config.addPlugInVirtualDataWindow("sample", "samplevdw",
    SampleVirtualDataWindowFactory.class.getName());
```

Your application may then create a named window backed by a virtual data window.

For example, assume that the `SampleEvent` event type is declared as follows:

```
create schema SampleEvent as (key1 string, key2 string, value1 int, value2 double)
```

The next EPL statement creates a named window `MySampleWindow` that provides `SampleEvent` events and is backed by a virtual data window provided by `SampleVirtualDataWindowFactory` as configured above:

```
create window MySampleWindow.sample:samplevdw() as SampleEvent
```

You may then access the named window, same as any other named window, for example by subquery, join, on-action, fire-and-forget query or by consuming its insert and remove stream. While this example uses Map-type events, the example code is the same for POJO or other events.

Your application may obtain a reference to the virtual data window from the engine context.

This code snippet looks up the virtual data window by the named window name:

```
try {
      return   (VirtualDataWindow)   epService.getContext().lookup("/virtualdw/
MySampleWindow");
}
catch (NamingException e) {
   throw  new  RuntimeException("Failed  to  look  up  virtual  data  window,  is  it
 created yet?");
}
```

17.2.1.1. Query Access Path

When you application registers a subquery, join or on-action query or executes a fire-and-forget query against a virtual data window the engine interacts with the virtual data window. The interaction is a two-step process.

At time of EPL statement creation (once), the engine analyzes the EPL where-clause, if present. It then compiles a list of hash-index and binary tree (btree, i.e. sorted) index properties. It passes the property names that are queried as well as the operators (i.e. =, >, range etc.) to the virtual data window. The virtual data window returns a lookup strategy object to the engine.

At time of EPL statement execution (repeatedly as triggered) , the engine uses that lookup strategy object to execute a lookup. It passes to the lookup all actual key values (hash, btree including ranges) to make fast and efficient lookup achievable.

To explain in detail, assume that your application creates an EPL statement with a subquery as follows:

```
select (select * from MySampleWindow where key1 = 'A1') from OtherEvent
```

At the time of creation of the EPL query above the engine analyzes the EPL query. It determines that the subquery queries a virtual data window. It determines from the where-clause that the lookup uses property `key1` and hash-equals semantics. The engine then provides this information as part of `VirtualDataWindowLookupContext` passed to the `getLookup` method. Your application may inspect hash and btree properties and may determine the appropriate store access method to use.

The hash and btree property lookup information is for informational purposes, to enable fast and performant queries that returns the smallest number of rows possible. Your implementation classes may use some or none of the information provided and may also instead return some or perhaps even all rows, as is practical to your implementation. The `where`-clause still remains in effect and gets evaluated on all rows that are returned by the lookup strategy.

Following the above example, the sub-query executes once when a `OtherEvent` event arrives. At time of execution the engine delivers the string value `A1` to the `VirtualDataWindowLookup` lookup implementation provided by your application. The lookup object queries the store and returns store rows as `EventBean` instances.

As a second example, consider an EPL join statement as follows:

```
select * from MySampleWindow, MyTriggerEvent where key1 = trigger1 and key2 =
 trigger2
```

The engine analyzes the query and passes to the virtual data window the information that the lookup occurs on properties `key1` and `key2` under hash-equals semantics. When a `MyTriggerEvent` arrives, it passes the actual value of the `trigger1` and `trigger2` properties of the current MyTriggerEvent to the lookup.

As a last example, consider an EPL fire-and-forget statement as follows:

```
select * from MySampleWindow key1 = 'A2' and value1 between 0 and 1000
```

The engine analyzes the query and passes to the virtual data window the lookup information. The lookup occurs on property `key1` under hash-equals semantics and on property `value1` under btree-open-range semantics. When you application executes the fire-and-forget query the engine passes `A2` and the range endpoints `0` and `1000` to the lookup.

For more information, please consult the JavaDoc API documentation for class `com.espertech.esper.client.hook.VirtualDataWindow`, `VirtualDataWindowLookupContext` or `VirtualDataWindowLookupFieldDesc`.

17.2.2. Implementing the Factory

For each named window that refers to the virtual data window, the engine instantiates one instance of the factory.

A virtual data window factory class is responsible for the following functions:

- Implement the `initialize` method that accepts a virtual data window factory context object as a parameter.
- Implement the `create` method that accepts a virtual data window context object as a parameter and returns a `VirtualDataWindow` implementation.
- Implement the `destroyAllContextPartitions` method that gets called once when the named window is stopped or destroyed.

The engine instantiates a `VirtualDataWindowFactory` instance for each named window created via `create window`. The engine invokes the `initialize` method once in respect to the named window being created passing a `VirtualDataWindowFactoryContext` context object.

If not using contexts, the engine calls the `create` method once after calling the `initialize` method. If using contexts, the engine calls the `create` method every time it allocates a context partition. If using contexts and your virtual data window implementation operates thread-safe, you may return the same virtual data window implementation object for each context partition. If using contexts and your implementation object is not thread safe, return a separate thread-safe implementation object for each context partition.

The engine invokes the `destroyAllContextPartitions` once when the named window is stopped or destroyed. If not using contexts, the engine calls the `destroy` method of the virtual data window implementation object before calling the `destroyAllContextPartitions` method on the factory object. If using contexts, the engine calls the `destroy` method on each instance associates to a context partition at the time when the associated context partition terminates.

The sample code shown here can be found among the examples in the distribution under `virtualdw`:

```
public class SampleVirtualDataWindowFactory implements VirtualDataWindowFactory
 {

    public void initialize(VirtualDataWindowFactoryContext factoryContext) {
  // Can add initialization logic here.
    }

  public VirtualDataWindow create(VirtualDataWindowContext context) {
   // This example allocates a new virtual data window (one per context partitions
 if using contexts).
    // For sharing the virtual data window instance between context partitions,
 return the same reference.
    return new SampleVirtualDataWindow(context);
  }

  public void destroyAllContextPartitions() {
    // Release shared resources here
  }
}
```

Your factory class must implement the `create` method which receives a `VirtualDataWindowContext` object. This method is called once for each EPL that creates a virtual data window (see example `create window` above).

The `VirtualDataWindowContext` provides to your application:

```
String namedWindowName; // Name of named window being created.
Object[] parameters;  // Any optional parameters provided as part of create-
window.
EventType eventType;  // The event type of events.
EventBeanFactory eventFactory;  // A factory for creating EventBean instances
 from store rows.
VirtualDataWindowOutStream outputStream;   // For stream output to consuming
 statements.
AgentInstanceContext agentInstanceContext;  // Other EPL statement information
 in statement context.
```

When using contexts you can decide whether your factory returns a new virtual data window for each context partition or returns the same virtual data window instance for all context partitions. Your extension code may refer to the named window name to identify the named window and may refer to the agent instance context that holds the agent instance id which is the id of the context partition.

17.2.3. Implementing the Virtual Data Window

A virtual data window implementation is responsible for the following functions:

- Accept the lookup context object as a parameter and return the `VirtualDataWindowLookup` implementation.
- Optionally, post insert and remove stream data.
- Implement the `destroy` method, which the engine calls for each context partition when the named window is stopped or destroyed, or once when a context partition is ended/terminated.

The sample code shown here can be found among the examples in the distribution under `virtualdw`.

The implementation class must implement the `VirtualDataWindow` interface like so:

```
public class SampleVirtualDataWindow implements VirtualDataWindow {

  private final VirtualDataWindowContext context;

  public SampleVirtualDataWindow(VirtualDataWindowContext context) {
    this.context = context;
  } ...
```

When the engine compiles an EPL statement and detects a virtual data window, the engine invokes the `getLookup` method indicating hash and btree access path information by passing a `VirtualDataWindowLookupContext` context. The lookup method must return a `VirtualDataWindowLookup` implementation that the EPL statement uses for all lookups until the EPL statement is stopped or destroyed.

The sample implementation does not use the hash and btree access path information and simply returns a lookup object:

```
public VirtualDataWindowLookup getLookup(VirtualDataWindowLookupContext desc) {

  // Place any code that interrogates the hash-index and btree-index fields here.

  // Return the lookup strategy.
  return new SampleVirtualDataWindowLookup(context);
}
```

If your virtual data window returns null instead of a lookup object, the EPL query creation fails and throws an `EPStatementException`.

The engine calls the `update` method when data changes because of on-merge, on-delete, on-update or insert-into. For example, if you have an on-merge statement that is triggered and that updates the virtual data window, the `newData` parameter receives the new (updated) event and the `oldData` parameters receives the event prior to the update. Your code may use these events to update the store or delete from the store, if needed.

If your application plans to consume data from the virtual data window, for example via `select * from MySampleWindow`, then the code must implement the `update` method to forward insert and remove stream events, as shown below, to receive the events in consuming statements. To post insert and remove stream data, use the `VirtualDataWindowOutStream` provided by the context object as follows.

```
public void update(EventBean[] newData, EventBean[] oldData) {
  // This sample simply posts into the insert and remove stream what is received.
  context.getOutputStream().update(newData, oldData);
}
```

Your application should not use `VirtualDataWindowOutStream` to post new events that originate from the store. The object is intended for use with on-action EPL statements. Use insert-into instead for any new events that originate from the store.

17.2.4. Implementing the Lookup

A lookup implementation is responsible for the following functions:

- Accept the lookup values as a parameter and return a set of `EventBean` instances.

The sample code shown here can be found among the examples in the distribution under `virtualdw`.

The implementation class must implement the `VirtualDataWindowLookup` interface:

```
public class SampleVirtualDataWindowLookup implements VirtualDataWindowLookup {

  private final VirtualDataWindowContext context;

  public SampleVirtualDataWindowLookup(VirtualDataWindowContext context) {
    this.context = context;
  } ...
```

When an EPL query fires, the engine invokes the lookup and provides the actual lookup values. The lookup values are provided in the same exact order as the access path information that the engine provided when obtaining the lookup.

Each store row must be wrapped as an `EventBean` instance. The context object provides an `EventBeanFactory` implementation returned by `getEventFactory()` that can be used to wrap rows.

The sample implementation does not use the lookup values and simply returns a hardcoded sample event:

```
public Set<EventBean> lookup(Object[] lookupValues) {
  // Add code to interogate lookup values here.

  // Create sample event.
   // This example uses Map events; Other underlying events such as POJO are
 exactly the same code.
  Map<String, Object> eventData = new HashMap<String, Object>();
  eventData.put("key1", "sample1");
  eventData.put("key2", "sample2");
  eventData.put("value1", 100);
  eventData.put("value2", 1.5d);
  EventBean event = context.getEventFactory().wrap(eventData);
  return Collections.singleton(event);
}
```

The `lookupValues` object array represents all actual joined property values or expression results if you where-clause criteria are expressions. The code may use these keys to for efficient store access.

When a key value is a range, the key value is an instance of `VirtualDataWindowKeyRange`.

17.3. Single-Row Function

Single-row functions return a single value. They are not expected to aggregate rows but instead should be stateless functions. These functions can appear in any expressions and can be passed any number of parameters.

The following steps are required to develop and use a custom single-row function with Esper.

1. Implement a class providing one or more public static methods accepting the number and type of parameters as required.
2. Register the single-row function class and method name with the engine by supplying a function name, via the engine configuration file or the configuration API.

You may not override a built-in function with a single-row function provided by you. The single-row function you register must have a different name then any of the built-in functions.

An example single-row function can also be found in the examples under the runtime configuration example.

17.3.1. Implementing a Single-Row Function

Single-row function classes have no further requirement then provide a public static method.

The following sample single-row function simply computes a percentage value based on two number values.

This sample class provides a public static method by name `computePercent` to return a percentage value:

```
public class MyUtilityClass {
  public static double computePercent(double amount, double total) {
    return amount / total * 100;
  }
}
```

17.3.2. Configuring the Single-Row Function Name

The class name of the class, the method name and the function name of the new single-row 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 runtime and static configuration API:

```
<esper-configuration
  <plugin-singlerow-function name="percent"
   function-class="mycompany.MyUtilityClass" function-method="computePercent" /
>
</esper-configuration>
```

Note that the function name and method name need not be the same.

The new single-row function is now ready to use in a statement:

```
select percent(fulfilled,total) from MyEvent
```

When selecting from a single stream, you may also pass wildcard to the single-row function and the function receives the underlying event:

```
select percent(*) from MyEvent
```

If the single-row function returns an object that provides further functions, you may chain function calls.

The following demonstrates a chained single-row function. The example assumes that a single-row function by name `calculator` returns an object that provides the `add` function which accepts two parameters:

```
select calculator().add(5, amount) from MyEvent
```

17.3.3. Value Cache

When a single-row function receives parameters that are all constant values or expressions that themselves receive only constant values, Esper can pre-evaluate the result of the single-row function at time of statement creation. By default, Esper does not pre-evaluate the single-row function unless you configure the value cache as enabled.

The following configuration XML enables the value cache for the single-row function:

```
<esper-configuration
  <plugin-singlerow-function name="getDate"
    function-class="mycompany.DateUtil" function-method="parseDate"
    value-cache="enabled" />
</esper-configuration>
```

When the single-row function receives constants as parameters, the engine computes the result once and returns the cached result for each evaluation:

```
select getDate('2002-05-30T9:00:00.000') from MyEvent
```

17.3.4. Single-Row Functions in Filter Predicate Expressions

Your EPL may use plug-in single row functions among the predicate expressions as part of the filters in a stream or pattern.

For example, the EPL below uses the function `computeHash` as part of a predicate expression:

```
select * from MyEvent(computeHash(field) = 100)
```

When you have many EPL statements or many context partitions that refer to the same function, event type and parameters in a predicate expression, the engine may optimize evaluation: The function gets evaluated only once per event.

While the optimization is enabled by default for all plug-in single row functions, you can also disable the optimization for a specific single-row function. By disabling the optimization for a single-row function the engine may use less memory to identify reusable function footprints but may cause the engine to evaluate each function more frequently then necessary.

The following configuration XML disables the filter optimization for a single-row function (by default it is enabled):

