Oracle® Fusion Middleware

Developing Applications for Oracle CQL Data Cartridges

12*c* Release (12.2.1.3.0)
E98670-01
August 2018

Oracle Fusion Middleware Developing Applications for Oracle CQL Data Cartridges, 12*c* Release (12.2.1.3.0)

E98670-01

Primary Author: Oracle® Corporation

Contents

A Oracle Spatial Command and API Reference

Preface

A complete description of the Oracle Continuous Query Language (Oracle CQL), a query language based on SQL with added constructs that support streaming data. Using Oracle CQL, you can express queries on data streams to perform event processing. Oracle CQL is a new technology but it is based on a subset of SQL99 is provided.

Audience

This document is intended for all users of Oracle CQL.

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 that have purchased support 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:

- Known Issues for Oracle SOA and BPM Products at: http://www.oracle.com/technetwork/middleware/soasuite/documentation/soaknownissues122120-3111966.html.
- *Oracle Fusion Middleware Administering Oracle Stream Analytics*
- *Oracle Fusion Middleware Developing Applications for Event Processing with Oracle Stream Analytics*
- *Oracle Fusion Middleware Getting Started with Event Processing for Oracle Stream Analytics*
- *Oracle Fusion Middleware Schema Reference for Oracle Stream Analytics*
- *Oracle Fusion Middleware Using Visualizer for Oracle Stream Analytics*
- *Oracle Fusion Middleware Customizing Event Processing for Oracle Stream Analytics*
- *Oracle Fusion Middleware Oracle CQL Language Reference*

- *Oracle Fusion Middleware Java API Reference for Oracle Stream Analytics*
- *Oracle Fusion Middleware Using Oracle Stream Analytics*
- *Oracle Fusion Middleware Getting Started with Oracle Stream Analytics*
- SQL99 Specifications (ISO/IEC 9075-1:1999, ISO/IEC 9075-2:1999, ISO/IEC 9075-3:1999, and ISO/IEC 9075-4:1999)
- Oracle Stream Analytics Forum: `http://forums.oracle.com/forums/forum.jspa?forumID=820`

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Syntax Diagrams

Syntax descriptions are provided in this book for various Oracle CQL, SQL, PL/SQL, or other command-line constructs in graphic form or Backus Naur Form (BNF).

What's New in This Guide

Screens shown in this guide may differ from your implementation, depending on the skin used. Any differences are cosmetic.

The product has been renamed from Oracle Stream Explorer to in this release.

Sections	Changes Made
HBase Big Data Cartridge	New section that describes the HBase Data Cartridge

1
Introduction to Data Cartridges

Oracle Stream Explorer data cartridges extend Oracle Continuous Query Language (Oracle CQL) to support domain-specific abstract data types of the following forms: simple types, complex types, array types, and domain-specific functions.

This chapter includes the following sections:

1.1 Oracle CQL Data Cartridge Framework

The Oracle CQL data cartridge framework enables you to tightly integrate arbitrary domain data types and functions with the Oracle CQL language. The tight integration means that you can use the data cartridge extensions within Oracle CQL queries in the same way that you use Oracle CQL native types and built-in functions. The framework supports both simple and complex data types. Complex data types allow you to use object-oriented programming.

Currently, Oracle Stream Explorer provides the following data cartridges:

- Oracle JDBC data cartridge: This data cartridge allows you to incorporate arbitrary SQL functions against multiple tables and data sources in Oracle CQL queries and views as you would Oracle CQL native types.
- Oracle Spatial: This data cartridge exposes Oracle Spatial types, methods, fields, and constructors that you can use in Oracle CQL queries and views as you would Oracle CQL native types.
- Hadoop Big Data cartridge: This data cartridge extends an Oracle CQL processor to access large quantities of data in a Hadoop distributed file system (HDFS).
- NoSQLDB Big Data cartridge: This data cartridge extends an Oracle CQL processor to access large quantities of data in an Oracle NoSQL Database
- Oracle Java data cartridge: This data cartridge exposes Java types, methods, fields, and constructors that you can use in Oracle CQL queries and views as you would Oracle CQL native types.

1.2 Names

Each data cartridge is identified by a unique data cartridge name that defines a name space for the data cartridge implementation. Use the data cartridge name to disambiguate references to types, methods, fields, and constructors.

How you access data cartridge types, methods, fields, and constructors using Oracle CQL is the same for all data cartridge implementations. For example, you can reference a data-cartridge function with *`func_expr`*, which optionally takes a link name.

What you access in each data cartridge is unique to each data cartridge implementation. For more information, see:

- Oracle Java Data Cartridge
- Oracle Spatial Data Cartridge
- Oracle Big Data Cartridges.

> **Note:**
>
> To simplify Oracle data cartridge type names, you can use aliases as described in Oracle Fusion Middleware Oracle CQL Language Reference for Oracle Stream Analytics.

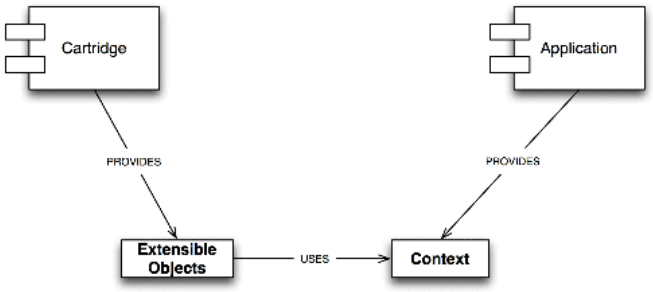
1.3 Application Context

Depending on the data cartridge implementation, you might be able to define an application context that the Oracle Stream Analytics server propagates to the functions and types that an instance of the data cartridge provides. For example, you might be able to configure an Oracle Stream Analytics server resource or a default data cartridge option and associate this application context information with a particular data cartridge instance.

Depending on the data cartridge implementation, you might be able to define an application context that the Oracle Stream Analytics server propagates to an instance of the data cartridge and the complex objects it provides.

The following figure illustrates this application context.

Figure 1-1 Data Cartridge Application Context



For example, you might be able to configure an Oracle Stream Analytics server resource or a default data cartridge option and associate this application context information with a particular data cartridge instance.

You define an application context for an instance of an Oracle Spatial data cartridge using a data cartridge implementation-provided element (call it *DATA_CARTRIDGE_CONTEXT*) in your Oracle Stream Analytics application's Event Processing Network (EPN) assembly file as the following example shows.

```
<DATA_CARTRIDGE_CONTEXT id="MyContext" ATTRIBUTE="" ... />
```

Where *DATA_CARTRIDGE_CONTEXT* is the name of the data cartridge implementation-provided element and *ATTRIBUTE* is one of one or more attributes that the data cartridge exposes for configuration.

In your Oracle CQL query, you use the `id` of the *DATA_CARTRIDGE_CONTEXT* (`MyContext` in the following example) in links instead of the *DATA_CARTRIDGE_NAME* alone. The Oracle Stream Analytics server will set the context object into the data cartridge instance before locating the data cartridge complex object.

> **Note:**
>
> The `id` value must not equal the *DATA_CARTRIDGE_NAME*.

In the following example, the default link (`@DATA_CARTRIDGE_NAME`) propagates the default application context to the `myMethod` call.

```
<view  id="view1">
  select com.mypackage.MyType.myMethod@DATA_CARTRIDGE_NAME( ... )
  from S[NOW]
</view>
```

In the following example, the link (`@MyContext`) propagates the user-defined application context to the `myMethod` call.

```
<view  id="view1">
  select com.mypackage.MyType.myMethod@MyContext(...)
  from S[NOW]
</view>
```

You can configure an application context for the following data cartridges:

- Oracle Spatial data cartridge
- Oracle JDBC data cartridge

2

Configure Oracle JDBC and Oracle Spatial Data Cartridges

How to configure the Oracle JDBC cartridge and Oracle Spatial cartridge, which extend Oracle Continuous Query Language (CQL) for use with Oracle Stream Explorer is described.

This chapter includes the following sections:

- How to Configure Oracle Spatial Application Context
- How to Configure Oracle JDBC Data Cartridge Application Context.

2.1 How to Configure Oracle Spatial Application Context

You define an application context for an instance of Oracle Spatial using element `spatial:context` in your Oracle Stream Analytics application's Event Processing Network (EPN) assembly file.

All constructors and methods from `com.oracle.cartridge.spatial.Geometry` and Oracle Spatial functions are aware of `spatial:context`. For example, the SRID is automatically set from the value in the Oracle Spatial application context.

For more information, see "SDO_SRID" in the *Oracle Spatial Developer's Guide* at: `http://download.oracle.com/docs/cd/E11882_01/appdev.112/e11830/sdo_objrelschema.htm#SPATL492`

To configure Oracle Spatial application context:

1. In Oracle JDeveloper, open the EPN diagram.
2. Import the package `com.oracle.cep.cartridge.spatial` into your Oracle Stream Analytics application's `MANIFEST.MF` file.
3. Right-click the EPN node and select **Configure Spatial Context** > **New Spatial Context**.
4. Edit the EPN file to add the required namespace and schema location entries as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:osgi="http://www.springframework.org/schema/osgi"
       xmlns:wlevs="http://www.bea.com/ns/wlevs/spring"
       xmlns:spatial="http://www.oracle.com/ns/ocep/spatial"
       xsi:schemaLocation="
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/osgi
  http://www.springframework.org/schema/osgi/spring-osgi.xsd
  http://www.bea.com/ns/wlevs/spring
  http://www.bea.com/ns/wlevs/spring/ocep-epn.xsd
```

```
http://www.oracle.com/ns/ocep/spatial
  http://www.oracle.com/ns/ocep/spatial/ocep-spatial.xsd">
```

5. Edit the EPN file to add a `spatial:context` element as follows.

```
<spatial:context id="SpatialGRS80" />
```

6. Assign a value to the `id` attribute that is unique in this EPN.

   This is the name you will use to reference this application context in subsequent Oracle CQL queries.

> **Note:**
>
> The `id` value must not equal the Oracle Spatial name `spatial`.

7. Configure the other attributes of the `spatial:context` element to suit your application requirements.

   Table 2-1 lists the attributes of the `spatial:context` element.

Table 2-1 spatial:context Element Attributes

Attribute	Description
`anyinteract-tolerance`	The default tolerance for contain or inside operator. Default: 0.0000005
`rof`	Defines the Reciprocal Of Flattening (`ROF`) parameter used for buffering and projection. Default: 298.257223563
`sma`	Defines the Semi-Major Axis (`SMA`) parameter used for buffering and projection. Default: 6378137.0
`srid`	SRID integer. Valid values are: • `CARTESIAN`: for cartesian coordinate system. • `LAT_LNG_WGS84_SRID`: for WGS84 coordinate system. • An integer value from the Oracle Spatial `SDO_COORD_SYS` table `COORD_SYS_ID` column. Default : `LAT_LNG_WGS84_SRID`
`tolerance`	The minimum distance to be ignored in geometric operations including buffering. Default: 0.000000001

   The following example shows how to create a spatial context named `SpatialGRS80` in an EPN assembly file using the Geodetic Reference System 1980 (GRS80) coordinate system (`srid="4269"`).

```
<spatial:context id="SpatialGRS80" srid="4269" sma="63787.0"
rof="298.25722101" />
```

8. Create Oracle CQL queries that reference this application context by name.

   The following example shows how to reference a `spatial:context` in an Oracle CQL query. In this case, the query uses link name `SpatialGRS80` to propagate this application context to the Oracle Spatial. The `spatial:context` attribute settings of

`SpatialGRS80` are applied to the `createPoint` method call. Because the application context defines the SRID, you do not need to pass that argument into the `createPoint` method.

```
<view  id="createPoint">
    select
com.oracle.cep.cartridge.spatial.Geometry.createPoint@SpatialGRS80(lng, lat, 0d)
    from CustomerPos[NOW]
</view>
```

2.2 How to Configure Oracle JDBC Data Cartridge Application Context

You define an application context for an instance of an Oracle JDBC data cartridge.

- A `jdbc:jdbc-context` element in the EPN assembly file.
- A `jc:jdbc-ctx` element in the component configuration file.

  The `jc:jdbc-ctx` element:

  – references one and only one `jdbc:jdbc-context`

  – references one and only one `data-source`

  – defines one or more SQL functions

> **Note:**
>
> You must provide alias names for every `SELECT` list column in the SQL function.

To configure Oracle JDBC data cartridge application context:

1. Open the EPN editor in the Oracle JDeveloper.
2. Right-click the EPN node and select **Configure Spatial Context** > **New Spatial Context**.
3. Edit the EPN file to add the required namespace and schema location entries as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:osgi="http://www.springframework.org/schema/osgi"
       xmlns:wlevs="http://www.bea.com/ns/wlevs/spring"
       xmlns:jdbc="http://www.oracle.com/ns/ocep/jdbc"
       xsi:schemaLocation="
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/osgi
  http://www.springframework.org/schema/osgi/spring-osgi.xsd
  http://www.bea.com/ns/wlevs/spring
 http://www.bea.com/ns/wlevs/spring/ocep-epn.xsd
  http://www.oracle.com/ns/ocep/jdbc
  http://www.oracle.com/ns/ocep/jdbc/ocep-jdbc.xsd">
```

4. Edit the EPN file to add a `jdbc:jdbc-context` element as follows.

```
<jdbc:jdbc-context id="JdbcCartridgeOne"/>
```

5. Assign a value to the `id` attribute that is unique in this EPN.

   This is the name you will use to reference this application context in subsequent Oracle CQL queries.

> **Note:**
>
> The `id` value must not equal the Oracle JDBC data cartridge name `jdbc`.

6. Right-click the desired processor and select **Go to Configuration Source**.
7. Edit the component configuration file to add the required namespace entries as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<wlevs:config
    xmlns:wlevs="http://www.bea.com/ns/wlevs/config/application"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jc="http://www.oracle.com/ns/ocep/config/jdbc
    xsi:schemaLocation="
       http://www.oracle.com/ns/ocep/config/jdbc
       http://www.oracle.com/ns/ocep/config/jdbc/ocep_jdbc_context_config.xsd">
```

8. Edit the component configuration file to add a `jc:jdbc-ctx` element as follows.

```
<jc:jdbc-ctx>
</jc:jdbc-ctx>
```

9. Add a name child element whose value is the name of the Oracle JDBC application context you defined in the EPN assembly file as follows.

```
<jc:jdbc-ctx>
    <name>JdbcCartridgeOne</name>
</jc:jdbc-ctx>
```

10. Add a `data-source` child element whose value is the name of a datasource defined in the Oracle Stream Analytics server `config.xml` file.

    The following example shows how to specify the data source named `StockDS`.

```
<jc:jdbc-ctx>
    <name>JdbcCartridgeOne</name>
    <data-source>StockDS</data-source>
</jc:jdbc-ctx>
```

11. Create one or more SQL functions using the `function` child element as follows.

```
<jc:jdbc-ctx>
    <name>JdbcCartridgeOne</name>
    <data-source>StockDS</data-source>
    <function name="getDetailsByOrderIdName">
        <param name="inpOrderId" type="int" />
        <param name="inpName" type="char" />
        <return-component-type>
            com.oracle.cep.example.jdbc_cartridge.RetEvent
        </return-component-type>
        <sql><![CDATA[
            SELECT
                Employee.empName as employeeName,
                Employee.empEmail as employeeEmail,
                OrderDetails.description as description
            FROM
                 PlacedOrders, OrderDetails , Employee
            WHERE
```

```
                PlacedOrders.empId = Employee.empId AND
                PlacedOrders.orderId = OrderDetails.orderId AND
                Employee.empName = :inpName AND
                PlacedOrders.orderId = :inpOrderId
        ></sql>
    </function>
</jc:jdbc-ctx>
```

> **Note:**
>
> You must provide alias names for every `SELECT` list column in the SQL query.

12. Create Oracle CQL queries that invoke the SQL functions using the Oracle JDBC data cartridge application context.

    The following example shows how to reference a `jdbc:jdbc-context` in an Oracle CQL query. In this case, the query uses link name `JdbcCartridgeOne` to propagate this application context to the Oracle JDBC data cartridge. The Oracle CQL query in invokes the function `getDetailsByOrderIdName` defined by Oracle JDBC data cartridge context `JdbcCartridgeOne`.

```
<processor>
        <name>Proc</name>
        <rules>
            <query id="q1"><![CDATA[
                RStream(
                   select
                     currentOrder.orderId,
                     details.orderInfo.employeeName,
                     details.orderInfo.employeeemail,
                     details.orderInfo.description
                   from
                     OrderArrival[now] as currentOrder,
                     TABLE(getDetailsByOrderIdName@JdbcCartridgeOne(
                               currentOrder.orderId, currentOrder.empName
                           ) as orderInfo
                     ) as details
                )
            ></query>
        </rules>
</processor>
```

3
Oracle JDBC Data Cartridge

You can use the Oracle Stream Explorer JDBC data cartridge to execute a SQL query against a database and use the returned results in a CQL query.

When using functionality provided by the cartridge, you are associating a SQL query with a JDBC cartridge function definition. Then, from a CQL query, you can call the JDBC cartridge function, which executes the associated SQL query against the database. The function call must be enclosed in the TABLE clause, which lets you use the SQL query results as a CQL relation in the CQL query making that function call.

> **Note:**
>
> Oracle recommends the Oracle JDBC data cartridge for accessing relational database tables from an Oracle CQL statement.

For information the TABLE clause, see Using the TABLE Clause.

This chapter includes the following sections:

- Understanding the Oracle Stream Explorer JDBC Data Cartridge
- Using the Event Processing JDBC Data Cartridge.

3.1 Understanding the Oracle Stream Explorer JDBC Data Cartridge

Oracle Stream Explorer streams contain streaming data, and a database typically stores historical data. Use the Oracle Stream Explorer JDBC data cartridge to associate historical data (stored in one or more tables) with the streaming data coming from Oracle Stream Explorer streams.

The Oracle Stream Explorer JDBC data cartridge executes arbitrary SQL query against a database and uses the results in the CQL query. This section describes how to associate streaming and historical data using the Oracle Stream Explorer JDBC data cartridge.

This section describes:

- Data Cartridge Name
- Scope
- Parameter Specification
- Oracle Stream Explorer JDBC Data Cartridge Application Context.

3.1.1 Data Cartridge Name

The Oracle Stream Explorer JDBC data cartridge uses the cartridge ID `com.oracle.cep.cartridge.jdbc`. This ID is reserved and cannot be used by any other cartridges.

For more information, see Oracle Stream Explorer JDBC Data Cartridge Application Context.

3.1.2 Scope

The Oracle Stream Analytics JDBC data cartridge supports arbitrarily complex SQL statements with the following restrictions:

- You can use only native SQL types in the `SELECT` list of the SQL query.
- You cannot use user-defined types and complex database types in the `SELECT` list.
- You can provide alias names for every `SELECT` list column in the SQL query. If you provide alias names, make sure the select list is consistent with the return type property names.

> **Note:**
>
> To use the Oracle Stream Analytics JDBC data cartridge, your data source must use Oracle JDBC driver version 11.2 or higher.

3.1.3 Parameter Specification

Use the `param` element to specify the parameters for JDBC functions. The parameters are specified as `name` and `value` pairs. The `name` attribute specifies event data of the specified `type`. The `type` attribute can be any Oracle CQL data type. See *Oracle Fusion Middleware Oracle CQL Language Reference* for information about Oracle CQL data types.

The following example shows an example configuration file that uses `param` and `type` pairs to specify parameters for the `getDetailsByOrderIdName` function.

> **Note:**
>
> The `RetEvent` class used in the example is an example of how to return a complex type as a table function. The full code for this class is shown in Using the Event Processing JDBC Data Cartridge.

```
...
<jc:jdbc-ctx>
    <name>JdbcCartridgeOne</name>
    <data-source>StockDS</data-source>
    <function name="getDetailsByOrderIdName">
        <param name="inpOrderId" type="int" />
        <param name="inpName" type="char" />
```

```
        <return-component-type>
            com.oracle.cep.example.jdbc_cartridge.RetEvent
        </return-component-type>
        <sql><![CDATA[
            SELECT
                Employee.empName as employeeName,
                Employee.empEmail as employeeEmail,
                OrderDetails.description as description
            FROM
                 PlacedOrders, OrderDetails , Employee
            WHERE
                PlacedOrders.empId = Employee.empId AND
                PlacedOrders.orderId = OrderDetails.orderId AND
                Employee.empName = :inpName AND
                PlacedOrders.orderId = :inpOrderId
        ></sql>
    </function>
</jc:jdbc-ctx>
...
```

3.1.4 Oracle Stream Explorer JDBC Data Cartridge Application Context

To use the Oracle Stream Explorer JDBC data cartridge, you must declare and configure one or more application-scoped JDBC cartridge context while developing an application, as described in the following steps:

- Declare a JDBC Cartridge Context in the EPN File
- Configure the JDBC Cartridge Context in the Application Configuration File.

3.1.4.1 Declare a JDBC Cartridge Context in the EPN File

To declare a JDBC cartridge context in the EPN file:

1. Edit your Oracle Stream Explorer application EPN assembly file to add the required namespace and schema location entries.
2. Add an entry with the tag `jdbc-context` in the EPN file and specify the `id` attribute. The `id` represents the name of this application-scoped context and is used in CQL queries that reference functions defined in this context. The `id` is also used when this context is configured in the application configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:osgi="http://www.springframework.org/schema/osgi"
       xmlns:wlevs="http://www.bea.com/ns/wlevs/spring"
       xmlns:jdbc="http://www.oracle.com/ns/ocep/jdbc"
       xsi:schemaLocation="
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/osgi
  http://www.springframework.org/schema/osgi/spring-osgi.xsd
  http://www.bea.com/ns/wlevs/spring
  http://www.bea.com/ns/wlevs/spring/http://www.bea.com/ns/wlevs/spring/ocep-epn.xsd
  http://www.oracle.com/ns/ocep/jdbc
  http://www.oracle.com/ns/ocep/jdbc/ocep-jdbc.xsd">
```

The following example shows how to create an Oracle Stream Explorer JDBC data cartridge application context named `JdbcCartridgeOne` in an EPN assembly file.

```
<jdbc:jdbc-context id="JdbcCartridgeOne"/>
```

3.1.4.2 Configure the JDBC Cartridge Context in the Application Configuration File

To configure the JDBC cartridge context, add the configuration details in the component configuration file that is typically placed under the application's /wlevs directory. This configuration is similar to configuring other EPN components such as channel and processor.

