Distribution Cache Integration for Esper

Version 5.4.0

by *EsperTech Inc.* [http://www.espertech.com]

Preface

This document describes the Distribution Cache Integration for Esper Complex Event Processing.

This documents targets software architects and developers. Some understanding and knowledge of Esper is assumed.

If you are new to Esper, the Esper reference manual as well as the tutorials, case studies and solution patterns available on the Esper public web site at *http://www.espertech.com/esper* provide the overall Esper documentation set

If you are new to the Distributed Cache Integration, please follow these steps:

1. Read *Chapter 1, Introduction* for an overview of concepts and capabilities.
2. Read *Chapter 2, Getting Started* for the quick start instructions.

Chapter 1. Introduction

1.1. Key Capabilities

The Distributed Cache Integration integrates distributed caching software with Esper Complex Event Processing in order to seamlessly query, insert, update, delete and listen to cache changes using EPL.

The Distributed Cache Integration is available for use with the following distributed caching products listed below in alphabetic order of product name:

- Oracle Coherence. Coherence is a regististered trademark of Oracle Corporation. Please visit *http://www.oracle.com*. We require version 12.1.2 or above.
- VMware vFabric GemFire. The names vFabric and GemFire are regististered trademarks of VMWare. Please visit *http://www.vmware.com*. We test against version 8.1.0 of GemFire.
- GigaSpaces XAP. GigaSpaces and eXtreme Application Platform (XAP) are registered trademarks of GigaSpaces Technologies. Please visit *http://www.gigaspaces.com*. The GigaSpaces version we test against is version 10.0.1 of GemFire.
- Hazelcast. Hazelcast is a trademark of Hazelcast Software. Please visit *http://www.hazelcast.com*. We require version 3.1.6 or above.
- Infinispan. Infinispan is a JBoss project. Please visit *http://www.jboss.org/infinispan*.
- Terracotta. Terracotta is a registered trademark of Software AG. Please visit *http://www.terracotta.org*. We require version 3.7.7 or above.
- IBM WebSphere eXtreme Scale. WebSphere and eXtreme scale are registered trademarks of IBM. Please visit *http://www.ibm.com*.

Please let us know if the distributed cache software you are using is not on this list.

The Distributed Cache Integration provides the capability to declare a named window backed by a specific cache instance that is managed by the distributed caching software. This document uses the term *cache instance* to mean the cache, map, grid, region or space instance, as the term is used by the specific distributed caching software.

EPL statements that join, subquery or perform fire-and-forget queries against named windows backed by a distributed cache execute the query against the cache instance. The Distributed Cache Integration analyses join, subquery or fire-and-forget where-clauses and performs key-based lookups, filter query lookups or full cache scan lookups to find matching rows. For the filter queries the Distributed Cache Integration uses the cache query language (if any) specific to the distributed caching software.

The Distributed Cache Integration can listen to changes in the cache instance. When unrelated applications add new entries to a cache instance, or update entries in a cache instance or delete entries from a cache instance then the Distributed Cache Integration can accordingly update EPL statements that refer to the named window. By refer we mean EPL statements that consume a named window stream for example to join, aggregate or detect patterns.

Any EPL statement that uses `insert into` to insert into a named window backed by a distributed cache inserts directly into the cache instance.

The EPL `on-merge`, `on-delete`, `on-update` statements when used against a named window backed by a distributed cache apply the respective change to the cache instance.

Statements that refer to the named window (consume its stream) and that use aggregations or detect patterns via match-recognize have the aggregation or pattern state initialized, at time of EPL statement creation, from the data available in a cache instance managed by the distributed cache software.

1.2. Known Limitations

This section lists known limitations.

1.2.1. GigaSpaces Support - Known Limitations

You may not insert Map-type or XML-type events into a GigaSpaces-managed cache instance (a space).

You may not use cache listeners with GigaSpaces.

You may not use the unique-flag to achieve unique semantics with Gigaspaces.

1.2.2. Websphere Support - Known Limitations

You may not use cache listeners with Websphere eXtreme Scale.

Chapter 2. Getting Started

This chapter is a step-by-step guide to get your application started using Distributed Cache Integration.

2.1. Quick Start

The quick start steps to Distributed Cache Integration are as follows:

1. Add the following files to your classpath:
   - The Distributed Cache Integration "common" jar file named `lib/esper-distcache-common-x.y.z.jar` (x.y.z being the release version number)
   - The Distributed Cache Integration jar file specific to the distributed caching software in use. The jar file name is `lib/esper-distcache-`*distcachename*`-x.y.z.jar`, with *distcachename* being the product name of the distributed caching software.
   - Any jar files required by the distributed caching software. The distribution provides a `readme` file for each distributed caching software that lists the jar files.
2. Add to the Esper engine configuration the plug-in for the distributed caching software (here shown for Coherence):

```
Configuration config = new Configuration();
config.addPluginLoader("coherence", CoherencePluginLoader.class.getName());
EPServiceProvider                        epService                        =
 EPServiceProviderManager.getDefaultProvider(config);
```

3. In EPL, declare a named window that is backed by the distributed cache as follows:

```
@ExternalDW(name='OrderCache') @ExternalDWKey(property='orderId')
create window OrderWindow.coherence:cache() as OrderEvent
```

After following the steps above, you are now ready to query the named window and insert, update or delete from the named window using EPL statements and fire-and-forget queries.