```
<esper-configuration
  <plugin-singlerow-function name="computeHash"
    function-class="mycompany.HashUtil" function-method="computeHash"
    filter-optimizable="disabled" />
</esper-configuration>
```

17.3.5. Single-Row Functions Taking Events as Parameters

Esper allows parameters to a single-row function to be events. In this case, declare the method parameter type to either take `EventBean`, `Collection<EventBean>` or the underlying class as a parameter.

Sample method footprints are:

```
public static double doCompute(EventBean eventBean) {...}
public static boolean doCheck(MyEvent myEvent, String text) {...}
public static String doSearch(Collection<EventBean> events) {...}
```

To pass the event, specify the stream alias, or wildcard `(*)` or the tag name when used in a pattern.

The EPL below shows example uses:

```
select * from MyEvent(doCompute(me) = 100) as me
```

```
select * from MyEvent where doCompute(*) = 100
```

```
select * from pattern[a=MyEvent -> MyEvent(doCheck(a, 'sometext'))]
```

```
select * from MyEvent.win:time(1 min) having doCompute(last(*))]
```

```
select * from MyEvent.win:time(1 min) having doSearch(window(*))]
```

Declare the method parameter as `Collection<EventBean>` if the method expects an expression result that returns multiple events.

Declare the method parameter as `EventBean` if the method expects an expression result that returns a single event.

17.3.6. Receiving a Context Object

Esper can pass an object containing contextual information such as statement name, function name, engine URI and context partition id to your method. The container for this information is `EPLMethodInvocationContext` in package `com.espertech.esper.client.hook`. Please declare your method to take `EPLMethodInvocationContext` as the last parameter. The engine then passes the information along.

A sample method footprint and EPL are shown below:

```
public static double computeSomething(double number, EPLMethodInvocationContext
 context) {...}
```

```
select computeSomething(10) from MyEvent
```

17.3.7. Exception Handling

By default the engine logs any exceptions thrown by the single row function and returns a null value. To have exceptions be re-thrown instead, which makes exceptions visible to any registered exception handler, please configure as discussed herein.

Set the `rethrow-exceptions` flag in the XML configuration or the `rethrowExceptions` flag in the API when registering the single row function to have the engine re-throw any exceptions that the single row function may throw.

17.4. Derived-value and Data Window View

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 examples source folder in the OHLC (open-high-low-close) example. The class names are `OHLCBarPlugInViewFactory` and `OHLCBarPlugInView`.

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

Section 17.4.3, “View Contract” outlines the requirements for correct behavior of a your custom view within the engine.

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.

17.4.1. Implementing a View Factory

A view factory class is responsible for the following functions:

- Accept zero, one or more view parameters. View parameters are themselves expressions. The view factory must validate and evaluate these expressions.
- 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 OHLCBarPlugInViewFactory 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 checks the number of parameters and retains the parameters passed to the method:

```
public class OHLCBarPlugInViewFactory extends ViewFactorySupport {
    private ViewFactoryContext viewFactoryContext;
    private List<ExprNode> viewParameters;
    private ExprNode timestampExpression;
    private ExprNode valueExpression;

    public void setViewParameters(ViewFactoryContext viewFactoryContext,
            List<ExprNode> viewParameters) throws ViewParameterException {
        this.viewFactoryContext = viewFactoryContext;
        if (viewParameters.size() != 2) {
            throw new ViewParameterException(
                "View requires a two parameters: " +
              "the expression returning timestamps and the expression supplying
 OHLC data points");
```

```
        }
        this.viewParameters = viewParameters;
    }
  ...
```

After the engine supplied view parameters to the factory, the engine will ask the view to attach to its parent view and validate any parameter expressions 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,
  StatementContext statementContext,
  ViewFactory optionalParentFactory,
  List<ViewFactory> parentViewFactories) throws ViewParameterException {

    ExprNode[] validatedNodes = ViewFactorySupport.validate("OHLC view",
       parentEventType, statementContext, viewParameters, false);

    timestampExpression = validatedNodes[0];
    valueExpression = validatedNodes[1];

    if ((timestampExpression.getExprEvaluator().getType() != long.class) &&
        (timestampExpression.getExprEvaluator().getType() != Long.class)) {
        throw new ViewParameterException(
            "View requires long-typed timestamp values in parameter 1");
    }
    if ((valueExpression.getExprEvaluator().getType() != double.class) &&
        (valueExpression.getExprEvaluator().getType() != Double.class)) {
        throw new ViewParameterException(
            "View requires double-typed values for in parameter 2");
    }
}
```

Finally, the engine asks the view factory to create a view instance, and asks for the type of event generated by the view:

```
public            View            makeView(AgentInstanceViewFactoryChainContext
 agentInstanceViewFactoryContext) {
             return    new    OHLCBarPlugInView(agentInstanceViewFactoryContext,
 timestampExpression, valueExpression);
}

public EventType getEventType() {
                                                                        return
 OHLCBarPlugInView.getEventType(viewFactoryContext.getEventAdapterService());
}
```

17.4.2. 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:groupwin` parent view

View classes simply subclass `com.espertech.esper.view.ViewSupport`:

```
public class MyTrendSpotterView extends ViewSupport { ...
```

Your view's `update` method will be processing incoming (insert stream) and outgoing (remove stream) events posted by the parent view (if any), 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 (EventBean object references) which the view generates as reference-equal remove stream events (EventBean object references) at a later time.

The view implementation must call the `updateChildren` method to post outgoing insert and remove stream events. Similar to the `update` method, the `updateChildren` method takes insert and remove stream events as parameters.

A sample `update` method implementation is provided in the OHLC example.

17.4.3. View Contract

The `update` method must adhere to the following conventions, to prevent memory leaks and to enable correct behavior within the engine:

- A view implementation that posts events to the insert stream must post unique `EventBean` object references as insert stream events, and cannot post the same `EventBean` object reference multiple times. The underlying event to the `EventBean` object reference can be the same object reference, however the `EventBean` object reference posted by the view into the insert stream must be a new instance for each insert stream event.
- If the custom view posts a continuous insert stream, then the views must also post a continuous remove stream (second parameter to the `updateChildren` method). If the view does not post remove stream events, it assumes unbound keep-all semantics.
- `EventBean` events posted as remove stream events must be the same object reference as the `EventBean` events posted as insert stream by the view. Thus remove stream events posted

by the view (the `EventBean` instances, does not affect the underlying representation) must be reference-equal to insert stream events posted by the view as part of an earlier invocation of the update method, or the same invocation of the update method.

- `EventBean` events represent a unique observation. The values of the observation can be the same, thus the underlying representation of an `EventBean` event can be reused, however event property values must be kept immutable and not be subject to change.
- Array elements of the insert and remove stream events must not carry null values. Array size must match the number of `EventBean` instances posted. It is recommended to use a `null` value for no insert or remove stream events rather then an empty zero-size array.

Your view implementation can register a callback indicating when a statement using the view, or a context partition using the view, is stopped or terminated. Your view code must implement, or provide an implementation, of the `com.espertech.esper.util.StopCallback` interface. Register the stop callback in order for the engine to invoke the callback:

```
agentInstanceContext.getTerminationCallbacks().add(this);
```

Please refer to the sample views for a code sample on how to implement `iterator` and `cloneView` methods.

In terms of multiple threads accessing view state, there is no need for your custom view factory or view implementation to perform any synchronization to protect internal state. The iterator of the custom view implementation does also not need to be thread-safe. The engine ensures the custom view executes in the context of a single thread at a time. If your view uses shared external state, such external state must be still considered for synchronization when using multiple threads.

17.4.4. 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="ohlc"
    factory-class="com.espertech.esper.example.ohlc.OHLCBarPlugInViewFactory" /
>
</esper-configuration>
```

The new view is now ready to use in a statement:

```
select * from StockTick.custom:ohlc(timestamp, price)
```

Note that the view must implement additional interfaces if it acts as a data window view, or works in a grouping context, as discussed in detail below.

17.4.5. Requirement for Data Window Views

Your custom view may represent an expiry policy and may retain events and thus act as a data window view. In order to allow the engine to validate that your view can be used with named windows, which allow only data window views, this section documents any additional requirement that your classes must fulfill.

Your view factory class must implement the `com.espertech.esper.view.DataWindowViewFactory` interface. This marker interface (no methods required) indicates that your view factory provides only data window views.

Your view class must implement the `com.espertech.esper.view.DataWindowView` interface. This marker interface indicates that your view is a data window view and therefore eligible to be used in any construct that requires a data window view.

17.4.6. Requirement for Derived-Value Views

Your custom view may compute derived information from the arriving stream, instead of retaining events, and thus act as a derived-value view.

Your view class should implement the `com.espertech.esper.view.DerivedValueView` interface. This marker interface indicates that your view is a derived-value view, affecting correct behavior of the view when used in joins.

17.4.7. Requirement for Grouped Views

Grouped views are views that operate under the `std:groupwin` view. When operating under one or more `std:groupwin` views, the engine instantiates a single view instance when the statement starts, and a new view instance per group criteria dynamically as new group criteria become known.

The next statement shows EPL for using a view instance per grouping criteria:

```
select * from StockTick.std:groupwin(symbol).custom:trendspotter(price)
```

Your view must implement the `com.espertech.esper.view.CloneableView` interface to indicate your view may create new views. This code snippet shows a sample implementation of the `cloneView` method required by the interface:

```
public View cloneView() {
  return new MyPlugInView(...); // pass any parameters along
}
```

17.5. Aggregation Function

Aggregation functions are stateful functions that aggregate events, event property values or expression results. Examples for built-in aggregation functions are `count(*)`, `sum(price * volume)`, `window(*)` or `maxby(volume)`.

Esper allows two different ways for your application to provide aggregation functions. We use the name *aggregation single-function* and *aggregation multi-function* for the two independent extension APIs for aggregation functions.

The aggregation single-function API is simple to use however it imposes certain restrictions on how expressions that contain aggregation functions share state and are evaluated.

The aggregation multi-function API is more powerful and provides control over how expressions that contain aggregation functions share state and are evaluated.

The next table compares the two aggregation function extension API's:

Table 17.1. Aggregation Function Extension API's

	Single-Function	Multi-Function	
Return Value	Can only return a single value or object. Cannot return an `EventBean` event, collection of `EventBean` events or collection or array of values for use with enumeration methods, for example.	Can return an `EventBean` event, a collection of `EventBean` events or a collection or array of objects for use with enumeration methods or to access event properties.	
Complexity of API	Simple (consists of 2 interfaces).	More complex (consists of 6 interfaces).	
State Sharing	State and parameter evaluation shared if multiple aggregation functions of the same name in the same statement (and context partition) take the exact same parameter expressions.	State and parameter evaluation sharable when multiple aggregation functions of a related name (related thru configuration) for the same statement (and context partition) exist, according to a sharing-key provided by your API implementation.	
Function Name	Each aggregation function expression receives its own factory object.	Multiple related aggregation function expressions share a single factory object.	

	Single-Function	Multi-Function	
Distinct Keyword	Handled by the engine transparently.	Indicated to the API implementation only.	

The following sections discuss developing an aggregation single-function first, followed by the subject of developing an aggregation multi-function.

Note

The aggregation multi-function API is a powerful and lower-level API to extend the engine. Any classes that are not part of the `client`, `plugin` or `agg.access` package are subject to change between minor and major releases of the engine.

17.5.1. Aggregation Single-Function Development

This section describes the *aggregation single-function* extension API for providing aggregation functions.

The following steps are required to develop and use a custom aggregation single-function with Esper.

1. Implement an aggregation function factory by implementing the interface `com.espertech.esper.client.hook.AggregationFunctionFactory`.
2. Implement an aggregation function by implementing the interface `com.espertech.esper.epl.agg.aggregator.AggregationMethod`.
3. Register the aggregation single-function factory class with the engine by supplying a function name, via the engine configuration file or the runtime and static configuration API.

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 single-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 single-function.

Custom aggregation functions can also be passed multiple parameters, as further described in *Section 17.5.1.4, “Aggregation Single-Function: Accepting Multiple Parameters”*. In the example below the aggregation function accepts a single parameter.

The code for the example aggregation function as shown in this chapter can be found in the runtime configuration example in the package `com.espertech.esper.example.runtimeconfig` by the name `MyConcatAggregationFunction`. The sample function simply concatenates string-type values.

17.5.1.1. Implementing an Aggregation Single-Function Factory

An aggregation function factory class is responsible for the following functions:

- Implement a `setFunctionName` method that receives the function name assigned to this instance.
- 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 aggregation function instances. For example, the built-in `count` aggregation function returns `Long.class` as it generates `long` -typed values.
- Implement a `newAggregator` method that instantiates and returns an aggregation function instance.

Aggregation function classes implement the interface `com.espertech.esper.client.hook.AggregationFunctionFactory`:

```
public class MyConcatAggregationFunctionFactory implements
 AggregationFunctionFactory { ...
```

The engine generally constructs one instance of the aggregation function factory class for each time the function is listed in an EPL statement, however the engine may decide to reduce the number of aggregation class instances if it finds equivalent aggregations.

The aggregation function factory instance receives the aggregation function name via set `setFunctionName` method.

The sample concatenation function factory provides an empty `setFunctionName` method:

```
public void setFunctionName(String functionName) {
  // no action taken
}
```

An aggregation function factory must provide an implementation of the `validate` method that is passed a `AggregationValidationContext` validation context object. Within the validation context you find the result type of each of the parameters expressions to the aggregation function as well as information about constant values and data window use. Please see the JavaDoc API documentation for a comprehensive list of validation context information.

Since the example concatenation function requires string types, it implements a type check:

```
public void validate(AggregationValidationContext validationContext) {
  if ((validationContext.getParameterTypes().length != 1) ||
    (validationContext.getParameterTypes()[0] != String.class)) {
     throw new IllegalArgumentException("Concat aggregation requires a single
 parameter of type String");
  }
}
```

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 factory implementation must provide a `newAggregator` method that returns instances of `AggregationMethod`. The engine invokes this method for each new aggregation state to be allocated.

