Oracle® Fusion Middleware
EPL Language Reference for Oracle Event Processing
11g Release 1 (11.1.1.9)
E14304-09

February 2015

Documentation for developers that provides a reference to Event Processing Language (EPL), a language for querying streaming data in Oracle Event Processing applications. Note that EPL is deprecated; new development should use Oracle Continuous Query Language (Oracle CQL).
# Contents

**Preface** ........................................................................................................................................................................... ix  
Audience ......................................................................................................................................................................................... ix  
Documentation Accessibility ..................................................................................................................................................................... ix  
Related Documents ............................................................................................................................................................................... x  
Conventions ..................................................................................................................................................................................... x  

1 Overview of the Event Processing Language (EPL)  

Overview of the EPL Language ................................................................................................................................................................. 1-1  
Event Representation .................................................................................................................................................................................. 1-2  
  Event Objects ......................................................................................................................................................................................... 1-2  
  Plain Old Java Object Events ................................................................................................................................................................. 1-3  
  Map Events ............................................................................................................................................................................................. 1-3  
  Event Properties .................................................................................................................................................................................... 1-3  
  Dynamic Event Properties ..................................................................................................................................................................... 1-5  
  Handling Other Event Properties Using a User-Defined Function ........................................................................................................ 1-6  
  Event Sinks ........................................................................................................................................................................................... 1-7  
Processing Model ....................................................................................................................................................................................... 1-8  
  Event Streams ......................................................................................................................................................................................... 1-8  
  Sliding Windows .................................................................................................................................................................................... 1-9  
    Row-Based Sliding Windows ............................................................................................................................................................... 1-9  
    Time-Based Sliding Windows ............................................................................................................................................................. 1-10  
  Batched Windows .................................................................................................................................................................................. 1-11  
    Time-Based Batched Windows ........................................................................................................................................................... 1-11  
    Row-Based Batched Windows ............................................................................................................................................................ 1-12  
  Subqueries and WHERE Clauses ......................................................................................................................................................... 1-13  
  Aggregation .......................................................................................................................................................................................... 1-14  
Use Cases ............................................................................................................................................................................................. 1-15  
  Computing Rates per Feed ................................................................................................................................................................. 1-15  
  Computing Highest Priced Stocks .......................................................................................................................................................... 1-15  
  Segmenting Location Data ................................................................................................................................................................. 1-15  
  Detecting Rapid Fall-off ................................................................................................................................................................. 1-16  
  Finding Network Anomalies ................................................................................................................................................................. 1-16  
  Detecting Absence of Event ............................................................................................................................................................ 1-16  
  Summarizing Terminal Activity Data .................................................................................................................................................. 1-17  
  Reading Sensor Data ............................................................................................................................................................................. 1-17
Combining Transaction Events ................................................................. 1-17
Monitoring Real-time Performance ....................................................... 1-17
Finding Dropped Transaction Events ................................................... 1-18

2 EPL Reference: Clauses

Overview of the Clauses You Can Use in an EPL Statement .................. 2-1
SELECT ........................................................................................................ 2-2
  Choosing Specific Event Properties ...................................................... 2-2
  Using Expressions ................................................................................ 2-2
  Aliasing Event Properties .................................................................. 2-2
  Choosing All Event Properties .......................................................... 2-2
  Selecting New and Old Events With ISTREAM and RSTREAM Keywords 2-3
FROM .......................................................................................................... 2-3
  Inner Joins ............................................................................................ 2-5
  Outer Joins ........................................................................................... 2-5
  Subquery Expressions ......................................................................... 2-6
  Parameterized SQL Queries ............................................................... 2-6
RETAIN ........................................................................................................ 2-8
  Keeping All Events .............................................................................. 2-8
  Specifying Window Size .................................................................. 2-8
  Specifying Batched Versus Sliding Windows ..................................... 2-9
  Specifying Time Interval .................................................................. 2-9
    BASED ON Clause ........................................................................... 2-10
  Specifying Property Name ................................................................ 2-10
  Using PARTITION BY Clause to Partition Window ............................ 2-10
  Using WITH Clause to Keep Largest/Smallest/Unique Values .......... 2-11
MATCHING .................................................................................................. 2-11
  FOLLOWED BY Operator .................................................................. 2-12
  AND Operator ..................................................................................... 2-12
  OR Operator ....................................................................................... 2-12
  NOT Operator .................................................................................... 2-13
  EVERY Operator ............................................................................... 2-13
  WITHIN Operator ............................................................................ 2-14
  Event Structure for Matched Pattern ............................................... 2-15
WHERE ....................................................................................................... 2-15
GROUP BY ................................................................................................ 2-15
HAVING ..................................................................................................... 2-16
  Interaction With MATCHING, WHERE and GROUP BY Clauses ....... 2-17
ORDER BY ................................................................................................ 2-18
OUTPUT ................................................................................................... 2-18
  Interaction With GROUP BY and HAVING Clauses ............................ 2-19
INSERT INTO ............................................................................................ 2-19
Simple and Correlated Subqueries .......................................................... 2-21
Parameterized Queries ............................................................................. 2-22
  General Usage .................................................................................. 2-22
  Using a Parameterized EPL Statement in the Processor Configuration File 2-23
  Programmatically Using a Prepared EPL Statement ......................... 2-24
3 EPL Reference: Operators

Overview of EPL Operators ............................................................................................................... 3-1
Arithmetic Operators .......................................................................................................................... 3-1
Logical and Comparison Operators ................................................................................................. 3-1
Concatenation Operators .................................................................................................................. 3-2
Binary Operators ............................................................................................................................. 3-2
Array Definition Operator ................................................................................................................. 3-2
List and Range Operators ............................................................................................................... 3-3
  IN Operator ........................................................................................................................................ 3-3
  BETWEEN Operator ......................................................................................................................... 3-3
String Operators ............................................................................................................................... 3-4
  LIKE Operator ................................................................................................................................... 3-4
  REGEXP Operator ........................................................................................................................... 3-5
Temporal Operators .......................................................................................................................... 3-5
  FOLLOWED BY Operator ................................................................................................................... 3-5
  WITHIN Operator ........................................................................................................................... 3-5
  EVERY Operator .............................................................................................................................. 3-6

4 EPL Reference: Functions

Single-row Functions ......................................................................................................................... 4-1
  The MIN and MAX Functions ........................................................................................................... 4-2
  The COALESCE Function ................................................................................................................. 4-3
  The CASE Control Flow Function ..................................................................................................... 4-3
  The PREV Function ......................................................................................................................... 4-3
    Previous Event Per Group ............................................................................................................... 4-4
    Restrictions .................................................................................................................................. 4-4
  The PRIOR Function ......................................................................................................................... 4-4
    Comparison to the PREV Function ................................................................................................... 4-5
  The INSTANCEOF Function .............................................................................................................. 4-5
  The CAST Function .......................................................................................................................... 4-6
  The EXISTS Function ...................................................................................................................... 4-6
Aggregate functions ........................................................................................................................... 4-7
User-Defined functions ...................................................................................................................... 4-8

5 Programmatic Interface to EPL

Java Programming Interfaces ........................................................................................................... 5-1
List of Examples

1–1 Enum Datatype ProcessStatus ........................................................................................................ 1-6
1–2 Event Using Enum Datatype ProcessStatus ................................................................................... 1-6
1–3 User-Defined Function to Evaluate Enum Datatype ...................................................................... 1-7
1–4 Registering the User-Defined Function in Application Assembly File ........................................ 1-7
1–5 Using the User-Defined Function to Evaluate Enum Datatype in an EPL Query ........... 1-7
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1–1</td>
<td>Example Event Stream</td>
<td>1-9</td>
</tr>
<tr>
<td>1–2</td>
<td>Example Row-Based Sliding Window</td>
<td>1-10</td>
</tr>
<tr>
<td>1–3</td>
<td>Example Time-Based Sliding Window</td>
<td>1-11</td>
</tr>
<tr>
<td>1–4</td>
<td>Example Time-Based Batched Window</td>
<td>1-12</td>
</tr>
<tr>
<td>1–5</td>
<td>Example Subquery and WHERE Clause</td>
<td>1-13</td>
</tr>
<tr>
<td>1–6</td>
<td>Events Arriving in a Row-Based Sliding Window</td>
<td>1-14</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1–1</td>
<td>Event Representation</td>
<td>1-2</td>
</tr>
<tr>
<td>1–2</td>
<td>Event Properties</td>
<td>1-3</td>
</tr>
<tr>
<td>1–3</td>
<td>Syntax of Dynamic Properties</td>
<td>1-5</td>
</tr>
<tr>
<td>2–1</td>
<td>Example EVERY Operators</td>
<td>2-13</td>
</tr>
<tr>
<td>3–1</td>
<td>Arithmetic Operators</td>
<td>3-1</td>
</tr>
<tr>
<td>3–2</td>
<td>Logical and Comparison Operators</td>
<td>3-2</td>
</tr>
<tr>
<td>3–3</td>
<td>Concatenation Operators</td>
<td>3-2</td>
</tr>
<tr>
<td>3–4</td>
<td>Binary Operators</td>
<td>3-2</td>
</tr>
<tr>
<td>4–1</td>
<td>Built-In Single-Row Functions</td>
<td>4-1</td>
</tr>
<tr>
<td>4–2</td>
<td>Aggregate Functions</td>
<td>4-7</td>
</tr>
</tbody>
</table>
Preface

This reference contains a complete description of the Oracle Event Processing Language (Oracle EPL), a query language based on SQL with added constructs that support streaming data. Using Oracle EPL, you can express queries on data streams to perform event processing using Oracle Event Processing.

Note: Oracle EPL is superseded by Oracle Continuous Query Language (Oracle CQL). For more information, see the Oracle Fusion Middleware CQL Language Reference for Oracle Event Processing.

Oracle Event Processing (formerly known as the WebLogic Event Server) is a Java server for the development of high-performance event driven applications. It is a lightweight Java application container based on Equinox OSGi, with shared services, including the Oracle Event Processing Service Engine, which provides a rich, declarative environment based on Oracle Continuous Query Language (Oracle CQL) - a query language based on SQL with added constructs that support streaming data - to improve the efficiency and effectiveness of managing business operations. Oracle Event Processing supports ultra-high throughput and microsecond latency using JRockit Real Time and provides Oracle Event Processing Visualizer and Oracle Event Processing IDE for Eclipse developer tooling for a complete real time end-to-end Java Event-Driven Architecture (EDA) development platform.

Audience

This document is intended for all users of Oracle EPL.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.
Related Documents

For more information, see the following:

- Oracle Fusion Middleware Getting Started Guide for Oracle Event Processing
- Oracle Fusion Middleware Administrator’s Guide for Oracle Event Processing
- Oracle Fusion Middleware Developer’s Guide for Oracle Event Processing for Eclipse
- Oracle Fusion Middleware Visualizer User’s Guide for Oracle Event Processing
- Oracle Fusion Middleware Java API Reference for Oracle Event Processing
- Oracle Fusion Middleware CQL Language Reference for Oracle Event Processing
- Oracle Event Processing Samples: http://www.oracle.com/technologies/soa/complex-event-processing.html

Conventions

The following text conventions are used in this document:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>boldface</td>
<td>Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.</td>
</tr>
<tr>
<td>italic</td>
<td>Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.</td>
</tr>
<tr>
<td>monospace</td>
<td>Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.</td>
</tr>
</tbody>
</table>
This chapter introduces Oracle Event Processing Language (EPL), a query language for streaming data. Using EPL, you can express queries on data streams with Oracle Event Processing. Note that EPL is deprecated; new development should use Oracle Continuous Query Language (Oracle CQL).

This chapter includes the following sections:

- Overview of the EPL Language
- Event Representation
- Processing Model
- Use Cases

Overview of the EPL Language

The event processor module can be broken down into the following functional components: event representation, processing model, programmatic interface, and language specification.

Events are represented as POJOs following the JavaBeans conventions. Event properties are exposed through getter methods on the POJO. When possible, the results from EPL statement execution are also returned as POJOs. However, there are times when un-typed events are returned such as when event streams are joined. In this case, an instance of the Map collection interface is returned. For more information, see Section, "Event Representation".