By default, the Distributed Cache Integration does not listen to cache changes. Please enable cache change listener as described below.

2.2. Required Esper Configurations

As briefly discussed before, your application must add the plug-in class for use with the distributed cache software to the Esper configuration.

The next programming example adds the Coherence plug-in to the Esper configuration via configuration API:

```
Configuration config = new Configuration();
config.addPluginLoader("coherence", CoherencePluginLoader.class.getName());
EPServiceProvider                          epService                          =
 EPServiceProviderManager.getDefaultProvider(config);
```

The example below is a snippet of Esper XML configuration that adds the Terracotta cache configuration to the Esper configuration XML.

```
<plugin-loader name="terracotta"
  class-name="com.espertech.esper.distcache.terracotta.TerracottaPluginLoader"/
>
```

The next table outlines, for each distributed caching software, the plug-in class and example code as well as example XML for use in the Esper configuration.

Table 2.1. Esper Configuration and Example

Cache Type	Class	Example

Coherence	`CoherencePluginLoad`	`config.addPluginLoader("coherence", CoherencePluginLoader.class.getName());`
		
		`<plugin-loader name="coherence" class-name="com.espertech.esper.distcache. coherence.CoherencePluginLoader"/>`

GemFire	`GemfirePluginLoader`	`config.addPluginLoader("gemfire", GemfirePluginLoader.class.getName());`
		
		`<plugin-loader name="gemfire" class-name="com.espertech.esper.distcache. gemfire.GemfirePluginLoader" />`

GigaSpaces	`GigaspacesPluginLoa`	`config.addPluginLoader("gigaspaces", GigaspacesPluginLoader.class.getName());`
		
		`<plugin-loader name="gigaspaces"`

Cache Type	Class	Example
		`class-name="com.espertech.esper.distcache.gigaspaces.GigaspacesPluginLoader" />`

Hazelcast	`HazelcastPluginLoad`	`config.addPluginLoader("hazelcast", HazelcastPluginLoader.class.getName());`
		
		`<plugin-loader name="hazelcast" class-name="com.espertech.esper.distcache.hazelcast.HazelcastPluginLoader" />`

Infinispan	`InfinispanPluginLoa`	`config.addPluginLoader("infinispan", InfinispanPluginLoader.class.getName());`
		
		`<plugin-loader name="Infinispan" class-name="com.espertech.esper.distcache.infinispan.InfinispanPluginLoader" />`

Terracotta	`TerracottaPluginLoa`	`config.addPluginLoader("terracotta", TerracottaPluginLoader.class.getName());`
		
		`<plugin-loader name="terracotta" class-name="com.espertech.esper.distcache.terracotta.TerracottaPluginLoader" />`

Websphere	`WebspherePluginLoad`	`config.addPluginLoader("websphere", WebspherePluginLoader.class.getName());`
		
		`<plugin-loader name="websphere" class-name="com.espertech.esper.distcache.websphere.WebspherePluginLoader" />`

2.3. EPL Create Window Syntax

Use the EPL `create window` syntax with a data window view name as *namespace*:`cache` to declare a named window backed by a distributed cache.

The following EPL statement declares a named window backed by a Coherence cache by name `OrderCache` wherein the event property `orderId` is the key and the value is `OrderEvent` events.

```
@ExternalDW(name='OrderCache') @ExternalDWKey(property='orderId')
create window OrderWindow.coherence:cache() as OrderEvent
```

The table below outlines, for each distributed caching software, the view namespace as well as example EPL to create a named window.

Table 2.2. EPL Create-Window Examples

Cache Type	Namespace	Example
Coherence	`coherence`	`create window OrderWindow.coherence:cache() as OrderEvent`
GemFire	`gemfire`	`create window OrderWindow.gemfire:cache() as OrderEvent`
GigaSpaces	`gigaspaces`	`create window OrderWindow.gigaspaces:cache() as OrderEvent`
Hazelcast	`hazelcast`	`create window OrderWindow.hazelcast:cache() as OrderEvent`
Infinispan	`infinispan`	`create window OrderWindow.infinispan:cache() as OrderEvent`
Terracotta	`terracotta`	`create window OrderWindow.terracotta:cache() as OrderEvent`
Websphere	`websphere`	`create window OrderWindow.websphere:cache() as OrderEvent`

Chapter 3. Guide

3.1. Associating a Cache Instance

You must provide the `@ExternalDW` annotation with `create window` to associate a named window to a cache instance that is managed by the distributed caching software.

The `@ExternalDW` annotation requires a value for the `name` attribute. The cache name is a required value: The Distributed Cache Integration uses the cache name for logging and can also use the cache name to obtain a cache instance from the distributed caching software.