```
public AggregationMethod newAggregator() {
  return new MyConcatAggregationFunction();
}
```

17.5.1.2. Implementing an Aggregation Single-Function

An aggregation function class is responsible for the following functions:

- 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.
- Implement a `clear` method that resets the current value.

Aggregation function classes implement the interface `AggregationMethod`:

```
public class MyConcatAggregationFunction implements AggregationMethod { ...
```

The class that provides the aggregation and implements `AggregationMethod` does not have to be threadsafe.

The constructor initializes the aggregation function:

```
public class MyConcatAggregationFunction implements AggregationMethod {
  private final static char DELIMITER = ' ';
  private StringBuilder builder;
  private String delimiter;

  public MyConcatAggregationFunction() {
    builder = new StringBuilder();
    delimiter = "";
  }
  ...
```

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;
}
```

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);
  }
}
```

Finally, the engine obtains the current aggregation value by means of the `getValue` method:

```
public Object getValue() {
  return builder.toString();
}
```

For on-demand queries the aggregation function must support resetting its value to empty or start values. Implement the `clear` function to reset the value as shown below:

```
public void clear() {
  builder = new StringBuilder();
  delimiter = "";
}
```

17.5.1.3. Configuring the Aggregation Single-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"
                                                                     factory-
class="com.espertech.esper.example.runtimeconfig.MyConcatAggregationFunctionFactory"
>
</esper-configuration>
```

The new aggregation function is now ready to use in a statement:

```
select concat(symbol) from StockTick.win:length(3)
```

17.5.1.4. Aggregation Single-Function: Accepting Multiple Parameters

Your plug-in aggregation function may accept multiple parameters, simply by casting the Object parameter of the `enter` and `leave` method to `Object[]`.

For instance, assume an aggregation function `rangeCount` that counts all values that fall into a range of values. The EPL that calls this function and provides a lower and upper bounds of 1 and 10 is:

```
select rangeCount(1, 10, myValue) from MyEvent
```

The `enter` method of the plug-in aggregation function may look as follows:

```
public void enter(Object value)  {
  Object[] params = (Object[]) value;
  int lower = (Integer) params[0];
  int upper = (Integer) params[1];
  int val = (Integer) params[2];
  if ((val >= lower) && (val <= upper)) {
    count++;
  }
}
```

Your plug-in aggregation function may want to validate parameter types or may want to know which parameters are constant-value expressions. Constant-value expressions are evaluated only once by the engine and could therefore be cached by your aggregation function for performance reasons. The engine provides constant-value information as part of the `AggregationValidationContext` passed to the `validate` method.

17.5.1.5. Aggregation Single-Function: Dot-Operator Use

When the custom aggregation function returns an object as a return value, the EPL can use parenthesis and the dot-operator to invoke methods on the return value.

The following example assumes that the `myAggregation` custom aggregation function returns an object that has `getValueOne` and `getValueTwo` methods:

```
select                (myAggregation(myValue)).getValueOne(),
 (myAggregation(myValue)).getValueTwo() from MyEvent
```

Since the above EPL aggregates the same value, the engine internally uses a single aggregation to represent the current value of `myAggregation` (and not two instances of the aggregation, even though `myAggregation` is listed twice).

17.5.2. Aggregation Multi-Function Development

This section introduces the aggregation multi-function API. Please refer to the JavaDoc for more complete class and method-level documentation.

Among the Esper examples we provide an example use of the aggregation multi-function API in the example by name Cycle-Detect. Cycle-Detect takes incoming transaction events that have from-account and to-account fields. The example detects a cycle in the transactions between accounts in order to detect a possible transaction fraud. Please note that the graph and cycle detection logic of the example is not part of Esper: The example utilizes the `jgrapht` library.

In the Cycle-Detect example, the vertices of a graph are the account numbers. For example the account numbers `Acct-1`, `Acct-2` and `Acct-3`. In the graph the edges are transaction events that identify a from-account and a to-account. An example edge is `{from:Acct-1, to:Acct-2}`. An example cycle is therefore in the three transactions `{from:Acct-1, to:Acct-2}`, `{from:Acct-2, to:Acct-3}` and `{from:Acct-3, to:Acct-1}`.

The code for the example aggregation multi-function as shown in this chapter can be found in the Cycle-Detect example in the package `com.espertech.esper.example.cycledetect`. The example provides two aggregation functions named `cycledetected` and `cycleoutput`:

1. The `cycledetected` function returns a boolean value whether a graph cycle is found or not.
2. The `cycleoutput` function outputs the vertices (account numbers) that are part of the graph cycle.

In the Cycle-Detect example, the following statement utilizes the two functions `cycledetected` and `cycleoutput` that share the same graph state to detect a cycle among the last 1000 events:

```
@Name('CycleDetector') select cycleoutput() as cyclevertices
from TransactionEvent.win:length(1000)
having cycledetected(fromAcct, toAcct)
```

If instead the goal is to run graph cycle detection every 1 second (and not upon arrival of a new event), this sample EPL statement uses a pattern to trigger cycle detection:

```
@Name('CycleDetector')
select (select cycleoutput(fromAcct, toAcct) from
 TransactionEvent.win:length(1000)) as cyclevertices
from pattern [every timer:interval(1)]
```

The following steps are required to develop and use a custom aggregation multi-function with Esper.

1. Implement an aggregation multi-function factory by implementing the interface `com.espertech.esper.plugin.PlugInAggregationMultiFunctionFactory`.
2. Implement one or more handlers for aggregation functions by implementing the interface `com.espertech.esper.plugin.PlugInAggregationMultiFunctionHandler`.
3. Implement an aggregation state key by implementing the interface `com.espertech.esper.epl.agg.access.AggregationStateKey`.
4. Implement an aggregation state factory by implementing the interface `com.espertech.esper.plugin.PlugInAggregationMultiFunctionStateFactory`.

5. Implement an aggregation state holder by implementing the interface `com.espertech.esper.epl.agg.access.AggregationState`.
6. Implement a state accessor by implementing the interface `com.espertech.esper.epl.agg.access.AggregationAccessor`.
7. Register the aggregation multi-function factory class with the engine by supplying one or more function names, via the engine configuration file or the runtime and static configuration API.

17.5.2.1. Implementing an Aggregation Multi-Function Factory

An aggregation multi-function factory class is responsible for the following functions:

- Implement the `addAggregationFunction` method that receives an invocation for each aggregation function declared in the statement that matches any of the function names provided at configuration time.
- Implement the `validateGetHandler` method that receives an invocation for each aggregation function to be validated in the statement that matches any of the function names provided at configuration time.

Aggregation multi-function factory classes implement the interface `com.espertech.esper.plugin.PlugInAggregationMultiFunctionFactory`:

```
public         class         CycleDetectorAggregationFactory         implements
 PlugInAggregationMultiFunctionFactory { ...
```

The engine constructs a single instance of the aggregation multi-function factory class that is shared for all aggregation function expressions in a statement that have one of the function names provided in the configuration object.

The engine invokes the `addAggregationFunction` method at the time it parses an EPL statement or compiles a statement object model (SODA API). The method receives a declaration-time context object that provides the function name as well as additional information.

The sample Cycle-Detect factory class provides an empty `addAggregationFunction` method:

```
public                                                                      void

 declarationContext) {
  // no action taken
}
```

The engine invokes the `validateGetHandler` method at the time of expression validation. It passes a `PlugInAggregationMultiFunctionValidationContext` validation context object that contains actual parameters expressions. Please see the JavaDoc API documentation for a comprehensive list of validation context information.

The `validateGetHandler` method must return a handler object the implements the `PlugInAggregationMultiFunctionHandler` interface. Return a handler object for each aggregation function expression according to the aggregation function name and its parameters that are provided in the validation context.

The example `cycledetect` function takes two parameters that provide the cycle edge (from-account and to-account):

```
public                                    PlugInAggregationMultiFunctionHandler

 validationContext) {
  if (validationContext.getParameterExpressions().length == 2) {
              fromExpression   =   validationContext.getParameterExpressions()
[0].getExprEvaluator();
               toExpression   =   validationContext.getParameterExpressions()
[1].getExprEvaluator();
  }
  return new CycleDetectorAggregationHandler(this, validationContext);
}
```

17.5.2.2. Implementing an Aggregation Multi-Function Handler

An aggregation multi-function handler class must implement the `PlugInAggregationMultiFunctionHandler` interface and is responsible for the following functions:

- Implement the `getAccessor` method that returns a reader object for the aggregation state.
- Implement the `getReturnType` method that returns information about the type of return values provided by the accessor reader object.
- Implement the `getAggregationStateUniqueKey` method that provides a key object used by the engine to determine which aggregation functions share state.
- Implement the `getStateFactory` method that returns a state factory object that the engine invokes, when required, to instantiate aggregation state holders.

Typically your API returns a handler instance for each aggregation function in an EPL statement expression.

In the Cycle-Detect example, the class `CycleDetectorAggregationHandler` is the handler for all aggregation functions.

```
public         class         CycleDetectorAggregationHandler         implements
 PlugInAggregationMultiFunctionHandler { ...
```

The `getAccessor` methods return a reader object according to whether the aggregation function name is `cycledetected` or `cycleoutput`:

```
public AggregationAccessor getAccessor() {
                                                                             if


 {
    return new CycleDetectorAggregationAccessorOutput();
  }
  return new CycleDetectorAggregationAccessorDetect();
}
```

The `getReturnType` method provided by the handler instructs the engine about the return type of each aggregation accessor. The class `com.espertech.esper.client.util.ExpressionReturnType` holds return type information.

In the Cycle-Detect example the `cycledetected` function returns a single boolean value. The `cycleoutput` returns a collection of vertices:

```
public ExpressionReturnType getReturnType() {
                                                                             if


 {
                                                                         return
 ExpressionReturnType.collectionOfSingleValue(factory.getFromExpression().getType());
  }
  return ExpressionReturnType.singleValue(Boolean.class) ;
}
```

The engine invokes the `getAggregationStateUniqueKey` method to determine whether multiple aggregation function expressions in the same statement can share the same aggregation state or should receive different aggregation state instances.

The `getAggregationStateUniqueKey` method must return an instance of `AggregationStateKey`. The engine uses equals-semantics (the `hashCode` and `equals` methods) to determine whether multiple aggregation function share the state object. If the key object returned for each aggregation function by the handler is an equal key object then the engine shares aggregation state between such aggregation functions for the same statement and context partition.

In the Cycle-Detect example the state is shared, which it achieves by simply returning the same key instance:

```
private static final AggregationStateKey CYCLE_KEY = new AggregationStateKey()
 {};

public AggregationStateKey getAggregationStateUniqueKey() {
  return CYCLE_KEY;   // Share the same aggregation state instance
}
```

The engine invokes the `getStateFactory` method to obtain an instance of `PlugInAggregationMultiFunctionStateFactory`. The state factory is responsible to instantiating separate aggregation state instances. If you statement does not have a `group by` clause, the engine obtains a single aggregation state from the state factory. If your statement has a `group by` clause, the engine obtains an aggregation state instance for each group when it encounters a new group.

In the Cycle-Detect example the method passes the expression evaluators providing the from-account and to-account expressions to the state factory:

```
public PlugInAggregationMultiFunctionStateFactory getStateFactory() {
   return new CycleDetectorAggregationStateFactory(factory.getFromExpression(),
 factory.getToExpression());
}
```

17.5.2.3. Implementing an Aggregation Multi-Function State Factory

An aggregation multi-function state factory class must implement the `PlugInAggregationMultiFunctionStateFactory` interface and is responsible for the following functions:

- Implement the `makeAggregationState` method that returns an aggregation state holder.

The engine invokes the `makeAggregationState` method to obtain a new aggregation state instance before applying aggregation state. If using `group by` in your statement, the engine invokes the `makeAggregationState` method to obtain a state holder for each group.

In the Cycle-Detect example, the class `CycleDetectorAggregationStateFactory` is the state factory for all aggregation functions:

```
public       class       CycleDetectorAggregationStateFactory       implements
 PlugInAggregationMultiFunctionStateFactory {
  private final ExprEvaluator fromEvaluator;
  private final ExprEvaluator toEvaluator;

    public  CycleDetectorAggregationStateFactory(ExprEvaluator  fromEvaluator,
 ExprEvaluator toEvaluator) {
    this.fromEvaluator = fromEvaluator;
    this.toEvaluator = toEvaluator;
  }
```

```
                                    public                  AggregationState
 makeAggregationState(PlugInAggregationMultiFunctionStateContext  stateContext)
 {
    return new CycleDetectorAggregationState(this);
  }

  public ExprEvaluator getFromEvaluator() {
    return fromEvaluator;
  }

  public ExprEvaluator getToEvaluator() {
    return toEvaluator;
  }
}
```

17.5.2.4. Implementing an Aggregation Multi-Function State

An aggregation multi-function state class must implement the `AggregationState` interface and is responsible for the following functions:

- Implement the `applyEnter` method that enters events, event properties or computed values.
- Optionally implement the `applyLeave` method that can remove events or computed values.
- Implement the `clear` method to clear state.

In the Cycle-Detect example, the class `CycleDetectorAggregationState` is the state for all aggregation functions. Please review the example for more information.

17.5.2.5. Implementing an Aggregation Multi-Function Accessor

An aggregation multi-function accessor class must implement the `AggregationAccessor` interface and is responsible for the following functions:

- Implement the `Object getValue(AggregationState state)` method that returns a result object for the aggregation state.
- Implement the `Collection<EventBean> getEnumerableEvents(AggregationState state)` method that returns a collection of events for enumeration, if applicable (or null).
- Implement the `EventBean getEnumerableEvent(AggregationState state)` method that returns an event, if applicable (or null).