The EPL processing model is continuous: results are output as soon as incoming events are received that meet the constraints of the statement. Two types of events are generated during output: insert events for new events entering the output window and remove events for old events exiting the output window. Listeners may be attached and notified when either or both type of events occur. For more information, see Section, "Processing Model".

Incoming events may be processed through either sliding or batched windows. Sliding windows process events by gradually moving the window over the data in single increments, while batched windows process events by moving the window over data in discrete chunks. The window size may be defined by the maximum number of events contained or by the maximum amount of time to keep an event.

The EPL programmatic interfaces allow statements to be individually compiled or loaded in bulk through a URL. Statements may be iterated over, retrieved, started and
stopped. Listeners may be attached to statements and notified when either insert and/or remove events occur.

The Event Processor Language 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 through sub-queries, and aggregation through grouping may 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 `RETAINT, MATCHING, and OUTPUT` clauses are also available to provide the missing SQL language constructs specific to event processing.

The `RETAINT` clause constraints the amount of data over which the query is run, essentially defining a virtual window over the stream data. Unlike relational database systems in which tables bound the extents of the data, event processing systems must provide an alternative, more dynamic means of limiting the queried data.

The `MATCHING` clause detects sequences of events matching a specific pattern. Temporal and logical operators such as `AND, OR, and FOLLOWED BY` enable both occurrence of and absence of events to be detected through arbitrarily complex expressions.

The `OUTPUT` clause throttles results of statement execution to prevent overloading downstream processors. Either all or a subset of the first or last resulting events may be passed on in either time or row-based batches.

A series of use cases is presented in the last section to illustrate the language features under realistic scenarios.

### Event Representation

Using EPL, you represent an event as an event object. For more information, see:

- Section, "Event Objects"
- Section, "Plain Old Java Object Events"
- Section, "Map Events"

All event objects include properties. For more information, see:

- Section, "Event Properties"
- Section, "Dynamic Event Properties"
- Section, "Handling Other Event Properties Using a User-Defined Function"

Events are consumed by event sinks. For more information, see Section, "Event Sinks."

### Event 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 object. An event is represented by either a POJO or a `com.bea.wlews.cep.event.MapEventObject` that extends the `java.util.Map` interface.

<table>
<thead>
<tr>
<th>Java Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java.lang.Object</code></td>
<td>Any Java POJO with getter methods following JavaBeans conventions.</td>
</tr>
</tbody>
</table>
Plain Old Java Object Events

Plain old Java object (POJO) 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 JavaBeans specification; however for the EPL engine to obtain event properties, the required JavaBeans getter methods must be present.

EPL supports JavaBeans-style event classes that extend a super class or implement one or more interfaces. Also, 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 EPL engine accepts events that are mutable as well.

Map Events

Events can also be represented by objects that implement the com.bea.wlevs.ede.api.MapEventObject interface that extends the java.util.Map interface. Event properties of Map events are the values of each entry accessible through the get method exposed by the java.util.Map 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 EPL statements. Values can be of any type. POJOs may also appear as values in a Map.

Event Properties

EPL expressions can include 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 JavaBeans objects graphs, XML structures and Map events.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>A property that has a single value that may be retrieved. The property type may be a primitive type (such as int, or java.lang.String) or another complex type.</td>
<td>name</td>
<td>sensorId</td>
</tr>
<tr>
<td>Nested</td>
<td>A nested property is a property that lives within another property of an event. Events represented as a Map may only nest other POJO events and not other Map events.</td>
<td>name.nestedname</td>
<td>sensor.value</td>
</tr>
<tr>
<td>Indexed</td>
<td>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). Events represented as a Map do not support Indexed properties.</td>
<td>name[index]</td>
<td>sensor[0]</td>
</tr>
</tbody>
</table>
Assume there is an EmployeeEvent event class as shown below. The mapped and indexed properties in this example return Java objects but could also return Java language primitive types (such as int or String). The Address object and Employee objects can themselves have properties that are nested within them, such as a street-Name in the Address object or a name of the employee in the Employee object.

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

Simple event properties require a getter-method that returns the property value. In the preceding 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.
- A method returns an array-type such as the getAllSubordinates getter method, which returns an array of Employee.

In an EPL statement, indexed properties are accessed via the `property[index]` syntax.

Mapped event properties require a getter-method that takes a String type key value and returns a 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 statement, nested properties are accessed via the `property.nestedProperty` syntax.

All EPL statements allow the use of indexed, mapped and nested properties (or a combination of these) at any place where one or more event property names are expected. The example below shows different combinations of indexed, mapped and nested properties.

```sql
address('home').streetName
subordinate[0].name='anotherName'
allSubordinates[1].name
subordinate[0].address('home').streetName
```

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.

```sql
SELECT firstName, address('work'), subordinate[0].name, subordinate[1].name
FROM EmployeeEvent RETAIN ALL
WHERE address('work').streetName = 'Park Ave'
```
Dynamic Event Properties

Dynamic (or unchecked) properties are event properties that need not be known at statement compilation time. Oracle Event Processing resolves these dynamic properties at runtime.

The idea behind dynamic properties is that for a given underlying event representation, the properties are not necessarily known in advance. An underlying event may have additional properties that are not known at statement compilation time, and these properties might need to be queried on using an EPL statement. 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. The following table describes the types of dynamic event properties and the syntax used to identify them.

<table>
<thead>
<tr>
<th>Event Property Type</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Simple</td>
<td>name?</td>
</tr>
<tr>
<td>Dynamic Indexed</td>
<td>name[index]?</td>
</tr>
<tr>
<td>Dynamic Mapped</td>
<td>name('key')?</td>
</tr>
<tr>
<td>Dynamic Nested</td>
<td>name?.nestedPropertyName</td>
</tr>
</tbody>
</table>

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

For 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`; which one is known only at runtime. Further assume that both the `Service` and `Product` classes provide a property named `price`. Using a dynamic property, you can specify a query that obtains the `price` property from `either` object (`Service` or `Product`), as shown in the following example:

```sql
SELECT item.price?
FROM OrderEvent RETAIN ALL EVENTS
```

As a second example, assume that the `Service` class contains a `serviceName` property that the `Product` class does not contain. At runtime, the following query returns the value of the `serviceName` property for `Service` objects; if, however, the query is run against a `Product` object, the query returns a null value because `Products` do not contain the `serviceName` property:

```sql
SELECT item.serviceName?
FROM OrderEvent RETAIN ALL EVENTS
```

Now consider the case where `OrderEvent` has multiple implementation classes, only some of which have a `timestamp` property. The following query returns the `timestamp` property of those implementations of the `OrderEvent` interface that feature the property:

```sql
SELECT timestamp?
FROM OrderEvent RETAIN ALL EVENTS
```
The preceding query returns a single column named `timestamp?` of type `java.lang.Object`.

When you nest dynamic properties, all properties under the dynamic property are also considered dynamic properties. In the next example, the query asks for the `direction` property of the object returned by the `detail` dynamic property:

```epl
SELECT detail?.direction
FROM OrderEvent RETAIN ALL EVENTS
```

The preceding query is equivalent to the following:

```epl
SELECT detail?.direction?
FROM OrderEvent RETAIN ALL EVENTS
```

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

- The `CAST` function casts the value of a dynamic property (or the value of an expression) to a given type. See Section, "The CAST Function."
- 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. See Section, "The EXISTS Function."
- The `INSTANCEOF` function checks whether the value of a dynamic property (or the value of an expression) is of any of the given types. See Section, "The INSTANCEOF Function."

### Handling Other Event Properties Using a User-Defined Function

If your event uses a datatype that EPL does not support, you can create a user-defined function to evaluate that datatype in an EPL query.

Consider the `enum` datatype that Example 1–1 shows. The event that Example 1–2 shows uses this `enum` datatype. EPL does not support `enum` datatypes.

**Example 1–1  Enum Datatype ProcessStatus**

```java
package com.oracle.app;

public enum ProcessStatus {
    OPEN(1),
    CLOSED(0)
}
```

**Example 1–2  Event Using Enum Datatype ProcessStatus**

```java
package com.oracle.app;

import com.oracle.capp.ProcessStatus;

public class ServiceOrder {
    private String serviceOrderId;
    private String electronicSerialNumber;
    private ProcessStatus status;
    ...
}
```

By creating the user-defined function that Example 1–3 shows and registering the function in your application assembly file as Example 1–4 shows, you can evaluate this `enum` datatype in an EPL query as Example 1–5 shows.
Example 1–3  User-Defined Function to Evaluate Enum Datatype

```java
package com.oracle.app;

import com.oracle.capp.ProcessStatus;
public class CheckIfStatusClosed {
    public boolean execute(Object[] args) {
        ProcessStatus arg0 = (ProcessStatus)args[0];
        if (arg0 == ProcessStatus.OPEN)
            return Boolean.FALSE;
        else
            return Boolean.TRUE;
    }
}
```

Example 1–4  Registering the User-Defined Function in Application Assembly File

```xml
<wlevs:processor id="testProcessor">
    <wlevs:listener ref="providerCache"/>
    <wlevs:listener ref="outputCache"/>
    <wlevs:cache-source ref="testCache"/>
    <wlevs:function function-name="statusClosed" exec-method="execute" />
        <bean class="com.oracle.app.CheckIfStatusClosed"/>
    </wlevs:function>
</wlevs:processor>
```

Example 1–5  Using the User-Defined Function to Evaluate Enum Datatype in an EPL Query

```xml
<query id="rule-04"><![CDATA[
    SELECT
        meter.electronicSerialNumber,
        meter.exceptionKind
    FROM
        MeterLogEvent AS meter,
        ServiceOrder AS svco
    WHERE
        meter.electronicSerialNumber = svco.electronicSerialNumber and
        svco.serviceOrderId IS NULL OR statusClosed(svco.status)
]]></query>
```

For more information, see Chapter 4, "EPL Reference: Functions".

Event Sinks

Event sinks provide a means of receiving programmatic notifications when events occur that meet the criteria specified in an EPL statement. Sinks may be notified when either:

- New events occur that meet the criteria specified in an EPL statement. These are termed ISTREAM events.
- Old events that previously met the criteria specified in an EPL statement are pushed out of the output window due to their expiration or due to new incoming events occurring that take their place. These are termed RSTREAM events.

Detailed examples illustrating when each of these notifications occur are provided in Section, "Processing Model."

To receive ISTREAM events, use the com.bea.wlevs.ede.api.EventSink interface. Your implementation must provide a single onEvent method that the engine invokes when results become available. With this interface, only the new events are sent to the listener.
public interface EventSink extends EventListener {
    void onEvent(List<Object> newEvents)
        throws RejectEventException;
}

The engine provides statement results to event sinks as a list of POJO or MapEventObject instances. For wildcard selects, the result will match the original event object type that was sent into the engine. For joins and select clauses with expressions, the resulting object will implement the com.bea.wlevs.ede.api.MapEventObject interface.

Processing Model

The EPL processing model is based on event streams as described in Section, "Event Streams."

You express queries that operate on event streams as described in:

- Section, "Sliding Windows"
- Section, "Batched Windows"
- Section, "Subqueries and WHERE Clauses"
- Section, "Aggregation"

Event Streams

The EPL processing model is continuous: Listeners to statements receive updated data as soon as the engine processes events for that statement, according to the statement's choice of event streams, retain clause restrictions, filters and output rates.

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 RETAIN ALL
```

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 update listeners as new events.

The diagram below shows a series of Withdrawal events 1 to 6 arriving over time. For this diagram as well as the others in this section, the number in parenthesis is the value of the amount property in the Withdrawal event.
The example statement above results in only new events and no old events posted by the engine to the statement’s listeners because no RETAIN clause is specified.

**Sliding Windows**

There are two types of sliding windows: row-based and time-based. Each of these is discussed in the following sections.

**Row-Based Sliding Windows**

A row-based sliding 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 RETAIN 5 EVENTS
```

The size of this statement’s window is five events. The engine enters all arriving Withdrawal events into the window. When the window is full, the oldest Withdrawal event is pushed out the window. The engine indicates to update 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 update 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.
As before, all arriving events are posted as new events to update 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 update 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.