There are two choices for associating a named window to a cache instance:

1. You provide only a cache name with the `@ExternalDW` annotation: The Distributed Cache Integration uses that cache name to obtain the cache instance.
2. You provide a cache name and a cache open function name with the `@ExternalDW` annotation: The Distributed Cache Integration uses the cache open function to obtain a cache instance.

You may specify the optional `functionOpen` and `functionClose` attributes of the `@ExternalDW` annotation to have your application provide the cache instance. If you provide a value for `functionOpen` the Distributed Cache Integration does not use the cache name to obtain a cache instance, and instead invokes the plug-in single-row function as provided to obtain the cache instance.

The instructions below refer to the following example `create window` EPL statement:

```
@ExternalDW(name='OrderCache') @ExternalDWKey(property='orderId')
create window OrderWindow.coherence:cache() as OrderEvent
```

Please replace the `coherence` namespace with the namespace of the distributed cache software that is in use by your application.

3.1.1. Cache Instance Association by Name

3.1.1.1. Coherence

For Coherence, a cache instance is a `com.tangosol.net.NamedCache` cache. The cache name provided as part of the `@ExternalDW` annotation is the Coherence cache name, or can be any value if the `functionOpen` attribute is provided instead.

If your `@ExternalDW` annotation only provides a cache name, the Distributed Cache Integration uses the `com.tangosol.net.CacheFactory.getCache(cacheName)` method to obtain a `NamedCache` instance.

If your `@ExternalDW` annotation provides the `functionOpen` attribute, the function your application provides must return a `NamedCache` instance.

3.1.1.2. GemFire

For GemFire, a cache instance is a `com.gemstone.gemfire.cache.Region` region. The cache name provided as part of the `@ExternalDW` annotation is the GemFire region name, or can be any value if the `functionOpen` attribute is provided instead.

If your `@ExternalDW` annotation only provides a cache name, the Distributed Cache Integration first instantiates a `com.gemstone.gemfire.cache.CacheFactory` by invoking `new CacheFactory()`. It then passes all properties that were provided by the plug-in loader configuration by calling `cacheFactory.set(key, value)`. It then invokes the `cacheFactory.create()` method to obtain an instance of `com.gemstone.gemfire.cache.Cache`. Finally, it obtains the `Region` instance by calling `cacheManager.getRegion(cacheName)`.

If your `@ExternalDW` annotation provides the `functionOpen` attribute, the function your application provides must return a `com.espertech.esper.distcache.gemfire.GemfireConnectionDescriptor` instance.

3.1.1.3. GigaSpaces

For GigaSpaces, a cache instance is a `org.openspaces.core.GigaSpace` space. The cache name provided as part of the `@ExternalDW` annotation is the GigaSpaces space URL, or can be any value if the `functionOpen` attribute is provided instead.

If your `@ExternalDW` annotation only provides a cache name, the Distributed Cache Integration first obtains a `com.j_spaces.core.IJSpace` by calling `IJSpace space = new UrlSpaceConfigurer(cacheName).space()`. It then calls `new GigaSpaceConfigurer(space).create()` to obtain a `GigaSpace` instance.

If your `@ExternalDW` annotation provides the `functionOpen` attribute, the function your application provides must return a `GigaSpace` instance.

3.1.1.4. Hazelcast

For Hazelcast, a cache instance is a `com.hazelcast.core.IMap` map. The cache name provided as part of the `@ExternalDW` annotation is the Hazelcast map name, or can be any value if the `functionOpen` attribute is provided instead.

If your `@ExternalDW` annotation only provides a cache name, the Distributed Cache Integration obtains a `IMap` instance by calling `Hazelcast.getMap(cacheName)`.

If your `@ExternalDW` annotation provides the `functionOpen` attribute, the function your application provides must return a `IMap` instance.

3.1.1.5. Infinispan

For Infinispan, a cache instance is a `org.infinispan.Cache` instance. The cache name provided as part of the `@ExternalDW` annotation is the DefaultCacheManager cache name, or can be any value if the `functionOpen` attribute is provided instead.

If your `@ExternalDW` annotation only provides a cache name, the Distributed Cache Integration obtains a `org.infinispan.manager.DefaultCacheManager` instance by calling `new DefaultCacheManager()` and obtains a cache instance by calling `defaultCacheManager.getCache(cacheName)`.

If your `@ExternalDW` annotation provides the `functionOpen` attribute, the function your application provides must return a `Cache` instance.

3.1.1.6. Terracotta

For Terracotta, a cache instance is a `net.sf.ehcache.Cache` cache. The cache name provided as part of the `@ExternalDW` annotation is the Terracotta cache name, or can be any value if the `functionOpen` attribute is provided instead.

If your `@ExternalDW` annotation only provides a cache name, the Distributed Cache Integration first obtains a `net.sf.ehcache.CacheManager` by calling `new CacheManager()`. It then invokes `cacheManager.getCache(cacheName)` on the Terracotta EHCache cache manager to obtain the `Cache`.

If your `@ExternalDW` annotation provides the `functionOpen` attribute, the function your application provides must return a `Cache` instance.