To configure the JDBC cartridge context in the application configuration file:

1. Before adding the JDBC context configuration, add the required namespace entry to the configuration XML file, as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<jdbcctxconfig:config xmlns:jdbcctxconfig="http://www.bea.com/ns/wlevs/config/
application"
    xmlns:jc="http://www.oracle.com/ns/ocep/config/jdbc">
```

2. The JDBC cartridge context configuration is done under the parent level tag `jdbc-ctx`. A context defines one or more functions, each of which is associated with a single SQL query. The configuration also specifies the data source representing the database against which the SQL queries are to be executed. Each function can have input parameters that are used to pass arguments to the SQL query defining the function, and each function specifies the return-component-type. Since the call to this function is always enclosed within a TABLE clause, the function always returns a Collection type. The return-component-type property indicates the type of the component of that collection.

   The value of the `name` property must match the value used for the `id` attribute in the EPN file.

> **Note:**
>
> The `RetEvent` class used in the example is an example of how to return a complex type as a table function. The full code for this class is shown in Using the Event Processing JDBC Data Cartridge.

```
...
<jc:jdbc-ctx>
    <name>JdbcCartridgeOne</name>
    <data-source>StockDS</data-source>
    <function name="getDetailsByOrderIdName">
        <param name="inpOrderId" type="int" />
        <param name="inpName" type="char" />
        <return-component-type>
            com.oracle.cep.example.jdbc_cartridge.RetEvent
        </return-component-type>
        <sql><![CDATA[
            SELECT
                Employee.empName as employeeName,
                Employee.empEmail as employeeEmail,
                OrderDetails.description as description
            FROM
                 PlacedOrders, OrderDetails , Employee
            WHERE
                PlacedOrders.empId = Employee.empId AND
                PlacedOrders.orderId = OrderDetails.orderId AND
```

```
                Employee.empName = :inpName AND
                PlacedOrders.orderId = :inpOrderId
        ></sql>
    </function>
</jc:jdbc-ctx>
...
<processor>
        <name>Proc</name>
        <rules>
            <query id="q1"><![CDATA[
                RStream(
                   select
                     currentOrder.orderId,
                     details.orderInfo.employeeName,
                     details.orderInfo.employeeemail,
                     details.orderInfo.description
                   from
                     OrderArrival[now] as currentOrder,
                     TABLE(getDetailsByOrderIdName@JdbcCartridgeOne(
                               currentOrder.orderId, currentOrder.empName
                           ) as orderInfo
                     ) as details
                )
            ></query>
        </rules>
</processor>
...
```

3.2 Using the Event Processing JDBC Data Cartridge

The different ways in which an Event Processing JDBC Data Cartridge can be used are explained.

In general, you use the Oracle Event Processing JDBC data cartridge as follows:

1. Declare and define an Oracle Event Processing JDBC cartridge application-scoped context.

   For more information, see Oracle Stream Explorer JDBC Data Cartridge Application Context.

2. Define one or more SQL statements in the `jc:jdbc-ctx` element in the component configuration file.

   For more information, see Defining SQL Statements: function Element.

3. If you specify the `function` element `return-component-type` child element as a Java bean, implement the bean and ensure that the class is on your Oracle Event Processing application classpath.

   The following example shows a typical implementation.

> **Note:**
>
> The `RetEvent` class is an example of how to return a complex type as a table function.

```
package com.oracle.cep.example.jdbc_cartridge;

public class RetEvent
{
```

```
    public String employeeName;
    public String employeeEmail;
    public String description;

    /* Default constructor is mandatory */
    public RetEvent1() {}

    /* May contain getters and setters for the fields */

    public String getEmployeeName() {
        return this.employeeName;
    }

    public void setEmployeeName(String employeeName) {
        this.employeeName = employeeName;
    }

    ...

    /* May contain other helper methods */

    public int getEmployeeNameLength() {
        return employeeName.length();
    }
}
```

You must declare the fields as public.

The return-component-type class for a JDBC cartridge context function must have a one-to-one mapping for fields in the SELECT list of the SQL query that defines the function. In other words, every field in the SELECT list of the SQL query defining a function must have a corresponding field (matching name) in the Java class that is declared to be the return-component-type for that function; otherwise Oracle Event Processing throws an error.

For more information, see return-component-type.

4. Define one or more Oracle CQL queries that call the SQL statements defined in the `jc:jdbc-ctx` element using the Oracle CQL `TABLE` clause and access the returned results by SQL `SELECT` list alias names.

   For more information, see Defining Oracle CQL Queries With the Oracle Stream Analytics JDBC Data Cartridge.

3.2.1 Defining SQL Statements: function Element

Within the `jc:jdbc-cxt` element in the component configuration file, you can define a JDBC cartridge context function using the `function` child element.

```
...
<jc:jdbc-ctx>
    <name>JdbcCartridgeOne</name>
    <data-source>StockDS</data-source>
    <function name="getDetailsByOrderIdName">
        <param name="inpOrderId" type="int" />
        <param name="inpName" type="char" />
        <return-component-type>
            com.oracle.cep.example.jdbc_cartridge.RetEvent
        </return-component-type>
        <sql><![CDATA[
            SELECT
                Employee.empName as employeeName,
```

```
                Employee.empEmail as employeeEmail,
                OrderDetails.description as description
            FROM
                 PlacedOrders, OrderDetails , Employee
            WHERE
                PlacedOrders.empId = Employee.empId AND
                PlacedOrders.orderId = OrderDetails.orderId AND
                Employee.empName = :inpName AND
                PlacedOrders.orderId = :inpOrderId
        ></sql>
    </function>
</jc:jdbc-ctx>
...
```

You may define one or more `function` elements within a given `jc:jdbc-cxt` element.

This section describes:

- function Element Attributes
- function Element Child Elements
- function Element Usage.

3.2.1.1 function Element Attributes

Each `function` element supports the attributes that Table 3-1 lists.

Table 3-1 function Element Attributes

Attribute	Description
`name`	The name of the JDBC cartridge context function. The combination of name and signature must be unique within a given Oracle Stream Analytics JDBC data cartridge application context. For more information, see Overloading JDBC Cartridge Context Functions.

3.2.1.2 function Element Child Elements

Each `function` element supports the following child elements:

- param
- return-component-type
- sql.

3.2.1.2.1 param

The `param` child element specifies an optional input parameter.

The SQL statement may take zero or more parameters. Each parameter is defined in a `param` element.

The `param` child element supports the attributes that Table 3-2 lists.

Table 3-2 param Element Attributes

Attribute	Description
`name`	The name of the input parameter. A valid parameter name is formed by a combination of A-Z,a-z,0-9 and _ (underscore).
`type`	The data type of the parameter.

Datatype Support – You may specify only Oracle CQL native `com.bea.wlevs.ede.api.Type` data types for the input parameter `param` element `type` attribute.

> **Note:**
>
> Datatype names are case sensitive. Use the case that the `com.bea.wlevs.ede.api.Type` class specifies.

For more information, see Table 3-3.

3.2.1.2.2 return-component-type

The `return-component-type` child element specifies the return type of the function. This child element is mandatory.

This represents the component type of the collection type returned by the JDBC data cartridge function. Because the function is always called from within an Oracle CQL `TABLE` clause, it always returns a collection type.

For more information, see Using the TABLE Clause.

Datatype Support – You may specify any one of the following types as the value of the `return-component-type` element:

- Oracle CQL native `com.bea.wlevs.ede.api.Type` datatype.
- Oracle CQL extensible Java cartridge type, such as a Java bean.

For more information, see:

- Table 3-3
- Oracle Java Data Cartridge.

3.2.1.2.3 sql

The `sql` child element specifies a SQL statement. This child element is mandatory.

Each `function` element may contain one and only one, single-line, SQL statement. You define the SQL statement itself within a `<![CDATA[>` block.

Within the SQL statement, you specify input parameters by `param` element `name` attribute using a colon (`:`) prefix.

> **Note:**
>
> You must provide alias names for every `SELECT` list column in the JDBC cartridge context function.

Datatype Support – Table 3-3 lists the SQL types you may use in your Oracle Stream Analytics JDBC data cartridge context functions and their corresponding Oracle Stream Analytics Java type and `com.bea.wlevs.ede.api.Type` type.

Table 3-3 SQL Column Types and Oracle Stream Analytics Type Equivalents

SQL Type	Oracle Stream Analytics Java Type	com.bea.wlevs.ede.api.Type
`NUMBER`	`java.math.BigDecimal`	bigdecimal
`NUMBER`	`long`	`bigint`
`RAW`	`byte[]`	`byte`
`CHAR, VARCHAR`	`java.lang.String`	`char`
`NUMBER`	`double`	`double`
`FLOAT, NUMBER`	`float`	`float`
`INTEGER, NUMBER`	`int`	`int`
`TIMESTAMP`	`java.sql.Timestamp`	`timestamp`

> **Note:**
>
> In cases where the size of the Java type exceeds that of the SQL type, your Oracle Stream Analytics application must restrict values to the maximum size of the SQL type. The choice of type to use on the CQL side should be driven by the range of values in the database column. For example, if the SQL column is a number that contains values in the range of integer, use the "int" type on CQL side. If you choose an incorrect type and encounter out-of-range values, Oracle Stream Analytics throws a numeric overflow error.

> **Note:**
>
> The Oracle Stream Analytics JDBC data cartridge does not support Oracle Spatial data types.

For more information, see function Element Usage.

3.2.1.3 function Element Usage

This section provides examples of different JDBC cartridge context functions you can define using the Oracle Stream Explorer JDBC data cartridge, including:

- Multiple Parameter JDBC Cartridge Context Functions
- Invoking PL/SQL Functions
- Complex JDBC Cartridge Context Functions
- Overloading JDBC Cartridge Context Functions.

3.2.1.3.1 Multiple Parameter JDBC Cartridge Context Functions

Using the Oracle Stream Explorer JDBC data cartridge, you can define JDBC cartridge context functions that take multiple input parameters.

The following example shows an Oracle Stream Explorer JDBC data cartridge application context that defines an JDBC cartridge context function that takes two input parameters.

```
...
<function name="getDetailsByOrderIdName">
    <param name="inpOrderId" type="int" />
    <param name="inpName" type="char" />
    <return-component-type>
        com.oracle.cep.example.jdbc_cartridge.RetEvent
    </return-component-type>
    <sql><![CDATA[
              SELECT
                    Employee.empName as employeeName,
                    Employee.empEmail as employeeEmail,
                    OrderDetails.description as description
              FROM
                     PlacedOrders, OrderDetails , Employee
              WHERE
                    PlacedOrders.empId = Employee.empId AND
                    PlacedOrders.orderId = OrderDetails.orderId AND
                    Employee.empName = :inpName AND
                    PlacedOrders.orderId = :inpOrderId
    ></sql>
</function>
...
```

3.2.1.3.2 Invoking PL/SQL Functions

Using the Oracle Stream Explorer JDBC data cartridge, you can define JDBC cartridge context functions that invoke PL/SQL functions that the database defines.

The following example shows an Oracle Stream Explorer JDBC data cartridge application context that defines a JDBC cartridge context function that invokes PL/SQL function `getOrderAmt`.

```
...
<function name="getOrderAmount">
    <param name="inpId" type="int" />
    <return-component-type>
        com.oracle.cep.example.jdbc_cartridge.RetEvent
    </return-component-type>
    <sql><![CDATA[
            SELECT getOrderAmt(:inpId) as orderAmt
            FROM dual
    ></sql>
</function>
...
```

3.2.1.3.3 Complex JDBC Cartridge Context Functions

Using the Oracle Stream Explorer JDBC data cartridge, you can define arbitrarily complex JDBC cartridge context functions including subqueries, aggregation, `GROUP BY`, `ORDER BY`, and `HAVING`.

The following example shows an Oracle Stream Explorer JDBC data cartridge application context that defines a complex JDBC cartridge context function.

```
...
<function name="getHighValueOrdersPerEmp">
    <param name="limit" type="int"/>
    <param name="inpName" type="char"/>
    <return-component-type>
        com.oracle.cep.example.jdbc_cartridge.RetEvent
    </return-component-type>
        <sql><![CDATA[
            select description as description,  sum(amt) as totalamt, count(*) as numTimes
            from  OrderDetails
                where orderid in (
                    select orderid from PlacedOrders where empid in (
                        select empid from Employee where  empName = :inpName
                    )
                )
            group by description
            having sum(amt) > :limit
        ></sql>
</function>
...
```

3.2.1.3.4 Overloading JDBC Cartridge Context Functions

Using the Oracle Stream Explorer JDBC data cartridge, you can define JDBC cartridge context functions with the same name in the same application context provided that each function has a unique signature.

The following example shows an Oracle Stream Explorer JDBC data cartridge application context that defines two JDBC cartridge context functions named `getDetails`. Each function is distinguished by a unique signature.

```
<jc:jdbc-ctx>
    <name>JdbcCartridgeOne</name>
    <data-source>StockDS</data-source>
         <function name="getDetails">
            <param name="inpName" type="char" />
            <return-component-type>
                com.oracle.cep.example.jdbc_cartridge.RetEvent
            </return-component-type>
            <sql><![CDATA[
                      SELECT
                             Employee.empName as employeeName,
                             Employee.empEmail as employeeEmail,
                             OrderDetails.description as description
                      FROM
                             PlacedOrders, OrderDetails , Employee
                      WHERE
                             PlacedOrders.empId = Employee.empId AND
                             PlacedOrders.orderId = OrderDetails.orderId AND
                             Employee.empName=:inpName
                      ORDER BY
                        description desc
            ></sql>
        </function>
        <function name="getDetails">
```

```
            <param name="inpOrderId" type="int" />
            <sql><![CDATA[ return-component-type
                      SELECT
                             Employee.empName as employeeName,
                             Employee.empEmail as employeeEmail,
                             OrderDetails.description as description
                      FROM
                             PlacedOrders, OrderDetails , Employee
                      WHERE
                             PlacedOrders.empId= Employee.empId AND
                             PlacedOrders.orderId = OrderDetails.orderId AND
                             PlacedOrders.orderId = :inpOrderId
            ></sql>
        </function>
</jc:jdbc-ctx>
```

3.2.2 Defining Oracle CQL Queries With the Oracle Stream Analytics JDBC Data Cartridge

This section describes how to define Oracle CQL queries that invoke SQL statements using the Oracle Stream Analytics JDBC data cartridge, including:

- Using SELECT List Aliases
- Using the TABLE Clause
- Using a Native CQL Type as a return-component-type.

3.2.2.1 Using SELECT List Aliases

Consider the Oracle Stream Explorer JDBC data cartridge context function.

```
<jc:jdbc-ctx>
    <name>JdbcCartridgeOne</name>
    <data-source>StockDS</data-source>
        <function name="getDetailsByOrderIdName">
            <param name="inpOrderId" type="int" />
            <param name="inpName" type="char" />
            <return-component-type>
                com.oracle.cep.example.jdbc_cartridge.RetEvent
            </return-component-type>
            <sql><![CDATA[
                    SELECT
                            Employee.empName as employeeName,
                            Employee.empEmail as employeeEmail,
                            OrderDetails.description as description
                    FROM
                            PlacedOrders, OrderDetails , Employee
                    WHERE
                            PlacedOrders.empId = Employee.empId AND
                            PlacedOrders.orderId = OrderDetails.orderId AND
                            Employee.empName = :inpName AND
                            PlacedOrders.orderId = :inpOrderId
            ></sql>
        </function>
</jc:jdbc-ctx>
```

You must assign an alias to each column in the `SELECT` list. When you invoke the JDBC cartridge context function in an Oracle CQL query, you access the columns in the result set by their SQL `SELECT` list aliases.

For more information, see Using the TABLE Clause.

3.2.2.2 Using the TABLE Clause

Consider the Oracle Stream Analytics JDBC data cartridge SQL statement.

```
...
<jc:jdbc-ctx>
    <name>JdbcCartridgeOne</name>
    <data-source>StockDS</data-source>
    <function name="getDetailsByOrderIdName">
        <param name="inpOrderId" type="int" />
        <param name="inpName" type="char" />
        <return-component-type>
            com.oracle.cep.example.jdbc_cartridge.RetEvent
        </return-component-type>
        <sql><![CDATA[
            SELECT
                Employee.empName as employeeName,
                Employee.empEmail as employeeEmail,
                OrderDetails.description as description
            FROM
                 PlacedOrders, OrderDetails , Employee
            WHERE
                PlacedOrders.empId = Employee.empId AND
                PlacedOrders.orderId = OrderDetails.orderId AND
                Employee.empName = :inpName AND
                PlacedOrders.orderId = :inpOrderId
        ></sql>
    </function>
</jc:jdbc-ctx>
...
```

The Oracle CQL query in the below example invokes the JDBC cartridge context function defined in the above example.

```
<processor>
        <name>Proc</name>
        <rules>
            <query id="q1"><![CDATA[
                RStream(
                   select
                     currentOrder.orderId,
                     details.orderInfo.employeeName,
                     details.orderInfo.employeeEmail,
                     details.orderInfo.description
                     details.orderInfo.getEmployeeNameLength()
                   from
                     OrderArrival[now] as currentOrder,
                     TABLE(getDetailsByOrderIdName@JdbcCartridgeOne(
                               currentOrder.orderId, currentOrder.empName
                           ) as orderInfo
                     ) as details
                )
            ></query>
        </rules>
</processor>
```

You must wrap the Oracle Stream Analytics JDBC data cartridge context function invocation in an Oracle CQL query `TABLE` clause.

You access the result set using:

TABLE_CLAUSE_ALIAS.JDBC_CARTRIDGE_FUNCTION_ALIAS.SQL_SELECT_LIST_ALIAS
or
TABLE_CLAUSE_ALIAS.JDBC_CARTRIDGE_FUNCTION_ALIAS.METHOD_NAME

Where:

- *TABLE_CLAUSE_ALIAS*: the outer AS alias of the TABLE clause.
- *JDBC_CARTRIDGE_FUNCTION_ALIAS*: the inner AS alias of the JDBC cartridge context function.
- *SQL_SELECT_LIST_ALIAS*: the JDBC cartridge context function SELECT list alias.
- *METHOD_NAME*: the name of the method that the return-component-type class provides.

You access the JDBC cartridge context function result set in the Oracle CQL query using:

```
details.orderInfo.employeeName
details.orderInfo.employeeEmail
details.orderInfo.description
details.orderInfo.getEmployeeNameLength()
```

The component type of the collection type returned by the JDBC data cartridge function is defined by the function element return-component-type child element. Because the function is always called from within an Oracle CQL TABLE clause, it always returns a collection type.

You can access both fields and methods of the return-component-type in an Oracle CQL query.

```
package com.oracle.cep.example.jdbc_cartridge;

public class RetEvent
{
    String employeeName;
    String employeeEmail;
    String description;

    /* Default constructor is mandatory */
    public RetEvent1() {}

    /* May contain getters and setters for the fields */

    public String getEmployeeName() {
        return this.employeeName;
    }

    public void setEmployeeName(String employeeName) {
        this.employeeName = employeeName;
    }

    ...

    /* May contain other helper methods */

    public int getEmployeeNameLength() {
        return employeeName.length();
    }
}
```

This class provides helper methods, like getEmployeeNameLength, that you can invoke within the Oracle CQL query.

For more information, see return-component-type.

3.2.2.3 Using a Native CQL Type as a return-component-type

Following is a JDBC cartridge context that defines a function that has a native CQL type `bigint` as return-component-type.

```
<jc:jdbc-ctx>
    <name>JdbcCartridgeOne</name>
    <data-source>myJdbcDataSource</data-source>
    <function name="getOrderAmt">
       <param name="inpId" type="int" />
       <return-component-type>bigint</return-component-type> <!-- native CQL as
return component type -->
       <sql><![CDATA[
                SELECT
                  getOrderAmt(:inpId) as orderAmt
                FROM (select :inpId as iid from
                  dual)>
       </sql>
    </function>
</jc:jdbc-ctx>
```

The following example shows how the `getOrderAmt` function in the above example can be used in a CQL query.