EPL supports optional `ISTREAM` and `RSTREAM` keywords on `SELECT` clauses and on `INSERT INTO` clauses. These instruct the engine to only forward events that enter or leave data windows, or select only current or prior aggregation values, i.e. the insert stream or the remove stream.

**Time-Based Sliding Windows**

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

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 RETAIN 4 SECONDS
```

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 1–3  Example Time-Based Sliding 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.

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 than 1000. The statement to solve this problem is shown below.

```sql
SELECT account, AVG(amount)
FROM Withdrawal RETAIN 4 SECONDS
GROUP BY account
HAVING amount > 1000
```

Batched Windows

Both row-based and time-based windows may be batched. The next sections explain each of these concepts in turn.

Time-Based Batched Windows

The time-based batch window buffers events and releases them every specified time interval in one update. Time-based batch 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:

```sql
SELECT * FROM Withdrawal RETAIN BATCH OF 4 SECONDS
```

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 1–4 Example Time-Based Batched Window**

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.

**Row-Based Batched Windows**

A row-based window may be batched as well. For example, the following query would wait to receive five events prior to doing any processing:

```sql
SELECT * FROM Withdrawal RETAIN BATCH OF 5 EVENTS
```
Once five events were received, the query would run and again wait for a new set of five events prior to processing.

### Subqueries and WHERE Clauses

Filters to event streams appear in a subquery expression and allow filtering events out of a given stream before events enter a data window. This filtering occurs prior to the WHERE clause executing. When possible, filtering should be done in a subquery as opposed to the WHERE clause, since this will improve performance by reducing the amount of data seen by the rest of the EPL statement.

The statement below shows a subquery that selects Withdrawal events with an amount value of 200 or more.

```
SELECT * FROM (SELECT * FROM Withdrawal WHERE amount >= 200) RETAIN 5 EVENTS
```

With the subquery, any Withdrawal events that have an amount of less than 200 do not enter the window of the outer query and are therefore not passed to update listeners.

### Figure 1–5  Example Subquery and WHERE Clause

![Diagram showing subquery and WHERE clause example](image)

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 instead of a subquery.

```
SELECT * FROM Withdrawal RETAIN 5 EVENTS WHERE amount >= 200
```

The WHERE clause applies to both new events and old events. As the diagram below shows, arriving events enter the window regardless of the value of the "amount" property. 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 update listeners as old events.
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 BigWithdrawal
    SELECT * FROM Withdrawal RETAIN ALL WHERE Math.ceil(amount) >= 200
SELECT * FROM BigWithdrawal RETAIN ALL
```

### Aggregation

Statements that aggregate events via aggregations functions also post remove stream events as aggregated values change. Consider the following statement that alerts when two Withdrawal events have been received:

```
SELECT COUNT(*) AS mycount
FROM Withdrawal RETAIN ALL
HAVING COUNT(*) = 2
```

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

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

```
SELECT ISTREAM COUNT(*) AS mycount
```
Use Cases

The use cases below illustrate through examples usage of various language features.

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 batches together 1 second of events from the market data event stream source. 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:

```plaintext
FROM Withdrawal RETAIN ALL
HAVING COUNT(*) = 2
```

Computing Highest Priced Stocks

For computing the highest priced stocks, we define a sliding window that retains 100 events for each unique stock symbol where the block size of the trade is greater than 10. For example, if there are 5,000 stock symbols, then 5,000 x 100 or 5,000,000 events would be kept. Only `MarketTrade` events with a block size of greater than 10 will enter the window and only the 100 highest priced events will be retained.

The results will be grouped by stock symbol and ordered alphabetically with stock symbols having an average price of less than 100 being filtered from the output.

```plaintext
SELECT symbol, AVG(price)
FROM (SELECT * FROM MarketTrade WHERE blockSize > 10)
RETAIN 100 EVENTS WITH LARGEST price PARTITION BY symbol
GROUP BY symbol
HAVING AVG(price) >= 100
ORDER BY symbol
```

Segmenting Location Data

We detect the route a car is taking based on the car location event data that contains information about the location and direction of a car on a highway. We first segment the data by `carId` to isolate information about a particular car and subsequently segment by expressway, direction and segment to plot its direction. We are then able to calculate the speed of the car based on this information.

The first `PARTITION BY carId` groups car location events by car while the following `PARTITION BY expressway PARTITION BY direction` further segment the data by more detailed location and direction property values. The number of events retained, 4 in this query, applies to the maximum number kept for the last `PARTITION BY` clause. Thus at most 4 events will be kept for each distinct segment property value.

```plaintext
SELECT carId, expressway, direction,
    SUM(segment)/(MAX(timestamp)-MIN(timestamp)) AS speed
FROM CarLocationEvent
RETAIN 4 events
```
PARTITION BY carId PARTITION BY expressway PARTITION BY direction

**Detecting Rapid 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
RETAIN 10 seconds
GROUP BY feed
HAVING cnt < AVG(cnt) * 0.75
```

**Finding Network Anomalies**

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 ci.term
MATCHING ci:=Checkin FOLLOWED BY
  ( OutOfOrder {term.id=ci.term.id} AND NOT
    (Cancelled {term.id=ci.term.id} OR
    Completed {term.id=ci.term.id}) WITHIN 3 MINUTES )
```

Each self-service terminal can publish any of the four events below.

- **Checkin** - Indicates a customer started a check-in dialogue.
- **Cancelled** - Indicates a customer cancelled a check-in dialogue.
- **Completed** - Indicates a customer completed a check-in dialogue.
- **OutOfOrder** - Indicates the terminal detected a hardware problem

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 `term.id`. Because all events carry similar information, we model each event as a subtype to a base class `TerminalEvent`, which will provide the terminal information that all events share. This enables us to treat all terminal events polymorphically, which simplifies our queries by allowing us to treat derived event types just like their parent event types.

**Detecting Absence of Event**

Because `Status` events arrive in regular intervals of 60 seconds, we can make use of temporal pattern matching using the `MATCHING` clause to find events that did not arrive in time. We can use the `WITHIN` operator to keep a 65 second window to account for a possible delay in transmission or processing and the `NOT` operator to detect the absence of a `Status` event with a `term.id` equal to `T1`:

```
SELECT 'terminal 1 is offline'
MATCHING NOT Status(term.id = 'T1') WITHIN 65 SECONDS
```
OUTPUT FIRST EVERY 5 MINUTES

**Summarizing Terminal Activity 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 TerminalEvent
RETAIN 10 MINUTES
GROUP BY type
OUTPUT ALL EVERY 1 MINUTE
```

**Reading Sensor Data**

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.

```
SELECT ID AS sensorId, SUM(countTags) AS numTagsPerSensor
FROM AutoIdRFIDExample
RETAIN 60 SECONDS
WHERE Observation[0].Command = 'READ_PALLET_TAGS_ONLY'
GROUP BY ID
```

**Combining Transaction Events**

In this example we compose an EPL statement to detect combined events in which each component of the transaction is present. We restrict the event matching to the events that arrived within the last 30 minutes. This statement uses the `INSERT INTO` syntax to generate a **CombinedEvent** event stream.

```
INSERT INTO CombinedEvent(transactionId, customerId, supplierId, latencyAC, latencyBC, latencyAB)
SELECT C.transactionId, customerId, supplierId,
    C.timestamp - A.timestamp,
    C.timestamp - B.timestamp,
    B.timestamp - A.timestamp
FROM TxnEventA A, TxnEventB B, TxnEventC C
RETAIN 30 MINUTES
WHERE A.transactionId = B.transactionId AND
      B.transactionId = C.transactionId
```

**Monitoring Real-time Performance**

To derive the minimum, maximum and average total latency from the events (difference in time between A and C) over the past 30 minutes we can use the EPL below. In addition, in order to monitor the event server, a dashboard UI will subscribe to a subset of the events to measure system performance such as server and end-to-end latency. It is not feasible to expect a UI to monitor every event flowing through the system, so there must be a way of rate limiting the output to a subset of the events that can be handled by the monitoring application. Only the single last event or all events can be output.
SELECT MIN(latencyAC) as minLatencyAC,
    MAX(latencyAC) as maxLatencyAC,
    AVG(latencyAC) as avgLatencyAC
FROM CombinedEvent
RETAIN 30 MINUTES
GROUP BY customerId
OUTPUT LAST 50 EVERY 1 SECOND

Finding Dropped Transaction Events

An OUTER JOIN allows us to detect a transaction that did not make it through all three events. When TxnEventA or TxnEventB events leave their respective time windows consisting of the last 30 minutes of events, EPL filters out rows in which no EventC row was found.

```
SELECT *
FROM TxnEventA A
    FULL OUTER JOIN TxnEventC C ON A.transactionId = C.transactionId
    FULL OUTER JOIN TxnEventB B ON B.transactionId = C.transactionId
RETAIN 30 MINUTES
WHERE C.transactionId is null
```
This chapter provides a reference to clauses in Event Processing Language (EPL). Note that EPL is deprecated; new development should use Oracle Continuous Query Language (Oracle CQL).

This chapter includes the following sections:

- Overview of the Clauses You Can Use in an EPL Statement
- SELECT
- FROM
- RETAIN
- MATCHING
- WHERE
- GROUP BY
- HAVING
- ORDER BY
- OUTPUT
- INSERT INTO
- Simple and Correlated Subqueries
- Parameterized Queries

### Overview of the Clauses You Can Use in an EPL Statement

The top-level BNF for the event processing language (EPL) is as follows:

```
[ INSERT INTO insert_into_def ]
SELECT select_list
{ FROM stream_source_list / MATCHING pattern_expression }
[ WHERE search_conditions ]
[ GROUP BY grouping_expression_list ]
[ HAVING grouping_search_conditions ]
[ ORDER BY order_by_expression_list ]
[ OUTPUT output_specification ]
```

Literal keywords are not case sensitive. Each clause is detailed in the following sections. For information on the built-in operators and functions, see Chapter 3, "EPL Reference: Operators" and Chapter 4, "EPL Reference: Functions."
SELECT

The SELECT clause is required in all EPL statements. The SELECT clause can be used to select all properties using 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.

The SELECT clause also offers optional ISTREAM and RSTREAM keywords to control how events are posted to update listeners attached to the statement.

The syntax for the SELECT clause is summarized below.

```
SELECT [RSTREAM | ISTREAM] ( expression_list | * )
```

The following examples use the FROM clause which defines the sources of the event data. The FROM clause is described in Section "FROM."

### Choosing Specific Event Properties

To choose the particular event properties to return:

```
SELECT event_property [, event_property] [, ...]
FROM stream_def
```

The following statement selects the count and standard deviation of the volume for the last 100 stock tick events.

```
SELECT COUNT, STDDEV(volume)
FROM StockTick RETAIN 100 EVENTS
```

### Using 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 RETAIN BATCH OF 30 SECONDS
```

### Aliasing Event Properties

Event properties and expressions can be aliased using below syntax.

```
SELECT [event_property | expression] AS identifier [,...]
```

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

```
SELECT volume * price AS volPrice
FROM StockTick RETAIN 100 EVENTS
```

### Choosing All Event Properties

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

```
SELECT *
FROM stream_def
```
The following statement selects all of the StockTick event properties for the last 30 seconds:

```sql
SELECT *
FROM StockTick RETAIN 30 SECONDS
```

In a join statement, using the `SELECT *` syntax selects event properties that contain the events representing the joined streams themselves.

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`:

```sql
SELECT *, price * volume AS pricevolume
FROM StockTick RETAIN ALL
```

### Selecting New and Old Events With ISTREAM and RSTREAM Keywords

The optional `ISTREAM` and `RSTREAM` keywords in the `SELECT` clause define the event stream posted to update listeners to the statement. If neither keyword is specified, the engine posts both insert and remove stream events to statement listeners. 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. Insert and remove events are explained in more detail in Section, “Event Sinks.”

By specifying the `ISTREAM` keyword you can instruct the engine to only post insert stream events to update listeners. The engine will then not post any remove stream events. By specifying the `RSTREAM` keyword you can instruct the engine to only post remove stream events to update listeners. The engine will then not post any insert stream events.