3.1.1.7. Websphere

For Websphere, a cache instance is a `com.ibm.websphere.objectgrid.ObjectMap` cache. The cache name provided as part of the `@ExternalDW` annotation is the Websphere map name, or can be any value if the `functionOpen` attribute is provided instead.

If your `@ExternalDW` annotation only provides a cache name, the Distributed Cache Integration first obtains an `com.ibm.websphere.objectgrid.ObjectGrid` by calling `ObjectGridManagerFactory.getObjectGridManager().createObjectGrid()`. The actual `ObjectMap` instance is obtained by the Distributed Cache Integration at runtime.

If your `@ExternalDW` annotation provides the `functionOpen` attribute, the function your application provides must return a `ObjectGrid` instance.

3.1.2. Cache Instance Association by Function

In this example we outline how to create a function that returns the cache instance. Use this mechanism when the default association using cache name is not sufficient for your needs or when your application requires complete control over cache instances. The example below assumes Coherence as the distributed cache for the purpose of illustration.

Provide a static method that takes a single `CacheOpenContext` as a parameter and that returns a cache instance. In this example the function returns `NamedCache` in the case of Coherence. We have instructions above that are specific to each distributed cache software.

A sample method providing a cache instance is shown below.

```
public class ExampleCoherenceCacheProvider {
  public static NamedCache getCacheInstance(CacheOpenContext cacheOpenContext) {
    return CacheFactory.getCache(cacheOpenContext.getCacheName());
  }
}
```

Next, register the function as a plug-in single-row function as part of the Esper configuration:

```
Configuration config = new Configuration();
config.addPlugInSingleRowFunction("getCacheInstance",
  ExampleCoherenceCacheProvider.class.getName(), "getCacheInstance");
```

Finally, provide the function name in the `functionOpen` attribute of the `@ExternalDW` annotation:

```
@ExternalDW(name='OrderCache', functionOpen='getCacheInstance') ...
```

When the named window gets destroyed, the Distributed Cache Integration invokes the function registered in the `functionClose` attribute of the `@ExternalDW` annotation, if a function name was provided.

A sample method for closing a cache instance is shown below.

```
public class ExampleCoherenceCacheProvider {
  public static void closeCacheInstance(CacheCloseContext cacheCloseContext) {
    // add logic to close the cache instance here
  }
}
```

Register the function as a plug-in single-row function as part of the Esper configuration:

```
Configuration config = new Configuration();
config.addPlugInSingleRowFunction("closeCacheInstance",
  ExampleCoherenceCacheProvider.class.getName(), "closeCacheInstance");
```

Provide the function name in the `functionClose` attribute of the `@ExternalDW` annotation:

```
@ExternalDW(name='OrderCache',
  functionOpen='getCacheInstance', functionClose='closeCacheInstance') ...
```

3.2. Determine Cache Key Objects

Use the `@ExternalDWKey` annotation with `create window` to identify key objects of entries managed by the cache instance. The `@ExternalDWKey` annotation is required in conjunction with the `@ExternalDW` annotation.

When an EPL statement inserts an event into the named window backed by a distributed cache it populates the cache with a key object and a value object by performing a `put(key, value)` operation.

When an EPL statement updates a named window backed by a distributed cache the statement identifies the rows to update by performing a query (see queries below) and by applying the `where`-clause. For each updated event the engine also performs a `put(key, value)` operation.

When an EPL statement deletes from a named window backed by a distributed cache the statement identifies the rows to delete by performing a query (see queries below) and by applying the `where`-clause. For each deleted event the engine also performs a `remove(key)` operation.

There are two choices for the engine to obtain the key object for each event that is inserted, updated or deleted:

1. You may provide an event property name in the `property` attribute of the `@ExternalDWKey` annotation: The Distributed Cache Integration uses the event property name to obtain the key object for the cache entry.
2. You may provide a function name in the `function` attribute of the `@ExternalDWKey` annotation: The Distributed Cache Integration invokes the function passing the event object and the function is expected to return the key object to use for the cache entry.

For GigaSpaces as the distributed cache software there is no explicit key associated to each GigaSpace space entry. It is therefore not necessary to use the `@ExternalDWKey` annotation.

3.2.1. Cache Key based on Event Property

Place the name of the event property that provides the value for the cache key into the `property` attribute of the `CacheKey` annotation.

For example, assume that you have an event that represents an order and is defined as a class as follows:

```
public class OrderEvent {
  private String orderId;
```

```
  private double price;

  public String getOrderId() {
    return orderId;
  }
  ...
}
```

In this example we assume that the `orderId` is a primary key for each order that uniquely identifies each order. You could thus organize the distributed cache such that the value of `orderId` is the key and the value is the `OrderEvent` itself. The cache `put` operation thus is `put(orderId, orderEvent)`. You only need to specify `@ExternalDWKey(property='orderId')` to instruct the Distributed Cache Integration to use the value of `orderId` as the cache key when putting entries into the cache.

The EPL example below specifies that the `orderId` property of the `OrderEvent` event type contains the value to use as the cache key:

```
@ExternalDW(name='OrderCache') @ExternalDWKey(property='orderId')
create window OrderWindow.coherence:cache() as OrderEvent
```

The type of object provided by the event property can be any application class or Java primitive or other type. If using an application class the class should implement the `hashCode` and `equals` methods.

