Red Hat JBoss Data Virtualization 6.4

Development Guide Volume 5: Caching Guide

This guide is intended for developers

Last Updated: 2018-02-13

Red Hat JBoss Data Virtualization 6.4 Development Guide Volume 5: Caching Guide

This guide is intended for developers

Red Hat Customer Content Services

Legal Notice

Abstract

This document teaches you how to use Red Hat JBoss Data Virtualization's Caching Functionality

Table of Contents

CHAPTER 1. SOME KEY DEFINITIONS

1.1. RESULT SET CACHING

Red Hat JBoss Data Virtualization allows you to cache the results of your queries and virtual procedure calls. This caching technique can yield significant performance gains if you find you are frequently running the same queries or procedures.

1.2. INTERNAL MATERIALIZATION

When results are served from memory it is called "Internal Materialization".

1.3. EXTERNAL MATERIALIZATION

When an external database is used to store the contents of a view, it is referred to as "External Materialization".

1.4. MATERIALIZED VIEWS

Materialized views are just like other views, but their transformations are pre-computed and stored like a regular table. When queries are issued against the views through the Red Hat JBoss Data Virtualization Server, the cached results are used. This saves the cost of accessing all the underlying data sources and re-computing the view transformations each time a query is executed.

Materialized views are appropriate when the underlying data does not change rapidly, or when it is acceptable to retrieve older data within a specified period of time, or when it is preferred for end-user queries to access staged data rather than placing additional query load on operational sources.

1.5. MATERIALIZATION TABLE

A Materialization Table is a cached view of a virtual table. When a query is issued against this virtual table, the contents can be served from memory space or redirected to a another table so results can be fetched from the original sources.

Internal materialization creates temporary tables to hold the materialized table. While these tables are not fully durable, they perform well in most circumstances and the data is present in each Red Hat JBoss Data Virtualiation instance, removing the risk of a single point of failure and the network overhead of an external database.

External materialized views cache their data in a matgerialization table in an external database system.

1.6. NOCACHE OPTION

When the NOCACHE option is used, the data is retrieved by the original source data query, not from the materialized cache.

CHAPTER 2. USING CACHING

2.1. CACHING

Red Hat JBoss Data Virtualization supports two kinds of caching, Result Set Caching and Materialized Views.

NOTE

To learn how to define external materializations using the Teiid DDL in a dynamic VDB, please look at the dynamicvdb-materialization quick start.

2.2. CREATE MATERIALIZED VIEWS

You can create materialized views and their corresponding physical materialized target tables in the Teiid Designer JBDS plug-in. This can be done through setting the materialized and target table manually, or by selecting the desired views, right-clicking, then selecting **`Modeling->Create Materialized Views`**.

Next, generate the DDL for your physical model materialization target tables. This can be done by selecting the model, right clicking, then choosing **`Export->Metadata Modeling->Data Definition Language (DDL) File`**. Use this script to create the chema for your materialization target on your source.

Determine a load and refresh strategy. With the schema created the simplest approach is to load the data. You can even load it through Red Hat JBoss Data Virtualization: **`insert into target_table select * from matview option nocache matview`**

That however may be too simplistic because your index creation may perform better if deferred until after the table has been created. Also full snapshot refreshes are best done to a staging table then swapping it for the existing physical table to ensure that the refresh does not impact user queries and to ensure that the table is valid prior to use.

Metadata-Based Materialization Management

Users when they are designing their views, they can define additional metadata on their views to control the loading and refreshing of external materialization cache. This option provides a limited but a powerful way to manage the materialization views. For this purpose, SYSADMIN Schema model in your VDB defines three stored procedures (loadMatView, updateMatView, matViewStatus) in its schema. Based on the defined metadata on the view, and these SYSADMIN procedures a simple scheduler automatically starts during the VDB deployment and loads and keeps the cache fresh.

To manage and report the loading and refreshing activity of materialization view, Red Hat JBoss Data Virtualization expects the user to define "Status" table with following schema in any one of the source models. Create this table on the physical database, before you do the import of this physical source.

```
CREATE TABLE status
(
  VDBName varchar(50) not null,
  VDBVersion integer not null,
  SchemaName varchar(50) not null,
  Name varchar(256) not null,
  TargetSchemaName varchar(50),
  TargetName varchar(256) not null,
```

```
  Valid boolean not null,
  LoadState varchar(25) not null,
  Cardinality long,
  Updated timestamp not null,
  LoadNumber long not null,
  PRIMARY KEY (VDBName, VDBVersion, SchemaName, Name)
);
```

NOTE

MariaDB have Silent Column Changes function, according to MariaDB document, 'long' type will silently change to 'MEDIUMTEXT' , so If execute above schema in MariaDB, 'Cardinality' and 'LoadNumber' column should change to 'bigint' type.