```
<query id="q1"><![CDATA[
                RStream(
                  select
                    currentOrder.orderId,
                    details.orderInfo as orderAmt
                  from
                    OrderArrival[now] as currentOrder,
                    TABLE(getOrderAmt@JdbcCartridgeTwo(currentOrder.orderId) as
orderInfo of bigint) as details
                )
 ></query>
```

Note that the alias `orderInfo` itself is of type `bigint` and can be accessed as `details.orderInfo as orderAmt` in the select list of the CQL query.

The "of bigint" clause used inside the TABLE construct is optional. If specified, the type mentioned should match the return-component-type.

4
Oracle Spatial Data Cartridge

A reference and guide to using the Oracle Spatial cartridge, which extends Oracle Continuous Query Language (Oracle CQL) to provide advanced spatial features for location-enabled applications is provided.

You can use Oracle Spatial types, methods, fields, and constructors in Oracle CQL queries and views as you would Oracle CQL native types when you create Oracle Stream Explorer applications.

This chapter includes the following sections:

- Understanding Oracle Spatial
- Using Oracle Spatial.

4.1 Understanding Oracle Spatial

Oracle Spatial is an Oracle Database option that provides advanced spatial features to support high-end geographic information systems (GIS) and location-enabled business intelligence solutions (LBS).

Oracle Spatial is an optional data cartridge that enables you to write Oracle CQL queries and views that seamlessly interact with Oracle Spatial classes in your Oracle Stream Explorer application.

With Oracle Spatial, you can configure Oracle CQL queries that perform the most important geographic domain operations such as storing spatial data, performing proximity and overlap comparisons on spatial data, and integrating spatial data with the Oracle Stream Explorer server by providing the ability to index on spatial data.

To use Oracle Spatial, you require a working knowledge of the Oracle Spatial API. For more information about Oracle Spatial, see:

- Oracle Spatial documentation: `http://www.oracle.com/pls/db112/portal.portal_db?selected=7&frame=#oracle_spatial_and_location_information`
- Oracle Spatial Java API reference: `http://download.oracle.com/docs/cd/E11882_01/appdev.112/e11829/toc.htm`

This section describes:

- Data Cartridge Name
- Scope
- Datatype Mapping
- Oracle Spatial Application Context.

4.1.1 Data Cartridge Name

Oracle Spatial uses the cartridge ID `com.oracle.cep.cartrdiges.spatial` and registers the server-scoped reserved link name `spatial`.

Use the `spatial` link name to associate an Oracle Spatial method call with the Oracle Spatial application context.

For more information, see:

- Oracle Spatial Application Context
- Geometry API.

4.1.2 Scope

Oracle Spatial is based on the Oracle Spatial Java API. Oracle Spatial exposes Oracle Spatial functionality in the `com.oracle.cep.cartridge.spatial.Geometry` class. Oracle Spatial functionality that is not in the Oracle Spatial Java API is not accessible from Oracle Spatial.

Using Oracle Spatial, your Oracle CQL queries can access the Oracle Spatial functionality that Table 4-1 describes.

Table 4-1 Oracle Spatial Scope

Oracle Spatial Feature	Scope
Geometry Types	The following geometry types from the Oracle Spatial Java API: • 2D points. • 2D circles, which support the Cartesian coordinate system and the geodetic (geographical) coordinates. • 2D simple polygons. • 2D rectangles. • Compound 2D geometries, which includes compound line strings and compound polygons. • 3D geometries, excluding 3D circles and compound 3D geometries. You can create a compound 3D geometry with the `Geometry3D.createGeometry` generic method. Be aware that spatial operations on the resulting compound 3D object raise an exception. • Solid (filled) 3D geometries The following geometry operations: • Creating geometry types • Accessing geometry type public member functions and public fields • Inside and contain operations on all 2D geometry objects. A 2D geometry object is inside when all of its points are within an outer geometry without touching any of the outer geometry boundaries. • Spatial operations between any two types of 2D geometries. You can execute spatial operations on any two arbitrary 2D geometries such as check whether a rectangle is inside a polygon. Note that any geometry that consists of arcs such as a compound polygon must use a non-zero tolerance to densify its arcs first. • Spatial operations on the following 3D geometries: 3D points, 3D lines, 3D rectangles, and 3D polygons. For more information, see: • Geometry Types • Element Info Array
Coordinate Systems	• Cartesian and WGS84 geodetic coordinates (default) • Specifying the default coordinate system through SRID • Using other geodetic coordinates For more information, see Ordinates and Coordinate Systems and the SDO_SRID.
Geometric Index	• R-Tree For more information, see Geometric Index.

Table 4-1 (Cont.) Oracle Spatial Scope

Oracle Spatial Feature	Scope
Geometric Relation Operators	• ANYINTERACT • CONTAIN • INSIDE • INSIDE3D • WITHINDISTANCE For more information, see Geometric Relation Operators.
Geometric Filter Operators	• FILTER • NN For more information, see Geometric Filter Operators.
Geometry API	For a complete list of the methods that `com.oracle.cep.cartridge.spatial.Geometry` provides, see Geometry API.
Geometric Aggregations	• `MBR` (minimum bounding rectangle) For more information, see Geometric Aggregations.

For more information on how to access these Oracle Spatial features using Oracle Spatial, see Using Oracle Spatial.

4.1.2.1 Geometry Types

The Oracle Spatial data model consists of geometries. A geometry is an ordered sequence of vertices. The semantics of the geometry are determined by its type. Oracle Spatial enables you to access the following Oracle Spatial types directly in Oracle CQL queries and views:

- `SDO_GTYPES`: Oracle Spatial supports the following geometry types:
  - 2D points
  - 2D simple polygons
  - 2D rectangles
  - 3D points
  - 3D lines
  - 3D rectangles
  - 3D polygons

  Table 4-2 describes the geometry types from the `com.oracle.cep.cartridge.spatial.Geometry` class that you can use.

Table 4-2 Oracle Spatial Geometry Types

Geometry Type	Description
`GTYPE_POINT`	Point geometry type that contains one point.

Table 4-2 (Cont.) Oracle Spatial Geometry Types

Geometry Type	Description
GTYPE_CURVE	Curve geometry type that contains one line string that can contain straight or circular arc segments, or both. LINE and CURVE are synonymous in this context.
GTYPE_POLYGON	Polygon geometry type that contains one polygon.
GTYPE_SURFACE	Polygon or surface geometry type that contains one polygon with or without holes or one surface consisting of one or more polygons. In a three-dimensional polygon, all points must be on the same plane.
GTYPE_COLLECTION	Collection geometry type that is a heterogeneous collection of elements. COLLECTION is a superset that includes all other types.
GTYPE_MULTIPOINT	Multipoint geometry type that has one or more points. MULTIPOINT is a superset of POINT.
GTYPE_MULTICURVE	Multiline or multicurve geometry type that has one or more line strings. MULTILINE and MULTICURVE are synonymous in this context, and each is a superset of both LINE and CURVE.
GTYPE_MULTIPOLYGON GTYPE_MULTISURFACE	Multipolygon or multisuraface geometry type that can have multiple, disjoint polygons (more than one exterior boundary) or surfaces. MULTIPOLYGON is a superset of POLYGON, and MULTISURFACE is a superset of SURFACE.
GTYPE_SOLID	Solid geometry that consists of multiple surfaces and is completely enclosed in a three-dimensional space. Can be a cuboid or a frustum.
GTYPE_MULTISOLID	Multisolid geometry that consists of multiple, disjoint solids (more than one exterior boundary). MULTISOLID is a superset of SOLID.

- SDO_ELEMENT_INFO: You can create the Element Info array using:
  - com.oracle.cep.cartridge.spatial.Geometry.createElemInfo static method
  - einfogenerator function

  For more information, see Element Info Array.
- ORDINATES: You can create the ordinates using the Oracle Spatial ordsgenerator function.

  For more information, see Ordinates and Coordinate Systems and the SDO_SRID.

For more information, see:

- How to Access Oracle Spatial Java API Geometry Types
- How to Create a Geometry
- How to Access Geometry Type Public Methods and Fields .

4.1.2.2 Element Info Array

The Element Info attribute is defined using a varying length array of numbers. This attribute specifies how to interpret the ordinates stored in the Ordinates attribute.

Oracle Spatial provides the following helper function for generating Element Info attribute values:

```
com.oracle.cep.cartridge.spatial.Geometry.createElemInfo(int SDO_STARTING_OFFSET,
int SDO_ETYPE , int SDO_INTERPRETATION)
```

You can also use the `einfogenerator` function.

For more information, see:

- createElemInfo
- einfogenerator.

4.1.2.3 Ordinates and Coordinate Systems and the SDO_SRID

Table 4-3 lists the coordinate systems that Oracle Spatial supports by default and the `SDO_SRID` value that identifies each coordinate system.

Table 4-3 Oracle Spatial Coordinate Systems

Coordinate System	SDO_SRID	Description
Cartesian	0	Cartesian coordinates are coordinates that measure the position of a point from a defined origin along axes that are perpendicular in the represented space.
Geodetic (WGS84)	8307	Geodetic coordinates (sometimes called geographic coordinates) are angular coordinates (longitude and latitude), closely related to spherical polar coordinates, and are defined relative to a particular Earth geodetic datum. This is the default coordinate system in Oracle Spatial.

You can specify the `SDO_SRID` value as an argument to each Oracle Spatial method and constructor you call or you can configure the `SDO_SRID` in the Oracle Spatial application context once and use `com.oracle.cep.cartridge.spatial.Geometry` methods without having to set the `SDO_SRID` as an argument each time. Using the application context, you can also specify any coordinate system that Oracle Spatial supports.

> **Note:**
>
> If you use a `com.oracle.cep.cartridge.spatial.Geometry` method that does not take an `SDO_SRID` value, then you must use the Oracle Spatial application context. For example, the following method call causes a runtime exception:
>
> ```
> com.oracle.cep.cartridge.spatial.Geometry.createPoint(lng, lat)
> ```
>
> Instead, you must use the `spatial` link name to associate the method call with the Oracle Spatial application context:
>
> ```
> com.oracle.cep.cartridge.spatial.Geometry.createPoint@spatial(lng, lat)
> ```
>
> If you use a `Geometry` method that takes an `SDO_SRID` value, then the use of the `spatial` link name is optional. For example, both the following method calls are valid:
>
> ```
> com.oracle.cep.cartridge.spatial.Geometry.createPoint(8307, lng, lat)
> com.oracle.cep.cartridge.spatial.Geometry.createPoint@spatial(lng, lat)
> ```
>
> For more information, see Oracle Spatial Application Context.

Ordinates define the array of coordinates for a geometry using a double array. Oracle Spatial provides the `ordsgenerator` helper function for generating the array of coordinates. For syntax, see ordsgenerator.

For more information, see:

- How to Use the Default Geodetic Coordinates
- How to Use Other Geodetic Coordinates .

4.1.2.4 Geometric Index

Oracle Spatial uses a spatial index to implement the primary filter. The purpose of the spatial index is to quickly create a subset of the data and reduce the processing burden on the secondary filter.

A spatial index, like any other index, provides a mechanism to limit searches, but in this case the mechanism is based on spatial criteria such as intersection and containment.

Oracle Spatial uses R-Tree indexing for the default indexing mechanism. A spatial R-tree index can index spatial data of up to four dimensions. An R-tree index approximates each geometry by a single rectangle that minimally encloses the geometry (called the Minimum Bounding Rectangle, or MBR)

For more information, see: Geometric Filter Operators.

4.1.2.5 Geometric Relation Operators

Oracle Spatial supports the following Oracle Spatial geometric relation operators:

- ANYINTERACT
- CONTAIN

- INSIDE
- INSIDE3D
- WITHINDISTANCE

You can use any of these operators in either the Oracle CQL query projection clause or where clause.

When you use a geometric relation operator in the where clause of an Oracle CQL query, Oracle Spatial enables Rtree indexing on the relation specified in the where clause.

Oracle Spatial supports only geometric relations between point and other geometry types.

For more information, see How to Use Geometry Relation Operators .

4.1.2.6 Geometric Filter Operators

Oracle Spatial supports the following Oracle Spatial geometric filter operators:

- FILTER
- NN

These filter operators perform primary filtering and so they may only appear in an Oracle CQL query where clause.

These filter operators use the spatial index to identify the set of spatial objects that are likely to interact spatially with the given object.

For more information, see:

- Geometric Index
- How to Use Geometry Filter Operators .

4.1.2.7 Geometric Aggregations

The geometry aggregation operator `MBR` may only appear in an Oracle CQL query projection clause.

For more information, see, How to Use Geometry Aggregate Operators .

4.1.2.8 Geometry API

Oracle Spatial is based on the Oracle Spatial Java API. Oracle Spatial exposes Oracle Spatial functionality in the `com.oracle.cep.cartridge.spatial.Geometry` class. This `Geometry` class also extends `oracle.spatial.geometry.J3D_Geometry`. Oracle Spatial supports 2D and 3D geometries and automatically zero-pads the Z coordinates for `J3D_Geometry` methods.

Oracle Spatial functionality inaccessible from the `Geometry` class (or not conforming to the scope and geometry types that Oracle Spatial supports) is inaccessible from Oracle Spatial.

This section describes:

- com.oracle.cep.cartridge.spatial.Geometry Methods

- oracle.spatial.geometry.JGeometry Methods

For more information, see:

- Scope
- ordsgenerator

4.1.2.8.1 com.oracle.cep.cartridge.spatial.Geometry Methods

Table 4-4 lists the public methods that the `Geometry` class provides.

Table 4-4 Oracle Spatial Geometry Methods

Type	Method
Buffers	• buffer • bufferPolygon
Circles	• createCircle
Conversions	• convertTo2D • convertTo3D
Distance	• distance
Element information	• createElemInfo
Geometries	• createGeometry
Linear line and multi line strings	• createLinearLineString • createLinearMultiLineString
Linear polygons	• createLinearPolygon
Minimum Bounding Rectangle (MBR)	• get2dMbr
Points	• createMultiPoint • createPoint
Rectangles	• createRectangle
Type and type conversion	• createGeometry • to_J3D_Geometry • to_JGeometry

> **Note:**
>
> `Geometry` class methods are case sensitive and you must use them in the case shown.

4.1.2.8.2 oracle.spatial.geometry.JGeometry Methods

The following `JGeometry` public methods are applicable to Oracle Spatial:

- `double area(double tolerance)`: returns the total planar surface area of a 2D geometry.
- `double length(double tolerance)`: returns the perimeter of a 2D geometry. All edge lengths are added.

- `double[] getMBR()`: returns the Minimum Bounding Rectangle (MBR) of this geometry. It returns a double array containing the `minX`, `minY`, `maxX`, and `maxY` value of the MBR for 2D.

For more information, see:

- http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28401/oracle/spatial/geometry/JGeometry.html
- http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28401/oracle/spatial/geometry/J3D_Geometry.html

4.1.3 Datatype Mapping

The Oracle Spatial cartridge supports one data type: `com.oracle.cep.cartridge.spatial.Geometry`.

The Geometry class extends `oracle.spatial.geometry.J3D_Geometry` and supports all the public methods, fields, and constructors that `J3D_Geometry` and its parent class `oracle.spatial.geometry.JGeometry` provide.

For a complete list of the methods that `com.oracle.cep.cartridge.spatial.Geometry` provides, see Geometry API.

4.1.4 Oracle Spatial Application Context

You can define an application context for an instance of Oracle Spatial and propagate this application context at runtime. This allows you to associate specific Oracle Spatial application defaults (such as an `SDO_SRID`) with a particular Oracle Spatial instance.

Before you can define an Oracle Spatial application context, edit your Oracle Stream Analytics application EPN assembly file to add the required namespace and schema location entries:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:osgi="http://www.springframework.org/schema/osgi"
       xmlns:wlevs="http://www.bea.com/ns/wlevs/spring"
       xmlns:spatial="http://www.oracle.com/ns/ocep/spatial"
       xsi:schemaLocation="
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/osgi
  http://www.springframework.org/schema/osgi/spring-osgi.xsd
  http://www.bea.com/ns/wlevs/spring
  http://www.bea.com/ns/wlevs/spring/spring-wlevs-v11_1_1_6.xsd"
  http://www.oracle.com/ns/ocep/spatial
  http://www.oracle.com/ns/ocep/spatial/ocep-spatial.xsd">
```

The following example shows how to create a spatial context named `SpatialGRS80` in an EPN assembly file using the Geodetic Reference System 1980 (GRS80) coordinate system.

```
<spatial:context id="SpatialGRS80" srid="4269" sma="6378137" rof="298.25722101" />
```

The following example shows how to reference a `spatial:context` in an Oracle CQL query. In this case, the query uses link name `SpatialGRS80` (defined in the above

example) to propagate this application context to Oracle Spatial. The `spatial:context` attribute settings of `SpatialGRS80` are applied to the `createPoint` method call.

```
<view  id="createPoint">
    select com.oracle.cep.cartridge.spatial.Geometry.createPoint@SpatialGRS80(
        lng, lat)
    from CustomerPos[NOW]
</view>
```

4.2 Using Oracle Spatial

Common use-cases that highlight how you can use Oracle Spatial in your Oracle Stream Explorer applications are described.

- How to Access Oracle Spatial Java API Geometry Types
- How to Create a Geometry
- How to Access Geometry Type Public Methods and Fields
- How to Use Geometry Relation Operators
- How to Use Geometry Filter Operators
- How to Use Geometry Aggregate Operators
- How to Use the Default Geodetic Coordinates
- How to Use Other Geodetic Coordinates

For more information, see Geometry API.

4.2.1 How to Access Oracle Spatial Java API Geometry Types

This procedure describes how to access Oracle Spatial geometry types `SDO_GTYPE`, `SDO_ELEMENT_INFO`, and `ORDINATES` using Oracle Spatial in an Oracle CQL query.

To access the geometry types that the Oracle Spatial Java API supports:

1. Import the package `com.oracle.cep.cartridge.spatial` into your Oracle Stream Analytics application's `MANIFEST.MF` file.
2. Define your Oracle Stream Analytics application event type using the appropriate Oracle Spatial data types.

   The following example shows how to define event type `MySpatialEvent` with two event properties `x` and `y` of type `com.oracle.cep.cartridge.spatial.Geometry`.

   ```
   <wlevs:event-type-repository>
       <wlevs:event-type type-name="MySpatialEvent">
           <wlevs:properties>
               <wlevs:property name="x" type="com.oracle.cep.cartridge.spatial.Geometry"/>
               <wlevs:property name="y" type="com.oracle.cep.cartridge.spatial.Geometry"/>
           </wlevs:properties>
       </wlevs:event-type>
   </wlevs:event-type-repository>
   ```

   You can use these event properties in an Oracle CQL query like this:

   ```
   CONTAIN@spatial(x, y, 20.0d)
   ```

   For more information, see *Oracle Fusion Middleware Developing Applications for Event Processing with Oracle Stream Analytics*.

3. Choose an `SDO_GTYPE`, for example, `GTYPE_POLYGON`.

   For more information, see Geometry Types.

4. Choose the Element Info appropriate for your ordinates.

   For more information, see Element Info Array

5. Define your coordinate values.

   For more information, see Ordinates and Coordinate Systems and the SDO_SRID.

6. Create your Oracle CQL query as the following example shows.

```
view id="ShopGeom">
    select  com.oracle.cep.cartridge.spatial.Geometry.createGeometry@spatial(
        com.oracle.cep.cartridge.spatial.Geometry.GTYPE_POLYGON,
        com.oracle.cep.cartridge.spatial.Geometry.createElemInfo(1, 1003, 1),
        ordsgenerator@spatial(
            lng1, lat1, lng2, lat2, lng3, lat3,
            lng4, lat4, lng5, lat5, lng6, lat6
        )
    ) as geom
    from ShopDesc
</view>
```

4.2.2 How to Create a Geometry

You can use Oracle Spatial to create a geometry in an Oracle CQL query by invoking:

- static methods in `com.oracle.cartridge.spatial.Geometry`
- methods in `oracle.spatial.geometry.JGeometry` that conform to the scope and geometry types that Oracle Spatial supports.
- constructor methods in `oracle.spatial.geometry.J3D_Geometry`
- static methods from `oracle.spatial.geometry.J3D_Geometry`

For more information, see Geometry API.

Using a Static Method in the Oracle Spatial Geometry Class

The following example shows how to create a point geometry using a static method in `com.oracle.cartridge.spatial.Geometry`. In this case, you must use a link (`@spatial`) to identify the data cartridge that provides this class. The advantage of using this approach is that the Oracle Spatial application context is applied to set the SRID and other Oracle Spatial options, either by default or based on an application context you configure (see Oracle Spatial Application Context).

```
<view  id="CustomerPosGeom">
    select com.oracle.cep.cartridge.spatial.Geometry.createPoint@spatial(
        lng, lat) as geom
    from CustomerPos[NOW]
</view>
```

Using an Oracle Spatial J3D_Geometry Constructor

The following example shows how to create a geometry using a constructor method in `oracle.spatial.geometry.J3D_Geometry`. In this case, you do not use a link (`@spatial`) because `J3D_Geometry` is just a Java class. The disadvantage of this approach compared with using `com.oracle.cartridge.spatial.Geometry` is that you must set the SRID because no application context is available.

```
<view  id="CustomerPosGeom">
    select oracle.spatial.geometry.J3D_Geometry(
        oracle.spatial.geometry.GTYPE_POINT, srid, x, y, z) as geom
    from CustomerPos[NOW]
</view>
```

Using a Static Method in the Oracle Spatial J3D_Geometry

The following example shows how to create a geometry using a static method in `oracle.spatial.geometry.J3D_Geometry`.

```
<view  id="CustomerPosGeom">
    select  oracle.spatial.geometry.J3D_Geometry.createArc@spatial(
        x1, y1, x2, y2, x3, y3) as geom
    from CustomerPos[NOW]
</view>
```

For more information, see Geometry Types.

4.2.3 How to Access Geometry Type Public Methods and Fields

Using Oracle Spatial, you can access the public member functions and public member fields of Oracle Spatial classes directly in Oracle CQL.

Oracle Spatial functionality inaccessible from the `Geometry` class (or not conforming to the scope and geometry types that Oracle Spatial supports) is inaccessible from Oracle Spatial.