3.2.2. Cache Key based on Function

If an event property cannot be used to provide the cache key object, you may provide an application function as described here. Place the function name that provides the value for the cache key into the `function` attribute of the `CacheKey` annotation.

Provide a static method that takes a single `Object` as a parameter and that returns the cache key object. In this example the function returns the `orderId` value.

A sample method providing a cache key object is shown below.

```
public class ExampleCacheKeyGetter {
  public static Object getCacheKey(Object event) {
    OrderEvent orderEvent = (OrderEvent) event;
    return orderEvent.getOrderId(); // simply return order id as cache key
  }
}
```

Next, register the function as a plug-in single-row function as part of the Esper configuration:

```
Configuration config = new Configuration();
config.addPlugInSingleRowFunction("getCacheKey",
  ExampleCacheKeyGetter.class.getName(), "getCacheKey");
```

Finally, provide the function name in the `function` attribute of the `@ExternalDWKey` annotation:

```
@ExternalDW(name='OrderCache') @ExternalDWKey(function='getCacheKey')
create window OrderWindow.coherence:cache() as OrderEvent
```

3.3. Determine Cache Value Object

Use the optional `@ExternalDWValue` annotation with `create window` to instruct the engine how events map to value objects of cache entries and how value objects of cache entries map to events.

The `@ExternalDWValue` annotation is not required in conjunction with the `@ExternalDW` annotation. If no `@ExternalDWValue` annotation is provided the value object is the underlying event object.

There are two choices for the engine to obtain the value object for each event that is inserted, updated or deleted, and for each value object that a cache listener may indicate:

1. If you provide no `@ExternalDWValue` annotation the event underlying object is the value object.
2. If you provide a `@ExternalDWValue` annotation you must specify both the `functionBeanToValue` and the `functionValueToBean`. The `functionBeanToValue` converts an event to a value object and `functionValueToBean` converts a value object obtained from a cache instance to an event.

3.3.1. Cache Value is Event Underlying Object

If you provide no `@ExternalDWValue` annotation (the default) the event underlying object is the value object.

When an EPL statement inserts or updates an event into the named window backed by a distributed cache it populates the cache with a key object and a value object by performing a `put(key, value)` operation. By default the value object is `EventBean.getUnderlying()`.

For example, lets assume the `OrderEvent` event type is the underlying class `com.mycompany.OrderEvent`. The following `create schema` declares the `OrderEvent` event type:

```
create schema OrderEvent as com.mycompany.OrderEvent
```

The next EPL statement creates a named window wherein the cache value object is the `OrderEvent`:

```
@ExternalDW(name='OrderCache') @ExternalDWKey(property='orderId')
create window OrderWindow.coherence:cache() as OrderEvent
```

The following EPL statement inserts all OrderEvent events into the named window:

```
insert into OrderWindow select * from OrderEvent
```

In the example above, as the cache key object is the value of order id, when a new `OrderEvent` event arrives the Distributed Cache Integration performs a `put(orderId, OrderEvent)` operation.

3.3.2. Cache Value Conversion Functions

Use the `@ExternalDWValue` annotation and provide the `functionBeanToValue` and `functionValueToBean` attributes to convert events to value objects and value objects to events. When the event underlying object is not the same class as the value object, or when different events map to different value objects or different value objects map to different events, the functions can be used to make a conversion as described here.

To convert an `EventBean` event instance to a value object for writing to the cache in a `put(key, value)` operation, provide a static method that takes a single `EventBean` parameter and returns the cache value object. This function is relevant for all writes to the cache instance.

To convert a cache value object to an `EventBean` event instance, provide a static method that takes a two parameters: An `Object` parameter that is the value object and an `EventBeanFactory` parameter that can be used to instantiate an event bean. This function is relevant for listening to new entries, updated entries and deleted entries by means of cache listener.

The sample methods below assumes that a class `OrderCacheValue` is defined that represents the value objects in the cache instance and that the event object is `OrderEvent`:

```
public class ExampleCacheValueConvertor {
  public static Object convertBeanToValue(EventBean event) {
    OrderEvent orderEvent = (OrderEvent) event.getUnderlying();
    return new OrderCacheValue(orderEvent);
  }

 public static EventBean convertValueToBean(Object valueObject, EventBeanFactory
 factory) {
    OrderCacheValue orderCacheValue = (OrderCacheValue) valueObject;
    return factory.wrap(new OrderEvent(orderCacheValue.getOrderId(), ...));
  }
}
```

Next, register the functions as plug-in single-row functions as part of the Esper configuration:

```
Configuration config = new Configuration();
config.addPlugInSingleRowFunction("convertBeanToValue",
  ExampleCacheValueConvertor.class.getName(), "convertBeanToValue");
config.addPlugInSingleRowFunction("convertValueToBean",
  ExampleCacheValueConvertor.class.getName(), "convertValueToBean");
```

Finally, provide the functions name as part of the `@ExternalDWValue` annotation:

```
@ExternalDW(name='OrderCache') @ExternalDWKey(property='orderId')
@ExternalDWValue(functionBeanToValue='convertBeanToValue',
 functionValueToBean='convertValueToBean')
create window OrderWindow.coherence:cache() as OrderEvent
```

3.4. Cache Query Considerations

When an EPL statement performs a join or subquery against the named window backed by a distributed cache, the Esper engine analyzes the join or subquery `where`-clause and the Distributed Cache Integration uses this information to determine a cache lookup strategy. The same is true for fire-and-forget queries that have a named window backed by a distributed cache in the `from`-clause.