In the Cycle-Detect example, the class `CycleDetectorAggregationAccessorDetect` returns state for the `cycledetected` aggregation function and the `CycleDetectorAggregationAccessorOutput` returns the state for the `cycleoutput` aggregation function.

17.5.2.6. Configuring the Aggregation Multi-Function Name

An aggregation multi-function configuration can receive one or multiple function names. You must also set a factory class name.

The sample XML snippet below configures an aggregation multi-function that is associated with the function names `func1` and `func2`.

```
<esper-configuration
  <plugin-aggregation-multifunction
       function-names="cycledetected,cycleoutput"    // a comma-separated list
 of function name
                                                                    factory-
class="com.espertech.esper.example.cycledetect.CycleDetectorAggregationFactory"/
>
</esper-configuration>
```

The next example uses the runtime configuration API to register the same:

```
String[] functionNames = new String[] {"cycledetected", "cycleoutput"};
ConfigurationPlugInAggregationMultiFunction        config        =        new

 CycleDetectorAggregationFactory.class.getName());
engine.getEPAdministrator().getConfiguration().addPlugInAggregationMultiFunction(config);
```

17.5.2.7. Aggregation Multi-Function Thread Safety

The engine shares an `AggregationAccessor` instance between threads. The accessor should be designed stateless and should not use any locking of any kind in the `AggregationAccessor` implementation unless your implementation uses other state. Since the engine passes an aggregation state instance to the accessor it is thread-safe as long as it relies only on the aggregation state passed to it.

The engine does not share an `AggregationState` instance between threads. There is no need to use locking of any kind in the `AggregationState` implementation unless your implementation uses other state.

17.6. 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.

17.6.1. Implementing a Guard Factory

A guard factory class is responsible for the following functions:

- Implement a `setGuardParameters` method that takes guard parameters, which are themselves expressions.
- 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<ExprNode> guardParameters,
   MatchedEventConvertor convertor) throws GuardParameterException {
     String message = "Count-to guard takes a single integer-value expression
 as parameter";
    if (guardParameters.size() != 1) {
        throw new GuardParameterException(message);
    }

    if (guardParameters.get(0).getExprEvaluator().getType() != Integer.class) {
        throw new GuardParameterException(message);
    }

    this.numCountToExpr = guardParameters.get(0);
    this.convertor = convertor;
}
```

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(PatternAgentInstanceContext context,
      MatchedEventMap beginState,
      Quitable quitable,
      Object stateNodeId,
      Object guardState) {

    Object parameter = PatternExpressionUtil.evaluate("Count-to guard",
        beginState, numCountToExpr, convertor);
    if (parameter == null) {
          throw new EPException("Count-to guard parameter evaluated to a null
 value");
    }

    Integer numCountTo = (Integer) parameter;
    return new MyCountToPatternGuard(numCountTo, quitable);
}
```

17.6.2. 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.

17.6.3. 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.

17.7. 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.

17.7.1. Implementing an Observer Factory

An observer factory class is responsible for the following functions:

- Implement a `setObserverParameters` method that takes observer parameters, which are themselves expressions.
- 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<ExprNode> expressionParameters,
   MatchedEventConvertor convertor) throws ObserverParameterException {
      String message = "File exists observer takes a single string filename
 parameter";
    if (expressionParameters.size() != 1) {
     throw new ObserverParameterException(message);
    }
          if  (!(expressionParameters.get(0).getExprEvaluator().getType()  ==
 String.class)) {
     throw new ObserverParameterException(message);
    }

    this.filenameExpression = expressionParameters.get(0);
    this.convertor = convertor;
}
```

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(PatternAgentInstanceContext context,
   MatchedEventMap beginState,
   ObserverEventEvaluator observerEventEvaluator,
   Object stateNodeId,
   Object observerState) {
     Object filename = PatternExpressionUtil.evaluate("File-exists observer ",
 beginState, filenameExpression, convertor);
    if (filename == null) {
     throw new EPException("Filename evaluated to null");
    }

        return  new  MyFileExistsObserver(beginState,  observerEventEvaluator,
 filename.toString());
}
```

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.

17.7.2. 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.

17.7.3. 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")]
```

17.8. Event Type And Event Object

Creating a plug-in event representation can be useful under any of these conditions:

- Your application has existing Java classes that carry event metadata and event property values and your application does not want to (or cannot) extract or transform such event metadata and event data into one of the built-in event representations (POJO Java objects, Map or XML DOM).
- Your application wants to provide a faster or short-cut access path to event data, for example to access XML event data through a Streaming API for XML (StAX).
- Your application must perform a network lookup or other dynamic resolution of event type and events.

Note that the classes to plug-in custom event representations are held stable between minor releases, but can be subject to change between major releases.

Currently, EsperIO provides the following additional event representations:

- Apache Axiom provides access to XML event data on top of the fast Streaming API for XML (StAX).

The source code is available for these and they are therefore excellent examples for how to implement a plug-in event representation. Please see the EsperIO documentation for usage details.

17.8.1. How It Works

Your application provides a plug-in event representation as an implementation of the `com.espertech.esper.plugin.PlugInEventRepresentation` interface. It registers the implementation class in the `Configuration` and at the same time provides a unique URI. This URI is called the root event representation URI. An example value for a root URI is `type://xml/apacheaxiom/OMNode`.

One can register multiple plug-in event representations. Each representation has a root URI. The root URI serves to divide the overall space of different event representations and plays a role in resolving event types and event objects.

There are two situations in an Esper engine instance asks an event representation for an event type:

1. When an application registers a new event type using the method `addPlugInEventType` on `ConfigurationOperations`, either at runtime or at configuration time.

2. When an EPL statement is created with a new event type name (a name not seen before) and the URIs for resolving such names are set beforehand via `setPlugInEventTypeNameResolutionURIs` on `ConfigurationOperations`.

The implementation of the `PlugInEventRepresentation` interface must provide implementations for two key interfaces: `com.espertech.esper.client.EventType` and `EventBean`. It must also implement several other related interfaces as described below.

The `EventType` methods provide event metadata including property names and property types. They also provide instances of `EventPropertyGetter` for retrieving event property values. Each instance of `EventType` represents a distinct type of event.

The `EventBean` implementation is the event itself and encapsulates the underlying event object.

17.8.2. Steps

Follow the steps outlined below to process event objects for your event types:

1. Implement the `EventType`, `EventPropertyGetter` and `EventBean` interfaces.
2. Implement the `PlugInEventRepresentation` interface, the `PlugInEventTypeHandler` and `PlugInEventBeanFactory` interfaces, then add the `PlugInEventRepresentation` class name to configuration.
3. Register plug-in event types, and/or set the event type name resolution URIs, via configuration.
4. Obtain an `EventSender` from `EPRuntime` via the `getEventSender(URI[])` method and use that to send in your event objects.

Please consult the JavaDoc for further information on each of the interfaces and their respective methods. The Apache Axiom event representation is an example implementation that can be found in the EsperIO packages.

17.8.3. URI-based Resolution

Assume you have registered event representations using the following URIs:

1. type://myFormat/myProject/myName
2. type://myFormat/myProject
3. type://myFormat/myOtherProject

When providing an array of child URIs for resolution, the engine compares each child URI to each of the event representation root URIs, in the order provided. Any event representation root URIs that spans the child URI space becomes a candidate event representation. If multiple root URIs match, the order is defined by the more specific root URI first, to the least specific root URI last.

During event type resolution and event sender resolution you provide a child URI. Assuming the child URI provided is `type://myFormat/myProject/myName/myEvent?param1=abc&param2=true`. In this example both root URIs #1 (the more specific) and #1 (the less specific) match, while root URI #3 is not a match. Thus at the time of type resolution the engine invokes the `acceptType` method on event presentation for URI #1 first (the more specific), before asking #2 (the less specific) to resolve the type.

The `EventSender` returned by the `getEventSender(URI[])` method follows the same scheme. The event sender instance asks each matching event representation for each URI to resolve the event object in the order of most specific to least specific root URI, and the first event representation to return an instance of `EventBean` ends the resolution process for event objects.

The `type://` part of the URI is an optional convention for the scheme part of an URI that your application may follow. URIs can also be simple names and can include parameters, as the Java software JavaDoc documents for class `java.net.URI`.

17.8.4. Example

This section implements a minimal sample plug-in event representation. For the sake of keeping the example easy to understand, the event representation is rather straightforward: an event is a `java.util.Properties` object that consists of key-values pairs of type string.

The code shown next does not document method footprints. Please consult the JavaDoc API documentation for method details.

17.8.4.1. Sample Event Type

First, the sample shows how to implement the `EventType` interface. The event type provides information about property names and types, as well as supertypes of the event type.

Our `EventType` takes a set of valid property names:

```
public class MyPlugInPropertiesEventType implements EventType {
  private final Set<String> properties;

  public MyPlugInPropertiesEventType(Set<String> properties) {
    this.properties = properties;
  }

  public Class getPropertyType(String property) {
    if (!isProperty(property)) {
      return null;
    }
    return String.class;
  }

  public Class getUnderlyingType() {
    return Properties.class;
  }

  //... further methods below
}
```

An `EventType` is responsible for providing implementations of `EventPropertyGetter` to query actual events. The getter simply queries the `Properties` object underlying each event:

```
  public EventPropertyGetter getGetter(String property) {
    final String propertyName = property;

    return new EventPropertyGetter() {
      public Object get(EventBean eventBean) throws PropertyAccessException {
          MyPlugInPropertiesEventBean propBean = (MyPlugInPropertiesEventBean)
eventBean;
        return propBean.getProperties().getProperty(propertyName);
      }

      public boolean isExistsProperty(EventBean eventBean) {
          MyPlugInPropertiesEventBean propBean = (MyPlugInPropertiesEventBean)
eventBean;
        return propBean.getProperties().getProperty(propertyName) != null;
      }

      public Object getFragment(EventBean eventBean) {
     return null; // The property is not a fragment
      }
    };
  }
```

Our sample `EventType` does not have supertypes. Supertypes represent an extends-relationship between event types, and subtypes are expected to exhibit the same event property names and types as each of their supertypes combined:

```
  public EventType[] getSuperTypes() {
    return null; // no supertype for this example
  }

  public Iterator<EventType> getDeepSuperTypes() {
    return null;
  }

  public String getName() {
    return name;
  }

  public EventPropertyDescriptor[] getPropertyDescriptors() {
    Collection<EventPropertyDescriptor> descriptorColl = descriptors.values();
                                    return         descriptorColl.toArray(new
EventPropertyDescriptor[descriptors.size()]);
  }

  public EventPropertyDescriptor getPropertyDescriptor(String propertyName) {
    return descriptors.get(propertyName);
  }

  public FragmentEventType getFragmentType(String property) {
    return null;  // sample does not provide any fragments
  }
```

The example event type as above does not provide fragments, which are properties of the event that can themselves be represented as an event, to keep the example simple.

17.8.4.2. Sample Event Bean

Each `EventBean` instance represents an event. The interface is straightforward to implement. In this example an event is backed by a `Properties` object:

```
public class MyPlugInPropertiesEventBean implements EventBean {
  private final MyPlugInPropertiesEventType eventType;
  private final Properties properties;

  public MyPlugInPropertiesEventBean(MyPlugInPropertiesEventType eventType,
        Properties properties) {
    this.eventType = eventType;
    this.properties = properties;
  }

  public EventType getEventType() {
    return eventType;
  }

  public Object get(String property) throws PropertyAccessException {
    EventPropertyGetter getter = eventType.getGetter(property);
    return getter.get(this);
  }

  public Object getFragment(String property) {
    EventPropertyGetter getter = eventType.getGetter(property);
    if (getter != null) {
      return getter.getFragment(this);
    }
    return null;
  }

  public Object getUnderlying() {
    return properties;
  }

  protected Properties getProperties() {
    return properties;
  }
}
```

17.8.4.3. Sample Event Representation

A `PlugInEventRepresentation` serves to create `EventType` and `EventBean` instances through its related interfaces.

The sample event representation creates `MyPlugInPropertiesEventType` and `MyPlugInPropertiesEventBean` instances. The `PlugInEventTypeHandler` returns the `EventType` instance and an `EventSender` instance.

Our sample event representation accepts all requests for event types by returning boolean true on the `acceptType` method. When asked for the `PlugInEventTypeHandler`, it constructs a new `EventType`. The list of property names for the new type is passed as an initialization value provided through the configuration API or XML, as a comma-separated list of property names:

```
public class MyPlugInEventRepresentation implements PlugInEventRepresentation {

  private List<MyPlugInPropertiesEventType> types;

  public void init(PlugInEventRepresentationContext context) {
    types = new ArrayList<MyPlugInPropertiesEventType>();
  }

  public boolean acceptsType(PlugInEventTypeHandlerContext context) {
    return true;
  }

  public PlugInEventTypeHandler getTypeHandler(PlugInEventTypeHandlerContext
 eventTypeContext) {
    String proplist = (String) eventTypeContext.getTypeInitializer();
    String[] propertyList = proplist.split(",");

    Set<String> typeProps = new HashSet<String>(Arrays.asList(propertyList));

    MyPlugInPropertiesEventType eventType = new
 MyPlugInPropertiesEventType(typeProps);
    types.add(eventType);

    return new MyPlugInPropertiesEventTypeHandler(eventType);
  }
  // ... more methods below
}
```

The `PlugInEventTypeHandler` simply returns the `EventType` as well as an implementation of `EventSender` for processing same-type events:

```
public class MyPlugInPropertiesEventTypeHandler implements
 PlugInEventTypeHandler {
  private final MyPlugInPropertiesEventType eventType;

       public MyPlugInPropertiesEventTypeHandler(MyPlugInPropertiesEventType
 eventType) {
    this.eventType = eventType;
  }

  public EventSender getSender(EPRuntimeEventSender runtimeEventSender) {
    return new MyPlugInPropertiesEventSender(eventType, runtimeEventSender);
  }

  public EventType getType() {
    return eventType;
  }
}
```

The `EventSender` returned by `PlugInEventTypeHandler` is expected process events of the same type or any subtype:

```
public class MyPlugInPropertiesEventSender implements EventSender {
  private final MyPlugInPropertiesEventType type;
  private final EPRuntimeEventSender runtimeSender;

  public MyPlugInPropertiesEventSender(MyPlugInPropertiesEventType type,
        EPRuntimeEventSender runtimeSender) {
    this.type = type;
    this.runtimeSender = runtimeSender;
  }

  public void sendEvent(Object event) {
    if (!(event instanceof Properties)) {
       throw new EPException("Sender expects a properties event");
    }
     EventBean eventBean = new MyPlugInPropertiesEventBean(type, (Properties)
 event);
    runtimeSender.processWrappedEvent(eventBean);
  }
}
```

17.8.4.4. Sample Event Bean Factory

The plug-in event representation may optionally provide an implementation of `PlugInEventBeanFactory`. A `PlugInEventBeanFactory` may inspect event objects and assign an event type dynamically based on resolution URIs and event properties.

Our sample event representation accepts all URIs and returns a `MyPlugInPropertiesBeanFactory`:

```
public class MyPlugInEventRepresentation implements PlugInEventRepresentation {

  // ... methods as seen earlier
  public boolean acceptsEventBeanResolution(
        PlugInEventBeanReflectorContext eventBeanContext) {
    return true;
  }

  public PlugInEventBeanFactory getEventBeanFactory(
        PlugInEventBeanReflectorContext eventBeanContext) {
    return new MyPlugInPropertiesBeanFactory(types);
   }
}
```

Last, the sample `MyPlugInPropertiesBeanFactory` implements the `PlugInEventBeanFactory` interface. It inspects incoming events and determines an event type based on whether all properties for that event type are present:

```
public class MyPlugInPropertiesBeanFactory implements PlugInEventBeanFactory {
  private final List<MyPlugInPropertiesEventType> knownTypes;

  public MyPlugInPropertiesBeanFactory(List<MyPlugInPropertiesEventType> types)
 {
    knownTypes = types;
  }

  public EventBean create(Object event, URI resolutionURI) {
    Properties properties = (Properties) event;

    // use the known types to determine the type of the object
    for (MyPlugInPropertiesEventType type : knownTypes) {
       // if there is one property the event does not contain, then its not
 the right type
      boolean hasAllProperties = true;
      for (String prop : type.getPropertyNames()) {
        if (!properties.containsKey(prop)) {
          hasAllProperties = false;
          break;
        }
      }

      if (hasAllProperties) {
        return new MyPlugInPropertiesEventBean(type, properties);
      }
    }
    return null; // none match, unknown event
  }
}
```

Chapter 18. Script Support

18.1. Overview

Esper allows the use scripting languages within EPL. You may use scripts for imperative programming to execute certain code as part of EPL processing by the engine.

The syntax and examples outlined below discuss how to declare a script that is visible to the same EPL statement that listed the script.

For declaring scripts that are visible across multiple EPL statements i.e. globally visible scripts please consult *Section 5.19.2, "Declaring a Global Script"* that explains the `create expression` clause.

Any scripting language that supports JSR 223 and also the MVEL scripting language can be specified in EPL. This section provides MVEL and JavaScript examples.

For more information on the MVEL scripting language and its syntax, please refer to *MVEL At Codehaus* [http://mvel.codehaus.org/]. MVEL is an expression language that has a natural syntax for Java-based applications and compiles to provide fast execution times. To use MVEL with Esper, please make sure to add the MVEL jar file to the application classpath.

For more information on JSR 223 scripting languages, please refer to external resources. As JSR 223 defines a standard API for script engines, your application may use any script engine that implements the API. Current JVM versions ship with a JavaScript script engine. Other script engines such as Groovy, Ruby and Python scripts can be used as implementations of JSR 223.

As an alternative to a script consider providing a custom single row function as described in *Section 17.3, "Single-Row Function"*

18.2. Syntax

The syntax for scripts is:

```
expression [return_type] [dialect_identifier:] script_name [ (parameters) ]
  [ script_body ]
```

Use the `expression` keyword to declare a script.

The *return_type* is optional. If the script declaration provides a return type the engine can perform strong type checking: Any expressions that invoke the script and use the return value are aware of the return type. If no return type is provided the engine assumes the script returns `java.lang.Object`.

The *dialect_identifier* is optional and identifies the scripting language. Use `mvel` for MVEL , `js` for JavaScript and `python` for Python and similar for other JSR 223 scripting languages. If no dialect identifier is specified, the default dialect that is configured applies, which is `js` unless your application changes the default configuration.

It follows the script name. You may use the same script name multiple times and thus overload providing multiple signatures under the same script name. The combination of script name and number of parameters must be unique however.

If you have script parameters, specify the parameter names for the script as a comma-separated list of identifiers in parenthesis. It is not necessary to list parameter types.

The *script body* is the actual MVEL or JavaScript or other scripting language script and is placed in square brackets: `[ ... script body ...]`.

18.3. Examples

The next example shows an EPL statement that calls a JavaScript script which computes the Fibonacci total for a given number:

```
expression double js:fib(num) [
fib(num);
function fib(n) {
  if(n <= 1)
    return n;
  return fib(n-1) + fib(n-2);
}
]
select fib(intPrimitive) from SupportBean;
```

The `expression` keyword is followed by the return type (`double`), the dialect (`js`) and the script name (`fib`) that declares a single parameter (`num`). The JavaScript code that computes the Fibonacci total is between square brackets `[]`.

The following example shows an EPL statement that calls a MVEL script which outputs all the different colors that are listed in the `colors` property of each `ColorEvent`:

```
expression mvel:printColors(colors) [
String c = null;
for (c : colors) {
   System.out.println(c);
}
]
select printColors(colors) from ColorEvent;
```

This example instead uses JavaScript to print colors and passes the event itself as a script parameter:

```
expression js:printColors(colorEvent) [
importClass (java.lang.System);
importClass (java.util.Arrays);
System.out.println(Arrays.toString(colorEvent.getColors()));
]
select printColors(colorEvent) from ColorEvent as colorEvent
```

18.4. Built-In EPL Script Attributes

The engine provides a built-in script object under the variable name `epl` to all scripts. Your scripts may use this script object to share and retain state by setting and reading script attributes.

The `epl` script object implements the interface `com.espertech.esper.client.hook.EPLScriptContext`. The `EPLScriptContext` interface has two methods: The `void setScriptAttribute(String attribute, Object value)` method to set an attribute value and the `Object getScriptAttribute(String attribute)` method to read an attribute value.

The engine maintains a separate script object per context partition, or per statement if not declaring a context. Therefore script attributes are not shared between statements.

The next example demonstrates the use of the `epl` script object. It outputs a flag value `true` when an RFID event matched because the location is `A`, and outputs a flag value `false` when an RFID event matched because the location is `B`. The example works the same for either MVEL or JavaScript dialects: You may simple replace the `js` dialect with `mvel`.

```
expression boolean js:setFlag(name, value, returnValue) [
  if (returnValue) epl.setScriptAttribute(name, value);
  returnValue;
]
expression js:getFlag(name) [
  epl.getScriptAttribute(name);
]
select getFlag('locA') as flag from RFIDEvent(zone = 'Z1' and
  (setFlag('locA', true, location = 'A') or setFlag('locA', false, location =
 'B')) )
```

The example above utilizes two scripts: The `setFlag` script receives an attribute name, attribute value and a return value. The script sets the script attribute only when the return value is true. The `getFlag` script simply returns the script attribute value.

18.5. Performance Notes

Upon EPL statement compilation, the engine resolves script parameter types and performs script compilation. At runtime the engine evaluates the script in its compiled form.

As the engine cannot inspect scripts if is not possible for the engine to perform query planning or many optimizations based on the information in scripts. It is thus recommended to structure EPL such that basic filter and join expressions are EPL expressions and not script expressions.

18.6. Additional Notes

Your EPL may declare a return type for the script. If no return type is declared and when using the MVEL dialect, the engine will infer the return type from the MVEL expression analysis result. If the return type is not provided and cannot be inferred or the dialect is not MVEL, the return type is `Object`.

If the EPL declares a numeric return type then engine performs coercion of the numeric result to the return type that is specified.

In the case that the EPL declares a return type that does not match the type of the actual script return value, the engine does not check return value type.

Chapter 19. Examples, Tutorials, Case Studies

19.1. Examples Overview

This chapter outlines the examples that come with Esper in the `examples` folder of the distribution. Each sample is in a separate folder that contains all files needed by the example, excluding jar files.

Here is an overview over the examples in alphabetical order:

Table 19.1. Examples

Name	Description
Section 19.3, "AutoID RFID Reader"	An array of RFID readers sense RFID tags as pallets are coming within the range of one of the readers. Shows the use of an XSD schema and XML event representation. A single statement shows a rolling time window, a where-clause filter on a nested property and a group-by.
Section 19.6, "Market Data Feed Monitor"	Processes a raw market data feed and reports throughput statistics and detects when the data rate of a feed falls off unexpectedly. Demonstrates a batch time window and a rolling time window with a having-clause. Multi-threaded example with a configurable number of threads and a simulator for generating feed data.
Section 19.12, "MatchMaker"	In the MatchMaker example every mobile user has an X and Y location and 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 certain properties match preferences. Dynamically creates and removes event patterns that use range matching based on mobile user events received.
Section 19.13, "Named Window Query"	A mini-benchmark that handles temperature sensor events. The sample creates a named window and fills it with a large number of events. It then executes a large number of pre-defined queries as well as fire-and-forget queries and reports times. Study this example if you are interested in named windows, Map event type representation, fire-and-forget queries as well as pre-defined queries via on-select, and the performance aspects.

Name	Description
Section 19.14, "Sample Virtual Data Window"	This example demonstrates the use of virtual data window to expose a (large) external data store, without any need to keep events in memory, and without sacrificing query performance.
Section 19.15, "Sample Cycle Detection"	This example showcases the aggregation multi-function extension API for use with a cycle-detection problem detecting cycles in transactions between accounts.
Section 19.7, "OHLC Plug-in View"	A plug-in custom view addressing a problem in the financial space: Computes open-high-low-close bars for minute-intervals of events that may arrive late, based on each event's timestamp. A custom plug-in view based on the extension API can be a convenient and reusable way to express a domain-specific analysis problem as a unit, and this example includes the code for the OHLC view factory and view as well as simulator to test the view.
Section 20.3, "Using the performance kit"	A benchmark that is further described in the performance section of this document under performance kit.
Section 19.16, "Quality of Service"	This example develops some code for measuring quality-of-service levels such as for a service-level agreement (SLA). This example combines patterns with select-statements, shows the use of the timer `'at'` operator and followed-by operator `->`, and uses the iterator API to poll for current results.
Section 19.10, "Assets Moving Across Zones - An RFID Example"	An example out of the RFID domain processes location report events. The example includes a simple Swing-based GUI for visualization allows moving tags from zone to zone visually. It also a contains comprehensive simulator to generate data for a large number of asset groups and their tracking. The example hooks up statements that aggregate and detect patterns in the aggregated data to determine when an asset group constraint is violated.
Section 19.4, "Runtime Configuration"	Example code to demonstrate various key runtime configuration options such as adding event types on-the-fly, adding new variables, adding plug-in single-row and aggregation functions and adding variant streams and revision type definition.
Section 19.5, "JMS Server Shell and Client"	The server shell is a Java Messaging Service (JMS) -based server and client that send and listens to messages on a JMS destination. It also demonstrates a simple Java Management Extension (JMX) MBean for remote statement management.

Name	Description
	A single EPL statement computes an average duration for each IP address on a rolling time window and outputs a snapshot every 2 seconds.
Section 19.11, “StockTicker”	An example from the financial domain that features event patterns to filter stock tick events based on price and symbol. The example is designed to provide a high volume of events and includes multithreaded unit test code as well as a simulting data generator. Perhaps this is a good example to learn the API and get started with event patterns. The example dynamically creates and removes event patterns based on price limit events received.
Section 19.9, “Self-Service Terminal”	A J2EE-based self-service terminal managing system in an airport that gets a lot of events from connected terminals. Contains a message-driven bean (EJB-MDB) for use in a J2EE container, a client and a simulator, as well as EPL statements for detecting various conditions. A version that runs outside of a J2EE container is also available.
Section 19.17, “Trivia Geeks Club”	Trivia Geeks Club demonstrates EPL for a scoring system computing scores in a trivia game.

19.2. Running the Examples

In order to compile and run the samples please follow the below instructions:

1. Make sure Java 1.6 or greater is installed and the JAVA_HOME environment variable is set.
2. Open a console window and change directory to examples/*example_name*/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 an example.
5. Now you are ready to run an example. Some examples require mandatory parameters that are also described in the file "readme.txt" in the "etc" folder.
6. Modify the logger logging level in the "log4j.xml" configuration file changing DEBUG to INFO on a class or package level to control the volume of text output.

Each example also provides Eclipse project `.classpath` and `.project` files. The Eclipse projects expect an `esper_runtime` user library that includes the runtime dependencies.

JUnit tests exist for the example code. The JUnit test source code for the examples can be found in each example's `src/test` folder. To build and run the example JUnit tests, use the Maven 2 goal `test`.

19.3. AutoID RFID Reader

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
```

19.4. Runtime Configuration

This example demonstrates various key runtime configuration options such as adding event types on-the-fly, adding new variables, adding plug-in single-row and aggregation functions and adding variant streams and revision type definition.

The classes for this example live in package `com.espertech.esper.example.runtimeconfig`.