In the following example, the view `ShopGeom` creates an Oracle Spatial geometry called `geom`. The view `shopMBR` calls `JGeometry` static method `getMBR` which returns a `double[]` as stream element `mbr`. The query `qshopMBR` accesses this `double[]` using regular Java API.

```
<view id="ShopGeom">
    select  com.oracle.cep.cartridge.spatial.Geometry.createGeometry@spatial(
        com.oracle.cep.cartridge.spatial.Geometry.GTYPE_POLYGON,
        com.oracle.cep.cartridge.spatial.Geometry.createElemInfo(1, 1003, 1),
        ordsgenerator@spatial(
            lng1, lat1, lng2, lat2, lng3, lat3,
            lng4, lat4, lng5, lat5, lng6, lat6
        )
    ) as geom
    from ShopDesc
</view>
<view  id="shopMBR">
    select geom.getMBR() as mbr
    from ShopGeom
</view>
<query  id="qshopMBR">
    select mbr[0], mbr[1], mbr[2], mbr[3]
    from shopMBR
</query>
```

For more information, see:

- Geometry Types
- Oracle Java Data Cartridge.

4.2.4 How to Use Geometry Relation Operators

Using Oracle Spatial, you can access the following Oracle Spatial geometry relation operators in either the `WHERE` or `SELECT` clause of an Oracle CQL query:

- ANYINTERACT
- CONTAIN
- INSIDE
- INSIDE3D
- WITHINDISTANCE

In the following example, the view `op_in_where` uses the `CONTAIN` geometry relation operator in the `WHERE` clause: in this case, Oracle Spatial uses R-Tree indexing. The view `op_in_proj` uses `CONTAIN` in the `SELECT` clause.

```
<view id="op_in_where">
    RStream(
        select
            loc.customerId,
            shop.shopId
        from
            LocGeomStream[NOW] as loc,
            ShopGeomRelation as shop
        where
            CONTAIN@spatial(shop.geom, loc.curLoc, 5.0d) = true
    )
</view>
<view id="op_in_proj">
    RStream(
        select
            loc.customerId,
            shop.shopId,
            CONTAIN@spatial(shop.geom, loc.curLoc, 5.0d)
        from
            LocGeomStream[NOW] as loc,
            ShopGeomRelation as shop
    )
</view>
```

For more information, see Geometric Relation Operators.

4.2.5 How to Use Geometry Filter Operators

Using Oracle Spatial, you can access the following Oracle Spatial geometry filter operators in the `WHERE` clause of an Oracle CQL query:

- FILTER
- NN

In the following example, the view `filter` uses the `FILTER` geometry filter operator in the `WHERE` clause.

```
<view id="filter">
    RStream(
        select loc.customerId, shop.shopId
        from LocGeomStream[NOW] as loc, ShopGeomRelation as shop
```

```
        where FILTER@spatial(shop.geom, loc.curLoc, 5.0d) = true
    )
</view>
```

For more information, see Geometric Filter Operators.

4.2.6 How to Use Geometry Aggregate Operators

Using the Oracle Spatial data cartridge, you can access the following Oracle Spatial aggregate operators in the SELECT clause of an Oracle CQL query:

- `MBR`

In the following example, the view `vaggrmbr` uses the `MBR` geometry aggregate operator in the `SELECT` clause. The query `qaggrmbr` access the `double[]` returned by the `MBR` geometry aggregate operator directly using standard Java API.

```
<view id="vaggrmbr">
    select MBR@spatial1(shop.geom) as mbr
    from ShopGeomRelation as shop
</view>
<query id="qaggrmbr">
    select mbr[0], mbr[1], mbr[2], mbr[3], mbr[4], mbr[5], mbr[6]
    from vaggrmbr
</query>
```

For more information, see Geometric Filter Operators.

4.2.7 How to Use the Default Geodetic Coordinates

When you create an Oracle CQL query using the default Oracle Spatial application context, the default `SRID` will be set to `CARTESIAN`.

The following example shows, the `createPoint` method call uses the default link (`@spatial`). This guarantees that the default Oracle Spatial application context is applied.

```
<view  id="createPoint">
    select com.oracle.cep.cartridge.spatial.Geometry.createPoint@spatial(
        lng, lat)
    from CustomerPos[NOW]
</view>
```

For more information, see:

- Oracle Spatial Application Context
- Ordinates and Coordinate Systems and the SDO_SRID.

4.2.8 How to Use Other Geodetic Coordinates

This procedure describes how to use the Oracle Spatial application context to specify a geodetic coordinate system other than the default Cartesian geodetic coordinate system in an Oracle CQL query:

For more information, see:

- Oracle Spatial Application Context

- Ordinates and Coordinate Systems and the SDO_SRID.

To use other geodetic coordinates:

1. Create an Oracle Spatial application context and define the `srid` attribute for the geodetic coordinate system you want to use.

   The following example shows how to create a spatial context named `SpatialGRS80` in an EPN assembly file using the Geodetic Reference System 1980 (GRS80) coordinate system.

   ```
   <spatial:context id="SpatialGRS80" srid="4269" sma="6378137"
   rof="298.25722101" />
   ```

2. In your Oracle CQL query, use the id of this `spatial:context` in your links.

   The following example shows how to reference a `spatial:context` in an Oracle CQL query. In this case, the query uses link name `SpatialGRS80` to propagate this application context to Oracle Spatial. The `spatial:context` attribute settings of `SpatialGRS80` are applied to the `createPoint` method call.

   ```
   <view  id="createPoint">
       select com.oracle.cep.cartridge.spatial.Geometry.createPoint@SpatialGRS80(
           lng, lat)
       from CustomerPos[NOW]
   </view>
   ```

5

Oracle Big Data Cartridges

Oracle Stream Explorer supports Big Data with the Hadoop, NoSQLDB, and HBase cartridges. Hadoop cartridge is extension for an Oracle CQL processor to access large quantities of data in a Hadoop distributed file system (HDFS). HDFS is a non-relational data store. NoSQL cartridge is extension for an Oracle CQL processor to access large quantities of data in an Oracle NoSQL database. The Oracle NoSQL database stores data in key-value pairs. HBase cartridge is extension for an Oracle CQL processor to access large quantities of data in a HBase database. HBase is a distributed column-oriented database built on top of the Hadoop file system.

This chapter includes the following sections:

5.1 What is Big Data?

Big Data is huge and complex data sets for which the traditional data processing applications are insufficient. *Big Data* describes a holistic information management strategy that includes and integrates many new types of data and data management alongside traditional data.

Big data has also been defined by the four *V*s:

- **Volume** — the amount of data. While volume indicates more data, it is the granular nature of the data that is unique. Big Data requires processing high volumes of low-density, unstructured Hadoop data—that is, data of unknown value, such as Twitter data feeds, click streams on a web page and a mobile application, network traffic, sensor-enabled equipment capturing data at the speed of light, and many more. It is the task of Big Data to convert such Hadoop data into valuable information.
- **Velocity** — the fast rate at which data is received and acted upon. The highest velocity data normally streams directly into memory versus being written to disk.
- **Variety** — new unstructured data types. Unstructured and semi-structured data types, such as text, audio, and video require additional processing to both derive meaning and the support metadata. Once understood, unstructured data has many of the same requirements as structured data, such as summarization, lineage, auditability, and privacy.
- **Value** — data has intrinsic value, but it must be discovered. There are a range of quantitative and investigative techniques to derive value from data.

5.2 Hadoop Data Cartridge

Hadoop is an open source technology that provides access to large data sets that are distributed across clusters. One strength of the Hadoop software is that it provides access to large quantities of data not stored in a relational database. The Oracle Stream Explorer data cartridge is based on the Cloudera distribution for Hadoop (CDH), version 3u5.

The content in this guide assumes that you are already familiar with, and likely running, a Hadoop system. If you need more information about Hadoop, start with the Hadoop project web site at `http://hadoop.apache.org/`.

> **Note:**
>
> You can use the Hadoop data cartridge on UNIX and Windows even through Hadoop itself runs only in the Linux environment.

5.2.1 Understanding the Oracle Stream Analytics Hadoop Data Cartridge

You can use the Hadoop data cartridge to integrate an existing Hadoop data source into an event processing network that can process data from files on the Hadoop distributed file system. With the data source integrated, you can write Oracle CQL query code that incorporates data from files on the Hadoop system.

When integrating a Hadoop system, keep the following guidelines in mind:

- The Hadoop cluster must have been started through its own mechanism and must be accessible. The cluster is not managed directly by Oracle Stream Analytics.
- A file from a Hadoop system supports only joins using a single key in Oracle CQL. However, any property of the associated event type may be used as key. In other words, with the exception of a key whose type is byte array, you can use keys whose type is other than a String type.
- Joins must use the equals operator. Other operators are not supported in a join condition.
- For the event type you define to represent data from the Hadoop file, only tuple-based event types are supported.
- The order of properties in the event type specification must match the order of fields in the Hadoop file.
- To avoid throwing a NullPointerExeption, wait for the Hadoop Data Cartridge to finish processing before attempting to shut down the server or undeploy.
- Only the following Oracle CQL to Hadoop types are supported. Any other type will cause a configuration exception to be raised.

Table 5-1 Mapping Between Datatypes for Oracle CQL and Hadoop

Oracle CQL Datatype	Hadoop Datatype
int	int
bigint	long
float	float
double	double
char	chararray
java.lang.String	chararray
byte	bytearray

5.2.1.1 Usage Scenario: Using Purchase Data to Develop Buying Incentives

To understand how a Hadoop data source might be used with an Oracle Stream Explorer application, consider a scenario with an application that requires quick access to a very large amount of customer purchase data in real time.

In this case, the data stored in Hadoop includes all purchases by all customers from all stores. Values in the data include customer identifiers, store identifiers, product identifiers, and so on. The purchase data includes information about which products are selling best in each store. To render the data to a manageable state, a MapReduce function is used to examine the data and produce a list of top buyers (those to whom incentives will be sent).

This data is collected and managed by a mobile application vendor as part of a service designed to send product recommendations and incentives (including coupons) to customers. The data is collected from multiple retailers and maintained separately for each retailer.

The Oracle Stream Explorer application provides the middle-tier logic for a client-side mobile application that is designed to offer purchase incentives to top buyers. It works in the following way:

1. Retailers arrange with the mobile application vendor to provide purchase data as part of a program to offer incentives to top buyers. The data, regularly refreshed from store sales data, is stored in a Hadoop system and a MapReduce function is used to identify top buyers.
2. The mobile application vendor provides the application for download, noting which retailers support the program.
3. App users each create a user ID that is correlated by the app vendor to data about customers from the retailers.
4. The mobile application is designed to send location data to the Oracle Stream Explorer application, along with the user ID. This information -- location coordinates and user ID -- forms the event data received by the Oracle Stream Explorer application.
5. As the Oracle Stream Explorer application receives event data from the mobile application, it uses Oracle CQL queries to:
   - Determine whether the user is near a store from a participating retailer.
   - Establish (from Hadoop-based data) whether the user is a top buyer for the retailer.

- Locate purchase information related to that user as a buyer from that retailer.
- If the user is a top buyer, the application correlates products previously purchased with incentives currently being offered to buyers of those products.

6. The Oracle Stream Explorer application pushes an incentive announcement to the user.

5.2.1.2 Data Cartridge Name

The Oracle Stream Explorer Hadoop cartridge uses the cartridge ID `com.oracle.cep.cartridge.hadoop`.

5.2.2 Using Hadoop Data Sources in Oracle CQL

You use the Hadoop support included with Oracle Stream Explorer by integrating a file in an existing Hadoop system into an event processing network. With the file integrated, you have access to data in the file from Oracle CQL code.

This section describes the following:

- Configuring Integration of Oracle Stream Analytics and Hadoop
- Integrating a File from a Hadoop System Into an EPN
- Using Hadoop Data in Oracle CQL.

5.2.2.1 Configuring Integration of Oracle Stream Analytics and Hadoop

In order to use Hadoop from Oracle Stream Analytics, you must first make configuration changes on both the Oracle Stream Analytics and Hadoop servers:

- On the Oracle Stream Analytics server, add the following Hadoop configuration files at the server's bootclasspath: `core-site.xml`, `hdfs.xml`, and `mapred.xml`.
- To the Hadoop server, copy the Pig JAR file to the lib directory and include it as part of the `HADOOP_CLASSPATH` defined in the `hadoop-env.sh` file.

> **Note:**
>
> A connection with a Hadoop data source through the cartridge might require many input/output operations, such that undeploying the application can time out or generate errors that prevent the application from being deployed again. Before undeploying an application that uses a Hadoop cartridge, be sure to discontinue event flow into the application.

5.2.2.2 Integrating a File from a Hadoop System Into an EPN

Integrating a file from an existing Hadoop system is similar to the way you might integrate a table from an existing relational database. For a Hadoop file, you use the `file` XML element from the Oracle Stream Explorer schema specifically added for Hadoop support.

The file element is from the http://www.oracle.com/ns/ocep/hadoop namespace. So your EPN assembly file needs to reference that namespace. The `file` element includes the following attributes:

- `id` -- Uniquely identifies the file in the EPN. You will use this attribute's value to reference the data source in a processor.
- `event-type` -- A reference to the event-type to which data from the file should be bound. The event-type must be defined in the EPN.
- `path` -- The path to the file in the Hadoop file system.
- `separator` -- Optional. The character delimiter to use when parsing the lines in the Hadoop file into separate fields. The default delimiter is the comma (',') character.
- `operation-timeout` -- Optional. The maximum amount of time, in milliseconds, to wait for the operation to complete.

With the Hadoop file to integrate specified with the file element, you use the table-source element to add the file as a data source for the Oracle CQL processor in which you will be using the file's data.

In the following example, note that the `http://www.oracle.com/ns/ocep/hadoop` namespace (and `hadoop` prefix) is referenced in the beans element. The `file` element references a `CustomerDescription.txt` file for data, along with a `CustomerDescription` event type defined in the event type repository.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:wlevs="http://www.bea.com/ns/wlevs/spring"
    xmlns:hadoop="http://www.oracle.com/ns/ocep/hadoop"
    xsi:schemaLocation="
        http://www.bea.com/ns/wlevs/spring
        http://www.bea.com/ns/wlevs/spring/ocep-epn.xsd
        http://www.oracle.com/ns/ocep/hadoop
        http://www.oracle.com/ns/ocep/hadoop/ocep-hadoop.xsd">
<!-- Some schema references omitted for brevity. -->

    <!-- Event types that will be used in the query. -->
    <wlevs:event-type-repository>
        <wlevs:event-type type-name="SalesEvent">
            <wlevs:class>com.bea.wlevs.example.SalesEvent</wlevs:class>
        </wlevs:event-type>
        <wlevs:event-type type-name="CustomerDescription">
            <wlevs:properties>
                <wlevs:property name="userId" type="char"/>
                <wlevs:property name="creditScore" type="int"/>
                <wlevs:property name="address" type="char"/>
                <wlevs:property name="customerName" type="char"/>
            </wlevs:properties>
        </wlevs:event-type>
    </wlevs:event-type-repository>

    <!-- Input adapter omitted for brevity. -->

    <!-- Channel sending SalesEvent instances to the processor. -->
    <wlevs:channel id="S1" event-type="SalesEvent" >
        <wlevs:listener ref="P1"/>
    </wlevs:channel>

    <!-- The file element to integrate CustomerDescription.txt file from
        the Hadoop system into the EPN. -->
```

```
    <hadoop:file id="CustomerDescription" event-type="CustomerDescription"
        path="CustomerDescription.txt" />

    <!-- The file from the Hadoop system tied into the query processor
        with the table-source element. -->
    <wlevs:processor id="P1">
        <wlevs:table-source ref="CustomerDescription" />
    </wlevs:processor>

    <!-- Other stages omitted for brevity. -->

</beans>
```

5.2.2.3 Using Hadoop Data in Oracle CQL

After you have integrated a Hadoop file into an event processing network, you can query the file from Oracle CQL code.

The following example illustrates how you can add a file from a Hadoop system into an EPN. With the file added to the EPN, you can query it from Oracle CQL code, as shown in the following example.

In the following example, the processor receives SalesEvent instances from a channel, but also has access to a file in the Hadoop system as CustomerDescription instances. The Hadoop file is essentially a CSV file that lists customers. Both event types have a `userId` property.

```
<n1:config
    xsi:schemaLocation="http://www.bea.com/ns/wlevs/config/application
wlevs_application_config.xsd"
    xmlns:n1="http://www.bea.com/ns/wlevs/config/application"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <processor>
        <name>P1</name>
        <rules>
            <query id="q1"><![CDATA[
                SELECT     customerName, creditScore, price, item
                FROM     S1 [Now], CustomerDescription as cust
                WHERE S1.userId = cust.userId
                AND S1.price > 1000
            ]></query>
        </rules>
    </processor>
</n1:config>
```

5.3 NoSQL Data Cartridge

The Oracle NoSQL Database is a distributed key-value database. In it, data is stored as key-value pairs, which are written to particular storage node(s). Storage nodes are replicated to ensure high availability, rapid failover in the event of a node failure and optimal load balancing of queries.

The content in this guide assumes that you are already familiar with, and likely running, an Oracle NoSQL database. If you need more information about Oracle NoSQL, be sure to see its Oracle Technology Network page at http://www.oracle.com/technetwork/database/database-technologies/nosqldb/documentation/index.html.

> **Note:**
>
> To use the NoSQL Data Cartridge, you must have a license for NoSQL Enterprise Edition.

5.3.1 Oracle CQL Processor Queries

You can use the Oracle Stream Explorer NoSQL Database data cartridge to refer to data stored in Oracle NoSQL Database as part of an Oracle CQL query. The cartridge makes it possible for queries to retrieve values from an Oracle NoSQL Database store by specifying a key in the query and then referring to fields of the value associated with the key.

When integrating an Oracle NoSQL database, keep the following guidelines in mind:

- The NoSQL database must have been started through its own mechanisms and must be accessible. It is not managed directly by Oracle Stream Explorer.
- This release of the cartridge provides access to the database using release 2.1.54 of the Oracle NoSQL Database API.
- The property used as a key in queries must be of type `String`. Joins can use a single key only.
- Joins must use the equals operator. Other operators are not supported in a join condition.
- Runaway queries that involve the NoSQL database are not supported. A runaway query has an execution time that takes longer than the execution time estimated by the optimizer.

5.3.2 Data Cartridge Name

The Oracle Stream Explorer NoSQL cartridge uses the cartridge ID `com.oracle.cep.cartridge.nosqldb`.

5.3.3 Using a NoSQL Database in Oracle CQL

To use the Oracle Stream Explorer NoSQL Database data cartridge in a CQL application, you must declare and configure it in one or more application-scoped cartridge contexts for the application.

5.3.3.1 Integrating a NoSQL Database Into an EPN

Integrating an existing NoSQL database is similar to the way you might integrate a table from a relational database. For a NoSQL database, you update the EPN assembly file in the following ways (see the example in step 3):

1. Add namespace declarations to support for the `store` element for referencing the NoSQL data source.

   Your changes should add a namespace schema location to the schemaLocation attribute, along with a namespace and prefix declaration:

- `http://www.oracle.com/ns/oep/nosqldb http://www.oracle.com/ns/oep/nosqldb/oep-nosqldb.xsd`
- `xmlns:nosqldb="http://www.oracle.com/ns/oep/nosqldb"`

2. Add the `store` element to integrate the NoSQL database into the event processing network as a relation source.

   The store element supports the following attributes, all of which are required:

   - `id` -- The name that will be used to refer to the key-value store in CQL queries.
   - `store-name` -- The name of the key-value store, which should match the name specified in the KVStoreConfig class when creating the store.
   - `store-locations` -- One or more host names and ports of active nodes in the store. The attribute value is a space-separated list in which each entry is formatted as "hostname:port". Nodes with the specified host name and port values will be contacted in order when connecting to the store initially.
   - `event-type` -- The object type for all objects retrieved for this relation from values in the store. The attribute value should correspond to the name of a wlevs:event-type entry specified in a wlevs:event-type-repository entry.

3. Add a `table-source` element to connect the NoSQL database to the processor in which queries will be executed.

The following example illustrates how you can connect an event processing network to a NoSQL database. The `store` element provides access to a store named "kvstore-customers", using port 5000 on host kvhost-alpha or port 5010 on host kvhost-beta to make the initial connection. It defines Oracle CQL processor P1 and makes the data in the key-value store available to it as a relation named "CustomerDescription".

The store can be referred to within Oracle CQL queries using the name "CustomerDescription". All values retrieved from the store should be serialized instances of the CustomerDescription class.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:osgi="http://www.springframework.org/schema/osgi"
       xmlns:wlevs="http://www.bea.com/ns/wlevs/spring"
       xmlns:nosqldb="http://www.oracle.com/ns/oep/nosqldb"
       xsi:schemaLocation="
 http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans.xsd
 http://www.springframework.org/schema/osgi
 http://www.springframework.org/schema/osgi/spring-osgi.xsd
 http://www.bea.com/ns/wlevs/spring
 http://www.bea.com/ns/wlevs/spring/ocep-epn.xsd
 http://www.oracle.com/ns/oep/nosqldb
 http://www.oracle.com/ns/oep/nosqldb/oep-nosqldb.xsd">

 <!-- Provide access to the CustomerDescription class, which represents
     the type of values in the store. -->
 <wlevs:event-type-repository>
     <wlevs:event-type type-name="CustomerDescription">
         <wlevs:class>com.bea.wlevs.example.CustomerDescription</wlevs:class>
     </wlevs:event-type>
     <wlevs:event-type type-name="SalesEvent">
         <wlevs:class>com.bea.wlevs.example.SalesEvent</wlevs:class>
     </wlevs:event-type>
 </wlevs:event-type-repository>
```

```
    <!-- The store element declares the key-value store, along with the
        event type to which incoming NoSQL data will be bound. -->
    <nosqldb:store store-name="kvstore-customers"
        store-locations="kvhost-alpha:5000 kvhost-beta:5010"
        id="CustomerDescription"
        event-type="CustomerDescription"/>


    <wlevs:channel id="S1" event-type="SalesEvent">
        <wlevs:listener ref="P1"/>
    </wlevs:channel>

    <!- The table-source element links the store to the CQL processor. -->
    <wlevs:processor id="P1">
        <wlevs:table-source ref="CustomerDescription" />
    </wlevs:processor>

</beans>
```

If Oracle CQL queries refer to entries in a store specified by a `store` element, then the values of those entries must be serialized instances of the type specified by the event-type attribute. The event type class must implement `java.io.Serializable`.