The Distributed Cache Integration performs cache lookups according to the chosen strategy and applies the `where`-clause criteria to the resulting cache rows.

Enable debug-level logging to have the Distributed Cache Integration output the query strategy, query text (if any) and detailed statistics on execution times and number of rows returned by the cache lookup.

There are four strategies for the Distributed Cache Integration to perform value object lookups:

1. If the `where`-clause contains the key property declared in `@ExternalDWKey(property='`*property_name*`')` the Distributed Cache Integration executes a cache `get(key)` operation (implementation details are listed below).

2. If the `where`-clause contains properties of the named window and eligible operators, the Distributed Cache Integration executes a cache query using the distributed caching software's query language (see below for cache software-specific details and limitations).

3. If the `where`-clause does not contain any properties of the named window or all expressions are ineligible for query planning, the Distributed Cache Integration performs a full cache scan (see below for cache software-specific implementation detail).

4. In any case, if the querying statement specifies the `@ExternalDWQuery` annotation the Distributed Cache Integration overrides the cache lookup strategy and instead invokes the given application function passing a context object.

3.4.1. Cache Lookup per Distributed Cache Software

3.4.1.1. Coherence

The following applies when using the Coherence distributed cache software.

The lookup strategy that employs a cache query to find cache values follows these steps: It instantiates a set of `com.tangosol.util.Filter` filters and passe the filters to `namedCache.entrySet(filters)`.

The lookup strategy that performs a full cache scan is implemented as `namedCache.entrySet()`.

The lookup strategy that performs a key-based lookup is implemented as `namedCache.get(key)`.

3.4.1.2. GemFire

The following applies when using the GemFire distributed cache software.

The lookup strategy that employs a cache query to find cache values follows these steps: It obtains a `com.gemstone.gemfire.cache.query.QueryService` by invoking `Cache.getQueryService()`. It then composes a query in the Gemfire query language and execute the query by calling `queryService.newQuery(queryText)`.

The lookup strategy that performs a full cache scan is implemented as `region.entrySet()`.

The lookup strategy that performs a key-based lookup is implemented as `region.get(key)`.

3.4.1.3. GigaSpaces

The following applies when using the GigaSpaces distributed cache software.

The lookup strategy that employs a cache query to find cache values follows these steps: It composes a query in the GigaSpaces query language and execute the query by calling `gigaSpace.readMultiple(new SQLQuery(underlyingClass, queryText))`.

The lookup strategy that performs a full cache scan is implemented as `gigaSpace.readMultiple(new SQLQuery(underlyingClass, "")`.

The lookup strategy that performs a key-based lookup is impleted as `gigaSpace.readMultiple(template)` and only available for use with the `@ExternalDWQuery` annotation.

3.4.1.4. Hazelcast

The following applies when using the Hazelcast distributed cache software.

The lookup strategy that employs a cache query to find cache values follows these steps: It composes a query in the Hazelcast query language and execute the query by calling `map.values(new SqlPredicate(queryText))`.

The lookup strategy that performs a full cache scan is implemented as `map.entrySet()`.

The lookup strategy that performs a key-based lookup is implemented as `map.get(key)`.

3.4.1.5. Infinispan

The following applies when using the Infinispan distributed cache software.

As Infinispan does not provide a cache query language, there is no implementation of a lookup strategy that employs a cache query to find cache values. The query planner indicates an exception for queries that require a cache query.

The lookup strategy that performs a full cache scan is implemented as `cache.entrySet()`.

The lookup strategy that performs a key-based lookup is implemented as `cache.get(key)`.

3.4.1.6. Terracotta

The following applies when using the Terracotta distributed cache software.

The lookup strategy that employs a cache query to find cache values follows these steps: It obtains a `net.sf.ehcache.search.Query` by invoking `cache.createQuery()`. It then calls `query.includeValues()` to include only cache value objects in the result, adds attributes to the query and executes the query by calling `query.execute()`.

The lookup strategy that performs a full cache scan is implemented as `cache.getKeys()` and individual operations of `cache.get(key)` for each key.

The lookup strategy that performs a key-based lookup is implemented as `cache.get(key)`.

3.4.1.7. Websphere

The following applies when using the WebSphere distributed cache software.

The lookup strategy that employs a cache query to find cache values follows these steps: It composes a query in the WebSphere query language. For execution of the query, it first obtains a `Session` from the `ObjectGrid` using a thread-local session cache. It then begins the session via `session.begin()` and fires the query by calling `session.createObjectQuery(queryText)` and `query.getResultIterator()`. Finally it commits the session via `session.commit()`.