19.5. JMS Server Shell and Client

19.5.1. 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.

19.5.2. 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.

19.5.3. 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, attach a listener to the statement and 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.

19.6. 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.

19.6.1. 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;
```

19.6.2. 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
```

19.6.3. 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
```

19.6.4. 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.

19.7. OHLC Plug-in View

This example contains a fully-functional custom view based on the extension API that computes OHLC open-high-low-close bars for events that provide a long-typed timestamp and a double-typed value.

OHLC bar is a problem out of the financial domain. The "Open" refers to the first datapoint and the "Close" to the last datapoint in an interval. The "High" refers to the maximum and the "Low" to the minimum value during each interval. The term "bar" is used to describe each interval results of these 4 values.

The example provides an OHLC view that is hardcoded to 1-minute bars. It considers the timestamp value carried by each event, and not the system time. The cutoff time after which an event is no longer considered for a bar is hardcoded to 5 seconds.

The view assumes that events arrive in timestamp order: Each event's timestamp value is equal to or higher then the timestamp value provided by the prior event.

The view may also be used together with `std:groupwin` to group per criteria, such as symbol. In this case the assumption of timestamp order applies per symbol.

The view gracefully handles no-event and late-event scenarios. Interval boundaries are defined by system time, thus event timestamp and system time must roughly be in-sync, unless using external timer events.

19.8. 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.

19.8.1. 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)

19.8.2. 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.

19.8.3. 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).

19.8.4. 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.

19.8.5. 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.

19.9. Self-Service Terminal

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.

19.9.1. 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.

19.9.2. 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)) )]
```

19.9.3. 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
```

19.9.4. 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
```

19.9.5. 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 standalone version that does not require a J2EE container is in "examples/terminalsvc-jse".

19.9.5.1. 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 "standalone/deployments".

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.

19.9.5.2. 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 7.1.1 and Maven 3.0.4.

The Maven build packages the EAR file for deployment to a JBoss application server with default deployment options.

19.9.5.3. 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). Before running please copy the `jboss-client.jar` file from your JBoss AS installation bin directory to the "terminalsvc/lib" folder.

The receiver and sender code use "guest" as user and "pass" as password. Add the "guest" user using the Jboss "add-user" script and assign the role "guest". Your JBoss server may need to start with "-c standalone-full.xml" to have the messaging subsystem available.

Add queue configurations to the messaging subsystem configuration as follows:

```
<jms-queue name="queue_a">
  <entry name="queue_a"/>
  <entry name="java:jboss/exported/jms/queue/queue_a"/>
</jms-queue>
<jms-queue name="queue_b">
  <entry name="queue_b"/>
  <entry name="java:jboss/exported/jms/queue/queue_b"/>
</jms-queue>
```

Disable persistence in the messaging subsystem for this example so we are not running out of disk space:

```
<persistence-enabled>false</persistence-enabled>
```

19.10. 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.

19.11. 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

19.12. 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

19.13. Named Window Query

This example handles very minimal temperature sensor events which are represented by `java.util.Map`. It creates a named window and fills it with a large number of events. It then executes a large number of pre-defined queries via on-select as well as performs a large number of fire-and-forget queries against the named window, and reports execution times.

19.14. Sample Virtual Data Window

Virtual data windows are an extension API used to integrate external stores and expose the data therein as a named window.

See the `virtualdw` folder for example code, compile and run scripts.

19.15. Sample Cycle Detection

The example is also discussed in the section on extension APIs specifically the aggregation multi-function development. The example uses the `jgrapht` library for a cycle-detection problem detecting cycles in transactions between accounts.

See the `examples/cycledetect` folder for example code, compile and run scripts.

19.16. Quality of Service

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.

19.17. Trivia Geeks Club

This example was developed for the DEBS 2011 conference and demonstrates how scoring rules for a trivia game can be implemented in EPL.

The EPL module that implements all scoring rules is located in the `etc` folder in file `trivia.epl`. The EPL is all required to run the solution without any custom functions required.

The trivia geeks club rules (the requirements) are provided in the `etc` folder in file `trivia_scoring_requirements.htm`.

The implementation we provide tests the questions, answers and scoring according to the data provided in `trivia_test_questions_small.htm` and `trivia_test_questions_large.htm`.

Chapter 20. 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.

20.1. 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.
```

20.2. Performance Tips

20.2.1. 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.

20.2.2. Input and Output Bottlenecks

Your application receives output events from Esper statements through the `UpdateListener` interface or via the strongly-typed subscriber POJO object. Such output events are delivered by the application or timer thread(s) that sends an input event into the engine instance.

The processing of output events that your listener or subscriber performs temporarily blocks the thread until the processing completes, and may thus reduce throughput. It can therefore be beneficial for your application to process output events asynchronously and not block the Esper engine while an output event is being processed by your listener, especially if your listener code performs blocking IO operations.

For example, your application may want to send output events to a JMS destination or write output event data to a relational database. For optimal throughput, consider performing such blocking operations in a separate thread.

Additionally, when reading input events from a store or network in a performance test, you may find that Esper processes events faster then you are able to feed events into Esper. In such case you may want to consider an in-memory driver for use in performance testing. Also consider decoupling your read operation from the event processing operation (sendEvent method) by having multiple readers or by pre-fetching your data from the store.

20.2.3. Theading

We recommend using multiple threads to send events into Esper. We provide a test class below. Our test class does not use a blocking queue and thread pool so as to avoid a point of contention.

A sample code for testing performance with multiple threads is provided:

```
public class SampleClassThreading {

    public static void main(String[] args) throws InterruptedException {

        int numEvents = 1000000;
        int numThreads = 3;

        Configuration config = new Configuration();
        config.getEngineDefaults().getThreading()
          .setListenerDispatchPreserveOrder(false);
        config.getEngineDefaults().getThreading()
            .setInternalTimerEnabled(false);    // remove thread that handles
 time advancing
        EPServiceProvider engine = EPServiceProviderManager
          .getDefaultProvider(config);

 engine.getEPAdministrator().getConfiguration().addEventType(MyEvent.class);

        engine.getEPAdministrator().createEPL(
          "create context MyContext coalesce by consistent_hash_crc32(id) " +
          "from MyEvent granularity 64 preallocate");
      String epl = "context MyContext select count(*) from MyEvent group by id";
        EPStatement stmt = engine.getEPAdministrator().createEPL(epl);
        stmt.setSubscriber(new MySubscriber());

        Thread[] threads = new Thread[numThreads];
        CountDownLatch latch = new CountDownLatch(numThreads);

        int eventsPerThreads = numEvents / numThreads;
        for (int i = 0; i < numThreads; i++) {
            threads[i] = new Thread(
              new MyRunnable(latch, eventsPerThreads, engine.getEPRuntime()));
        }
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < numThreads; i++) {
            threads[i].start();
        }

        latch.await(10, TimeUnit.MINUTES);
        if (latch.getCount() > 0) {
            throw new RuntimeException("Failed to complete in 10 minute");
        }
        long delta = System.currentTimeMillis() - startTime;
        System.out.println("Took " + delta + " millis");
    }

    public static class MySubscriber {
        public void update(Object[] args) {
        }
    }

    public static class MyRunnable implements Runnable {
        private final CountDownLatch latch;
        private final int numEvents;
        private final EPRuntime runtime;

      public MyRunnable(CountDownLatch latch, int numEvents, EPRuntime runtime) {
            this.latch = latch;
            this.numEvents = numEvents;
            this.runtime = runtime;
        }

        public void run() {
            Random r = new Random();
            for (int i = 0; i < numEvents; i++) {
                runtime.sendEvent(new MyEvent(r.nextInt(512)));
            }
            latch.countDown();
        }
    }

    public static class MyEvent {
        private final int id;

        public MyEvent(int id) {
            this.id = id;
        }

        public int getId() {
            return id;
        }
    }
}
```

We recommend using Java threads as above, or a blocking queue and thread pool with `sendEvent()` or alternatively we recommend configuring inbound threading if your application does not already employ threading. Esper provides the configuration option to use engine-level queues and threadpools for inbound, outbound and internal executions. See *Section 14.7.1, "Advanced Threading"* for more information.

We recommend the outbound threading if your listeners are blocking. For outbound threading also see the section below on tuning and disabling listener delivery guarantees.

If enabling advanced threading options keep in mind that the engine will maintain a queue and thread pool. There is additional overhead associated with entering work units into the queue, maintaining the queue and the hand-off between threads. The Java blocking queues are not necessarily fast on all JVM. It is not necessarily true that your application will perform better with any of the advanced threading options.

We found scalability better on Linux systems and running Java with `-server` and pinning threads to exclusive CPUs and after making sure CPUs are available on your system.

We recommend looking at LMAX Disruptor, an inter-thread messaging library.

20.2.4. 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
```

20.2.5. Prefer stream-level filtering over where-clause filtering

Esper stream-level filtering is very well optimized, while filtering via the where-clause post any data windows is not optimized.

The same is true for named windows. If your application is only interested in a subset of named window data and such filters are not correlated to arriving events, place the filters into parenthesis after the named window name.

20.2.5.1. Examples without named windows

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.

20.2.5.2. Examples using named windows

The next two example EPL queries put the account number filter criteria directly into parenthesis following the named window name:

```
// Better performance : stream-level filtering
select * from WithdrawalNamedWindow(accountNumber = '123')
```

```
// Better performance : example with subquery
select *, (select * from LoginSucceededWindow(accountNumber = '123'))
from WithdrawalNamedWindow(accountNumber = '123')
```

20.2.5.3. Common computations in where-clauses

If you have a number of queries performing a given computation on incoming events, consider moving the computation from the where-clause to a plug-in user-defined function that is listed as part of stream-level filter criteria. The engine optimizes evaluation of user-defined functions in filters such that an incoming event can undergo the computation just once even in the presence of N queries.

```
// Prefer stream-level filtering with a user-defined function
select * from MarketData(vstCompare(*))
```

```
// Less preferable when there are N similar queries:
// Move the computation in the where-clause to the "vstCompare" function.
select * from MarketData where (VST * RT) – (VST / RT) > 1
```

20.2.6. 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)
```

20.2.7. Remove Unneccessary Constructs

If your statement uses `order by` to order output events, consider removing `order by` unless your application does indeed require the events it receives to be ordered.

If your statement specifies `group by` but does not use aggregation functions, consider removing `group by`.

If your statement specifies `group by` but the filter criteria only allows one group, consider removing `group by`:

```
// Prefer:
select * from MarketData(symbol = 'GE') having sum(price) > 1000

// Don't use this since the filter specifies a single symbol:
select * from MarketData(symbol = 'GE') group by symbol having sum(price) > 1000
```

If your statement specifies the grouped data window `std:groupwin` but the window being grouped retains the same set of events regardless of grouping, remove `std:groupwin`:

```
// Prefer:
create window MarketDataWindow.win:keepall() as MarketDataEventType

// Don't use this, since keeping all events
// or keeping all events per symbol is the same thing:
create window MarketDataWindow.std:groupwin(symbol).win:keepall() as
 MarketDataEventType

// Don't use this, since keeping the last 1-minute of events
// or keeping 1-minute of events per symbol is the same thing:
create window MarketDataWindow.std:groupwin(symbol).win:time(1 min) as
 MarketDataEventType
```

It is not necessary to specify a data window for each stream.

```
// Prefer:
select * from MarketDataWindow

// Don't have a data window if just listening to events, prefer the above
select * from MarketDataWindow.std:lastevent()
```

If your statement specifies unique data window but the filter criteria only allows one unique criteria, consider removing the unique data window:

```
// Prefer:
select * from MarketDataWindow(symbol = 'GE').std:lastevent()

// Don't have a unique-key data window if your filter specifies a single value
select * from MarketDataWindow(symbol = 'GE').std:unique(symbol)
```

20.2.8. End Pattern Sub-Expressions

In patterns, the `every` keyword in conjunction with followed by (`->`) starts a new sub-expression per match.

For example, the following pattern starts a sub-expression looking for a B event for every A event that arrives.