If a query retrieves a value from the store that is not a valid serialized form, or if the value is not the serialized form for the specified class, then Oracle Stream Analytics throws an exception and event processing is halted. You can declare multiple `store` elements to return values of different types from the same or different stores.

5.3.3.2 Using NoSQL Data in Oracle CQL

After you have integrated a NoSQL database into an event processing network, you can access data from Oracle CQL code. The query can look up an entry from the store by specifying an equality relation in the query's `WHERE` clause.

```
<n1:config
    xsi:schemaLocation="http://www.bea.com/ns/wlevs/config/application
wlevs_application_config.xsd"
    xmlns:n1="http://www.bea.com/ns/wlevs/config/application"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <processor>
        <name>P1</name>
        <rules>
            <query id="q1"><![CDATA[
                SELECT customerName, creditScore, price, item
                FROM S1 [Now], CustomerDescription as cust
                WHERE S1.userId = cust.userId
                AND creditScore > 5
            ></query>
        </rules>
    </processor>
</n1:config>
```

In this example, the event type instances representing data from the S1 channel and CustomerDescription NoSQL data source are both implemented as JavaBeans classes. Because both event types are JavaBeans classes, the Oracle CQL query can access the customer description associated with a particular event by equating the event's user ID with that of the customer description in the `WHERE` clause, treating both as JavaBeans properties:

```
WHERE S1.userId = CustomerDescription.userId
```

This clause requests that an entry be retrieved from the store that has the key specified by the value of the event's `userId` field. Only equality relations are supported for obtaining entries from the store.

Once an entry from the store has been selected, fields from the value retrieved from the store can be referred to in the `SELECT` portion of the query or in additional clauses in the `WHERE` clause.

The `creditScore` value specified in the `SELECT` clause will include the value of the `creditScore` field of the `CustomerDescription` object retrieved from the store in the query output. The reference to `creditScore` in the `WHERE` clause will also further restrict the query to events where the value of the `CustomerDescription` `creditScore` field is greater than 5.

5.3.3.2.1 Formatting the Key Used to Obtain Entries from the NoSQL Store

The key used to obtain entries from the store can be formatted in one of two ways: by beginning the value with a forward slash ('/') or by omitting a slash.

If the value specified on the left hand side of the equality relation starts with a forward slash, then the key is treated as a full key path that specifies one or more major components, as well as minor components if desired. For more details on the syntax of key paths, see the information about the `oracle.kv.Key` class in the Oracle NoSQL Database API documentation at `http://docs.oracle.com/cd/NOSQL/html/javadoc/index.html`.

For example, if the `userId` field of a `SalesEvent` object has the value "/users/user42/-/custDesc", then that value will be treated as a full key path that specifies "users" as the first major component, the user ID "user42" as the second major component, and a minor component named "custDesc".

As a convenience, if the value specified on the left hand side of the equality relation does not start with a forward slash, then it is treated as a single major component that comprises the entire key.

Note that keys used to retrieve entries from the store must be specified in full by a single field accessed by the Oracle CQL query. In particular, if a key path with multiple components is required to access entries in the key-value store, then the full key path expression must be stored in a single field that is accessed by the query.

5.4 HBase Big Data Cartridge

HBase Big Data Cartridge is an integration of HBase with Oracle Stream Explorer. HBase is a type of NOSQL database that is distributed, versioned, and a non-relational database.

HBase Big Data Cartridge does not support SQL as a primary means to access data. HBase provides Java APIs to retrieve the data. Every row has a key. All columns belong to particular column family. Each column family consists of one or more qualifiers. Hence, a combination of row key, column family and column qualifier is required to retrieve the data. HBase is suitable for storing Big Data without an RDBMS.

Every table has a row key like a relational database. The HBase column qualifier is similar to the concept of minor keys in NoSqlDB. For example, the major key for

records could be the name of a person, whereas the minor key would be the different pieces of information that you want to store for the person.

5.4.1 Understanding HBase Cartridge

In HBase, the priority is to store Big Data efficiently and not perform any complex data retrieval operations.

The code snippets below give an idea of how data can be stored and retrieved in HBase:

```
HBaseConfiguration config = new HBaseConfiguration();
batchUpdate.put("myColumnFamily:columnQualifier1",
"columnQualifier1value".getBytes());
Cell cell = table.get("myRow", "myColumnFamily:columnQualifier1");
String valueStr = new String(cell.getValue());
```

HBase is used to store metadata for various applications. For example, a company may store customer information associated with various sales in an HBase database. In this case, you can use the HBase Big Data Cartridge that enables you to write CQL queries using HBase as an external data source.

The HBase database store EPN component is provided as a data cartridge. The HBase database cartridge provides a `<store>` EPN component with the following properties:

- `id`: id of the EPN component.
- `store-location`: location in the form of `domain:client port` of an HBase database server.
- `event-type`: schema for store as seen by the CQL processor. The event type must be a Java class that implements the `java.io.Serializable interface`.
- `table-name`: name of the HBase table.

The EPN component has a related <column-mappings> component to specify the mappings from the CQL event attributes to the HBase column family/qualifier. This component is declared in an HBase Big Data Cartridge configuration file similar to the JDBC cartridge configuration. This component has the following properties:

- `name`: the id of the `<store>` EPN component for which the mappings are being declared.
- `rowkey`: the row key attribute used for the HBase table. This must be a `String`.
- `cql-attribute`: the CQL attribute name used in the CQL query. This name should match with a corresponding field declared in the Java event type.
- `hbase-family`: the HBase column family.
- `hbase-qualifer`: the HBase column qualifier.

5.4.2 Using HBase Cartridge

To use a HBase Cartidge, you need to specify the `hbase-family` and `hbase-qualifier` if the `cql-attribute` is a primitive data type.

The `<hbase:store>` component is linked to a CQL processor using the 'table-source' element, as in the following example:

```
<hbase:store id="User" table-name="User" event-type="UserEvent" store-
location="localhost:5000">
</hbase:store>
<wlevs:processor id="P1">
  <wlevs:table-source ref="User"/>
</wlevs:processor>
```

You must specify the column mappings for the `<hbase:store>` component in the Oracle Stream Analytics HBase configuration file as shown in the following example:

Example 5-1 HBase Cartridge Column Mappings

In the example below, the CQL column `address` is a map as it holds all the column qualifiers from the `address` column family. The CQL columns `firstname`, `lastname`, `email` and `role` hold primitive data types. These are the specific column qualifiers from the `data` column family. The `userName` field from the event type is the row key and hence it does not have any mapping to an HBase column family or qualifier.

```
<hbase:column-mappings>
    <name>User</name>
    <rowkey>userName</rowkey>
    <mapping cql-attribute="address" hbase-family="address"/>
    <mapping cql-attribute="firstname" hbase-family="data"  hbase-
qualifier="firstname"/>
    <mapping cql-attribute="lastname" hbase-family="data"  hbase-
qualifier="lastname"/>
    <mapping cql-attribute="email" hbase-family="data"  hbase-qualifier="email"/>
    <mapping cql-attribute="role" hbase-family="data" hbase-qualifier="role"/>
</hbase:column-mappings>
```

The `<UserEvent>` class has the following fields:

```
String userName;
java.util.Map address;
String first name;
String lastname;
String email;
String role;
```

The HBase schema is dynamic in nature and additional column families and/or column qualifiers can be added at any point after an HBase table is created. Oracle Stream Analytics allows you to retrieve the event fields as a map which contains all dynamically added column qualifiers. In this case, you need to declare a `java.util.Map` as one of the event fields in the Java event type. Hence the `UserEvent` event type must have a `java.util.Map` field with name and address. The cartridge does not support dynamically added column families. So, the event type needs to be modified if the Oracle Stream Analytics application needs to use a newly added column family.

An HBase database may be run as a cluster. The `hostname` and `client port` of the master node need to be configured.

Supported Operators

Currently, only the `=`, `!=`, `like`, `<`, and `>` operators are supported. The first sub-clause in the query must be an equality join with the HBase data source based on row key.

```
S1.userName = user.userName
```

The `like` operator accepts a Java regular expression as argument. This operator is for `String` only.

```
user.firstname like Y.*
```

The < and > operators are for `integer` and `double` data types only.

5.4.3 Limitations of HBase Cartridge in 12.2.1 Release

The HBase Cartridge has a few limitations in the 12.2.1 release.

The limitations of the HBase Cartridge are as listed below:

1. Only the HBase server version 0.94.8 supported.
2. When an HBase server is unreachable, you cannot deploy a HBase cartridge application to the Oracle Stream Explorer server.
3. You cannot reconnect to an HBase server unless you restart the Oracle Stream Explorer server, when a HBase server is shut down after a HBase application is deployed to the Oracle Stream Explorer.
4. The event property data type must be `String` when you want to join it with the HBase table rowkey.
5. In HBase cartridge application, the name of both `store id` and `table-name` must be the same for HBase cartridge in the spring file. Else, the table identification will fail.
6. Use the wrapper data type integer or double for the HBase event property data type to avoid runtime exceptions.
7. A clear error message is not shown when there is a syntax error in the CQL query with HBase cartridge application.
8. A runtime exception is thrown when you try to join a null string value with HBase rowkey.
9. The first sub-clause in the CQL query must be an equality join with the HBase data source based on row key.
10. You must suffix the letter *d* to the `double` data type value when you use `=` or `!=` operator for `double` data type in the CQL query.

6
Oracle Java Data Cartridge

How to use the Oracle Java Data Cartridge, an extension of Oracle Continuous Query Language (Oracle CQL). You can use Oracle CQL to write CQL code that interacts with Java classes in your Oracle Stream Explorer application is described.

This chapter describes the types, methods, fields, and constructors that the Oracle Java data cartridge exposes. You can use these types, methods, fields, and constructors in Oracle CQL queries and views as you would Oracle CQL native types.

This chapter includes the following sections:

- Understanding the Oracle Java Data Cartridge
- Using the Oracle Java Data Cartridge.

6.1 Understanding the Oracle Java Data Cartridge

The Oracle Java data cartridge is a built-in Java cartridge that enables you to write Oracle CQL queries and views that interact with the Java classes in your Oracle Stream Explorer application.

- Data Cartridge Name
- Class Loading
- Method Resolution
- Datatype Mapping
- Oracle CQL Query Support for the Oracle Java Data Cartridge.

6.1.1 Data Cartridge Name

The Oracle Java data cartridge uses the cartridge ID `com.oracle.cep.cartrdiges.java`.

The Oracle Java data cartridge is the default Oracle Stream Analytics data cartridge.

For types under the default Java package name or types under the system package of `java.lang`, you can reference the Java type in an Oracle CQL query unqualified by package or data cartridge name:

```
<query id="q1"><![CDATA[
    select String(“foo") …
></query>
```

> **Note:**
>
> To simplify Oracle Java data cartridge type names, you can use aliases as described in Oracle Fusion Middleware Oracle CQL Language Reference for Oracle Stream Analytics.

For more information, see: Class Loading.

6.1.2 Class Loading

The Oracle Java data cartridge supports the following policies for loading the Java classes that your Oracle CQL queries reference:

- Application Class Space Policy
- No Automatic Import Class Space Policy
- Server Class Space Policy

For more information, see:

- Class Loading Example
- Method Resolution

6.1.2.1 Application Class Space Policy

This is the default class loading policy.

In this mode, the Oracle Java data cartridge uses the class-space of the application in scope when searching for a Java class.

This is only applicable when a type is specified only by its local name, that is, there is a single identifier, and no other identifiers are being used for its package. That is:

```
select String(“foo") …
```

And not:

```
select java.lang.String(“foo") …
```

In this case the procedure is as follows:

- Attempt to load the class defined by the single identifier (call it `ID1`) using the application's class-space as usual; if this fails then:
- Verify if the application defines any class within its bundle's internal class-path whose name matches `ID1`, independent of the package; if this fails then:
- Verify if application specifies an `Import-Package` `MANIFEST` header statement which in conjunction with `ID1` can be used to load a Java class.

For an example, see Class Loading Example.

6.1.2.2 No Automatic Import Class Space Policy

This is an optional class loading policy. To use this policy, you must include the following `MANIFEST` header entry in your Oracle Stream Explorer application:

```
OCEP_JAVA_CARTRIDGE_CLASS_SPACE: APPLICATION_NO_AUTO_IMPORT_CLASS_SPACE
```

This mode is similar to the application class space policy except that Oracle Event Processing does not attempt to combine the package with `ID1`.

For more information, see Application Class Space Policy.

6.1.2.3 Server Class Space Policy

This is an optional class loading policy. To use this policy, you must include the following `MANIFEST` header entry in your Oracle Stream Explorer application:

```
OCEP_JAVA_CARTRIDGE_CLASS_SPACE: SERVER_CLASS_SPACE
```

An Oracle CQL query can reference any exported Java class, regardless of whether or not the class package is imported into the application or bundle

The query can also access all classes visible to the OSGi framework's parent class-loader, which includes the runtime JDK classes.

This means that an Oracle CQL application may contain an Oracle CQL query that references classes defined by other Oracle Stream Explorer applications, as long as they are exported. This behavior facilitates the creation of Java-based cartridges whose sole purpose is to provide new Java libraries.

> **Note:**
>
> You can only reference a Java class that is part of the internal class path of an Oracle Stream Explorer application if it is exported, even when a processor within this application defines the Oracle CQL query.

For an example, see Class Loading Example.

6.1.2.4 Class Loading Example

Consider the example that Figure 6-1 shows: application `B1` imports package `package2` that application `B2` exports.

Figure 6-1 Example Oracle Stream Analytics Event Processing Applications

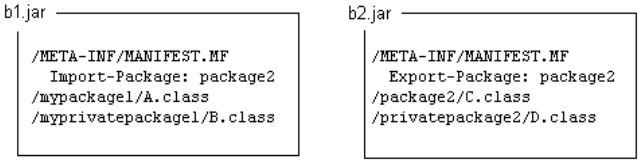


Table 6-1 summarizes which classes these two different applications can access depending on whether they are running in the application class space or server class space.

Table 6-1 Class Accessibility by Class Loading Policy

Class Loading Policy	Application B1	Application B2
Application Class Space	• `mypackage1.A` • `myprivatepackage1.B` • `package2.C`	• `package2.C` • `privatepackage2.D`
Server Class Space	• `package2.C`	• `package2.C`

In application B1, you can use any of the Java classes A, B, and C in your Oracle CQL queries:

```
select A …
select B …
select C …
```

However, in application B2, you cannot use Java classes A and B in your Oracle CQL queries. You can only use Java classes C and D:

```
select C …
select D …
```

6.1.3 Method Resolution

An Oracle CQL expression that accesses a Java method uses the following algorithm to resolve the method:

1. All parameter types are converted to Java types as Datatype Mapping describes.

   For example, an Oracle CQL `INTEGER` is converted to a Java primitive `int`.

2. Standard Java method resolution rules are applied as the Java Language Specification, Third Edition, Section 15.12, "Method Invocation Expressions" describes.

> **Note:**
>
> Variable arity methods are not supported. For more information, see the Java Language Specification, Third Edition, Section 12.12.2.4.

As an example, consider the following Oracle CQL expression:

```
attribute.methodA(10)
```

Where `attribute` is of type `mypackage.MyType` which defines the following overloaded methods:

- `methodA(int)`
- `methodA(Integer)`
- `methodA(Object)`
- `methodA(long)`

As the literal 10 is of the primitive type `int`, the order of precedence is:

- `methodA(int)`
- `methodA(long)`
- `methodA(Integer)`
- `methodA(Object)`

For more information, see Class Loading.

6.1.4 Datatype Mapping

The Oracle Java data cartridge applies a fixed, asymmetrical mapping between Oracle CQL native data types and Java data types.

- Table 6-2 lists the mappings between Oracle CQL native data types and Java data types.
- Table 6-3 lists the mappings between Java data types and Oracle CQL native data types.

Table 6-2 Oracle Java Data Cartridge: Oracle CQL to Java Data Type Mapping

Oracle CQL Native Data Type	Java Data Type
BIGINT	long
BOOLEAN	boolean
BYTE	byte[]
CHAR	java.lang.String
DOUBLE	double
FLOAT	float
INTEGER	int
INTERVAL	long
INTERVAL_DAY	long, java.lang.String
INTERVAL_DAY_TO_SECOND	java.lang.String
INTERVAL_YEAR	long, java.lang.String
INTERVAL_MONTH	long, java.lang.String
INTERVAL_YEAR_TO_MONTH	java.lang.String
XMLTYPE	java.lang.String

Table 6-3 Oracle Java Data Cartridge: Java Data Type to Oracle CQL Mapping

Java Datatype	Oracle CQL Native Data Type
long	BIGINT
boolean	BOOLEAN
byte[]	BYTE
java.lang.String	CHAR
double	DOUBLE
float	FLOAT

Table 6-3 (Cont.) Oracle Java Data Cartridge: Java Data Type to Oracle CQL Mapping

Java Datatype	Oracle CQL Native Data Type
`int`	`INTEGER`
`java.sql.Date`	`INTERVAL`
`java.sql.Timestamp`	
`java.sql.SQLXML`	`XMLTYPE`

All other Java classes are mapped as a complex type.

For more information on these datatype mappings:

- Java Data Type String and Oracle CQL Data Type CHAR
- Literals
- Arrays
- Collections

6.1.4.1 Java Data Type String and Oracle CQL Data Type CHAR

Oracle CQL data type `CHAR` is mapped to `java.lang.String` and `java.lang.String` is mapped to Oracle CQL data type `CHAR`. This means you can access `java.lang.String` member fields and methods for an attribute defined as Oracle CQL `CHAR`. For example, if `a1` is declared as type Oracle CQL `CHAR`, then you can write a query like this:

```
<query id="q1"><![CDATA[

    select a1.substring(1,2)

]></query>
```

6.1.4.2 Literals

You cannot access member fields and methods on literals, even Oracle CQL `CHAR` literals. For example, the following query is *not* allowed:

```
<query id="q1-forbidden"><![CDATA[

    select "hello".substring(1,2)

]></query>
```

6.1.4.3 Arrays

Java arrays are converted to Oracle CQL data cartridge arrays, and Oracle CQL data cartridge arrays are converted to Java arrays. This applies to both complex types and simple types.

You can use the data cartridge `TABLE` clause to access the multiple rows returned by a data cartridge function in the `FROM` clause of an Oracle CQL query.

For more information, see Collections.

6.1.4.4 Collections

Typically, the Oracle Java data cartridge converts an instance that implements the `java.util.Collection` interface to an Oracle CQL complex type.

An Oracle CQL query can iterate through the members of the `java.util.Collection`.

You can use the data cartridge `TABLE` clause to access the multiple rows returned by a data cartridge function in the `FROM` clause of an Oracle CQL query.

For more information, see Arrays.

6.1.5 Oracle CQL Query Support for the Oracle Java Data Cartridge

You may use Oracle Java data cartridge types in expressions within a `SELECT` clause and `WHERE` clause.

You may not use Oracle Java data cartridge types in expressions within an `ORDER BY` clause.

For more information, see Using the Oracle Java Data Cartridge.

6.2 Using the Oracle Java Data Cartridge

Common use-cases that highlight how you can use the Oracle Java data cartridge in your Oracle Stream Analytics applications are described.

- How to Query Using the Java API
- How to Query Using Exported Java Classes

For more information, see Oracle CQL Query Support for the Oracle Java Data Cartridge.

6.2.1 How to Query Using the Java API

This procedure describes how to use the Oracle Java data cartridge in an Oracle Stream Explorer application that uses one event type defined as a tuple (`Student`) that has an event property type defined as a Java class (`Address.java`).

To query with Java classes:

1. Implement the `Address.java` class.

```
package test;

class Address {
    String street;
    String state;
    String city;
    String [] phones;
}
```

   In this example, assume that the `Address.java` class belongs to this application.

If the `Address.java` class belonged to another Oracle Stream Explorer application, it must be exported in its parent application. For more information, see How to Query Using Exported Java Classes.

2. Define the event type repository.

```
<event-type-repository>
    <event-type name="Student">
        <properties>
            <property name="name" type="char"/>
            <property name="address" type="Address"/>
        </properties>
    </event-type>

    <event-type name="Address">
        <class-name>test.Address</class-name>
    </event-type>
<event-type-repository>
```

Because the `test.Address` class belongs to this application, it can be declared in the event type repository. This automatically makes the class globally accessible within this application; its package does not need to be exported.

3. Assume that an adapter is providing `Student` events to channel `StudentStream`:

```
<channel id="StudentStream" event-type="Student"/>
```

4. Assume that the `StudentStream` is connected to a processor with the Oracle CQL query `q1`.

```
<processor>
    <rules>

       <query id="q1"><![CDATA[

            select
                name,
                address.street as street,
                address.phones[0] as primary_phone
            from
                StudentStream

       ></query>

    </rules>
</processor>
```

The Oracle Java data cartridge allows you to access the `address` event property from within the Oracle CQL query using normal Java API.