The lookup strategy that performs a full cache scan is implemented as `session.createObjectQuery("select m from cacheName m")`.

The lookup strategy that performs a key-based lookup is implemented as `objectMap.get(key)`.

3.4.2. Cache Lookup Strategy Override

If the default lookup strategy is not optimal or your application requires complete control over cache lookup, specify the `@ExternalDWQuery` annotation on the EPL statement that performs the query against the named window (not the create-window statement).

There are two ways to override the cache lookup strategy with `@ExternalDWQuery`:

1. The `functionKeys` attribute of the annotation can point to the name of the application-provided function that receives a lookup context as a parameter and that returns a collection of cache keys objects.

2. The `functionValues` attribute of the annotation can point to the name of the application-provided function that receives a lookup context as a parameter and that returns a collection of cache value objects.

Use the `functionKeys` attribute when you want to provide an application function that returns only key values and wherein the Distributed Cache Integration performs a cache lookup for each key object returned by the function.

Use the `functionValues` attribute when you want to provide an application function that returns cache values. The Distributed Cache Integration does not perform a cache lookup in this case.

3.4.2.1. Lookup Function Returning Keys

Specify the `functionKeys` annotation attribute for the `@ExternalDWQuery` annotation to provide a function that, given a lookup context, returns a set of cache keys objects.

A sample implementation is provided below:

```
public class ExampleCacheLookupKeyResolver {
  public static Collection getLookupKeys(CacheLookupContext context) {
    // the context contains lookup values and more
    // this sample returns a simple order id as a key
    return Collections.singleton("123");
  }
}
```

Next, register the function as a plug-in single-row function as part of the Esper configuration:

```
Configuration config = new Configuration();
config.addPlugInSingleRowFunction("getLookupKeys",
  ExampleCacheLookupKeyResolver.class.getName(), "getLookupKeys");
```

Finally, provide the function name as part of the `@ExternalDWQuery` annotation:

```
@ExternalDWQuery(functionKeys='getLookupKeys')
select * from OrderWindow where orderId = '123'
```

In above simple query against the named window backed by a distributed cache, the `where`-clause looks for a given order id value of `123`. The value `123` is passed to the function as part of the lookup context. The function may then return a collection of keys that are the cache key objects for order id `123`.

3.4.2.2. Lookup Function Returning Values

Specify the `functionValues` annotation attribute for the `@ExternalDWQuery` annotation to provide a function that, given a lookup context, returns a set of value objects.

A sample implementation is provided below:

```
public class ExampleCacheLookupValueResolver {
  public static Collection getLookupKeys(CacheLookupContext context) {
    // the context contains lookup values and more
    // this sample returns an OrderEvent instance as value object
    return Collections.singleton(new OrderEvent(...));
  }
}
```

Next, register the function as a plug-in single-row function as part of the Esper configuration:

```
Configuration config = new Configuration();
config.addPlugInSingleRowFunction("getLookupValues",
  ExampleCacheLookupValueResolver.class.getName(), "getLookupValues");
```

Finally, provide the function name as part of the `@ExternalDWQuery` annotation:

```
@ExternalDWQuery(functionValues='getLookupValues')
select * from OrderWindow where orderId = '123'
```

In above simple query against the named window backed by a distributed cache, the `where`-clause looks for a given order id value of `123`. The value `123` is passed to the function as part of the lookup context. The function may then return a collection of value objects, for example instances of `OrderEvent`.

3.5. Cache Listener

Most distributed cache software allows to register a listener that gets notified when cache operations take place. The cache listener gets invoked by the distributed cache software when a new cache entry is created, or an existing entry is updated or when an entry is deleted.

By default, the Distributed Cache Integration does not register a listener to cache operations. Use the `@ExternalDWListener` annotation and the `create window` syntax to have the Distributed Cache Integration listen to cache operations.

An example `create window` EPL statement that enables cache listening is shown next. Replace the `coherence` namespace with the namespace of the distributed cache software that is in use by your application:

```
@ExternalDW(name='OrderCache') @ExternalDWKey(property='orderId')
@ExternalDWListener
create window OrderWindow.coherence:cache() as OrderEvent
```

When your application code adds a new entry to the cache the cache listener receives the new value, converts the new value into an event and make the event available as an insert stream event for statements that consume the stream of the named window.

When your application code updates an entry in the cache the cache listener receives the new value and old value. It converts the new and old value into an event. It pushes the new-value event as an insert stream event and the old-value event as a remove stream event to statements that consume the stream of the named window.

When your application code deletes an entry in the cache the cache listener receives the old value. It converts the old value into an event and pushes the event as a remove stream event to statements that consume the stream of the named window.

For example, your EPL statement may select from the named window as follows. This statement employs the `irstream` keyword to select both the insert stream and remove stream:

```
select irstream * from OrderWindow
```

The statement above receives all new, updated and deleted entries, as each cache operation is performed by your unrelated application code, as events.