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

```sql
SELECT RSTREAM *
FROM StockTick RETAIN 30 SECONDS
```

The `ISTREAM` and `RSTREAM` keywords in the `SELECT` clause are matched by same-name keywords available in the `INSERT INTO` clause as explained in Section, “INSERT INTO.” While the keywords in the `SELECT` clause control the event stream posted to update listeners to the statement, the same keywords in the insert into clause specify the event stream that the engine makes available to other statements.

### FROM

Either the `FROM` or the `MATCHING` clause is required in all EPL statements. The `FROM` clause specifies one or more event streams as the source of the event data. The `MATCHING` clause is discussed in Section, “MATCHING.”

```sql
FROM stream_expression [ inner_join | outer_join ]
```

with `inner_join` specified as a comma separated list of stream expressions:

```sql
{, stream_expression }*
```

and `outer_join` defined as:
((LEFT|RIGHT|FULL) OUTER JOIN stream_expression ON prop_name = prop_name)*

Inner joins are discussed in detail in Section , "Inner Joins" while outer joins are discussed in Section , "Outer Joins."

A stream_expression may simply define the name of the event type used as the source of the stream data, or in more complex scenarios define either a subquery expression as a nested EPL statement or a parameterized SQL query to access JDBC data. In all of these cases, the stream_expression may optionally include an alias as an identifier to qualify any ambiguous property name references in other expressions and a RETAIN clause to define the window of stream data seen by the rest of the query:

(stream_name | subquery_expr | param_sql_query) [[AS] alias] [RETAIN retain_expr]
subquery_expr: { epl_statement }
param_sql_query: database_name ('parameterized_sql_query')

The subquery_expr defines a subquery or nested EPL statement in parenthesis. A subquery is used to pre-filter event stream data seen by the outer EPL statement. For example, the following query would restrict the data seen by the outer EPL statement to only StockTick events coming from a Reuters feed.

SELECT stockSymbol, AVG(price)
FROM (SELECT * FROM StockTick WHERE feedName = 'Reuters' )
RETAIN 1 MINUTE PARTITION BY stockSymbol
GROUP BY stockSymbol

Subqueries may be arbitrarily nested, but may not contain an INSERT INTO or an OUTPUT clause. Unlike with a top level EPL statement, a RETAIN clause is optional within a subquery. Subquery expressions are discussed in more detail in Section , "Subquery Expressions."

The param_sql_query specifies a parameterized SQL query in quotes surrounded by parenthesis that enables reference and historical data accessible through JDBC to be retrieved. The database_name identifies the name of the database over which the query will be executed. Configuration information is associated with this database name to establish a database connection, control connection creation and removal, and to setup caching policies for query results. Parameterized SQL queries are discussed in more detail in Section , "Parameterized SQL Queries."

The RETAIN clause defines the quantity of event data read from the streams listed in the FROM clause prior to query processing. Each stream may have its own RETAIN clause if each require different retain policies. Otherwise, the RETAIN clause may appear at the end of the FROM clause for it to apply to all streams. Essentially the RETAIN clause applies to all streams that appear before it in the FROM clause.

For example, in the following EPL statement, five StockTick events will be retained while three News events will be retained:

SELECT t.stockSymbol, t.price, n.summary
FROM StockTick t RETAIN 5 EVENTS, News n RETAIN 3 EVENTS
WHERE t.stockSymbol = n.stockSymbol

However, in the following statement, four StockTick and four News events will be retained:

SELECT t.stockSymbol, t.price, n.summary
FROM StockTick t, News n RETAIN 4 EVENTS
WHERE t.stockSymbol = n.stockSymbol
With the exception of subquery expressions, all stream sources must be constrained by a retain clause. Thus at a minimum the from clause must contain at least one retain clause at the end for top level EPL statements. The external data from parameterized SQL queries is not affected by the retain clause. The retain clause is discussed in more detail in section "RETAI N."

Inner Joins

Two or more event streams can be part of the from clause and thus both streams determine the resulting events. The where clause lists the join conditions that EPL uses to relate events in two or more streams. If the condition is failed to be met, for example if no event data occurs for either of the joined stream source, no output will be produced.

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

This example joins two 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,
       'withdraw1Fraud' AS desc
FROM FraudWarningEvent AS fraud RETAIN 30 MIN,
     WithdrawalEvent AS withdraw RETAIN 30 SEC
WHERE fraud.accountNumber = withdraw.accountNumber
```

Outer Joins

Left outer joins, right outer joins and full outer joins between an unlimited number of event streams are supported by EPL. Depending on the left, right, or full qualifier, in the absence of event data from either stream source, output may still occur.

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 stream OrderList.

```
SELECT *
FROM RfidEvent AS rfid
     LEFT OUTER JOIN
     OrderList AS orderlist
     ON rfid.itemId = orderList.itemId
RETAIN 30 SECONDS
```

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 stream RfidEvent.

```
SELECT *
FROM RfidEvent AS rfid
     RIGHT OUTER JOIN
     OrderList AS orderlist
     ON rfid.itemId = orderList.itemId
RETAIN 30 SECONDS
```
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.

```
SELECT *
FROM RfidEvent AS rfid
FULL OUTER JOIN
OrderList AS orderlist
ON rfid.itemId = orderlist.itemId
RETAIN 30 SECONDS
```

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

### Subquery Expressions

A subquery expression is a nested EPL statement that appears in parenthesis in the FROM clause. A subquery may not contain an INSERT INTO clause or an OUTPUT clause, and unlike top level EPL statements, a RETAIN clause is optional.

Subquery expressions execute prior to their containing EPL statement and thus are useful to pre-filter event data seen by the outer statement. For example, the following query would calculate the moving average of a particular stock over the last 100 StockTick events:

```
SELECT AVG(price)
FROM (SELECT * FROM StockTick WHERE stockSymbol = 'ACME' )
RETAIN 100 EVENTS
```

If the WHERE clause had been placed in the outer query, StockTick events for other stock symbols would enter into the window, reducing the number of events used to calculate the average price.

In addition, a subquery may be used to a) transform the structure of the inner event source to the structure required by the outer EPL statement or b) merge multiple event streams to form a single stream of events. This allows a single EPL statement to be used instead of multiple EPL statements with an INSERT INTO clause connecting them. For example, the following query merges transaction data from EventA and EventB and then uses the combined data in the outer query:

```
SELECT custId, SUM(latency)
FROM (SELECT A.customerId AS custId, A.timestamp -B.timestamp AS latency
      FROM EventA A, EventB B
      WHERE A.txnId = B.txnId)
RETAIN 30 MIN
GROUP BY custId
```

A subquery itself may contain subqueries thus allowing arbitrary levels of nesting.

### Parameterized SQL Queries

Parameterized SQL queries enable reference and historical data accessible through JDBC to be queried via SQL within EPL statements. In order for such data sources to become accessible to EPL, some configuration is required.

The following restrictions currently apply:
- Only one event stream and one SQL query may be joined; Joins of two or more event streams with an SQL query are not supported.

- Constraints specified in the RETAIN clause are ignored for the stream for the SQL query; that is, one cannot create a time-based or event-based 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 metadata at compilation time. Most major databases provide this function.

The query string is single or double quoted and surrounded by parentheses. The query may contain one or more substitution parameters. The query string is passed to your database software unchanged, allowing you to write any SQL query syntax that your database understands, including stored procedure calls.

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

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

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

```
SELECT custId, cust_name
FROM CustomerCallEvent,
MyCustomerDB (' SELECT cust_name FROM Customer WHERE cust_id = \$\{custId\} ') 
RETAIN 10 MINUTES
```

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 customerId and customerName to demonstrate how the naming of columns in an SQL query can be used in the SELECT clause in the EQL query. The example uses explicit stream names via the AS keyword.

```
SELECT customerId, customerName
FROM CustomerCallEvent AS cce RETAIN 30 SECONDS,
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 events as events enter the window, and remove 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 cases. As a performance optimization, the ISTREAM or RSTREAM keywords in the SELECT clause can be used to instruct the engine to only join insert or remove events, reducing the number of SQL query executions.

Parameterized SQL queries may be used in outer joins as well. 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.

```sql
SELECT custId, custName
FROM CustomerCallEvent AS cce
LEFT OUTER JOIN MyCustomerDB
  ('SELECT cust_id, cust_name AS custName
   FROM Customer WHERE cust_id = ${cce.custId})* AS cq
ON cce.custId = cq.cust_id
RETAIN 10 MINUTES
```

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.

**RETAIN**

At least one RETAIN clause is a required in the FROM clause. The RETAIN clause applies to all stream sources listed in the FROM clause that precedes it. Conceptually it defines a window of event data for each stream source over which the query will be executed. The RETAIN clause has the following syntax:

```sql
RETAIN
  ( ALL [EVENTS] ) |
  [BATCH OF]
  ( integer (EVENT|EVENTS) ) | ( time_interval (BASED ON prop_name)* )
  ( WITH [n] (LARGEST | SMALLEST | UNIQUE) prop_name )* 
  ( PARTITION BY prop_name )*
```

Each aspect of the RETAIN clause is discussed in detail below.

**Keeping All Events**

To keep all events for a stream source, specify the ALL [EVENTS] in the RETAIN clause.

```sql
SELECT AVG(price)
FROM StockTick RETAIN ALL EVENTS
```

In this case, the average price will be calculated based on all StockTick events that occur. Care must be taken with this option, however, since memory may run out when making calculations that require all or part of each event object to be retained under high volume scenarios. One such example would be in calculating a weighted average.

**Specifying Window Size**

The amount of event data to keep when running the query may be determined in two ways. The first option is to specify the maximum number of events kept. For example, the query below would keep a maximum of 100 StockTick events on which the average price would be computed:

```sql
SELECT AVG(price)
FROM StockTick RETAIN (BATCH OF 100)
```
FROM StockTick RETAIN 100 EVENTS

As each new StockTick event comes in, the average price would be computed, with a maximum of 100 events being used for the calculation.

The second option is to specify the time interval in which to collect event data. For example, the query below would keep 1 minute’s worth of StockTick events and compute the average price for this data:

```
SELECT AVG(price)
FROM StockTick RETAIN 1 MINUTE
```

In this case, as each new StockTick event comes in, again the average price would be computed. However, events that arrived more than one minute ago would be removed from the window with the average price being recalculated based on the remaining events in the window.

### Specifying Batched Versus Sliding Windows

By default, the windows holding event data are sliding. With sliding windows, as a new event enters the window, an old event falls off the end of the window once the window is at capacity. Sliding windows cause the query to be re-executed as each new event enters and/or old event leaves the window. An alternative is to specify that the event data should be batched prior to query execution. Only when the window is full, is the query is executed. After this, new event data will again be collected until the window is once again full at which time the query will be re-executed.

For example, the query below would batch together 100 events prior to executing the query to compute the average price:

```
SELECT AVG(price)
FROM StockTick RETAIN BATCH OF 100 EVENTS
```

Once executed, it would batch the next 100 events together prior to re-executing the query.

For more detail on sliding versus batched windows, see Section, "Processing Model."

### Specifying Time Interval

The time interval for the RETAIN clause may be specified in days, hours, minutes, seconds, and/or milliseconds:

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

Some examples of time intervals are:

- 10 seconds
- 10 minutes 30 seconds
- 20 sec 100 msec
- 0.5 minutes
- 1 day 2 hours 20 minutes 15 seconds 110 milliseconds
BASED ON Clause

By default, the elapse of a time interval is based on the internal system clock. However, in some cases, the time needs to be based on a timestamp value appearing as an event property. In this case, the BASED ON clause may be used to specify the property name containing a long-typed timestamp value. In this example, the StockTick events would be expected to have a timestamp property of type long whose value would control inclusion into and removal from the window:

```
SELECT AVG(price) FROM StockTick RETAIN 1 MINUTE BASED ON timestamp
```

When using the BASED ON clause, each stream source listed in the FROM clause must have an associated timestamp property listed or Oracle Event Processing will throw an exception.

Specifying Property Name

A property may be referred to by simply using its property name within the RETAIN clause. However, if ambiguities exist because the same property name exists in more than one stream source in the FROM clause, it must be prefixed with its alias name followed by a period (similar to the behavior of properties referenced in the SELECT clause).