6.2.2 How to Query Using Exported Java Classes

This procedure describes how to use the Oracle Java data cartridge in an Oracle Stream Analytics application that uses one event type defined as a tuple (`Student`) that has an event property type defined as a Java class (`Address.java`). In this procedure, the Address.java class belongs to a separate Oracle Stream Analytics application. It is exported in its parent application to make it accessible to other Oracle Stream Analytics applications deployed to the same Oracle Stream Analytics server.

To query with Java classes:

1. Implement the `Address.java` class.

```
package test;

class Address {
    String street;
    String state;
    String city;
    String [] phones;
}
```

2. Export the `test` package that contains the `Address.java` class.

   For more information, see *Oracle Fusion Middleware Developing Applications for Event Processing with Oracle Stream Analytics*.

   The `test` package may be part of this Oracle Stream Analytics application or it may be part of some other Oracle Stream Analytics application deployed to the same Oracle Stream Analytics server as this application.

3. Define the event type repository.

```
<event-type-repository>
    <event-type name="Student">
        <property name="name" type="char"/>
        <property name="address" type="Address"/>
    </event-type>
<event-type-repository>
```

4. Assume that an adapter is providing `Student` events to channel `StudentStream`:

```
<channel id="StudentStream" event-type="Student"/>
```

5. Assume that the `StudentStream` is connected to a processor with the Oracle CQL query `q1`.

```
<processor>
    <rules>

       <query id="q1"><![CDATA[

            select
                name,
                address.street as street,
                address.phones[0] as primary_phone
            from
                StudentStream

       ></query>

    </rules>
</processor>
```

   The Oracle Java data cartridge allows you to access the `address` event property from within the Oracle CQL query using normal Java API.

6.2.3 Java Cast Function

The Java cartridge provides the Java Cast function that enables a Java extensible type to be cast to another Java extensible type, providing the latter can be assigned from the former. To use this function, you must have the Java cartridge installed.

Syntax

```
T cast@java(l-value, class-literal<T>)
```

Parameters

`l-value`: A event attribute that contains the data that you want to cast. If `l-value` cannot be assigned from `T`, then Java Cartridge raises a `RuntimeInvocationException` during the invocation of the cast function

`class-literal<T>`: The name of the class to which you want to cast. For example, if you want to cast an `int` to `long`, then `class-literal<T>` is `Long.class`.

Example

Consider the following class hierarchy:

```
public class Parent
{
...
}

public class Child extends Parent
{
...
}
```

The following example casts an object of type `Child`.

```
cast@java(S.parent, Child.class)
```

7

Data Cartridge Framework

The Data Cartridge Framework is a service provider interface (SPI) that enables users and vendors to create cartridges to extend Oracle CQL functionality. The Hadoop and NoSQL cartridges described in Oracle Big Data Cartridges are examples of cartridges created with the Data Cartridge Framework.

For example, with the Data Cartridge Framework, you can extend Oracle CQL functionality to support the development of telematic applications. Telematic applications encompass telecommunications (electrical signals and electromagnetic waves), automotive technologies, transportation, electrical engineering (sensors, instrumentation, and wireless communications), and computer science (Internet of Things).

This chapter includes the following sections:

7.1 About the SPI

An Oracle Stream Analytics cartridge is a single manageable unit that defines external functions, types, indexes, Java classes, and data sources.

The user of the cartridge references the available functions, types, indexes, Java classes, and data sources from Oracle CQL code with links of the following form:

```
myFunction@myCartridge(arg1)
```

A cartridge created with the Oracle Stream Analytics Data Cartridges Framework is an Oracle Stream Analytics library. This means that you deploy the cartridge the same way that you deploy a library, which is from the command line or in Oracle JDeveloper. Once you deploy a cartridge, all of the external functions, types, indexes, Java classes, and data sources are available to use in Oracle CQL queries. You must deploy a cartridge before you deploy the application. You can update the cartridge without updating your application.

7.2 Interfaces

The `com.oracle.cep.cartridge` package contains the Cartridge Framework Java interfaces.

This section describes what you *can* do and what you *must* do when you use the interfaces. Brief descriptions of the interfaces and exceptions follow.

You Can:

- Use any type system for the table and stream attribute types in Oracle CQL.
- Provide your own index data structure for invoking functions.
- Provide new Java classes that are visible within an Oracle CQL query. When you deploy the cartridge, include the application or library that has the new Java classes. Applications that access the new Java classes, must import the correct Java packages in their `MANIFEST.MF` file with the `Import-Package` header entry.

You Must:

- Provide an MBean for all deployed cartridges that contains a list of all functions that the cartridge supports. When the cartridge is undeployed the MBean instance is unregistered.
- Implement the `ExternalFunctionProvider.listFunctions` method.
- Provide a bean-stage MBean for table sources that you tie to a cartridge external data source. This MBean provides a list of the table source custom properties including its `id` and `provider` name.

  Optionally, a table source Spring bean factory can implement the `com.bea.wlevs.management.configuration.spring.StageFactoryAccess` interface to customize how to access the table source properties.

7.2.1 Interface Descriptions

The `com.oracle.cep.cartridge` package provides the following Java interfaces.

`CapabilityProvider`: An `ExternalConnection` can implement this interface to specify the supported capabilities such as less than `<`, `AND`, `OR`, and so on.

`ExternalConnection`: Connect to an `ExternalDataSource`.

`ExternalConstants`: Define general constants used by the data cartridge. This interface provides two constants: `EQUALS` for external connection capabilities, and `SERVER_CONTEXT_LINK_ID` to denote an `ExternalFunctionProvider` link `id`.

`ExternalDataSource`: Use the `getConnnection` method to connect to an external source of contextual data to join with Oracle CQL processor events. The external data source must support the configuration of its properties. For example, a NoSQLDB data source supports the configuration of a host, a port, and a store name.

The external data source specifies the functions it supports. By default, all external data sources support the equality function, for example:

```
SELECT * FROM S[NOW], MyExternalDataSource
WHERE S.id = MyExternalDataSource.id
```

To make the data source available to Oracle CQL processors, register a Spring Bean that implements the `com.oracle.cep.cartridge.ExternalDataSource` interface and make that Spring bean the target of a table source (`wlevs:table-source` tag).

`ExternalFunction`: A function provided by an `ExternalFunctionProvider` or other external entity.

`ExternalFunctionDefinition`: Specify the metadata for functions used in Oracle CQL queries and views that are provided by an `ExternalFunctionProvider` or other external entity.

`ExternalFunctionProvider`: Defines a set of functions that can be directly accessed from Oracle CQL queries and views. Use the `getID` method to register an external function provider as an OSGi service. Also, the provider must specify the `ExternalContants.SERVER_CONTEXT_LINK_ID` service property to indicate the link ID to use in Oracle CQL queries and views to identify the provider.

`ExternalPredicate`: Represent prepared statement predicates with attributes and a predicate clause.

`ExternalPreparedStatement`: Represent a prepared statement from an external function provider to execute the same or similar functions repeatedly and efficiently.

7.2.2 Exceptions

The `com.oracle.cep.cartridge` package provides the following exceptions:

`AmbiguousFunctionException`: Thrown when referenced function cannot be determined by the `ExternalFunctionProvider` due to ambiguity.

`CartridgeException`: Root cartridge exception.

`FunctionNotFoundException`: Thrown when the referenced function in an Oracle CQL statement is not supported by `ExternalFunctionProvider`.

7.3 Cartridge Examples

To make the cartridges available for Oracle CQL queries within Oracle Stream Explorer applications, deploy each cartridge as a separate application library. After you deploy the cartridges, deploy the Oracle Stream Explorer application or applications that use the cartridges.

This section describes two cartridge examples: an arithmetic cartridge and a data source cartridge. The arithmetic cartridge makes arithmetic functions available to Oracle CQL queries similar to the spatial cartridge, which contains only functions, described in Oracle Spatial Data Cartridge. The data cartridge defines a data source similar to Hadoop described in Oracle Big Data Cartridges.

7.3.1 Arithmetic Cartridge

The arithmetic cartridge has the following function classes:

- A set of Java classes that provide the functionality for addition, array, and exception operations.
- The `ExceptionFunction.java` and `ArrayFunciton.java` classes to provide array and exception functionality so that you can use arrays and throw exceptions from an Oracle CQL query.
- An `ArithmeticActivator.java` class starts and stops the cartridge bundle.

All of the function classes implement the `com.oracle.cep.cartridge.ExternalFunctionProvider` interface and have a `getName` method that returns the name of the function to use in an Oracle CQL query.

For example the `AddFunction.java` and `AddLongFunction.java` `getName` methods return `plus` for the function name. You use the function name in the Oracle CQL query to call the function. The following query uses the `plus` function in the `arithmetic` cartridge to add two integers from `inputChannel`.

```
SELECT plus@arithmetic(typeInt, typeInt2) AS typeInt FROM inputChannel
```

7.3.2 Data Source Cartridge

Data Source Cartridge Files

The cartridge example uses a set of Java classes that define the data source.

The `MyCartridgeSource.java` class implements the `com.oracle.cep.cartridge.ExternalDataSource interface`. It defines the data source connection functionality, and reads event data from and writes event data to the database.

The `MyActivator.java` class implements `org.osgi.framework.BundleActivator` and provides code to start and stop the cartridge bundle.

The `MyHandler.java` class implements `org.springframework.beans.factory.xml.NamespaceHandler`, and provides code to manage the cartridge name space and register the Udds factory bean.

The `UddsDefinitionParser.java` class extends `org.springframework.beans.factory.xml.AbstractSingleBeanDefinitionParser` and provides code to parse and register `UddsFactoryBean` objects.

The `UddsFactoryBean.java` class extends `org.springframework.beans.factory.config.AbstractFactoryBean` and provides code to manage events and the event type repository.

7.4 Source Code

The source code for the data source application and cartridge, and the arithmetic cartridge is provided.

- Arithmetic Cartridge
- Data Source Cartridge.

7.4.1 Arithmetic Cartridge

- AddFunction.java
- ArithmeticActivator.java
- ArrayFunction.java
- ExceptionFunction.java
- UserDefineFunctionClass.java

AddFunction.java

```
package tests.functional.cartridge.userdefine.common.libs.arithmetic;

import java.util.Map;
```

```
import com.oracle.cep.cartridge.ExternalFunction;

public class AddFunction implements ExternalFunction{

  @Override
  public String getName() {
    return "plus";
  }

  @Override
  public Class<?>[] getParameterTypes() {
    Class<?>[] parameters = new Class<?>[2];
    parameters[0] = java.lang.Integer.class;
    parameters[1] = java.lang.Integer.class;
    return parameters;
  }

  @Override
  public Class<?> getReturnType() {
        return java.lang.Integer.class;
  }

  @Override
  public Object execute(Object[] args, String caller, Map<String, Object> context)
                                throws Exception {
    if(args.length != 2)
      throw new IllegalArgumentException("add function need an 2 parameters");
    if(!(args[0] instanceof java.lang.Integer && args[1]
                instanceof java.lang.Integer)) {
      throw new IllegalArgumentException("add function only
                                          support java.lang.Integer");
    }
    java.lang.Integer arg1 = (Integer) args[0];
    java.lang.Integer arg2 = (Integer) args[1];
    return new java.lang.Integer(arg1 + arg2);
  }
}
```

ArithmeticActivator.java

```
package tests.functional.cartridge.userdefine.common.libs.arithmetic;

import java.util.Hashtable;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;
import com.oracle.cep.cartridge.ExternalFunctionProvider;

public class ArithmeticActivator  implements BundleActivator {
  private ServiceRegistration reg;

  @Override
  public void start(BundleContext context) throws Exception {
    Hashtable props = new Hashtable();
    props.put("server.context.link.id", "arithmetic");

    this.reg = context.registerService(ExternalFunctionProvider.class.getName(),
                                    new UserDefineFunction(), props);
  }

  @Override
```

```
  public void stop(BundleContext arg0) throws Exception {
    this.reg.unregister();
  }
}
```

ArrayFunction.java

```
package tests.functional.cartridge.userdefine.common.libs.arithmetic;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import com.oracle.cep.cartridge.ExternalFunction;

public class ArrayFunction implements ExternalFunction {

  @Override
    public String getName() {
    return "array";
  }

  @Override
  public Class<?>[] getParameterTypes() {
    Class<?>[] parameters = new Class<?>[2];
    parameters[0] = Integer.class;    parameters[1] = Integer.class;
    return parameters;
  }

  @Override
  public Class<?> getReturnType() {
    return List.class;
  }

  @Override
  public Object execute(Object[] args, String caller, Map<String, Object> context)
                        throws Exception {
    if(args.length == 0) {
      return null;
    }
    if(!(args[0] instanceof java.lang.Integer)) {
      throw new IllegalArgumentException("median function only supports
                                          java.lang.Integer");
    }
    List ret = new ArrayList();
    for(Object obj:args) {
      ret.add(obj);
    }
    return ret;
  }
}
```

ExceptionFunction.java

```
package tests.functional.cartridge.userdefine.common.libs.arithmetic;

import java.util.Map;
import com.oracle.cep.cartridge.ExternalFunction;

public class ExceptionFunction implements ExternalFunction{

  @Override
```

```
  public String getName() {
    return "exception"
  }

  @Override
  public Class<?>[] getParameterTypes() {
    return new Class<?>[]{Integer.class};
  }

  @Override
  public Class<?> getReturnType() {
    return Integer.class;
  }

  @Override
  public Object execute(Object[] args, String caller, Map<String, Object> context)
                        throws Exception {
    throw new NullPointerException("I am an excpetion");
  }
}
```

UserDefineFunctionClass.java

```
package tests.functional.cartridge.userdefine.common.libs.arithmetic;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Set;
import com.oracle.cep.cartridge.AmbiguousFunctionException;
import com.oracle.cep.cartridge.ExternalFunction;
import com.oracle.cep.cartridge.ExternalFunctionProvider;
import com.oracle.cep.cartridge.FunctionNotFoundException;

public class UserDefineFunction implements ExternalFunctionProvider {

  private ArrayList<ExternalFunction> functions = new
                                    ArrayList<ExternalFunction>();

  public UserDefineFunction() {
    functions.add(new AddFunction());
    functions.add(new ArrayFunction());
    functions.add(new ExceptionFunction());
  }

  @Override
  public ExternalFunction getFunction(String functionName, Class<?> []
                       parameterTypes, String caller, Map<String, Object> context)
    throws AmbiguousFunctionException, FunctionNotFoundException {
    if("plus".equalsIgnoreCase(functionName)) {
      return new AddFunction();
    } else if("array".equalsIgnoreCase(functionName)) {
      return new ArrayFunction();
    } else if("exception".equalsIgnoreCase(functionName)) {
      return new ExceptionFunction();
    }
    throw new FunctionNotFoundException(functionName+" is not supported in
                                        arithmetic");
  }

  @Override
```

```
  public String getId() {
    return "arithmetic";
  }

  @Override
  public List<ExternalFunction> listFunctions(String caller,
    Map<String, Object> context) {
    ArrayList<ExternalFunction> functionList = new ArrayList<ExternalFunction>();
    functionList.addAll(functions);
    return functionList;
  }
}
```

7.4.2 Data Source Cartridge

The Data Source cartridge is comprised of the following Java class files:

- MyCartridgeSource.java
- MyActivator.java
- MyHandler.java
- UddsFactoryBean.java

MyCartridgeSource.java

```
package tests.functional.cartridge.userdefine.common.libs.datasource;

import java.math.BigDecimal;
import java.sql.Timestamp;
import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import oracle.cep.dataStructures.external.TupleValue;
import org.springframework.osgi.extensions.annotation.ServiceReference;
import com.bea.wlevs.ede.api.EventProperty;
import com.bea.wlevs.ede.api.EventType;
import com.bea.wlevs.ede.api.EventTypeRepository;
import com.bea.wlevs.ede.api.Type;
import com.bea.wlevs.management.configuration.spring.StageFactoryAccess;
import com.oracle.cep.cartridge.ExternalConnection;
import com.oracle.cep.cartridge.ExternalDataSource;
import com.oracle.cep.cartridge.ExternalPredicate;
import com.oracle.cep.cartridge.ExternalPreparedStatement;

public class MyCartridgeSource implements StageFactoryAccess,
  ExternalDataSource {
  //
  @Override
  public Map<?, ?> getCacheDataSource() {
    return null;
  }

  private EventTypeRepository etr;

  @ServiceReference
```

```
public void setEventTypeRepository(EventTypeRepository etr) {
  this.etr = etr;
}

private String eventType;

@Override
public String getEventType() {
  System.out.println("event type:" + this.eventType);
  return eventType;
}

public void setEventType(String eventType) {
  this.eventType = eventType;
}

private long maxThreshhold = 0;

@Override
  public long getExternalRowsThreshold() {
  return maxThreshhold;
}

public void setExternalRowsThreshold(long maxThreshhold) {
  this.maxThreshhold = maxThreshhold;
}

private String pattern;

public String getPattern() {
  return pattern;
}

public void setPattern(String pattern) {
  this.pattern = pattern;
}

private String singularity;

public String getSingularity() {
  return singularity;
}

public void setSingularity(String singularity) {
      this.singularity = singularity;
}

private String id;

@Override
public String getId() {
  return id;
}

public void setId(String id) {
  this.id = id;
}

@Override
public String getJDBCDataSource() {
      return null;
```

```
    }

private Class keyClass;

@Override
public Class getKeyClass() {
      return Long.class;
}

public void setKeyClass(String className) throws ClassNotFoundException {
  this.keyClass = Class.forName(className);
}

private String[] keyPropertyNames;

@Override
public String[] getKeyPropertyNames() {
  return keyPropertyNames;
}

public void setKeyProperty(String names) {
  keyPropertyNames = names.split(",");
}

@Override
public String getTableName() {
  return null;
}

public Class getObjectType() {
  return ExternalDataSource.class;
}

@Override
public ExternalConnection getConnection() throws Exception {
  MyExternalConnection connection = new
  MyExternalConnection(this.etr.getEventType(this.eventType));
  connection.setPattern(pattern);
  connection.setSingularity(singularity);
  return connection;
}

public static class MyExternalConnection implements ExternalConnection {

private final EventType targetEventType;

public MyExternalConnection(EventType eventtype) {
      this.targetEventType = eventtype;
}
private String pattern;

public void setPattern(String pattern) {
  this.pattern = pattern;
}
private String singularity;

public void setSingularity(String singularity) {
  this.singularity = singularity;
}

@Override
```

```
  public void close() throws Exception {
  }

  @Override
  public ExternalPreparedStatement prepareStatement(String relationName,
    List<String> relationAttrs, ExternalPredicate predicate)
          throws Exception {
    return new MyExternalPreparedStatement(this.targetEventType,
    predicate,this.pattern,this.singularity);
  }

  @Override
  public boolean supportsPredicate(ExternalPredicate predicate)
      throws Exception {
    return true;
  }
}

public static class MyExternalPreparedStatement implements
                                        ExternalPreparedStatement {
  private ExternalPredicate predicate;
  private Object[] keys = new Object[10];
  private final EventType targetEventType;
  private  Pattern pattern;
  private  Pattern singularity;

  public MyExternalPreparedStatement(EventType targetEventType,
  ExternalPredicate predicate,String pattern,String singularity) {
    this.targetEventType = targetEventType;
    this.predicate = predicate;
    if (pattern == null) {
      this.pattern = Pattern.compile(".*");
    } else {
      this.pattern = Pattern.compile(pattern);
    }
    if (singularity == null) {
      this.singularity = Pattern.compile("$.^");
    } else {
      this.singularity = Pattern.compile(singularity);
    }
  }

  @Override
  public void close() throws Exception {}

  @Override
  public Iterator<Object> executeQuery() throws Exception {
    List<Object> result = new ArrayList<Object>();
    List attrs = predicate.getAttributes();
    String value="";
    for(int i = 0;i<attrs.size();i++) {
      if(keys[i+1] ==null) {
        System.out.println("empty="+keys[i+1]);
        return result.iterator();
      }
      value = keys[i+1].toString();
      Matcher m = this.pattern.matcher(value);
      if(!m.matches()) {
        System.out.println("empty="+value);
        return result.iterator();
      }
```

```
    }
  TupleValue event = (TupleValue) this.targetEventType.createEvent();
  EventProperty[] properties = this.targetEventType.getProperties();
  for (int i = 0; i < properties.length; i++) {
    properties[i].setValue(event, createValue(properties[i], value));
  }
  System.out.println("one="+value);
  result.add(event);

  Matcher s = this.singularity.matcher(value);
  if(s.matches()) {
    System.out.println("double="+value);
    result.add(event);
  }
  return result.iterator();
}

  private Object createValue(EventProperty property, String value) {
    Type propertyType = property.getType();
    Object ret;
    if (Type.INT == propertyType) {
      ret = Integer.valueOf(value);
    } else if (Type.BIGINT == propertyType) {
      ret = Long.valueOf(value);
    } else if (Type.FLOAT == propertyType) {
      ret = Float.valueOf(value);
    } else if (Type.DOUBLE == propertyType) {
      ret = Double.valueOf(value);
    } else if (Type.BYTE == propertyType) {
      ret = value.getBytes();
    } else if (Type.BOOLEAN == propertyType) {
      ret = false;
    } else if (Type.TIMESTAMP == propertyType) {
      ret = new Date();
    } else if (Type.INTERVAL == propertyType) {
      ret = Long.valueOf(value);
    } else {
      ret = value;
    }
    return ret;
  }

  @Override
  public void setBigDecimal(int paramIndex, BigDecimal x)
    throws Exception {
    this.keys[paramIndex] = x;
  }

  @Override
  public void setBoolean(int paramIndex, boolean x) throws Exception {
    this.keys[paramIndex] = x;
  }

  @Override
  public void setBytes(int paramIndex, byte[] x) throws Exception {
    this.keys[paramIndex] = x;
  }

  @Override
  public void setDouble(int paramIndex, double x) throws Exception {
    this.keys[paramIndex] = x;
```

```
  }

  @Override
  public void setFloat(int paramIndex, float x) throws Exception {
    this.keys[paramIndex] = x;
  }

  @Override
  public void setInt(int paramIndex, int x) throws Exception {
    this.keys[paramIndex] = x;
  }

  @Override
  public void setLong(int paramIndex, long x) throws Exception {
    this.keys[paramIndex] = x;
  }

  @Override
  public void setNull(int paramIndex, int x) throws Exception {
    this.keys[paramIndex] = x;
  }

  @Override
  public void setString(int paramIndex, String x) throws Exception {
    this.keys[paramIndex] = x;
  }

  @Override
  public void setTimestamp(int paramIndex, Timestamp x) throws Exception {
    this.keys[paramIndex] = x;
  }
}

  // @Override
  public Class getBeanClass() {
    return this.getClass();
  }

  // @Override
    public Map getInstancePropertiesAsMap() {
    return null;
  }

  // @Override
  public String getProvider() {
    return null;
  }
}
```