```
every A -> B
```

Determine under what conditions a subexpression should end so the engine can stop looking for a B event. Here are a few generic examples:

```
every A -> (B and not C)
every A -> B where timer:within(1 sec)
```

20.2.9. 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, EPServiceProvider epServiceProvider) {
        String symbol = (String) symbolGetter.get(eventBeans[0]);
        long volume = (Long) volumeGetter.get(eventBeans[0]);
        // some more logic here
    }
```

20.2.10. 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
    }
```

20.2.11. Turn off logging and audit

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.

Esper provides the `@Audit` annotation for statements. For performance testing and production deployment, we recommend removing `@Audit`.

20.2.12. Disable view sharing

By default, Esper compares streams and views in use with existing statement's streams and views, and then reuses 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);
```

20.2.13. Tune or 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);
```

Additional configuration options are available and described in the configuration section that specify timeout values and spin or thread context switching.

Esper logging will log the following informational message when guaranteed delivery order to listeners is enabled and spin lock times exceed the default or configured timeout : `Spin wait timeout exceeded in listener dispatch`. The respective message for delivery from `insert into` statements to consuming statements is `Spin wait timeout exceeded in insert-into dispatch`.

If your application sees messages that spin lock times are exceeded, your application has several options: First, disabling preserve order is an option. Second, ensure your listener does not perform (long-running) blocking operations before returning, for example by performing output event processing in a separate thread. Third, change the timeout value to a larger number to block longer without logging the message.

20.2.14. Use a Subscriber Object to Receive Events

The subscriber object is a technique to receive result data that has performance advantages over the `UpdateListener` interface. Please refer to *Section 14.3.3, “Setting a Subscriber Object”*.

20.2.15. Consider Data Flows

Data flows offer a high-performance means to execute EPL select statements and use other built-in data flow operators. The data flow `Emitter` operator allows sending underlying event objects directly into a data flow. Thereby the engine does not need to wrap each underlying event into a `EventBean` instance and the engine does not need to match events to statements. Instead, the underling event directly applies to only that data flow instance that your application submits the event to, and no other continuous query statements or data flows see the same event.

Data flows are described in *Chapter 13, EPL Reference: Data Flow*.

20.2.16. High-Arrival-Rate Streams and Single Statements

A context partition is associated with certain context partition state that consists of current aggregation values, partial pattern matches, data windows or other view state depending on whether your statement uses such constructs. When an engine receives events it updates context partition state under locking such that context partition state remains consistent under concurrent multi-threaded access.

For high-volume streams, the locking required to protected context partition state may slow down or introduce blocking for very high arrival rates of events that apply to the very same context partition and its state.

Your first choice should be to utilize a context that allows for multiple context partitions, such as the hash segmented context. The hash segmented context usually performs better compared to the keyed segmented context since in the keyed segmented context the engine must check whether a partition exists or must be created for a given key.

Your second choice is to split the statement into multiple statements that each perform part of the intended function or that each look for a certain subset of the high-arrival-rate stream. There is very little cost in terms of memory or CPU resources per statement, the engine can handle larger number of statements usually as efficiently as single statements.

For example, consider the following statement:

```
// less effective in a highly threaded environment
select venue, ccyPair, side, sum(qty)
from CumulativePrice
where side='O'
group by venue, ccyPair, side
```

The engine protects state of each context partition by a separate lock for each context partition, as discussed in the API section. In highly threaded applications threads may block on a specific context partition. You would therefore want to use multiple context partitions.

Consider creating a keyed segmented context, for example:

```
create context MyContext partition by venue, ccyPair, side from
 CumulativePrice(side='O')
```

The keyed segmented context instructs the engine to employ a context partition per `venue, ccyPair, side` key combination. As locking is on the level of context partition, the locks taken by the engine are very fine grained allowing for highly concurrent processing.

The new statement that refers to the context as created above is:

```
context MyContext select venue, ccyPair, side, sum(qty) from CumulativePrice
```

For testing purposes or if your application controls concurrency, you may disable context partition locking, see *Section 15.4.23.3, “Disable Locking”*.

20.2.17. Subqueries versus Joins And Where-clause And Data Windows

When joining streams the engine builds a product of the joined data windows based on the `where` clause. It analyzes the `where` clause at time of statement compilation and builds the appropriate indexes and query strategy. Avoid using expressions in the join `where` clause that require evaluation, such as user-defined functions or arithmatic expressions.

When joining streams and not providing a `where` clause, consider using the `std:unique` data window or `std:lastevent` data window to join only the last event or the last event per unique key(s) of each stream.

The sample query below can produce up to 5,000 rows when both data windows are filled and an event arrives for either stream:

```
// Avoid joins between streams with data windows without where-clause
select * from StreamA.win:length(100), StreamB.win:length(50)
```

Consider using a subquery, consider using separate statements with insert-into and consider providing a `where` clause to limit the product of rows.

Below examples show different approaches, that are not semantically equivalent, assuming that an `MyEvent` is defined with the properties symbol and value:

```
// Replace the following statement as it may not perform well
select a.symbol, avg(a.value), avg(b.value)
from MyEvent.win:length(100) a, MyEvent.win:length(50) b

// Join with where-clause
select a.symbol, avg(a.value), avg(b.value)
from MyEvent.win:length(100) a, MyEvent.win:length(50) b
where a.symbol = b.symbol

// Unidirectional join with where-clause
select a.symbol, avg(b.value)
from MyEvent unidirectional, MyEvent.win:length(50) b
where a.symbol = b.symbol

// Subquery
select
  (select avg(value) from MyEvent.win:length(100)) as avgA,
  (select avg(value) from MyEvent.win:length(50)) as avgB,
  a.symbol
from MyEvent
```

```
// Since streams cost almost nothing, use insert-into to populate and a
 unidirectional join
insert into StreamAvgA select symbol, avg(value) as avgA from
 MyEvent.win:length(100)
insert into StreamAvgB select symbol, avg(value) as avgB from
 MyEvent.win:length(50)
select a.symbol, avgA, avgB from StreamAvgA unidirectional,
 StreamAvgB.std:unique(symbol) b
where a.symbol = b.symbol
```

A join is multidirectionally evaluated: When an event of any of the streams participating in the join arrive, the join gets evaluated, unless using the unidirectional keyword. Consider using a subquery instead when evaluation only needs to take place when a certain event arrives:

```
// Rewrite this join since we don't need to join when a LoginSucceededWindow
 arrives
// Also rewrite because the account number always is the value 123.
select * from LoginSucceededWindow as l, WithdrawalWindow as w
where w.accountNumber = '123' and w.accountNumber = l.accountNumber

// Rewritten as a subquery,
select *, (select * from LoginSucceededWindow where accountNumber=’123’)
from WithdrawalWindow(accountNumber=’123’) as w
```

20.2.18. Patterns and Pattern Sub-Expression Instances

The `every` and repeat operators in patterns control the number of sub-expressions that are active. Each sub-expression can consume memory as it may retain, depending on the use of tags in the pattern, the matching events. A large number of active sub-expressions can reduce performance or lead to out-of-memory errors.

During the design of the pattern statement consider the use of `timer:within` to reduce the amount of time a sub-expression lives, or consider the `not` operator to end a sub-expression.

The examples herein assume an `AEvent` and a `BEvent` event type that have an `id` property that may correlate between arriving events of the two event types.

In the following sample pattern the engine starts, for each arriving AEvent, a new pattern sub-expression looking for a matching BEvent. Since the AEvent is tagged with `a` the engine retains each AEvent until a match is found for delivery to listeners or subscribers:

```
every a=AEvent -> b=BEvent(b.id = a.id)
```

One way to end a sub-expression is to attach a time how long it may be active.

The next statement ends sub-expressions looking for a matching BEvent 10 seconds after arrival of the AEvent event that started the sub-expression:

```
every a=AEvent -> (b=BEvent(b.id = a.id) where timer:within(10 sec))
```

A second way to end a sub-expression is to use the `not` operator. You can use the `not` operator together with the `and` operator to end a sub-expression when a certain event arrives.

The next statement ends sub-expressions looking for a matching BEvent when, in the order of arrival, the next BEvent that arrives after the AEvent event that started the sub-expression does not match the id of the AEvent:

```
every a=AEvent -> (b=BEvent(b.id = a.id) and not BEvent(b.id != a.id))
```

The `every-distinct` operator can be used to keep one sub-expression alive per one or more keys. The next pattern demonstrates an alternative to `every-distinct`. It ends sub-expressions looking for a matching BEvent when an AEvent arrives that matches the id of the AEvent that started the sub-expression:

```
every a=AEvent -> (b=BEvent(b.id = a.id) and not AEvent(b.id = a.id))
```

20.2.19. Pattern Sub-Expression Instance Versus Data Window Use

For some use cases you can either specify one or more data windows as the solution, or you can specify a pattern that also solves your use case.

For patterns, you should understand that the engine employs a dynamic state machine. For data windows, the engine employs a delta network and collections. Generally you may find patterns that require a large number of sub-expression instances to consume more memory and more CPU then data windows.

For example, consider the following EPL statement that filters out duplicate transaction ids that occur within 20 seconds of each other:

```
select * from TxnEvent.std:firstunique(transactionId).win:time(20 sec)
```

You could also address this solution using a pattern:

```
select * from pattern [every-distinct(a.transactionId) a=TxnEvent where
 timer:within(20 sec)]
```

If you have a fairly large number of different transaction ids to track, you may find the pattern to perform less well then the data window solution as the pattern asks the engine to manage a pattern sub-expression per transaction id. The data window solution asks the engine to manage expiry, which can give better performance in many cases.

Similar to this, it is generally preferable to use EPL join syntax over a pattern that cardinally detects relationships i.e. `pattern [every-distinct(...) ... -> every-distinct(...) ...]`. Join query planning is a powerful Esper feature that implements fast relational joins.

20.2.20. The Keep-All Data Window

The `std:keepall` data window is a data window that retains all arriving events. The data window can be useful during the development phase and to implement a custom expiry policy using `on-delete` and named windows. Care should be taken to timely remove from the keep-all data window however. Use `on-select` or on-demand queries to count the number of rows currently held by a named window with keep-all expiry policy.

20.2.21. Statement Design for Reduced Memory Consumption - Diagnosing OutOfMemoryError

This section describes common sources of out-of-memory problems.

If using the keep-all data window please consider the information above. If using pattern statements please consider pattern sub-expression instantiation and lifetime as discussed prior to this section.

When using the `group-by` clause or `std:groupwin` grouped data windows please consider the hints as described below. Make sure your grouping criteria are fields that don't have an unlimited number of possible values or specify hints otherwise.

The `std:unique` unique data window can also be a source for error. If your uniqueness criteria include a field which is never unique the memory use of the data window can grow, unless your application deletes events.

When using the `every-distinct` pattern construct parameterized by distinct value expressions that generate an unlimited number of distinct values, consider specifying a time period as part of the parameters to indicate to the pattern engine how long a distinct value should be considered.

In a match-recognize pattern consider limiting the number of optional events if optional events are part of the data reported in the `measures` clause. Also when using the partition clause, if your partitioning criteria include a field which is never unique the memory use of the match-recognize pattern engine can grow.

A further source of memory use is when your application creates new statements but fails to destroy created statements when they are no longer needed.

In your application design you may also want to be conscious when the application listener or subscriber objects retain output data.

An engine instance, uniquely identified by an `engine URI` is a relatively heavyweight object. Optimally your application allocates only one or a few engine instances per JVM. A statement instance is associated to one engine instance, is uniquely identified by a statement name and is a medium weight object. We have seen applications allocate 100,000 statements easily. A statement's context partition instance is associated to one statement, is uniquely identified by a context partition id and is a light weight object. We have seen applications allocate 5000 context partitions for 100 statements easily, i.e. 5,000,000 context partitions. An aggregation row, data window row, pattern etc. is associated to a statement context partition and is a very lightweight object itself.

The `prev`, `prevwindow` and `prevtail` functions access a data window directly. The engine does not need to maintain a separate data structure and grouping is based on the use of the `std:groupwin` grouped data window. Compare this to the use of event aggregation functions such as `first`, `window` and `last` which group according to the `group by` clause. If your statement utilizes both together consider reformulating to use `prev` instead.

20.2.22. 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.

20.2.23. Consider using Hints

The @Hint annotation provides a single keyword or a comma-separated list of keywords that provide instructions to the engine towards statement execution that affect runtime performance and memory-use of statements. Also see *Section 5.2.7.8, “@Hint”*.

The hints for use with joins, outer joins, unidirectional joins, relational and non-relational join query planning are described in *Section 5.12.5, “Hints Related to Joins”*.

The hint for use with `group by` to specify how state for groups is reclaimed is described in *Section 5.6.2.1, “Hints Pertaining to Group-By”* and *Section 12.3.2, “Grouped Data Window (std:groupwin)”*.

The hint for use with `group by` to specify aggregation state reclaim for unbound streams and timestamp groups is described in *Section 5.6.2.1, “Hints Pertaining to Group-By”*.

The hint for use with `match_recognize` to specify iterate-only is described in *Section 7.4.6, “Eliminating Duplicate Matches”*.

To tune subquery performance when your subquery selects from a named window, consider the hints discussed in *Section 5.11.7, “Hints Related to Subqueries”*.

The `@NoLock` hint to remove context partition locking (also read caution note) is described at *Section 14.7, “Engine Threading and Concurrency”*.

The hint for influencing query planning is elaborated in *Section 20.2.33, “Query Planning Index Hints”* and query planning is described in *Section 20.2.32, “Notes on Query Planning”*.

20.2.24. Optimizing Stream Filter Expressions

Assume your EPL statement invokes a static method in the stream filter as the below statement shows as an example:

```
select * from MyEvent(MyHelperLibrary.filter(field1, field2, field3,
 field4*field5))
```

As a result of starting above statement, the engine must evaluate each MyEvent event invoking the `MyHelperLibrary.filter` method and passing certain event properties. The same applies to pattern filters that specify functions to evaluate.

If possible, consider moving some of the checking performed by the function back into the filter or consider splitting the function into a two parts separated by `and` conjunction. In general for all expressions, the engine evaluates expressions left of the `and` first and can skip evaluation of the further expressions in the conjunction in the case when the first expression returns false. In addition the engine can build a reverse index for fields provided in stream or pattern filters.

For example, the below statement could be faster to evaluate:

```
select * from MyEvent(field1="value" and
  MyHelperLibrary.filter(field1, field2, field3, field4*field5))
```

20.2.25. Statement and Engine Metric Reporting

You can use statement and engine metric reporting as described in *Section 14.15, “Engine and Statement Metrics Reporting”* to monitor performance or identify slow statements.

20.2.26. Expression Evaluation Order and Early Exit

The term "early exit" or "short-circuit evaluation" refers to when the engine can evaluate an expression without a complete evaluation of all sub-expressions.

Consider an expression such as follows:

```
where expr1 and expr2 and expr3
```

If expr1 is false the engine does not need to evaluate expr2 and expr3. Therefore when using the `AND` logical operator consider reordering expressions placing the most-selective expression first and less selective expressions thereafter.

The same is true for the `OR` logical operator: If expr1 is true the engine does not need to evaluate expr2 and expr3. Therefore when using the `OR` logical operator consider reordering expressions placing the least-selective expression first and more selective expressions thereafter.

The order of expressions (here: expr1, expr2 and expr3) does not make a difference for the join and subquery query planner.

Note that the engine does not guarantee short-circuit evaluation in all cases. The engine may rewrite the where-clause or filter conditions into another order of evaluation so that it can perform index or reverse index lookups.

20.2.27. Large Number of Threads

When using a large number of threads with the engine, such as more then 100 threads, we provide a setting in the configuration that instructs the engine to reduce the use of thread-local variables. Please see *Section 15.4.23, “Engine Settings related to Execution of Statements”* for more information.

20.2.28. Context Partition Related Information

As the engine locks on the level of context partition, high concurrency under threading can be achieved by using context partitions.

Generally context partitions require more memory then the more fine-grained grouping that can be achieved by `group by` or `std:groupwin`.

20.2.29. Prefer Constant Variables over Non-Constant Variables

The create-variable syntax as well as the APIs can identify a variable as a constant value. When a variable's value is not intended to change it is best to declare the variable as constant.

For example, consider the following two EPL statements that each declares a variable. The first statement declares a constant variable and the second statement declares a non-constant variable:

```
// declare a constant variable
create constant variable CONST_DEPARTMENT = 'PURCHASING'
```

```
// declare a non-constant variable
create variable VAR_DEPARTMENT = 'SALES'
```

When your application creates a statement that has filters for events according to variable values, the engine internally inspects such expressions and performs filter optimizations for constant variables that are more effective in evaluation.

For example, consider the following two EPL statements that each look for events related to persons that belong to a given department:

```
// perfer the constant
select * from PersonEvent(department=CONST_DEPARTMENT)
```

```
// less efficient
select * from PersonEvent(department=VAR_DEPARTMENT)
```

The engine can more efficiently evaluate the expression using a variable declared as constant. The same observation can be made for subquery and join query planning.

20.2.30. Prefer Object-array Events

Object-array events offer the best read access performance for access to event property values. In addition, object-array events use much less memory then Map-type events. They also offer the best write access performance.

A comparison of different event representations is in *Section 2.13, “Comparing Event Representations”*.

First, we recommend that your application sends object-array events into the engine, instead of Map-type events. See *Section 2.7, “Object-array (Object[]) Events”* for more information.

Second, we recommend that your application sets the engine-wide configuration of the default event representation to object array, as described in *Section 15.4.11.1, “Default Event Representation”*. Alternatively you can use the `@EventRepresentation(array=true)` annotation with individual statements.

20.2.31. Composite or Compound Keys

If your uniqueness, grouping, sorting or partitioning keys are composite keys or compound keys, this section may apply. A composite key is a key that consists of 2 or more properties or expressions.

In the example below the `firstName` and `lastName` expressions are part of a composite key:

```
... group by firstName, lastName
...std:unique(firstName, lastName)...
...order by firstName, lastName
```

Note

The example above is not a comprehensive discussion where composite or compound keys may be used in EPL. Other places where composite keys may apply are patterns, partitioned contexts and grouped data windows (we may have missed one).

You application could change the EPL to instead refer to a single value `fullName`:

```
... group by fullName
...std:unique(fullName)...
...order by fullName
```

The advantage in using a single expression as the uniqueness, grouping and sorting key is that the engine does not need to compute multiple expressions and retain a separate data structure in memory that represents the composite key, resulting in reduced memory use and increased throughput.

20.2.32. Notes on Query Planning

Query planning takes place for subqueries, joins (any type), named window on-actions (on-select, on-merge, on-insert, on-update, on-select) and fire-and-forget queries. Query planning affects query execution speed. Enable query plan logging to output query plan information.

For query planning, the engine draws information from:

1. The `where`-clauses, if any are specified. `Where`-clauses correlate streams, patterns, named windows etc. with more streams, patterns and named windows and are thus the main source of information for query planning.
2. The data window(s) declared on streams and named windows. The `std:unique` and the `std:firstunique` data window instruct the engine to retain the last event per unique criteria.
3. For named windows, the explicit indexes created via `create unique index` or `create index`.
4. For named windows, the previously created implicit indexes. The engine can create implicit indexes automatically if explicit indexes do not match correlation requirements.
5. Any hints specified for the statement in question and including hints specified during the creation of named windows with `create window`.

The engine prefers unique indexes over non-unique indexes.

20.2.33. Query Planning Index Hints

Currently index hints are only supported for the following types of statements:

1. Named window on-action statements (on-select, on-merge, on-insert, on-update, on-select).
2. Statements that have subselects against named windows that have index sharing enabled (the default is disabled).
3. Fire-and-forget queries.

For the above statements, you may dictate to the engine which explicit index (created via `create index` syntax) to use.

Specify the name of the explicit index in parentheses following `@Hint` and the `index` literal.

The following example instructs the engine to use the `UserProfileIndex` if possible:

```
@Hint('index(UserProfileIndex)')
```

Add the literal `bust` to instruct the engine to use the index, or if the engine cannot use the index fail query planning with an exception and therefore fail statement creation.

The following example instructs the engine to use the `UserProfileIndex` if possible or fail with an exception if the index cannot be used:

```
@Hint('index(UserProfileIndex, bust)')
```

Multiple indexes can be listed separated by comma (`,`).

The next example instructs the engine to consider the `UserProfileIndex` and the `SessionIndex` or fail with an exception if either index cannot be used:

```
@Hint('index(UserProfileIndex, SessionIndex, bust)')
```

The literal `explicit` can be added to instruct the engine to use only explicitly created indexes.

The final example instructs the engine to consider any explicitly create index or fail with an exception if any of the explicitly created indexes cannot be used:

```
@Hint('index(explicit, bust)')
```

20.2.34. Measuring Throughput

We recommend using `System.nanoTime()` to measure elapsed time when processing a batch of, for example, 1000 events.

Note that `System.nanoTime()` provides nanosecond precision, but not necessarily nanosecond resolution.

Therefore don't try to measure the time spent by the engine processing a single event: The resolution of `System.nanoTime()` is not sufficient. Also, there are reports that `System.nanoTime()` can be actually go "backwards" and may not always behave as expected under threading. Please check your JVM platform documentation.

In the default configuration, the best way to measure performance is to take nano time, send a large number of events, for example 10.000 events, and take nano time again reporting on the difference between the two numbers.

If your configuration has inbound threading or other threading options set, you should either monitor the queue depth to determine performance, or disable threading options when measuring performance, or have your application use multiple threads to send events instead.

20.2.35. Do not create the same EPL Statement X times

It is vastly more efficient to create an EPL statement once and attach multiple listeners, then to create the same EPL statement X times.

Since your goal will be to make all test code as realistic, real-world and production-like as possible, we recommend against production code or test code creating the same EPL statement multiple times. Instead consider creating the same EPL statement once and attaching multiple listeners. Certain important optimizations that the engine can perform when EPL statements realistically differ, may not take place. The engine also does not try to detect duplicate EPL statements, since that can easily be done by your application using public APIs.

20.2.36. Comparing Single-Threaded and Multi-Threaded Performance

The Java Virtual Machine optimizes locks such that the time to obtain a read lock, for example, differs widely between single-threaded and multi-threaded applications. We compared code that obtains an unfair `ReentrantReadWriteLock` read lock 100 million times, without any writer. We measured 3 seconds for a single-threaded application and 15 seconds for an application with 2 threads. It can therefore not be expected that scaling from single-threaded to 2 threads will always double performance. There is a base cost for multiple threads to coordinate.

20.2.37. Incremental Versus Recomputed Aggregation for Named Window Events

Whether aggregations of named window rows are computed incrementally or are recomputed from scratch depends on the type of query.

When the engine computes aggregation values incrementally, meaning it continuously updates the aggregation value as events enter and leave a named window, it means that the engine internally subscribes to named window updates and applies these updates as they occur. For some applications this is the desired behavior.

For some applications re-computing aggregation values from scratch when a certain condition occurs, for example when a triggering event arrives or time passes, is beneficial. Re-computing an aggregation can be less expensive if the number of rows to consider is small and/or when the triggering event or time condition triggers infrequently.

The next paragraph assumes that a named window has been created to hold some historical financial data per symbol and minute:

```
create window HistoricalWindow.win:keepall() as (symbol string, int minute,
 double price)
```

```
insert into HistoricalWindow select symbol, minute, price from HistoricalTick
```

For statements that simply select from a named window (excludes on-select) the engine computes aggregation values incrementally, continuously updating the aggregation, as events enter and leave the named window.

For example, the below statement updates the total price incrementally as events enter and leave the named window. If events in the named window already exist at the time the statement gets created, the total price gets pre-computed once when the statement gets created and incrementally updated when events enter and leave the named window:

```
select sum(price) from HistoricalWindow(symbol='GE')
```

The same is true for uncorrelated subqueries. For statements that sub-select from a named window, the engine computes aggregation values incrementally, continuously updating the aggregation, as events enter and leave the named window. This is only true for uncorrelated subqueries that don't have a where-clause.

For example, the below statement updates the total price incrementally as events enter and leave the named window. If events in the named window already exist at the time the statement gets created, the total price gets pre-computed once when the statement gets created and incrementally updated when events enter and leave the named window:

```
// Output GE symbol total price, incrementally computed
// Outputs every 15 minutes on the hour.
select (sum(price) from HistoricalWindow(symbol='GE'))
from pattern [every timer:at(0, 15, 30, 45), *, *, *, *, 0)]
```

If instead your application uses `on-select` or a correlated subquery, the engine recomputes aggregation values from scratch every time the triggering event fires.

For example, the below statement does not incrementally compute the total price (use a plain select or subselect as above instead). Instead the engine computes the total price from scratch based on the where-clause and matching rows:

```
// Output GE symbol total price (recomputed from scratch) every 15 minutes on
 the hour
on pattern [every timer:at(0, 15, 30, 45), *, *, *, *, 0)]
select sum(price) from HistoricalWindow where symbol='GE'
```

Unidirectional joins against named windows also do not incrementally compute aggregation values.

Joins and outer joins, that are not unidirectional, compute aggregation values incrementally.

20.2.38. When Does Memory Get Released

Java Virtual Machines (JVMs) release memory only when a garbage collection occurs. Depending on your JVM settings a garbage collection can occur frequently or infrequently and may consider all or only parts of heap memory.

Esper is optimized towards latency and throughput. Esper does not force garbage collection or interfere with garbage collection. For performance-sensitive code areas, Esper utilizes thread-local buffers such as arrays or ringbuffers that can retain small amounts of recently processed state. Esper does not try to clean such buffers after every event for performance reasons. It does clean such buffers when destroying the engine and stopping or destroying statements. It is therefore normal to see a small non-increasing amount of memory to be retained after processing events that the garbage collector may not free immediately.

20.3. Using the performance kit

20.3.1. 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 *Performance Wiki* [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 *Performance Wiki* [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 2.1) :

```
# classpath on Unix/Linux (on one single line)
etc:build:lib/esper-4.10.0.jar:lib/commons-logging-1.1.1.jar:lib/cglib-
nodep-2.2.jar
   :lib/antlr-runtime-3.2.jar:lib/log4j-1.2.16.jar
@rem  classpath on Windows (on one single line)
etc;build;lib\esper-4.10.0.jar;lib\commons-logging-1.1.1.jar;lib\cglib-
nodep-2.2.jar
   ;lib\antlr-runtime-3.2.jar;lib\log4j-1.2.16.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. By default the benchmark registers a subscriber to the statement(s). Use `-Desper.benchmark.ul` to use an UpdateListener instead. Note that the subscriber contains suitable update(..) methods for the default proposed statement in the `etc/statements.properties` file but might not be suitable if you change statements due to the strong binding with statement results. Refer to *Section 14.3.2, “Receiving Statement Results”*.

6. 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.

7. 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."
```

20.3.2. 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 21. References

21.1. 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
-----------------------------------------------
 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
                                          [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
  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.5.6. 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
                                           [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
  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.6.6. 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]
```

Appendix B. Reserved Keywords

The words in the following table are explicitly reserved in EPL, however certain keywords are allowed as event property names in expressions and as column names in the rename syntax of the `select` clause.

Most of the words in the table are forbidden by standard SQL as well. A few are reserved because EPL needs them.

Names of built-in functions and certain auxiliary keywords are permitted as identifiers for use either as event property names in expressions and for the column rename syntax. The second column in the table below indicates which keywords are acceptable. For example, `count` is acceptable.

An example of permitted use is:

```
select last, count(*) as count from MyEvent
```

This example shows an incorrect use of a reserved keyword:

```
// incorrect
select insert from MyEvent
```

The table of explicitly reserved keywords and permitted keywords:

Table B.1. Reserved Keywords

Keyword	Property Name and Rename Syntax
after	-
all	-
and	-
as	-
at	yes
asc	-
avedev	yes
avg	yes
between	-
by	-
case	-
cast	yes

Keyword	Property Name and Rename Syntax
coalesce	yes
context	-
count	yes
create	-
current_timestamp	-
dataflow	-
day	-
days	-
delete	-
define	yes
desc	-
distinct	-
else	-
end	-
escape	yes
events	yes
every	yes
exists	-
expression	-
false	yes
first	yes
for	yes
from	-
full	yes
group	-
having	-
hour	-
hours	-
in	-
initiated	-
inner	-
insert	-
instanceof	yes
into	-

Keyword	Property Name and Rename Syntax
irstream	-
is	-
istream	-
join	yes
last	yes
lastweekday	yes
left	yes
limit	-
like	-
max	yes
match_recognize	-
matched	-
matches	-
median	yes
measures	yes
merge	-
metadatasql	yes
min	yes
minute	yes
minutes	yes
msec	yes
millisecond	yes
milliseconds	yes
new	-
not	-
null	-
offset	-
on	-
or	-
order	-
outer	yes
output	-
partition	-
pattern	yes

Keyword	Property Name and Rename Syntax
prev	yes
prior	yes
regexp	-
retain-union	yes
retain-intersection	yes
right	yes
rstream	-
sec	-
second	-
seconds	-
select	-
set	-
some	-
snapshot	yes
sql	yes
start	-
stddev	yes
sum	yes
terminated	-
then	-
true	-
unidirectional	yes
until	yes
update	-
using	yes
variable	yes
weekday	yes
when	-
where	-
while	-
window	yes

Index