When specifying `@ExternalDWListener`, the Distributed Cache Integration allocates one thread to handle cache listener events. This design makes sure that the cache listener thread is free to continue its work.

You may disable the thread creation and use the thread provided by the distributed cache software instead by setting the `threaded` attribute to false, for example via `@ExternalDWListener(threaded=false)`.

You may also allocate more then 1 threads to handle cache listener events. Set the `numThreads` attribute to the desired number of threads, for example via `@ExternalDWListener(numThreads=2)`.

Please review the implementation notes regarding each of the distributed cache softwares below.

3.5.1. Cache Listener per Distributed Cache Software

3.5.1.1. Coherence

The following applies when using the Coherence distributed cache software.

The implementation adds a cache listener to the `NamedCache` instance by calling `namedCache.addMapListener(...)`. There are no known limitations to the map listener.

3.5.1.2. GemFire

The following applies when using the GemFire distributed cache software.

The implementation adds a cache listener to the `Region` instance by calling `region.getAttributesMutator().addCacheListener(...)`. There are no known limitations to the entry listener.

3.5.1.3. GigaSpaces

The following applies when using the GigaSpaces distributed cache software.

The `@ExternalDWListener` annotation is not supported with GigaSpaces.

3.5.1.4. Hazelcast

The following applies when using the Hazelcast distributed cache software.

The implementation adds an entry listener to the `IMap` instance by calling `imap.cache.addEntryListener(..., true)`. There are no known limitations to the region cache listener.

3.5.1.5. Infinispan

The following applies when using the Infinispan distributed cache software.

The implementation adds an cache listener to the `Cache` instance by calling `cache.addListener(...)`.

3.5.1.6. Terracotta

The following applies when using the Terracotta distributed cache software.

The implementation adds an cache event listener to the `Cache` instance by calling `cache.getCacheEventNotificationService().registerListener(this)`. Please note that the Terracotta cache event listener does make the old value available when cache entries are updated by application code.

3.5.1.7. Websphere

The following applies when using the WebSphere distributed cache software.

The `@ExternalDWListener` annotation is not supported with WebSphere.

3.6. Late Consumers

When EPL statements perform insert, update and delete operations against the named window and therefore the cache instance managed by the distributed cache software, then statements that consume from the named window receive such changes as insert and remove stream events.

In addition, if you specify the `@ExternalDWListener` annotation, the Distributed Cache Integration enables listening to cache operations. This means that statements that consume from the named window receive any changes that application code may perform to the cache in addition to any change EPL statements may make to the cache.

When the cache is already populated with data and not empty, any EPL statements that aggregate upon the named window get initialized by performing a full cache scan (the default behavior).

For example, consider the following EPL statement that returns the total quantity per product code:

```
select productCode, sum(qty) from OrderWindow
```

At time of statement creation, the Distributed Cache Integration performs a full cache read to build the initial state of the above statement, namely the product code and total quantity per product code. Please see the full scan lookup strategy description for information specific to the distributed caching software in use.

You may use the `@ExternalDWSetting` annotation to turn off full cache scans for statement initialization. Set the `iterable` attribute to false to instruct the Distributed Cache Integration to not perform a full scan.

For example:

```
@ExternalDW(name='OrderCache') @ExternalDWKey(property='orderId')
@ExternalDWSetting(iterable=false)
```

```
create window OrderWindow.coherence:cache() as OrderEvent
```

3.7. Achieving Unique Semantics Without Cache Listener

This section applies when not using a cache listener, since a distributed cache's cache listener receives old and new value as follows. When your application code updates an entry in the cache the cache listener receives the new value and old value. It converts the new and old value into an event. It pushes the new-value event as an insert stream event and the old-value event as a remove stream event to statements that consume the stream of the named window.

By default and without cache listener the named window backed by a distributed cache does not exhibit the same behavior as a `std:unique` data window. By default, when events are inserted, the engine does not check whether any existing event(s) exist in the cache for the same key. The default behavior therefore does not read cache values that are being overwritten and posts the old cache values as remove stream events to consumers.

The Distributed Cache Integration provides a `unique` flag as part of the `ExternalDW` annotation to instruct the engine to read the old value for a given key from the cache before inserting data into the cache, and post the previous value as an event into the remove stream.

For example, consider the following two statements:

```
@ExternalDW(name='OrderCache', unique=true) @ExternalDWKey(property='orderId')
create window OrderWindow.coherence:cache() as OrderEvent
```

```
insert into OrderWindow select * from OrderEvent
```

In the above example, assume an OrderEvent arrives for order id "00012". The engine reads the cache to find an existing OrderEvent for order id "00012". The engine then writes the new OrderEvent into cache. If a previous OrderEvent is found, the engine posts the previous OrderEvent as a remove stream event and the new OrderEvent as insert stream event. This allows consumers of the named window to receive insert and remove stream events.

Chapter 4. Performance

4.1. Performance Tips

4.1.1. Locking

By default the named window backed by a distributed cache is protected by a read-write lock, allowing any number of threads for reading and only one thread for writing to the named window.

You may specify the `@NoLock` annotation as part of the `create window` statement to remove the read-write lock and operate without locks.