MyActivator.java

```
package tests.functional.cartridge.userdefine.common.libs.datasource;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;

public class MyActivator  implements BundleActivator {
  private ServiceRegistration reg;

  @Override
```

```
  public void start(BundleContext context) throws Exception {
  }

  @Override
  public void stop(BundleContext arg0) throws Exception {
  }

 }
```

MyHandler.java

```
package tests.functional.cartridge.userdefine.common.libs.datasource;

import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.config.BeanDefinitionHolder;
import org.springframework.beans.factory.xml.NamespaceHandler;
import org.springframework.beans.factory.xml.NamespaceHandlerSupport;
import org.springframework.beans.factory.xml.ParserContext;
import org.w3c.dom.Element;
import org.w3c.dom.Node;

import
tests.functional.cartridge.externaldatasource.common.apps.cart2.spring.FileDefinition
Parser;

public class MyHandler implements NamespaceHandler {
  private NamespaceHandlerSupport support = new NamespaceHandlerSupport() {
    public void init() {
     registerBeanDefinitionParser("udds", new UddsDefinitionParser());
    }
  };

  @Override
  public BeanDefinitionHolder decorate(Node node, BeanDefinitionHolder definition,
                                       ParserContext parserContext) {
    return this.support.decorate(node, definition, parserContext);
  }

  @Override
  public void init() {
    this.support.init();
  }

  @Override
  public BeanDefinition parse(Element element, ParserContext parserContext) {
    return this.support.parse(element, parserContext);
  }
}
```

UddsDefinitionParser.java

```
package tests.functional.cartridge.userdefine.common.libs.datasource;

import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.xml.AbstractSingleBeanDefinitionParser;
import org.springframework.core.Conventions;
import org.w3c.dom.Attr;
import org.w3c.dom.Element;
import org.w3c.dom.NamedNodeMap;

public class UddsDefinitionParser extends AbstractSingleBeanDefinitionParser {
```

```
  protected Class<?> getBeanClass(Element element) {
    return UddsFactoryBean.class;
  }

  protected void doParse(Element element, BeanDefinitionBuilder builder) {
    NamedNodeMap attributes = element.getAttributes();

    for (int x = 0; x < attributes.getLength(); x++) {
    Attr attribute = (Attr) attributes.item(x);
    String name = attribute.getLocalName();

    if ("id".equals(name))
      continue;
      builder.addPropertyValue(
      Conventions.attributeNameToPropertyName(name),
      attribute.getValue());
    }
  }
}
```

UddsFactoryBean.java

```
package tests.functional.cartridge.userdefine.common.libs.datasource;

import org.osgi.framework.BundleContext;
import org.springframework.beans.factory.BeanNameAware;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.config.AbstractFactoryBean;
import org.springframework.osgi.context.BundleContextAware;
import org.springframework.osgi.extensions.annotation.ServiceReference;
import com.bea.wlevs.ede.api.EventTypeRepository;

public class UddsFactoryBean extends AbstractFactoryBean<MyCartridgeSource>
                  implements InitializingBean, BeanNameAware, BundleContextAware {

  private EventTypeRepository etr;
  private BundleContext bundleContext;
  private String beanName;
  private String eventType;
  private String pattern;
  private String singularity;

  @ServiceReference
  public void setEventTypeRepository(EventTypeRepository etr) {
    this.etr = etr;
  }

  @Override
  public void setBundleContext(BundleContext context) {
    this.bundleContext = context;
  }

  @Override
  public void setBeanName(String name) {
        this.beanName = name;
  }

  public String getEventType() {
    return this.eventType;
  }
```

```
  public void setEventType(String eventType) {
    this.eventType = eventType;
  }

  private String keyProperty;

  public String getKeyProperty() {
    return keyProperty;
  }

  public void setKeyProperty(String names) {
    keyProperty = names;
  }

  @Override
  protected MyCartridgeSource createInstance() throws Exception {
    MyCartridgeSource ret = new MyCartridgeSource();
    System.out.println("id="+this.beanName+",eventType="+this.eventType);
    ret.setId(this.beanName);
    ret.setEventType(this.eventType);
    ret.setPattern(this.pattern);
    ret.setSingularity(singularity);
    ret.setKeyProperty(keyProperty);
    return ret;
  }

  @Override
  public Class<?> getObjectType() {
    return MyCartridgeSource.class;
  }

  public void setPattern(String pattern) {
    this.pattern = pattern;
  }

  public String getPattern() {
    return pattern;
  }

  public void setSingularity(String singularity) {
    this.singularity = singularity;
  }

  public String getSingularity() {
    return singularity;
  }
}
```

A

Oracle Spatial Command and API Reference

Syntax and example information for Oracle Spatial commands and APIs that apply to Event Processing are provided.

- ANYINTERACT
- buffer
- bufferPolygon
- CONTAIN
- convertTo2D
- convertTo3D
- createCircle
- createElemInfo
- createGeometry
- createLinearLineString
- createLinearMultiLineString
- createLinearPolygon
- createMultiPoint
- createPoint
- createRectangle
- distance
- einfogenerator
- FILTER
- get2dMbr
- INSIDE
- INSIDE3D
- NN
- ordsgenerator
- to_Geometry
- to_J3D_Geometry
- to_JGeometry
- WITHINDISTANCE.

A.1 ANYINTERACT

The `ANYINTERACT` Oracle Spatial geometric relation operator returns `true` when the key interacts with the geometry (`geom`), and `false` otherwise.

Syntax

```
ANYINTERACT@spatial(geom, key, tol)
```

- `geom`: Any supported geometry type.
- `key`: A `GTYPE_POINT`, `GTYPE_CURVE`, `GTYPE_POLYGON`, `GTYPE_SURFACE`, `GTYPE_COLLECTION`, `GTYPE_MULTIPOINT`, `GTYPE_MULTICURVE`, `GTYPE_MULTIPOLYGON`, `GTYPE_SOLID`, or `GTYPE_MULTISOLID` geometry type.

  The geometry type of this geometry must be `GTYPE_POINT` or a `RUNTIME_EXCEPTION` will be thrown.
- `tol`: The tolerance as a `double` value. The tolerance value expands the thickness of the boundaries.

Example

```
<view id="op_in_where">
    RStream(
        select
            loc.customerId,
            shop.shopId
        from
            LocGeomStream[NOW] as loc,
            ShopGeomRelation as shop
        where
            ANYINTERACT@spatial(shop.geom, loc.curLoc, 5.0d) = true
    )
</view>
<view id="op_in_proj">
    RStream(
        select
            loc.customerId,
            shop.shopId,
            ANYINTERACT@spatial(shop.geom, loc.curLoc, 5.0d)
        from
            LocGeomStream[NOW] as loc,
            ShopGeomRelation as shop
    )
</view>
```

A.2 buffer

The `com.oracle.cep.cartridge`.spatial.Geometry `buffer` method returns a new `oracle.spatial.geometry.JGeometry` object that is the buffered version of the input geometry.

Syntax

- `bufferWidth`: The distance value used for this buffer as a `double`.

This value is assumed to be in the same unit as the Unit of Projection for projected geometry. If the geometry is geodetic, this buffer width should be in meters.

- `SMA`: The Semi Major Axis as a `double`.

  Set this parameter when the geometry is geodetic.

- `iFlat`: The Flattening from CS parameters as a `double`.

  Set this parameter when the geometry is geodetic.

- `arcT`: The `arc_tolerance` for geodetic arc densification as a `double`.

```
com.oracle.cep.cartridge,spatial.geometry.buffer(bufferWidth, SMA, iFlat, actT)
```

Example

```
<view id="LocGeomStream" schema="customerId curLoc">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.buffer(13, 2, 4, 7)
    from
        CustomerLocStream
</view>
```

A.3 bufferPolygon

The `com.oracle.cep.cartridge.spatial.Geometry bufferPolygon` method returns a `com.oracle.cep.cartridge.spatial.Geometry` object that is the buffered version of the input `oracle.spatial.geometry.JGeometry` polygon. This method creates buffered polygons to a specified distance around the input features.

Syntax

```
com.oracle.cep.cartridge.spatial.Geometry.bufferPolygon(polygon, distance)
```

- `polygon`: An `oracle.spatial.geometry.JGeometry` polygon.
- `distance`: A `double` value that specifies the distance around the input features.

  The `distance` value is assumed to be in the same unit as the Unit of Projection for projected geometry. If the geometry is geodetic, the buffer `distance` should be in meters.

Example

This method obtains parameters from the Oracle Spatial application context. You must use the `spatial` link name (`@spatial`) to associate the method call with the Oracle Spatial application context See Oracle Spatial Application Context.

```
com.oracle.cep.cartridge.spatial.Geometry.bufferPolygon@spatial(geom, 1300)
```

The following example creates a buffered polygon. Because this example depends on the Oracle Spatial application context, it uses the `spatial` link name.

```
<view id="LocGeomStream" schema="customerId curLoc">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.bufferPolygon@spatial(geom, 13)
    from
        CustomerLocStream
</view>
```

A.4 CONTAIN

The Oracle Spatial geometric relation `CONTAIN` operator returns `true` when a geometry is contained by another geometry, and `false` otherwise.

Syntax

```
CONTAIN@spatial(geom, key)
```

- `geom`: Any supported geometry type.
- `key`: A `GTYPE_POINT`, `GTYPE_CURVE`, `GTYPE_POLYGON`, `GTYPE_SURFACE`, `GTYPE_COLLECTION`, `GTYPE_MULTIPOINT`, `GTYPE_MULTICURVE`, `GTYPE_MULTIPOLYGON`, `GTYPE_SOLID`, or `GTYPE_MULTISOLID` geometry type.

Example

```
<view id="op_in_where">
    RStream(
        select
            loc.customerId,
            shop.shopId
        from
            LocGeomStream[NOW] as loc,
            ShopGeomRelation as shop
        where
            CONTAIN@spatial(shop.geom, loc.curLoc, 5.0d) = true
    )
</view>
<view id="op_in_proj">
    RStream(
        select
            loc.customerId,
            shop.shopId,
            CONTAIN@spatial(shop.geom, loc.curLoc, 5.0d)
        from
            LocGeomStream[NOW] as loc,
            ShopGeomRelation as shop
    )
</view>
```

A.5 convertTo2D

The `com.oracle.cep.cartridge.spatial.Geometry convertTo2D` method converts an `oracle.spatial.geometry.JGeometry` 3D object to an `oracle.spatial.geometry.JGeometry` 2D object.

Syntax

```
com.oracle.cep.cartridge.spatial.Geometry.convertTo2D(geom)
```

The `geom` parameter is an oracle.spatial.geometry.JGeometry 3D object.

Example

```
<view id="LocGeomStream" schema="customerId curLoc">
    select
        customerId,
```

```
        com.oracle.cep.cartridge.spatial.Geometry.convertTo2D(geom)
    from
        CustomerLocStream
</view>
```

A.6 convertTo3D

The `com.oracle.cep.cartridge.spatial.Geometry` `convertTo3D` method converts an `oracle.spatial.geometry.JGeometry` 2D object into an `oracle.spatial.geometry.JGeometry` 3D object. The conversion pads `z` coordinates to zero.

Syntax

```
com.oracle.cep.cartridge.spatial.Geometry.convertTo3D(geom)
```

The `geom` parameter is an `oracle.spatial.geometry.JGeometry` 2D object.

Example

```
<view id="LocGeomStream" schema="customerId curLoc">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.convertTo3D(geom)
    from
        CustomerLocStream
</view>
```

A.7 createCircle

The `com.oracle.cep.cartridge.spatial.Geometry` `createCircle` method returns a `com.oracle.cep.cartridge.spatial.Geometry` object that is a 2D or 3D circle.

Create a 2D Circle Syntax

```
com.oracle.cep.cartridge.spatial.Geometry.createCircle(x, y, radius)
com.oracle.cep.cartridge.spatial.Geometry.createCircle(x, y, radius, srid)
```

- `x`: The x ordinate of the circle's center as a `double`.
- `y`: The y ordinate of the circle's center as a `double`.
- `radius`: The `arc_tolerance` for geodetic arc densification as a `double`.
- `srid`: The optional `SDO_SRID` of the circle as an `int`. When the `srid` parameter is omitted, add the spatial link name as shown in the examples.

Create a 3D Circle Syntax

```
com.oracle.cep.cartridge.spatial.Geometry.createCircle(x1, y1, x2, y2, x3, y3)
com.oracle.cep.cartridge.spatial.Geometry.createCircle(srid, x1, y1, x2, y2, x3,
                                                                          y3)
```

Specify three coordinates to form the circumference with the following arguments:

- `x1`: The x ordinate of point 1 as a `double`.
- `y1`: The y ordinate of point 1 as a `double`.
- `x2`: The x ordinate of point 2 as a `double`.

- `y2`: The y ordinate of point 2 as a `double`.
- `x3`: The x ordinate of point 3 as a `double`.
- `y3`: The y ordinate of point 3 as a `double`.
- `srid`: The optional `SRID` of the circle as an `int`. When you omit the `srid` parameter, add the spatial link name (`@spatial`) as shown in the examples.

Examples

If you omit the optional `srid` parameter, then the method obtains parameters from the Oracle Spatial data cartridge application context. In this case, use the spatial link name (`@spatial`) to associate the method call with the Oracle Spatial data cartridge application context. See Oracle Spatial Application Context

```
com.oracle.cep.cartridge.spatial.Geometry.createCircle@spatial(x, y)
```

The following example creates a 2D circle with the `srid` parameter. Because this example uses the `srid` parameter, it does not need the `spatial` link name.

```
<view id="LocGeomStream" schema="customerId curLoc">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.createCircle(x, y, 300, srid)
    from
        CustomerLocStream
</view>
```

A.8 createElemInfo

The `com.oracle.cep.cartridge.spatial.Geometry createElemInfo` method returns a single element info value as an `int[]` from the given arguments. See einfogenerator for an alternative.

Syntax

```
com.oracle.cep.cartridge.spatial.Geometry.createElemInfo(offset, etype, interp)
```

- `soffset`: The offset, as an `int`, within the ordinates array where the first ordinate for this element is stored.

  `SDO_STARTING_OFFSET` values start at 1 and not at 0. Thus, the first ordinate for the first element will be at `SDO_GEOMETRY.Ordinates(1)`. If there is a second element, its first ordinate will be at `SDO_GEOMETRY.Ordinates(`*`n`*` * 3 + 2)`, where *n* reflects the position within the `SDO_ORDINATE_ARRAY` definition.

- `etype`: The type of the element as an `int`.

  Oracle Spatial supports `SDO_ETYPE` values 1, 1003, and 2003 are considered simple elements (not compound types). They are defined by a single triplet entry in the element info array. These types are:

  – 1: point.
  – 1003: exterior polygon ring (must be specified in counterclockwise order).
  – 2003: interior polygon ring (must be specified in clockwise order).

  These types are further qualified by the `SDO_INTERPRETATION`.

> **Note:**
>
> Do not mix 1-digit and 4-digit `SDO_ETYPE` values in the same geometry.

- `interp`: The interpretation as an `int`.

  For an `SDO_ETYPE` that is a simple element (1, 1003, or 2003), the `SDO_INTERPRETATION` attribute determines how the sequence of ordinates for this element is interpreted. For example, a polygon boundary may be made up of a sequence of connected straight line segments.

  If a geometry consists of more than one element, then the last ordinate for an element is always one less than the starting offset for the next element. The last element in the geometry is described by the ordinates from its starting offset to the end of the ordinates varying length array.

  Table A-1 describes the relationship between `SDO_ETYPE` and `SDO_INTERPREATION`.

Table A-1 SDO_ETYPE and SDO_INTERPRETATION

SDO_ETYPE	SDO_INTERPRETATION	Description
0	Any numeric value	Use to model geometry types not supported by Oracle Spatial.
1	1	Point type.
1	0	Orientation for an oriented point.
1003 or 2003	1	Simple polygon with vertices connected by straight line segments. You must specify a point for each vertex; and the last point specified must be exactly the same point as the first (within the tolerance value), to close the polygon. For example, for a 4-sided polygon, specify 5 points, with point 5 the same as point 1.
1003 or 2003	3	Rectangle type (optimized rectangle). A bounding rectangle such that only two points, the lower-left and the upper-right, are required to describe it. The rectangle type can be used with geodetic or non-geodetic data. However, with geodetic data, use this type only to create a query window (not for storing objects in the database).

Example

```
<view id="ShopGeom">
    select  com.oracle.cep.cartridge.spatial.Geometry.createGeometry@spatial(
        com.oracle.cep.cartridge.spatial.Geometry.GTYPE_POLYGON,
        com.oracle.cep.cartridge.spatial.Geometry.createElemInfo(1, 1003, 1),
        ordsgenerator@spatial(
            lng1, lat1, lng2, lat2, lng3, lat3,
            lng4, lat4, lng5, lat5, lng6, lat6
        )
    ) as geom
    from ShopDesc
</view>
```

A.9 createGeometry

The `com.oracle.cep.cartridge.spatial.Geometry createGeometry` method returns a new 2D `oracle.cep.cartridge.spatial.Geometry` object.

Syntax

```
com.oracle.cep.cartridge.spatial.Geometry(gtype, elemInfo, ordinates)
com.oracle.cep.cartridge.spatial.Geometry(gtype, srid, elemInfo, ordinates)
```

- `gtype`: The geometry type as an `int`.

  For more information, see Table A-2.
- `eleminfo`: The geometry element info as an `int[]`.

  For more information, see createElemInfo.
- `ordinates`: The geometry ordinates as a `double[]`.
- `srid`: The optional `SDO_SRID` of the geometry as an `int`. When you omit the `srid` parameter, add the spatial link name (`@spatial`) as shown in the examples.

Examples

If you omit the `srid` parameter, then this method obtains parameters from the Oracle Spatial application context. In this case, you must use the `spatial` link name to associate the method call with the Oracle Spatial application context: For more information, see Oracle Spatial Application Context.

```
com.oracle.cep.cartridge.spatial.Geometry.createGeometry@spatial(gtype, eleminfo,
                                                                 ordinates)
```

The following examples creates a geometry with the srid parameter. Because this example uses the `srid` argument, it does not need the `@spatial` link name.

```
<view id="ShopGeom">
    select  com.oracle.cep.cartridge.spatial.Geometry.createGeometry(
        com.oracle.cep.cartridge.spatial.Geometry.GTYPE_POLYGON,
        srid,
        com.oracle.cep.cartridge.spatial.Geometry.createElemInfo(1, 1003, 1,
                                                                 srid),
        ordsgenerator@spatial(
            lng1, lat1, lng2, lat2, lng3, lat3,
            lng4, lat4, lng5, lat5, lng6, lat6
        )
    ) as geom
    from ShopDesc
</view>
```

A.10 createLinearLineString

The `com.oracle.cep.cartridge.spatial.Geometry createLinearLineString` method returns a new 3D `com.oracle.cep.cartridge.spatial.Geometry` geometry that is a linear line string with element info of `{1, 2, 1}`. If the dimensionality of the given coordinates is 2, the z coordinates are padded to zero.

Syntax

```
com.oracle.cep.cartridge.spatial.Geometry.createLinearLineString(coords, dim)
com.oracle.cep.cartridge.spatial.Geometry.createLinearLineString(srid, coords,
                                                                  dim)
```

- `coords`: The coordinates of the linear line string as a `double[]`.
- `dim`: The dimensionality of the given coordinates as an `int`.
- `srid`: The optional `SDO_SRID` of the geometry as an `int`. When the srid parameter is omitted, add the spatial link name as shown in the examples.

  For more information, see Oracle Spatial Application Context.

Examples

If you omit the `srid` parameter, then this method obtains parameters from the Oracle Spatial data cartridge application context. You must use the spatial link name (`@spatial`) to associate the method call with the Oracle Spatial data cartridge application context. See Oracle Spatial Application Context.

```
com.oracle.cep.cartridge.spatial.Geometry.createLinearLineString@spatial(coords,
                                                                         dim)
```

The following examples creates a linear line string with the srid paramter. Because this example uses the `srid` parameter, it does not use the `@spatial` link.

```
<view id="LocGeomStream" schema="customerId curLoc">
 select
  customerId,
  com.oracle.cep.cartridge.spatial.Geometry.createLinearLineString(coords, dim, srid)
 from
  CustomerLocStream
</view>
```

A.11 createLinearMultiLineString

The `com.oracle.cep.cartridge.spatial.Geometry` `createMultiLineString` method returns a new 3D `com.oracle.cep.cartridge.spatial.Geometry` geometry that is a linear multiline string. If the dimensionality of the given coordinates is 2, then the z coordinates are padded to zero.