Create Views and corresponding physical materialized target tables in Designer or using DDL in Dynamic VDB. This can be done through setting the materialized and target table manually, or by selecting the desired views, right clicking, then selecting Modeling->"Create Materialized Views" in the Designer.

Define the following extension properties on the view.

Table 2.1. Extension Properties

Property	Description	Optional	Default
teiid_rel:ALLOW_MATVIEW_MANAGEMENT	Allow Red Hat JBoss Data Virtualization-based management	False	False
teiid_rel:MATVIEW_STATUS_TABLE	fully qualified Status Table Name defined above	False	NA
teiid_rel:MATVIEW_BEFORE_LOAD_SCRIPT	semi-colon(;) separated DDL/DML commands to run before the actual load of the cache, typically used to truncate staging table	True	When not defined, no script will be run
teiid_rel:MATVIEW_LOAD_SCRIPT	semi-colon(;) separated DDL/DML commands to run for loading of the cache	True	will be determined based on view transformation

Property	Description	Optional	Default
teiid_rel:MATVIEW_AFTER_LOAD_SCRIPT	semi-colon(;) separated DDL/DML commands to run after the actual load of the cache. Typically used to rename staging table to actual cache table. Required when MATVIEW_LOAD_SCRIPT is not defined in order to copy data from the teiid_rel:MATVIEW_STAGE_TABLE the MATVIEW table.	True	When not defined, no script will be run
teiid_rel:MATVIEW_SHARE_SCOPE	Allowed values are {NONE, VDB, SCHEMA}, which define if the cached contents are shared among different VDB versions and different VDBs as long as schema names match	True	None
teiid_rel:MATERIALIZED_STAGE_TABLE	When MATVIEW_LOAD_SCRIPT property not defined, Red Hat JBoss Data Virtualization loads the cache contents into this table. Required when MATVIEW_LOAD_SCRIPT not defined	False	NA
teiid_rel:ON_VDB_START_SCRIPT	DML commands to run start of vdb	True	NA
teiid_rel:ON_VDB_DROP_SCRIPT	DML commands to run at VDB un-deploy; typically used for cleaning the cache/status tables	True	NA
teiid_rel:MATVIEW_ONERROR_ACTION	Action to be taken when mat view contents are requested but cache is invalid. Allowed values are (THROW_EXCEPTION = throws an exception, IGNORE = ignores the warning and supplied invalidated data, WAIT = waits until the data is refreshed and valid then provides the updated data)	True	WAIT

Property	Description	Optional	Default
teiid_rel:MATVIEW_TTL	time to live in milliseconds. Provide property or cache hint on view transformation - property takes precedence.	True	2^63 milliseconds - effectively the table will not refresh, but will be loaded a single time initially

Once the VDB (with a model with the properties specified above) has been defined and deployed, the following sequence of events will take place.

Upon the VDB deployment, teiid_rel:ON_VDB_START_SCRIPT will be run on completion of the deployment.

Based on the teiid_rel:MATVIEW_TTL defined a scheduler entry will be created to run SYSADMIN.loadMatView procedure, which loads the cache contents.

This procedure first inserts/updates an entry in teiid_rel:MATVIEW_STATUS_TABLE, which indicates that the cache is being loaded, then teiid_rel:MATVIEW_BEFORE_LOAD_SCRIPT will be run if defined. Next, teiid_rel:MATVIEW_LOAD_SCRIPT will be run if defined, otherwise one will be automatically created based on the view's transformation logic. Then, teiid_rel:MATVIEW_AFTER_LOAD_SCRIPT will be run, to close out and create any indexes on the mat view table.

The procedure then sets the teiid_rel:MATVIEW_STATUS_TABLE entry to "LOADED" and valid.

Based on the teiid_rel:MATVIEW_TTL, the SYSADMIN.matViewStatus is run by the Scheduler, to queue further cache re-loads.

When VDB is un-deployed (not when server is restarted) the teiid_rel:ON_VDB_DROP_SCRIPT script will be run.

Run the SYSADMIN.updateMatView procedure at any time to partially update the cache contents rather than complete refresh of contents with SYSADMIN.loadMatview procedure. When partial update is run the cache expiration time is renewed for new term based on Cache Hint again.