Using PARTITION BY Clause to Partition Window

The PARTITION BY clause allows a window to be further subdivided into multiple windows based on the unique values contained in the properties listed. For example, the following query would keep 3 events for each unique stock symbol:

```
SELECT stockSymbol, price FROM StockTick RETAIN 3 EVENTS PARTITION BY stockSymbol
```

Conceptually this is similar to the GROUP BY functionality in SQL or EPL. However, the PARTITION BY clause only controls the size and subdivision of the window and does not cause event data to be aggregated as with the GROUP BY clause. However, in most cases, the PARTITION BY clause is used in conjunction with the GROUP BY clause with same properties specified in both.

The following examples illustrate the interaction between PARTITION BY and GROUP BY. In the first example, with the absence of the PARTITION BY clause, a total of 10 events will be kept across all stock symbols.

```
SELECT stockSymbol, AVG(price) FROM StockTick RETAIN 10 EVENTS GROUP BY stockSymbol
```

The average price for each unique set of stock symbol will be computed based on these 10 events. If a stock symbol of AAA comes into the window, it may cause a different stock symbol such as BBB to leave the window. This would cause the average price for both the AAA group as well as the BBB group to change.

The second example includes the PARTITION BY clause and the GROUP BY clause.

```
SELECT stockSymbol, AVG(price) FROM StockTick RETAIN 10 EVENTS PARTITION BY stockSymbol GROUP BY stockSymbol
```

In this case, 10 events will be kept for each unique stock symbol. If a stock symbol of AAA comes into the window, it would only affect the sub-window associated with that...
symbol and not other windows for different stock symbols. Thus, in this case, only the average price of AAA would be affected.

**Using WITH Clause to Keep Largest/Smallest/Unique Values**

The WITH clause allows the largest, smallest, and unique property values to be kept in the window. For example, to keep the two highest priced stocks, the following statement would be used:

```sql
SELECT stockSymbol, price
FROM StockTick RETAIN ALL WITH 2 LARGEST price
```

In the case of time-based windows, the [n] qualifier before the LARGEST or SMALLEST keyword determines how many values are kept. For example, the following statement would keep the two smallest prices seen over one minute:

```sql
SELECT stockSymbol, price
FROM StockTick RETAIN 1 MINUTE WITH 2 SMALLEST price
```

In the absence of this qualifier, the single largest or smallest value is kept.

The UNIQUE qualifier causes the window to include only the most recent among events having the same value for the specified property. For example, the following query would keep only the last stock tick for each unique stock symbol:

```sql
SELECT *
FROM StockTick RETAIN 1 DAY WITH UNIQUE stockSymbol
```

Prior events of the same property value would be posted as old events by the engine.

**MATCHING**

Either a MATCHING or a FROM clause must appear in an EPL statement. The MATCHING clause is an alternate mechanism for determining which events are used by the EPL statement. It allows for the detection of a series of one or more events occurring that satisfies a specified pattern. Pattern expressions consist of references to streams separated by logical operators such as AND, OR, and FOLLOWED BY to define the sequence of events that compose the pattern. You may include an optional RETAIN clause, as specified in Section , "RETAIN," to define the characteristics of the window containing the matched events. The MATCHING clause executes prior to the WHERE or HAVING clauses.

The MATCHING clause syntax is as follows:

```sql
MATCHING pattern_expression [RETAIN retain_clause]
```

with pattern_expression having the following syntax:

```sql
[NOT|EVERY] stream_expression
  ( ( AND | OR | [NOT] FOLLOWED BY ) stream_expression )*
[WITHIN time_interval]
```

You can use the NOT operator to detect the absence of an event and the EVERY operator to control how pattern matching continues after a match. The stream_expression is a stream source name optionally bound to a variable and filtered by a parenthesized expression:

```sql
stream_expression: [var_name:=]stream_name [( filter_expression )]
```

Alternatively, a stream_expression may itself be a pattern_expression allowing for arbitrarily complex nesting of expressions:
The `var_name` is bound to the event object occurring that triggers the match. It may be referenced as any other event property in filter expressions that follow as well as in other clauses such as the SELECT and WHERE clauses. The `stream_name` may optionally be followed by a parenthesized expression to filter the matching events of that type. The expression act as a precondition for events to enter the corresponding window and has the same syntax as a WHERE clause expression. Previously bound variables may be used within the expression to correlate with already matched events.

The `time_interval` is a time interval as specified in Section , "Specifying Time Interval" that follows the optional WITHIN keyword to determine how long to wait before giving up on the preceding expression to be met.

In the example below we look for RFIDEvent event with a category of "Perishable" followed by an RFIDError within 10 seconds whose `id` matches the ID of the matched RFIDEvent object.

```sql
SELECT *
MATCHING a:=RFIDEvent(category='Perishable')
  FOLLOWED BY RFIDError(id=a.id) WITHIN 10 seconds
RETAIN 1 MINUTE
```

The following sections discuss the syntax, semantics, and additional operators available in the MATCHING clause to express temporal constraints for pattern matching.

**FOLLOWED BY Operator**

The FOLLOWED BY temporal operator matches on the occurrence of several event conditions in a particular order. It specifies that first the left hand expression must turn true and only then will the right hand expression be evaluated for matching events.

For example, the following pattern looks for event A and if encountered, looks for event B:

```sql
A FOLLOWED BY B
```

This does not mean that event A must immediately be followed by event B. Other events may occur between the event A and the event B and this expression would still evaluate to true. If this is not the desired behavior, used the NOT operator as described in Section , "NOT Operator."

**AND Operator**

The AND logical operator requires both nested pattern expressions to turn true before the whole expression returns true. In the context of the MATCHING clause, the operator matches on the occurrence of several event conditions but not necessarily in a particular order.

For example, the following pattern matches when both event A and event B are found:

```sql
A AND B
```

The pattern matches on any sequence of A followed by B in either order. In addition, it is not required that a B event immediately follow an A event - other events may appear in between the occurrence of an A event and a B event for this expression to return true.

**OR Operator**

The OR logical operator requires either one of the expressions to turn true before the whole expression returns true. In the context of the MATCHING clause, the operator
matches on the occurrence of either of several event conditions but not necessarily in a particular order.

For example, the following pattern matches for either event A or event B:

A OR B

The following would detect all stock ticks that are either above a certain price or above a certain volume.

StockTick(price > 1.0) OR StockTick(volume > 1000)

**NOT Operator**

The NOT operator negates the truth value of an expression. In the context of the MATCHING clause, the operator allows the absence of an event condition to be detected.

The following pattern matches only when an event A is encountered followed by event B but only if no event C was encountered before event B.

{ A FOLLOWED BY B } AND NOT C

**EVERY Operator**

The EVERY operator indicates that the pattern sub-expression should restart when the sub-expression qualified by the EVERY keyword evaluates to true or false. In the absence of the EVERY operator, an implicit EVERY operator is inserted as a qualifier to the first event stream source found in the pattern not occurring within a NOT expression.

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 sub-expression looking for more event(s) or timing conditions that match the pattern sub-expression.

Let’s consider an example event sequence as follows:

A₁ B₁ C₁ B₂ A₂ D₁ A₃ B₃ E₁ A₄ F₁ B₄

**Table 2–1 Example EVERY Operators**

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVERY ( A FOLLOWED BY B )</td>
<td>Detect event A followed by event B. At the time when B occurs the pattern matches, then the pattern matcher restarts and looks for event A again. Matches on B₁ for combination (A₁, B₁). Matches on B₃ for combination (A₂, B₃). Matches on B₄ for combination (A₄, B₄)</td>
</tr>
<tr>
<td>EVERY A FOLLOWED BY B</td>
<td>The pattern fires for every event A followed by an event B. Matches on B₁ for combination (A₁, B₁) Matches on B₃ for combination (A₂, B₃) and (A₃, B₃). Matches on B₄ for combination (A₄, B₄)</td>
</tr>
</tbody>
</table>
The examples show that it is possible that a pattern fires for multiple combinations of events that match a pattern expression.

Let’s consider the **EVERY** operator in conjunction with a sub-expression that matches three events that follow each other:

\[ \text{EVERY (A FOLLOWED BY B FOLLOWED BY C)} \]

The pattern first looks for event A. When event A arrives, it looks for event B. After event B arrives, the pattern looks for event C. Finally, when event C arrives the pattern matches. The engine then starts looking for event A again.

Assume that between event B and event C a second event A₂ arrives. The pattern would ignore the A₂ entirely since it’s then looking for event C. As observed in the prior example, the **EVERY** operator restarts the sub-expression **A FOLLOWED BY B FOLLOWED BY C** only when the sub-expression fires.

In the next statement the every operator applies only to the A event, not the whole sub-expression:

\[ \text{EVERY A FOLLOWED BY B FOLLOWED BY C} \]

This pattern now matches for any event A that is followed by an event B and then event C, regardless of when the event A arrives. This can often be impractical unless used in combination with the **AND NOT** syntax or the **RETAIN** syntax to constrain how long an event remains in the window.

### WITHIN Operator

The **WITHIN** qualifier acts like a stopwatch. If the associated pattern expression does not become true within the specified time period it is evaluated by the engine as false. The **WITHIN** qualifier takes a time period as a parameter as specified in Section, "Specifying Time Interval."

This pattern fires if an A event arrives within 5 seconds after statement creation.

\[ \text{A WITHIN 5 SECONDS} \]

This pattern fires for all A events that arrive within 5 second intervals.

This pattern matches for any one A or B event in the next 5 seconds.

\[ \text{(A or B) WITHIN 5 SECONDS} \]

This pattern matches for any two errors that happen 10 seconds within each other.

\[ \text{A(status='ERROR') FOLLOWED BY B(status='ERROR') WITHIN 10 SECONDS} \]

This pattern matches when a Status event does not occur within 10 seconds:
NOT Status WITHIN 10 SECONDS

Event Structure for Matched Pattern

The structure of the events produced when a pattern matches is determined by the structure of the union of the variables bound within the MATCHING clause. Thus variable bindings must be present in order to retrieve data from the matched events.

For example, given the following pattern:

```
tick:=StockTick FOLLOWED BY news:=News(stockSymbol = tick.stockSymbol)
```

Events that match would have a composite event type with two properties: a tick property with a type of StockTick and a news property with a type of News.

WHERE

The WHERE clause is an optional clause in EPL statements. Using the WHERE clause event streams can be joined and events can be filtered. Aggregate functions may not appear in a WHERE clause. To filter using aggregate functions, the HAVING clause should be used.

```
WHERE aggregate_free_expression
```

Comparison operators =, <, >, >=, <=, !=, <=> IS NULL, IS NOT NULL and logical combinations using AND and OR are supported in the WHERE clause. 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
```

GROUP BY

The GROUP BY clause is optional in 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 this can produce confusing results.

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

```
SELECT symbol, SUM(price)
FROM StockTickEvent RETAIN 30 SEC
GROUP BY symbol
```

The syntax of the GROUP BY clause is:

```
GROUP BY aggregate_free_expression [, aggregate_free_expression] [, …]
```

EPL places the following restrictions on expressions in the GROUP BY clause:

- Expressions in the GROUP BY clause cannot contain aggregate functions
- Event properties that are used within aggregate functions in the SELECT clause cannot also be used in a GROUP BY expression

You can list more than 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 grouped by symbol and tick data feed. EPL posts one event for each group to statement update listeners:

```
SELECT symbol, tickDataFeed, MEDIAN(volume)
FROM StockTickEvent RETAIN 30 SECONDS
GROUP BY symbol, tickDataFeed
```

In the statement above the event properties in the select list (symbol and 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.

EPL 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 RETAIN 30 SECONDS
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, EPL 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 RETAIN 30 SECONDS
GROUP BY symbol, tickDataFeed
```

Expressions are also allowed in the GROUP BY list:

```
SELECT symbol * price, count(*)
FROM StockTickEvent RETAIN 30 SECONDS
GROUP BY symbol * price
```

If the GROUP BY expression results in a null value, the null value becomes its own group. All null values are aggregated into the same group. The COUNT(expression) aggregate function does not count null values and 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 RETAIN 30 SECONDS
WHERE volume > 100
GROUP BY symbol
```

**HAVING**

The **HAVING** clause is optional in EPL statements. Use the **HAVING** clause to pass or reject events defined by the GROUP BY clause. The **HAVING** clause sets conditions for the GROUP

2-16  EPL Language Reference for Oracle Event Processing
HAVING

BY clause in the same way WHERE sets conditions for the SELECT clause, except the WHERE clause cannot include aggregate functions, while HAVING often does.

**HAVING expression**

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 than 1000.