Syntax

```
com.oracle.oep.cartridge.spatial.Geometry.createMultiLineString(coords, dim)
com.oracle.cep.cartridge.spatial.Geometry.createMultiLineString(srid, coords, dim)
```

- `coords`: the coordinates of the linear line string as a `double[][]`.
- `dim`: the dimensionality of the given coordinates as an `int`.
- `srid`: the optional `SRID` of the geometry as an `int`. When you omit the srid parameter, add the spatial link name (`@spatial`) as shown in the examples.

Examples

If you omit the `srid` parameter, then this method obtains parameters from the Oracle Spatial data cartridge application context. You must use the `spatial` link name

(@spatial) to associate the method call with the Oracle Spatial data cartridge application context. See Oracle Spatial Application Context.

```
com.oracle.cep.cartridge.spatial.Geometry.createLinearMultiLineString@spatial(
                                                                    coords, dim)
```

The following example creates a linear multiline linear string. Because this example uses the srid argument, it does not use the spatial link name.

```
<view id="LocGeomStream" schema="customerId curLoc">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.createLinearMultiLineString(coords, dim,
                                                                               srid)
    from
        CustomerLocStream
</view>
```

A.12 createLinearPolygon

The com.oracle.cep.cartridge.spatial.Geometry createLinearPolygon method returns a new com.oracle.cep.cartridge.spatial.Geometry object that is a 2D simple linear polygon without holes. If the coordinate array does not close itself (the last coordinate is not the same as the first), then this method copies the first coordinate and appends this coordinate value to the end of the input coordinates array.

Syntax

```
com.oracle.cep.cartridge.spatial.Geometry.createLinearPolygon(coords{])
com.oracle.cep.cartridge.spatial.Geometry.createLinearPolygon(srid, coords[])
```

- coords: the coordinates of the linear polygon as a double[].
- srid: the optional SRID of the geometry as an int. When you omit the srid parameter, add the spatial link name (@spatial) as shown in the examples.

Examples

If you omit the srid parameter, then the method obtains parameters from the Oracle Spatial application context. In this case, you must use the spatial link name (@spatial) to associate the method call with the Oracle Spatial application context. See Oracle Spatial Application Context.

```
com.oracle.cep.cartridge.spatial.Geometry.createLinearPolygon@spatial(coords)
```

The following example creates a linear polygon with the srid parameter. Because this example uses the srid argument, it does not use the spatial link name.

```
<view id="LocGeomStream" schema="customerId curLoc">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.createLinearPolygon(coords, srid)
    from
        CustomerLocStream
</view>
```

A.13 createMultiPoint

The com.oracle.cep.cartridge.spatial.Geometry createMultiPoint method returns a com.oracle.cep.cartridge.spatial.Geometry object which is a multipoint geometry

Syntax

```
com.oracle.cep.cartridge.spatial.Geometry.createMultiPoint(coords[][], dim)
com.oracle.cep.cartridge.spatial.Geometry.createMultiPoint(srid, coords[][], dim)
```

- `coords`: the array of arrays of type `double` each containing one point.
- `dim`: the dimensionality of each point as an `int`.
- `srid`: the optional `SRID` of the geometry as an `int`. When you omit the srid parameter, add the spatial link name (`@spatial`) as shown in the examples.

Examples

If you omit the `srid` parameter, then this method obtains parameters from the Oracle Spatial data cartridge application context. In this case, you must use the spatial link name (`@spatial`) to associate the method call with the Oracle Spatial data cartridge application context. See Oracle Spatial Application Context.

```
com.oracle.cep.cartridge.spatial.Geometry.createMultiPoint@spatial(coords, dim)
```

The following example creates a multipoint geometry with the `srid` parameter. Because this example uses the `srid` parameter, it does not use the `spatial` link name.

```
<view id="LocGeomStream" schema="customerId curLoc">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.createMultiPoint(coords, dim, srid)
    from
        CustomerLocStream
</view>
```

A.14 createPoint

The `com.oracle.cep.cartridge.spatial.Geometry` `createPoint` method returns a new `com.oracle.cep.cartridge.spatial.Geometry` object that is a 3D point.

Syntax

```
com.oracle.cep.cartridge.spatial.Geometry.createPoint(x, y)
com.oracle.cep.cartridge.spatial.Geometry.createPoint(srid, x, y)
```

- `x`: the x coordinate of the lower left as a `double`.
- `y`: the y coordinate of the lower left as a `double`.
- `srid`: the optional `SRID` of the geometry as an `int`. When you omit the `srid` parameter, add the spatial link name (`@spatial`) as shown in the examples.

If you omit the `srid` parameter, then this method obtains parameters from the Oracle Spatial application context. In this case, you must use the spatial link name (`@spatial`) to associate the method call with the Oracle Spatial application context. See Oracle Spatial Application Context

```
com.oracle.cep.cartridge.spatial.Geometry.createPoint@spatial(x, y)
```

The following example creates a point with the `srid` parameter. Because this example uses the `srid` parameter, it does not use the `spatial` link name.

```
<view id="LocGeomStream" schema="customerId curLoc">
    select
```

```
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.createPoint(lng, lat, srid)
    from
        CustomerLocStream
</view>
```

A.15 createRectangle

The `com.oracle.cep.cartridge.spatial.Geometry` `createRectangle` method returns a new `com.oracle.cep.cartridge.spatial.Geometry` object that is a 2D rectangle

Syntax

```
com.oracle.cep.cartridge.spatial.Geometry.createRectangle(x1, y1, x2, y2)
com.oracle.cep.cartridge.spatial.Geometry.createRectangle(srid, x1, y1, x2, y2)
```

- `x1`: the x coordinate of the lower left as a `double`.
- `y1`: the y coordinate of the lower left as a `double`.
- `x2`: the x coordinate of the upper right as a `double`.
- `y2`: the y coordinate of the upper right as a `double`.
- `srid`: the optional `SRID` of the geometry as an `int`.

Examples

If you omit the `srid` parameter, then this method obtains parameters from the Oracle Spatial application context. In this case, you must use the `spatial` link name (`@spatial`) to associate the method call with the Oracle Spatial application context. See Oracle Spatial Application Context.

```
com.oracle.cep.cartridge.spatial.Geometry.createRectangle@spatial(x1, y1, x2, y2)
```

The following example creates a rectangle. Because this example uses the `srid` parameter, it does not need the `spatial` link name.

```
<view id="LocGeomStream" schema="customerId curLoc">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.createRectangle(x1, y1, x2, y2, srid)
    from
        CustomerLocStream
</view>
```

A.16 distance

The `com.oracle.cep.cartridge.spatial.Geometry` `distance` method calculates the distance between two geometries as a `double`.

Syntax

```
com.oracle.cep.cartridge.spatial.Geometry.distance(g1, g2)
com.oracle.cep.cartridge.spatial.Geometry.distance(geoParam, g1, g2)
```

To calculate the distance between a `com.oracle.cep.cartridge.spatial.Geometry` object and another, use the non-static `distance` method of the current `Geometry` object with the following arguments:

- `g`: the other `com.oracle.cep.cartridge.spatial.Geometry` object.

To calculate the distance between two `com.oracle.cep.cartridge.spatial.Geometry` objects, use the static `distance` method with the following arguments:

- `g1`: the first `com.oracle.cep.cartridge.spatial.Geometry` object.
- `g2`: the second `com.oracle.cep.cartridge.spatial.Geometry` object.

Examples

This method obtains parameters from the Oracle Spatial application context. You must use the `spatial` link name to associate the method call with the Oracle Spatial application context. See Oracle Spatial Application Context.

```
com.oracle.cep.cartridge.spatial.Geometry.distance@spatial(geom)
com.oracle.cep.cartridge.spatial.Geometry.distance@spatial(geom1, geom2)
```

The following example calculates the distance between two geometries. Because the `distance` method depends on the Oracle Spatial application context, it must use the `spatial` link name.

```
<view id="LocGeomStream" schema="customerId curLoc">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.createRectangle(x1, y1, x2, y2, srid)
    from
        CustomerLocStream
    where
        com.oracle.cep.cartridge.spatial.Geometry.distance@spatial(geom1, geom2) < 5
</view>
```

A.17 einfogenerator

The `einfogenerator` Oracle CQL function returns a single `info` element value as in `int[]` from the given arguments. Alternately, see createElemInfo if you prefer to use the `com.oracle.cep.cartridge.spatial.Geometry.createElemInfo` method.

Syntax

```
einfogenerator@spatial(offset, etype, interp)
```

- `offset`: the offset, as an `int`, within the ordinates array where the first ordinate for this element is stored.

  `SDO_STARTING_OFFSET` values start at 1 and not at 0. Thus, the first ordinate for the first element will be at `SDO_GEOMETRY.Ordinates(1)`. If there is a second element, its first ordinate will be at `SDO_GEOMETRY.Ordinates(`*`n`*` * 3 + 2)`, where *n* reflects the position within the `SDO_ORDINATE_ARRAY` definition.

- `etype`: the type of the element as an `int`.

  Oracle Spatial supports `SDO_ETYPE` values 1, 1003, and 2003 are considered simple elements (not compound types). They are defined by a single triplet entry in the element info array. These types are:

  – 1: point.

  – 1003: exterior polygon ring (must be specified in counterclockwise order).

  – 2003: interior polygon ring (must be specified in clockwise order).

  These types are further qualified by the `SDO_INTERPRETATION`.

> **Note:**
>
> You cannot mix 1-digit and 4-digit SDO_ETYPE values in a single geometry.

- interp: the interpretation as an int.

  For an SDO_ETYPE that is a simple element (1, 1003, or 2003) the SDO_INTERPRETATION attribute determines how the sequence of ordinates for this element is interpreted. For example, a polygon boundary may be made up of a sequence of connected straight line segments.

  If a geometry consists of more than one element, then the last ordinate for an element is always one less than the starting offset for the next element. The last element in the geometry is described by the ordinates from its starting offset to the end of the ordinates varying length array.

  Table A-2 describes the relationship between SDO_ETYPE and SDO_INTERPREATION.

Table A-2 SDO_ETYPE and SDO_INTERPRETATION

SDO_ETYPE	SDO_INTERPRETATION	Description
0	Any numeric value	Used to model geometry types not supported by Oracle Spatial.
1	1	Point type.
1	0	Orientation for an oriented point.
1003 or 2003	1	Simple polygon whose vertices are connected by straight line segments. You must specify a point for each vertex; and the last point specified must be exactly the same point as the first (within the tolerance value), to close the polygon. For example, for a 4-sided polygon, specify 5 points, with point 5 the same as point 1.
1003 or 2003	3	Rectangle type (sometimes called optimized rectangle). A bounding rectangle such that only two points, the lower-left and the upper-right, are required to describe it. The rectangle type can be used with geodetic or non-geodetic data. However, with geodetic data, use this type only to create a query window (not for storing objects in the database).

Examples

This is an Oracle CQL function so you invoke this function with the spatial link name and without a package prefix. The following example creates the element information for a geometry.

```
view id="ShopGeom">
    select  com.oracle.cep.cartridge.spatial.Geometry.createGeometry@spatial(
        com.oracle.cep.cartridge.spatial.Geometry.GTYPE_POLYGON,
        einfogenerator@spatial(1, 1003, 1),
        ordsgenerator@spatial(
            lng1, lat1, lng2, lat2, lng3, lat3,
            lng4, lat4, lng5, lat5, lng6, lat6
        )
```

```
    ) as geom
    from ShopDesc
</view>
```

A.18 FILTER

The `FILTER` Oracle Spatial geometric filter operator returns true for object pairs that are non-disjoint, and `false` otherwise.

```
FILTER@spatial(key, tol)
```

- `key`: A `GTYPE_POINT`, `GTYPE_CURVE`, `GTYPE_POLYGON`, `GTYPE_SURFACE`, `GTYPE_COLLECTION`, `GTYPE_MULTIPOINT`, `GTYPE_MULTICURVE`, `GTYPE_MULTIPOLYGON`, `GTYPE_SOLID`, or `GTYPE_MULTISOLID` geometry type.
- `tol`: the tolerance as a `double` value.

Example

This is an Oracle Spatial geometric filter operator so you invoke this function with the spatial link name and without a package prefix. The following example test for object pairs that are non-disjoint.

```
<view id="filter">
    RStream(
        select loc.customerId, shop.shopId
        from LocGeomStream[NOW] as loc, ShopGeomRelation as shop
        where FILTER@spatial(loc.curLoc, 5.0d) = true
    )
</view>
```

A.19 get2dMbr

The `com.oracle.cep.cartridge.spatial.Geometry` `get2dMbr` method returns the Minimum Bounding Rectangle (MBR) of a given `Geometry` as a `double[][]`.

Syntax

```
com.oracle.cep.cartridge.spatial.Geometry.get2DMbr(geom)
```

The `geom` parameter is a `com.oracle.cep.cartridge.spatial.Geometry` object for which the method returns the bounding rectangle. The returned bounding rectangle contains the following values:

- `[0][0]`: `minX`
- `[0][1]`: `maxX`
- `[1][0]`: `minY`
- `[1][1]`: `maxY`

Examples

The following example returns a bounding rectangle for `geom`.

```
<view id="LocGeomStream" schema="customerId mbr">
  select
    customerId,
    com.oracle.cep.cartridge.spatial.Geometry.get2dMbr(geom)
```

```
  from
    CustomerLocStream
  where
    com.oracle.cep.cartridge.spatial.Geometry.distance@spatial(geom1, geom2) < 5
</view>
```

A.20 INSIDE

The `INSIDE` Oracle Spatial geometric relation returns `true` if `GTYPE_POINT` is inside the geometry, and `false` otherwise.

Syntax

```
INSIDE@spatial(geom, key)
```

- `geom`: any supported geometry type.
- `key`: A `GTYPE_POINT`, `GTYPE_CURVE`, `GTYPE_POLYGON`, `GTYPE_SURFACE`, `GTYPE_COLLECTION`, `GTYPE_MULTIPOINT`, `GTYPE_MULTICURVE`, `GTYPE_MULTIPOLYGON`, `GTYPE_SOLID`, or `GTYPE_MULTISOLID` geometry type.

Example

The following Oracle CQL query tests whether a point is inside the geometry.

```
<view id="op_in_where">
    RStream(
        select
            loc.customerId,
            shop.shopId
        from
            LocGeomStream[NOW] as loc,
            ShopGeomRelation as shop
        where
            INSIDE@spatial(shop.geom, loc.curLoc, 5.0d) = true
    )
</view>
<view id="op_in_proj">
    RStream(
        select
            loc.customerId,
            shop.shopId,
            INSIDE@spatial(shop.geom, loc.curLoc, 5.0d)
        from
            LocGeomStream[NOW] as loc,
            ShopGeomRelation as shop
    )
</view>
```

A.21 INSIDE3D

The `INSIDE`3D Oracle Spatial geometric relation returns `true` if the 3D geometry, `geom1`, is inside the 3D space of `geom2`, and `false` otherwise.

Syntax

```
INSIDE3D@spatial(geom1, geom2)
INSIDE3D@spatial(geom1, geom2)
```

- `geom1`: The contained geometry, which can be any supported 3D geometry.
- `geom2`: The containing geometry, which can be any supported 3D geometry.

Example

The following Oracle CQL query tests whether a point is inside a 3D geometry.

```
<view id="op_in_where">
    RStream(
        select
            loc.customerId,
            shop.shopId
        from
            LocGeomStream[NOW] as loc,
            ShopGeomRelation as shop
        where
            INSIDE3D@spatial(shop.geom1, shop.geom2) = true
    )
</view>
<view id="op_in_proj">
    RStream(
        select
            loc.customerId,
            shop.shopId,
            INSIDE@spatial(shop.geom1, shop.geom2)
        from
            LocGeomStream[NOW] as loc,
            ShopGeomRelation as shop
    )
</view>
```

A.22 NN

The NN Oracle Spatial geometric filter operator returns the objects (nearest neighbors) from geom that are nearest to the key. To determine how near two geometry objects are to each other, Oracle Event Processing uses the shortest possible distance between any two points on the surface of each object used.

Syntax

```
NN@spatial(geom, key, tol)
```

- `geom`: any supported geometry type.
- `key`: A `GTYPE_POINT`, `GTYPE_CURVE`, `GTYPE_POLYGON`, `GTYPE_SURFACE`, `GTYPE_COLLECTION`, `GTYPE_MULTIPOINT`, `GTYPE_MULTICURVE`, `GTYPE_MULTIPOLYGON`, `GTYPE_SOLID`, or `GTYPE_MULTISOLID` geometry type.
- `tol`: the tolerance as a `double` value.

Examples

The following Oracle CQL query tests for nearest neighbors.

```
<view id="filter">
    RStream(
        select loc.customerId, shop.shopId
        from LocGeomStream[NOW] as loc, ShopGeomRelation as shop
        where NN@spatial(shop.geom, loc.curLoc, 5.0d) = true
```

```
    )
</view>
```

A.23 ordsgenerator

The `ordsgenerator` Oracle CQL function returns a `double` array of 2D coordinates from coordinate parameter values.

Syntax

```
ordsgenerator@spatial(x1, y1, ..., xN, yN)
```

The parameter values form a comma-separated list of coordinate values. This function returns a `double` array of 2D coordinates from the input.

Example

The following example creates an Oracle Spatial double array out of six `double` coordinate values.

```
view id="ShopGeom">
    select  com.oracle.cep.cartridge.spatial.Geometry.createGeometry@spatial(
        com.oracle.cep.cartridge.spatial.Geometry.GTYPE_POLYGON,
        com.oracle.cep.cartridge.spatial.Geometry.createElemInfo(1, 1003, 1),
        ordsgenerator@spatial(
            lng1, lat1, lng2, lat2, lng3, lat3,
            lng4, lat4, lng5, lat5, lng6, lat6
        )
    ) as geom
    from ShopDesc
</view>
```

A.24 to_Geometry

The `com.oracle.cep.cartridge.spatial.Geometry to_Geometry` method converts an `oracle.spatial.geometry.JGeometry` type to a 3D `com.oracle.cep.cartridge.spatial.Geometry` type.

Syntax

```
com.oracle.cep.cartridge.spatial.Geometry.to_Geometry(geom)
```

The `geom` parameter is the `oracle.spatial.geometry.JGeometry` object to convert. If the given geometry is already a `Geometry` type and a 3D geometry, then no conversion is done. If the given geometry is a 2D geometry, then the given geometry is converted to 3D by padding z coordinates.

Example

The following example converts the 2D geometry, `geo`, to a 3D geometry.

```
<view id="LocStream" schema="customerId loc">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.to_Geometry(geom)
    from
        CustomerLocStream
</view>
```

A.25 to_J3D_Geometry

The `com.oracle.cep.cartridge.spatial.Geometry` to_J3D_Geometry method converts a `com.oracle.cep.cartridge.spatial.Geometry` object to an `oracle.spatial.geometry.J3D_Geometry` object.

Syntax

```
com.oracle.cep.cartridge.spatial.Geometry.to_J3D_Geometry(g)
```

The `g` parameter is the com.oracle.cep.cartridge.spatial.Geometry object to convert.

Example

The following example shows how to use the `to_J3D_Geometry` method.

```
<view id="LocStream" schema="customerId loc">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.to_J3D_Geometry(geom)
    from
        CustomerLocStream
</view>
```

A.26 to_JGeometry

The `com.oracle.cep.cartridge.spatial.Geometry.to_JGeometry` method converts a `com.oracle.cep.cartridge.spatial.Geometry` object to an `oracle.spatial.geometry.JGeometry` 2D type.

Syntax

```
com.oracle.cep.cartridge.spatial.Geometry.to_JGeometry(g)
```

The `g` parameter is the com.oracle.cep.cartridge.spatial.Geometry object to convert.

Example

The following example converts the 2D geometry object, `geom`, to a 2D JGeometry object.

```
<view id="LocStream" schema="customerId loc">
    select
        customerId,
        com.oracle.cep.cartridge.spatial.Geometry.to_JGeometry(geom)
    from
        CustomerLocStream
</view>
```

A.27 WITHINDISTANCE

The `WITHINDISTANCE` Oracle CQL query returns `true` when the `GTYPE_POINT` is within the given distance of the geometry, and `false` otherwise.

Syntax

```
WITHINDISTANCE@spatial(geom, key, dist)
```

- `geom`: any supported geometry type.
- `key`: A `GTYPE_POINT`, `GTYPE_CURVE`, `GTYPE_POLYGON`, `GTYPE_SURFACE`, `GTYPE_COLLECTION`, `GTYPE_MULTIPOINT`, `GTYPE_MULTICURVE`, `GTYPE_MULTIPOLYGON`, `GTYPE_SOLID`, or `GTYPE_MULTISOLID` geometry type.
- `dist`: the distance as a `double` value.

Example

The following Oracle CQL query tests whether `loc.curLoc` is within the `5.0d` distance of `shop.geom`.

```
<view id="op_in_where">
    RStream(
        select
            loc.customerId,
            shop.shopId
        from
            LocGeomStream[NOW] as loc,
            ShopGeomRelation as shop
        where
            WITHINDISTANCE@spatial(shop.geom, loc.curLoc, 5.0d) = true
    )
</view>
<view id="op_in_proj">
    RStream(
        select
            loc.customerId,
            shop.shopId,
            WITHINDISTANCE@spatial(shop.geom, loc.curLoc, 5.0d)
        from
            LocGeomStream[NOW] as loc,
            ShopGeomRelation as shop
    )
</view>
```