```
SELECT symbol, SUM(price)
FROM StockTickEvent RETAIN 30 SEC
GROUP BY symbol
HAVING SUM(price) > 1000
```

To include more than 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 than 1000 and an average volume less than 500.

```
SELECT symbol, SUM(price), AVG(volume)
FROM StockTickEvent RETAIN 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 than the current running average price of all stock tick events in the last 30 seconds.

```
SELECT symbol, price, AVG(price)
FROM StockTickEvent RETAIN 30 SEC
HAVING price < AVG(price)
```

**Interaction With MATCHING, WHERE and GROUP BY Clauses**

When an EPL statement includes subqueries, a MATCHING clause, WHERE conditions, a GROUP BY clause, and HAVING conditions, the sequence in which each clause executes determines the final result:

1. Any subqueries present in the statement run first. The subqueries act as a filter for events to enter the window of the outer query.
2. The event stream’s filter conditions in the MATCHING clause, if present, dictates which events enter a window. The filter discards any events not meeting filter criteria.
3. The WHERE clause excludes events that do not meet its search condition.
4. Aggregate functions in the SELECT list calculate summary values for each group.
5. 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 (SELECT * FROM StockTickEvent WHERE symbol='ACME')
RETAIN 10 EVENTS
WHERE volume > 1000
```
GROUP BY tickDataFeed  
HAVING STDDEV(price) > 0.8

EPL filters events using the subquery for the event stream `StockTickEvent`. In the example above, only events with symbol `ACME` enter the window over the last 10 events, all other events are simply discarded. The `WHERE` clause removes any events posted into the window (events entering the window and event leaving the window) that do not match the condition of volume greater than 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. EPL applies the `HAVING` clause and only lets events pass for `tickDataFeed` groups with a standard deviation of price greater than 0.8.

**ORDER BY**

The `ORDER BY` clause is optional in EPL. It is used for ordering output events by their properties, or by expressions involving those properties. For example, the following statement batches 1 minute of stock tick events sorting them first by price and then by volume.

```sql
SELECT symbol  
FROM StockTickEvent RETAIN BATCH OF 1 MINUTE  
ORDER BY price, volume
```

Here is the syntax for the `ORDER BY` clause:

```
ORDER BY expression [ASC | DESC] [, expression [ASC | DESC] [,...]]
```

EPL places the following restrictions on the expressions in the `ORDER BY` clause:
- All aggregate functions that appear in the `ORDER BY` clause must also appear in the `SELECT` expression.

Otherwise, any kind of expression that can appear in the `SELECT` clause, as well as any alias defined in the `SELECT` clause, is also valid in the `ORDER BY` clause.

**OUTPUT**

The `OUTPUT` clause is optional in EPL and is used to control or stabilize the rate at which events are output. For example, the following statement batches old and new events and outputs them at the end of every 90 second interval.

```sql
SELECT *  
FROM StockTickEvent RETAIN 5 EVENTS  
OUTPUT EVERY 90 SECONDS
```

Here is the syntax for output rate limiting:

```
OUTPUT [ALL | (FIRST | LAST) [number]] EVERY number [EVENTS | time_unit]
```

where

```
time_unit: MIN | MINUTE | MINUTES | SEC | SECOND | SECONDS | MILLISECONDS | MS
```

The `ALL` keyword is the default and specifies that all events in a batch should be output. The batch size can be specified in terms of time or number of events.

The `FIRST` keyword specifies that only the first event in an output batch is to be output. The optional number qualifier allows more than one event to be output. The `FIRST` keyword instructs the engine to output the first matching event(s) as soon as
they arrive, and then ignore 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 same cycle starts again.

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. The optional number qualifier allows more than one event to be output.

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

```sql
SELECT *
FROM StockTickEvent RETAIN 5 EVENTS
OUTPUT EVERY 1.5 MINUTES
```

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

```sql
SELECT *
FROM StockTickEvent RETAIN 30 SECONDS
OUTPUT EVERY 5 EVENTS
```

Additionally, event output can be further modified by the optional LAST keyword, which causes output of only the last event(s) to arrive into an output batch. For the example below, the last five events would be output every three minutes.

```sql
SELECT *
FROM StockTickEvent RETAIN 30 SECONDS
OUTPUT LAST 5 EVERY 3 MINUTES
```

Using the FIRST keyword you can be notified at the start of the interval. This allows one to be immediately notified each time a rate falls below a threshold.

```sql
SELECT *
FROM TickRate RETAIN 30 SECONDS
WHERE rate < 100
OUTPUT FIRST EVERY 60 SECONDS
```

**Interaction With GROUP BY and HAVING Clauses**

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 do not satisfy the HAVING clause, then the actual number of events output by the statement could be fewer than n.

Second, the LAST and ALL keywords have special meanings when used in a statement with aggregate functions and the GROUP BY clause. The LAST keyword specifies that only groups whose aggregate values have been updated with the most recent batch of events should be output. 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.

**INSERT INTO**

The INSERT INTO clause is optional in EPL. This clause can be specified to make the results of a statement available as an event stream for use in further statements. The clause can also be used to merge multiple event streams to form a single stream of events.
The `INSERT INTO` clause in the above statement generates events of type `CombinedEvent`. Each generated `CombinedEvent` event has two event properties named `custId` and `latency`. The events generated by the above statement can be used in further statements. For example, the statement below uses the generated events.

```sql
SELECT custId, SUM(latency)
FROM CombinedEvent RETAIN 30 MIN
GROUP BY custId
```

The `INSERT INTO` clause can consist of just an event type alias, or of an event type alias and one or more event property names. The syntax for the `INSERT INTO` clause is as follows:

```
INSERT [ISTREAM | RSTREAM] INTO event_type_alias [(prop_name [,prop_name, [...]])]
```

The `ISTREAM` (default) and `RSTREAM` keywords are optional. If neither keyword is specified, the engine supplies the insert stream events generated by the statement to attached update listeners. The insert stream consists of the events entering the respective window(s) or stream(s). If the `RSTREAM` keyword is specified, the engine supplies the remove stream events generated by the statement. The remove stream consists of the events leaving the respective window(s).

The `event_type_alias` is an identifier that names the events generated by the engine. The identifier can be used in statements to filter and process events of the given name.

The engine also allows update listeners to be attached to a statement that contain an `INSERT INTO` clause.

To merge event streams, simply use the same `event_type_alias` identifier in any EPL statements that you would like to be merged. Make sure to use the same number and names of event properties and that event property types match up.

EPL places the following restrictions on the `INSERT INTO` clause:

- 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
- If the event type alias has already been defined by a prior statement and the event property names and types do not match, an exception is thrown at statement creation time.

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

```sql
INSERT INTO CombinedEvent (custId, latency)
SELECT A.customerId, A.timestamp - B.timestamp
FROM EventA A, EventB B RETAIN 30 MIN
WHERE A.txnId = B.txnId
```

The `RSTREAM` keyword is used 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 than when events enter a window. The statement below generates `CombinedEvent` events when `EventA` and `EventB` leave the window after 30 minutes.

```sql
INSERT RSTREAM INTO CombinedEvent
SELECT A.customerId AS custId, A.timestamp - B.timestamp AS latency
FROM EventA A, EventB B RETAIN 30 MIN
```
WHERE A.txnId = B.txnId

**Simple and Correlated Subqueries**

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

EPL supports both simple subqueries as well as correlated subqueries. In a simple subquery, the inner query does not reference any elements (rows) from the outer query. The following example shows a simple subquery within a `SELECT` clause:

```sql
SELECT assetId,
   (SELECT zone
      FROM ZoneClosed.std:lastevent) AS lastClosed
FROM RFIDEvent
SELECT assetId,
   (SELECT zone
      FROM ZoneClosed RETAIN ALL EVENTS) AS lastClosed
FROM RFIDEvent
```

If the inner query is dependent on the outer query, it is referred to as a correlated subquery, as shown in the following example. In the query, the `WHERE` clause in the inner query involves a stream from the outer query:

```sql
SELECT *
FROM RfidEvent AS RFID
WHERE 'Dock 1' =
   (SELECT name
      FROM Zones RETAIN ALL EVENTS WITH UNIQUE zoneId
      WHERE zoneId = RFID.zoneId)
```

The preceding example 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 `WITH UNIQUE` subclause in the `RETAIN` clause to guarantee that only the last event per zone ID is held from processing by the subquery.

The following example is a correlated subquery within a `SELECT` clause. In this query, the `SELECT` clause retrieves the zone name by means of a subquery against the `Zones` set of events correlated by zone id:

```sql
SELECT zoneId,
   (SELECT name
      FROM Zones RETAIN ALL EVENTS WITH UNIQUE zoneId
      WHERE zoneId = RFID.zoneId) AS name
FROM RFIDEvent
```

When a simple or correlated subquery returns multiple rows, Oracle Event Processing returns a null value as the subquery result. To limit the number of events returned by a subquery, consider using `WITH UNIQUE` or `PARTITION BY` in the `RETAIN` clause.

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:

```sql
SELECT
   (SELECT *
      FROM MarketData RETAIN 1 EVENT) AS md
MATCHING WITHIN 10 SECONDS
```
The output events of the preceding statement contain the underlying MarketData event in a property named \texttt{md}. The statement populates the last MarketData event into a property named \texttt{md} every 10 seconds following the pattern definition, or populates a null value if no MarketData event has been encountered so far.

The following restrictions apply to subqueries:

- 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.
- You cannot use aggregation functions in subqueries. Instead, use the \texttt{INSERT into} clause to provide aggregation results for use in subqueries.
- Subqueries can consist only of a \texttt{SELECT} clause, a \texttt{FROM} clause, and a \texttt{WHERE} clause. The \texttt{GROUP BY} and \texttt{HAVING} clauses, as well as joins, outer-joins and output rate limiting are not permitted within subqueries.

The performance of your statement that contains 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 using equals (\texttt{=}), Oracle Event Processing 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 (\texttt{WHERE} clause), as each row in the subquery stream must evaluate against the \texttt{WHERE} clause filter.

**Parameterized Queries**

Parameterized queries allow you to put placeholders inside of an EPL query in the form of a question mark. At runtime you bind these placeholders with values and they are then compiled into regular statements. The process is much like the \texttt{PreparedStatement} in JDBC.

Use of parameterized queries prevents requiring separate rules to be written for each possible parameter value. For example, the signal trading application has 300 rules that only differ in the value that selects which stock symbol to be matched against. In a real-world application, maintaining 300 separate rules would be a maintenance nightmare, because a change in the rule logic would require an identical change in all 300 rules. Instead, the parameterized query approach allows a developer to put a placeholder in the query which is later bound to a particular stock symbol value. Once bound with values, parameterized queries compile down to regular queries, and so the performance of the queries is not be adversely affected.

To change the parameters associated with a parameterized EPL query at runtime without coding in Java, use the “wlevs.Admin Command-Line Reference” in the Oracle Fusion Middleware Administrator’s Guide for Oracle Event Processing.

**General Usage**

You use question marks in a parameterized EPL query to indicate the location of a placeholder. You can use placeholders in any expression where a constant value is currently allowed; you put the placeholder in the same location as the constant would be. In practice this means that you can specify placeholders only in the \texttt{SELECT} and \texttt{WHERE} clauses of a query. You cannot use placeholders in the \texttt{RETAIN} clause.

The following example shows a parameterized query:

```
SELECT symbol, AVG(price) AS average, ? AS market
FROM StockTick t RETAIN ALL EVENTS
WHERE symbol = ?
```
The example shows how to specify a placeholder in both the `SELECT` and `WHERE` clause. At runtime, the two placeholders are bound with values, such as NASDAQ and ORCL so that the actual EPL statement that is executed looks like the following:

```
SELECT symbol, AVG(price) AS average, NASDAQ AS market
FROM StockTick t RETAIN ALL EVENTS
WHERE symbol = ORCL
```

### Using a Parameterized EPL Statement in the Processor Configuration File

If you have configured the EPL rules for a particular processor in its component configuration file, you specify the EPL statement as usual using the `<rule>` element. Specify the placeholders using question marks, as described in Section, "General Usage."

Then use the `<binding>` element to specify one or more `<params>` elements that correspond to the comma-separated list of values that you want to pass to the parameterized EPL statement at runtime. Each `<params>` element effectively causes a new EPL query to execute with the new parameters. Use the `id` attribute of `<binding>` to reference the particular EPL rule to which the binding applies. Each `<params>` element contains a single set of parameters; the order of the parameters corresponds to the order in which the question marks appear in the parameterized query.

Use the `id` attribute of `<params>` to uniquely identify each individual parameter set; this is so later you can dynamically delete single parameter sets using JMX or wlevs.Admin.

As with the EPL rules in the configuration file, group the `<binding>` elements together using a parent `<bindings>` element.

The following example shows how to specify a parameterized query and its runtime parameters for the query described in Section, "General Usage."

```
<n1:config xmlns:n1="http://www.bea.com/ns/wlevs/config/application"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <processor>
      <name>myProcessor</name>
      <rules>
         <rule id="MarketRule"><![CDATA[
            SELECT symbol, AVG(price) AS average, ? AS market
            FROM StockTick t RETAIN ALL EVENTS
            WHERE symbol = ?
         ]]>></rule>
      </rules>
      <bindings>
         <binding id="MarketRule">
            <params id="nasORCL">NASDAQ,ORCL</params>
            <params id="nyJPM">NYSE,JPM</params>
            <params id="nyWFC">NYSE,WFC</params>
         </binding>
      </bindings>
   </processor>
</n1:config>
```

In the preceding example, the `MarketRule` EPL query includes two placeholders: one in the `SELECT` clause and another in the `WHERE` clause. The `<binding id="MarketRule">` element specifies the list of parameter sets that will be passed to `MarketRule` at runtime. Each parameter set is specified with a single `<params>` element. Because there are two placeholders in the parameterized query, each `<params>` element specifies two values separated by a comma.
At runtime, the preceding parameterized query effectively breaks down into the following three queries:

\[
\begin{align*}
&\text{SELECT symbol, AVG(price) AS average, } \text{NASDAQ} \text{ AS market} \\
&\text{FROM StockTick t RETAIN ALL EVENTS} \\
&\text{WHERE symbol = ORCL} \\
&\text{SELECT symbol, AVG(price) AS average, } \text{NYSE} \text{ AS market} \\
&\text{FROM StockTick t RETAIN ALL EVENTS} \\
&\text{WHERE symbol = JPM} \\
&\text{SELECT symbol, AVG(price) AS average, } \text{NYSE} \text{ AS market} \\
&\text{FROM StockTick t RETAIN ALL EVENTS} \\
&\text{WHERE symbol = WFC}
\end{align*}
\]

**Programmatically Using a Prepared EPL Statement**

If you use the Oracle Event Processing APIs in your application to create and execute EPL rules, and you want to use parameterized rules, use the com.bea.wlevs.ede.api.PreparedStatement interface to create instances of the com.bea.wlevs.ede.api.BoundStatement interfaced based on provided parameter values.

Both PreparedStatement and BoundStatement extend Statement, which is the basic object that represents a compiled EPL query. PreparedStatement has additional methods to bind parameters to the placeholders in the query, return the number of placeholders in the query, and return an instance of the BoundStatement based on a binding id. BoundStatement has additional methods for getting the parameter values used to bind the EPL query and getting the actual parameterized query itself.

If you call the start() or stop() methods on a PreparedStatement, Oracle Event Processing starts or stops all of the BoundStatements that have been created from the PreparedStatement; this is because a parameterized query is not itself executable, only the BoundStatements that have been created from it with specific parameter values are actually executable.

See Oracle Fusion Middleware Java API Reference for Oracle Event Processing for the full reference documentation on these APIs.

The following sample Java snippet shows how to create a BoundStatement using the query described in Section "General Usage":

```java
String stmtString = 
    "select symbol, avg(price) as average, ? as market " + 
    "from StockTick t " + 
    "retain all events " + 
    "where symbol = ?" 
    ;

processorImpl.createStatement(stmtString);
PreparedStatement preparedStmt = 
(PreparedStatement)processorImpl.createStatement(stmtString);
BoundStatement orclBoundStmt = preparedStmt.bind(new String[] 
{"NASDAQ","ORCL"});
```
This chapter provides a reference to operators in Event Processing Language (EPL). Note that EPL is deprecated; new development should use Oracle Continuous Query Language (Oracle CQL).

This chapter includes the following sections:

- Overview of EPL Operators
- Arithmetic Operators
- Logical and Comparison Operators
- Concatenation Operators
- Binary Operators
- Array Definition Operator
- List and Range Operators
- String Operators
- Temporal Operators

**Overview of EPL Operators**

The precedence of arithmetic and logical operators in EPL follows Java standard arithmetic and logical operator precedence.

**Arithmetic Operators**

The table below outlines the arithmetic operators available.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, -</td>
<td>As unary operators they denote a positive or negative expression. As binary operators they add or subtract.</td>
</tr>
<tr>
<td>* /</td>
<td>Multiplication and division are binary operators.</td>
</tr>
<tr>
<td>%</td>
<td>Modulo binary operator.</td>
</tr>
</tbody>
</table>

**Logical and Comparison Operators**

The table below outlines the logical and comparison operators available.
## Concatenation Operators

The table below outlines the concatenation operators available.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Binary Operators

The table below outlines the binary operators available.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Bitwise AND if both operands are numbers; conditional AND if both operands are Boolean.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Bitwise exclusive OR (XOR)</td>
</tr>
</tbody>
</table>

## 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:

```java
{ [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 `RFIDEEvent` events.

```sql
SELECT {observation, command} AS actions
FROM RFIDEEvent RETAIN ALL
```

The engine determines the array type based on the types returned by the expressions in the array definition. For example, if all expressions in the array definition return

---

### Table 3–2  Logical and Comparison Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>Returns true if the following condition is false, returns false if it is true.</td>
</tr>
<tr>
<td>OR</td>
<td>Returns true if either component condition is true, returns false if both are false</td>
</tr>
<tr>
<td>AND</td>
<td>Returns true if both component conditions are true, returns false if either is false</td>
</tr>
<tr>
<td>=, !=, &lt;, &gt;, &lt;=, =&gt;</td>
<td>Comparison operators</td>
</tr>
</tbody>
</table>

### Table 3–3  Concatenation Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 3–4  Binary Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Bitwise AND if both operands are numbers; conditional AND if both operands are Boolean.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Bitwise exclusive OR (XOR)</td>
</tr>
</tbody>
</table>
integer values then the type of the array is `java.lang.Integer[]`. If the types returned by all expressions are a 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 RETAIN ALL 
WHERE Filter.myFilter(zone, {1,2,3})
```

List and Range Operators

This section describes the following two operations:

- Section , "IN Operator"
- Section , "BETWEEN Operator"

IN Operator

The `IN` operator determines if a given value matches any value in a list. The syntax of the operator is:

```
test_expression [NOT] IN (expression [,expression [,...]] )
```

The `test_expression` is any valid expression. The `IN` keyword is followed by a list of expressions to test for a match. The optional `NOT` keyword specifies that the result of the predicate be negated.

The result of an `IN` expression is of type `Boolean`. If the value of `test_expression` is equal to any expression from the comma-separated list, the result value is `true`. Otherwise, the result value is `false`. All expressions must be of the same type or a type compatible with `test_expression`.

The next example shows how the `IN` keyword can be applied to select certain command types of `RFIDEvents`:

```
SELECT * 
FROM RFIDEvent RETAIN ALL 
WHERE command IN ('OBSERVATION', 'SIGNAL')
```

The statement is equivalent to:

```
SELECT * 
FROM RFIDEvent RETAIN ALL 
WHERE command = 'OBSERVATION' OR symbol = 'SIGNAL'
```

BETWEEN Operator

The `BETWEEN` operator specifies a range to test. The syntax of the operator is:

```
test_expression [NOT] BETWEEN begin_expression AND end_expression
```

The `test_expression` is any valid expression and is the expression to test for the range being inclusively within the expressions 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 than or equal to the value of begin_expression and less than or equal to the value of end_expression, the result is true.

The next example shows how the BETWEEN keyword can be used to select events with a price between 55 and 60 (inclusive).

```
SELECT *
FROM StockTickEvent RETAIN ALL
WHERE price BETWEEN 55 AND 60
```

The equivalent expression without using the BETWEEN keyword is:

```
SELECT *
FROM StockTickEvent RETAIN ALL
WHERE price >= 55 AND price <= 60
```

The begin_expression and end_expression may occur in either order without affecting the query. For example, the following is equivalent to the above example:

```
SELECT *
FROM StockTickEvent RETAIN ALL
WHERE price BETWEEN 60 AND 55
```

**String Operators**

This section describes the following string operators:

- Section, "LIKE Operator"
- Section, "REGEXP Operator"

**LIKE Operator**

The LIKE operator 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 EPL, 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 used to escape the _ 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 shown below:

```
SELECT *
FROM PersonLocationEvent RETAIN ALL
WHERE name LIKE '%Jack%'
```

In this example the WHERE clause matches events where the suffix property is a single _ character.

```
SELECT *
FROM PersonLocationEvent RETAIN ALL
WHERE suffix LIKE '!_' ESCAPE '!'
```
**REGEXP Operator**

The **REGEXP** operator is a form of pattern matching based on regular expressions implemented through the Java `java.util.regex` package. The syntax of **REGEXP** is:

\[
\text{test_expression} \ [\text{NOT}] \ \text{REGEXP} \ \text{pattern_expression}
\]

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** operator is below.

```
SELECT *
FROM PersonLocationEvent RETAIN ALL
WHERE name REGEXP '*Jack*
```

**Temporal Operators**

This section describes the following temporal operations:

- Section, "FOLLOWED BY Operator"
- Section, "WITHIN Operator"
- Section, "EVERY Operator"

**FOLLOWED BY Operator**

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.

For example, the following pattern looks for event A and if encountered, looks for event B:

```
A FOLLOWED BY B
```

This does not mean that event A must immediately be followed by event B. Other events may occur between the event A and the event B and this expression would still evaluate to true. If this is not the desired behavior, the **NOT** operator can be used.

**WITHIN Operator**

The **WITHIN** qualifier acts like a stopwatch. If the associated pattern expression does not become true within the specified time period it is evaluated by the engine as false. The **WITHIN** qualifier takes a time period as a parameter as specified in Section, "Specifying Time Interval."

This pattern fires if an A event arrives within 5 seconds after statement creation.

```
A WITHIN 5 seconds
```

This pattern fires for all A events that arrive within 5 second intervals.
EVERY Operator

The EVERY operator indicates that the pattern sub-expression should restart when the sub-expression qualified by the EVERY keyword evaluates to true or false. In the absence of the EVERY operator, an implicit EVERY operator is inserted as a qualifier to the first event stream source found in the pattern not occurring within a NOT expression.

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 sub-expression looking for more event(s) or timing conditions that match the pattern sub-expression.

This pattern fires when an A event is followed by a B event and continues attempting to match again after the B event:

EVERY (A FOLLOWED BY B)

This pattern also fires when an A event is followed by a B event, but continues attempting to match again after the A event:

EVERY A FOLLOWED BY B

The EVERY in this pattern is optional, since it would implicitly be placed here if it was absent.
This chapter provides a reference to functions in Event Processing Language (EPL). Note that EPL is deprecated; new development should use Oracle Continuous Query Language (Oracle CQL).

This chapter includes the following sections:

- Single-row Functions
- Aggregate functions
- User-Defined functions

### Single-row Functions

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.

EPL allows static Java library methods as single-row functions, and also features built-in single-row functions.

EPL 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:

```epl
SELECT symbol, Math.round(volume/1000)
FROM StockTickEvent RETAIN 30 SECONDS
```

Other arbitrary Java classes may also be used, however their names must be fully qualified or configured to be imported. For more information, see Section, "User-Defined functions."

The table below outlines the built-in single-row functions available.

<table>
<thead>
<tr>
<th>Single-row Function</th>
<th>Result</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MAX(expression, expression [, expression [,...]])</code></td>
<td>Returns the highest numeric value among the two or more comma-separated expressions.</td>
<td>Section, &quot;The MIN and MAX Functions&quot;</td>
</tr>
</tbody>
</table>
You may also create user-defined single-row functions. See Section , "User-Defined functions".

The MIN and MAX Functions

The MIN and MAX functions take two or more expression parameters. The MIN function returns the lowest numeric value among these 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.

```sql
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 AS ticks, NewsEvent AS news RETAIN 30 SECONDS
WHERE ticks.symbol = news.symbol
```

The **MIN** and **MAX** functions are also available as aggregate functions. See Section, "Aggregate functions" for a description of this usage.

**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** type result with a value of **foo**.

```
SELECT COALESCE(NULL, 'foo')
```

**The CASE Control Flow Function**

The **CASE** control flow function has two versions. The first version takes a value and a list of compare values to compare against, and returns the result where the first value equals the compare value. The second version takes a list of conditions and returns the result for the first condition that is true.

The return type of a **CASE** expression is the compatible aggregated type of all return values.

The example below shows the first version of a **CASE** statement. It has a **String** return type and returns the value **one**.

```
SELECT CASE 1 WHEN 1 THEN 'one' WHEN 2 THEN 'two' ELSE 'more' END
```

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
```

**The PREV Function**

The **PREV** function returns the property value of a previous event. The first parameter denotes the \(i\)th previous event in the order established by the data window. The second parameter is a property name for which the function returns the value for the previous event.

This example selects the value of the price property of the second previous event from the current **Trade** event.

```
SELECT PREV(2, price)
```

Because 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 three **Trade** events that had the largest, second-largest and third-largest volume.
SELECT PREV(0, symbol), PREV(1, symbol), PREV(2, symbol)
FROM Trade RETAIN 10 EVENTS WITH LARGEST volume

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 a 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, RankSelectionEvent RETAIN 10 EVENTS WITH LARGEST volume

The \(\text{PREV}\) 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 \(\text{PREV}\) function returns a null value for any events in which the previous event is not in the same batch of events. The \(\text{PRIOR}\) function as discussed below can be used if a null value is not the desired result.

SELECT PREV(1, symbol)
FROM Trade RETAIN BATCH OF 1 MINUTE

**Previous Event Per Group**

The combination of the \(\text{PREV}\) function and the \text{PARTITION BY} clause returns the property value for a previous event in the given group.

For example, assume we want to obtain the price of the previous event of the same symbol as the current event.

The statement that follows solves this problem. It partitions the window on the symbol property over a time window of one minute. As a result, when the engine encounters a new symbol value that it hasn’t seen before, it creates a new window specifically to hold events for that symbol. Consequently, the \(\text{PREV}\) function returns the previous event within the respective time window for that event’s symbol value.

SELECT PREV(1, price) AS prevPrice
FROM Trade RETAIN 1 MIN PARTITION BY symbol

**Restrictions**

The following restrictions apply to the \(\text{PREV}\) functions and its results:

- The function always returns a null value for remove stream (old data) events.
- The function may only be used on streams that are constrained by a \text{RETAIN} clause.

**The PRIOR Function**

The \(\text{PRIOR}\) function returns the property value of a prior event. The first parameter is an integer value that denotes the \(i^{th}\) prior event in the natural order of arrival. The second parameter is a property name for which the function returns the value for the prior event.

This example selects the value of the price property of the second prior event to the current Trade event.

SELECT PRIOR(2, price)
FROM Trade RETAIN ALL

The \(\text{PRIOR}\) function can be used on any event stream or view and does not require a stream to be constrained by a \text{RETAIN} clause as with the \(\text{PREV}\) function. The function operates based on the order of arrival of events in the event stream that provides the events.
The next statement uses a length batch window to compute an average volume for every 3 Trade events, posting results every 3 events. The SELECT clause employs the PRIOR function to select the current average and the average before the current average:

```
SELECT AVG(volume) AS avgVolume, AVG(PRIOR(3, volume))
FROM Trade RETAIN BATCH OF 3 EVENTS
```

**Comparison to the PREV Function**

The PRIOR function is similar to the PREV function. The key differences between the two functions are as follows:

- The PREV function returns previous events in the order provided by the window, while the PRIOR function returns prior events in the order of arrival in the stream.
- The PREV function requires a RETAIN clause while the PRIOR function does not.
- The PREV function returns the previous event taking into account any grouping, while the PRIOR function returns prior events regardless of any 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.

**The INSTANCEOF 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 INSTANCEOF 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 time.

The following 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 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 following 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 test, for example:

```
INSTANCEOF(product, org.myproducer.Product)
```

Valid parameters for the type parameter list are:

The `INSTANCEOF` function considers an event class's superclasses as well as all the directly or indirectly-implemented interfaces by superclasses.

### 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 and the second parameter is the type to cast to.

Valid parameters for the second (type) parameter are:

- Any of the Java built-in types: `int, long, byte, short, char, double, float, string`, where `string` is a short notation for `java.lang.String`. The type name is not case-sensitive. For example:
  ```
  cast(price, double)
  ```

- The fully-qualified class name of the class to cast to, for example:
  ```
  cast(product, org.myproducer.Product)
  ```

The `CAST` function is often used to provide a type for dynamic (unchecked) properties. Dynamic properties are properties whose type is not known at compile time. These properties are always of type `java.lang.Object`.

The following example shows how to use the `CAST` function to cast the `price` dynamic 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 utilizes `java.lang.Number` to convert numeric types, if required.

- For casts to `string` or `java.lang.String`, the `CAST` 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.

### 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.

Use the `EXISTS` function with dynamic (unchecked) properties. Dynamic properties are properties whose type is not known at compile time. Dynamic properties return a null value if the dynamic property does not exist on an event, or if the dynamic property exists but the value of the dynamic property is null.
The following example of using the EXISTS function 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
```

### 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:

```
SELECT SUM(price)
FROM StockTickEvent RETAIN 30 SECONDS
```

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, EPL generates a single value.

The 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 RETAIN 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.

The MIN and MAX aggregate functions are also available as single row functions. See Section , "The MIN and MAX Functions" for a description of this usage.

The syntax of the aggregation functions and the results they produce are shown in table below.

<table>
<thead>
<tr>
<th>Aggregate Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUM([ALL</td>
<td>DISTINCT] expression)</td>
</tr>
<tr>
<td>AVG([ALL</td>
<td>DISTINCT] expression)</td>
</tr>
<tr>
<td>COUNT([ALL</td>
<td>DISTINCT] expression)</td>
</tr>
<tr>
<td>COUNT(*)</td>
<td>Number of events, returning a value of long type.</td>
</tr>
<tr>
<td>MAX([ALL</td>
<td>DISTINCT] expression)</td>
</tr>
<tr>
<td>MIN([ALL</td>
<td>DISTINCT] expression)</td>
</tr>
</tbody>
</table>
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 "ACME" only, for the last 10 stock tick events regardless of their symbol.

```sql
SELECT 'ACME stats' AS title, AVG(price) AS avgPrice, SUM(price) AS sumPrice
FROM StockTickEvent RETAIN 10 EVENTS
WHERE symbol='ACME'
```

In the preceding example the length window of 10 elements is not affected by the `WHERE` clause, in other words, all events enter and leave the length window regardless of their symbol. If we only care about the last 10 `ACME` events, we need to add a `MATCHING` clause as shown below.

```sql
SELECT 'ACME stats' AS title, AVG(price) AS avgPrice, SUM(price) AS sumPrice
FROM (SELECT * FROM StockTickEvent WHERE symbol='ACME')
RETAIN 10 EVENT
```

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

- You can use `SUM`, `AVG`, `MEDIAN`, `STDDEV`, `AVEDEV` with numeric event properties only.

EPL 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.

You may also create an aggregate user-defined function. For more information, see Section, "User-Defined functions".

### User-Defined functions

A user-defined function can be invoked anywhere as an expression itself or within an expression. The function must simply be a public static method that the class loader can resolve at statement creation time. The engine resolves the function reference at statement creation time and verifies parameter types.

<table>
<thead>
<tr>
<th>Aggregate Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>`MEDIAN([ALL</td>
<td>DISTINCT] expression)`</td>
</tr>
<tr>
<td>`STDDEV([ALL</td>
<td>DISTINCT] expression)`</td>
</tr>
<tr>
<td>`AVEDEV([ALL</td>
<td>DISTINCT] expression)`</td>
</tr>
<tr>
<td><code>TREND(expression)</code></td>
<td>Number of consecutive up ticks (as positive number), down ticks (as negative number), or no change (as zero) for <code>expression</code>.</td>
</tr>
</tbody>
</table>

---

**Table 4–2 (Cont.) Aggregate Functions**

<table>
<thead>
<tr>
<th>Aggregate Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>`MEDIAN([ALL</td>
<td>DISTINCT] expression)`</td>
</tr>
<tr>
<td>`STDDEV([ALL</td>
<td>DISTINCT] expression)`</td>
</tr>
<tr>
<td>`AVEDEV([ALL</td>
<td>DISTINCT] expression)`</td>
</tr>
<tr>
<td><code>TREND(expression)</code></td>
<td>Number of consecutive up ticks (as positive number), down ticks (as negative number), or no change (as zero) for <code>expression</code>.</td>
</tr>
</tbody>
</table>
The example below assumes a class `MyClass` that exposes a public static method `myFunction` accepting two parameters, and returning a numeric type such as `double`.

```sql
SELECT 3 * MyClass.myFunction(price, volume) as myValue
FROM StockTick RETAIN 30 SECONDS
```

User-defined functions also take array parameters as this example shows. Section, "Array Definition Operator" outlines in more detail the types of arrays produced.

```sql
SELECT *
FROM RFIDEVENT RETAIN 10 MINUTES
WHERE com.mycompany.rfid.MyChecker.isInZone(zone, {10, 20, 30})
```

Oracle Event Processing supports both single-row and aggregate user-defined functions.

For more information, see "wlevs:function" in the *Oracle Fusion Middleware Developer’s Guide for Oracle Event Processing for Eclipse*. 

---

**Caution:** Only idempotent user-defined functions in EPL queries are allowed. That is, multiple invocations of the user-defined function does not change the result.
This chapter describes Java APIs for working with Event Processing Language (EPL). Note that EPL is deprecated; new development should use Oracle Continuous Query Language (Oracle CQL).

This chapter includes the following sections:
- Java Programming Interfaces

### Java Programming Interfaces

The Java programmatic interface for the EPL is rooted at the `com.bea.wlevs.ede.api.Processor` interface. This interface provides methods to load, compile, start, stop, and retrieve EPL statements.

EPL statements are loaded and compiled individually through the following method:

```java
Statement createStatement(String query) throws StatementException;
```

If the query fails to compile, a `StatementException` will be thrown. Alternatively, multiple statements may be loaded from a URL using the following method:

```java
List<Statement> loadStatements(URL location) throws MultiStatementException;
```

If the queries fail to compile, a `MultiStatementException` will be thrown. The structure of the rules file is explained in "Configuring EPL Processors" in the Oracle Fusion Middleware Developer’s Guide for Oracle Event Processing for Eclipse. Individual queries compiled through the `createStatement` are not persisted and have no effect on the rule files located at the URL location.

The `com.bea.wlevs.ede.api.Statement` interface allows event sinks to be attached to an EPL statement using the following method:

```java
void addEventSink(EventSink listener);
```

The engine calls the following method on the `com.bea.wlevs.ede.api.EventSink` interface when events are added to the output window as a result of executing the statement:

```java
void onEvent(List newEvents);
```

For more information, see the complete Oracle Fusion Middleware Java API Reference for Oracle Event Processing.