Here is a sample dynamic VDB:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="sakila" version="1">
    <description>Shows how to call JPA entities</description>

    <model name="pg">
        <source name="pg" translator-name="postgresql-override"
connection-jndi-name="java:/sakila-ds"/>
    </model>

    <model name="sakila" type="VIRTUAL">
    <metadata type="DDL"><![CDATA[
        CREATE VIEW actor (
           actor_id integer,
           first_name varchar(45) NOT NULL,
           last_name varchar(45) NOT NULL,
           last_update timestamp NOT NULL
        ) OPTIONS (MATERIALIZED 'TRUE', UPDATABLE 'TRUE',
```

```
                MATERIALIZED_TABLE 'pg.public.mat_actor',
                "teiid_rel:MATERIALIZED_STAGE_TABLE"
'pg.public.mat_actor_staging',
                "teiid_rel:ALLOW_MATVIEW_MANAGEMENT" 'true',
                "teiid_rel:MATVIEW_STATUS_TABLE" 'pg.public.status',
                "teiid_rel:MATVIEW_BEFORE_LOAD_SCRIPT" 'execute
pg.native(''truncate table mat_actor_staging'');',
                "teiid_rel:MATVIEW_AFTER_LOAD_SCRIPT" 'execute
pg.native(''ALTER TABLE mat_actor RENAME TO mat_actor_temp'');execute
pg.native(''ALTER TABLE mat_actor_staging RENAME TO mat_actor'');execute
pg.native(''ALTER TABLE mat_actor_temp RENAME TO mat_actor_staging;'')',
                "teiid_rel:MATVIEW_SHARE_SCOPE" 'NONE',
                "teiid_rel:MATVIEW_ONERROR_ACTION" 'THROW_EXCEPTION',
                "teiid_rel:MATVIEW_TTL" 300000,
                "teiid_rel:ON_VDB_DROP_SCRIPT" 'DELETE FROM
pg.public.status WHERE Name=''actor'' AND schemaname = ''sakila''')
            AS SELECT actor_id, first_name, last_name, last_update from
pg."public".actor;

    </metadata>
    </model>
    <translator name="postgresql-override" type="postgresql">
        <property name="SupportsNativeQueries" value="true"/>
    </translator>
</vdb>
```

2.2.1. Configure External Materialization In Teiid Designer

1. Build a VDB using the Teiid Designer for your use case.
2. Identify all the "Virtual Tables", that you think can use caching,
3. Click on the table, then in the **`Properties`** panel, switch the **`Materialized`** property to "true".
4. Right click on each materialized table, then choose **Modeling** - **Create Materialized Views**.
5. Click on **`...`** button on the **`Materialization Model`** input box.
6. Select a "physical model" that already exists or create a new name for "physical model".
7. Click **`Finish`**.

   This will create the new model (if applicable) and a table with exact schema as your selected virtual table.
8. Verify that the "Materialization Table" property is now updated with name of table that has just been created.
9. Navigate to the new materialized table that has been created, and click on "Name In Source" property and change it from "MV1000001" to "mv_{your_table_name}". Typically this could be same name as your virtual table name, (for example, you might name it "mv_UpdateProduct".)
10. Save your model.

NOTE

The data source this materialized view physical model represents will be the data source for storing the materialized tables. You can select different "physical models" for different materialized tables, creating multiple places to store your materialized tables.

11. Once you are have finished creating all materialized tables, right click on each model (in most cases this will be a single physical model used for all the materialized views), then use **Export** - **Teiid Designer** - **Data Definition Language (DDL) File** to generate the DDL for the physical model.

12. Select the type of the database and DDL file name and click **`Finish`**.

    A DDL file that contains all of the "create table" commands is generated..

13. Use your favorite "client" tool for your database and create the database using the DDL file created.

14. Go back to your VDB and configure the data source and translator for the "materialized" physical model to the database you just created.

15. Once finished, deploy the VDB to the Red Hat JBoss Data Virtualization Server and make sure that it is correctly configured and active.

IMPORTANT

It is important to ensure that the key/index information is defined as this will be used by the materialization process to enhance the performance of the materialized table.

2.3. EXTERNAL MATERIALIZATION OPTIONS

If you are trying to load the materialized table "Portfolio.UpdateProduct", for which the materialization table is defined as "mv_view.UpdateProduct", use any JDBC Query tool like SquirreL and make a JDBC connection to the VDB you created and issue following SQL command: **`INSERT INTO mv_view.mv_UpdateProduct SELECT * FROM Portfolio.UpdateProduct OPTION NOCACHE`**

Here is how you would create an AdminShell script to automatically load the materialized table:

```
sql=connect(${url}, ${user}, ${password});
sql.execute("DELETE FROM mv_view.mv_UpdateProduct");
sql.execute("INSERT INTO mv_view.mv_UpdateProduct SELECT * FROM
Portfolio.UpdateProduct OPTION NOCACHE");
sql.close();
```

Use this command to execute the script: **`adminshell.sh . load.groovy`**

NOTE

- If you want to set up a job to run this script frequently at regular intervals, then on Red Hat Enterprise Linux use "cron tab" or on Microsoft Windows use "Windows Scheduler" to refresh the rows in the materialized table. Every time the script runs it will refresh the contents.
- This job needs to be run only when user access is restricted.

IMPORTANT

There are some situation in which this process of loading the cache will not work. Here are some situations in which it will not work:

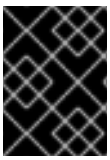
- If it is updating all the rows in the materialized table, and you only need to update only few rows to avoid long refresh time.
- If it takes an hour for your reload your materialized table, queries executed during that time will fail to povide correct results.
- Also ensure that you create indexes on your materialization table after the data is loaded, as having indexes during the load process slows down the loading of data, especially when you are dealing with a large number of rows.

2.4. EXTERNAL MATERIALIZATION AND RED HAT JBOSS DATA GRID

Red Hat JBoss Data Grid can be used as an external materialized data source system in order to improve query result performance. You can use a JDBC data source for the same reason.

NOTE

You can also define the materialization through a dynamic VDB. To learn how, look at the `[EAP_HOME]/quickstarts/jdg7.1-remote-cache-materialization` quick start.



IMPORTANT

When DDL metadata changes, the only way to promote the change is restart the Red Hat JBoss Data Grid instance. This also impacts VDB versioning.

IMPORTANT

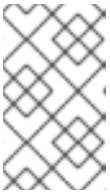
You cannot query a cache if it contains multiple pojos.

2.4.1. Materializing a View

Prerequisites

You must have two caches and the **teiid-alias-naming-cache** in Red Hat JBoss Data Grid. (The **teiid-alias-naming-cache** only needs to be created once because it is shared across all the materializations stored in the Red Hat JBoss Data Grid instance.)

1. Using Teiid Designer, click on a view that is to be materialized.

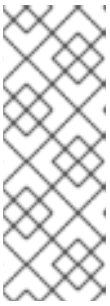


NOTE

Make sure the view has a primary key defined as the Red Hat JBoss Data Grid source table needs one for updates. Otherwise, you will need to manually create a primary key on each of the new JDG source tables.

2. Right-click on the view and click **Modeling->Materialize**.
3. Enter the **primary** and **staging** cache names.

   Optionally, you can change the **JDGSource** model name and the directory in which the model is saved.



NOTE

Red Hat JBoss Data Grid restricts the name of the source model because the protobuf code is based on the Java package naming constraints. The model name becomes the package name in the `.proto` file. This is due to a limitation in the way that the protobuf is defined. Because Red Hat JBoss Data Grid uses Java, the package name must follow the Java package naming standards. Dashes, for instance, are not allowed.

4. Click **`Finish`**.
5. To control the materialization process, update the materialized view extension properties on the above selected view:
   - MATVIEW_TTL - to set the refresh rate, in milliseconds

   If the materialization management status table is used, then set the following extension properties:
   - ALLOW_MATVIEW_MANAGEMENT = true
   - MATVIEW_STATUS_TABLE = {status table name}
6. Create the VDB, using the models needed for materialization.

   For the JDGSource model, be sure the JNDI is mapped to the JDG data source. Also, enable native queries. To do this, create a translator override for the infinispan-hotrod translator. Click the **supportsDirectQueryProcedure** property and set the value to **true**.
7. Deploy the VDB.

2.5. INTERNAL MATERIALIZATION

Internal materialization creates Data Virtualization temporary tables to hold the materialized table. While these tables are not fully durable, they perform well in most circumstances and the data is present in each Red Hat JBoss Data Virtualization instance which removes the single point of failure and network overhead of an external database. Internal materialization also provides more built-in facilities for refreshing and monitoring.

The materialized option must be set for the view to be materialized. The Cache Hint, when used in the context of an internal materialized view transformation query, provides the ability to fine tune the materialized table. The caching options are also settable via extension metadata:

Table 2.2. Mapping

Property Name	Description
teiid_rel:ALLOW_MATVIEW_MANAGEMENT	Allow Teiid based management of the ttl and initial load rather than the implicit behavior
teiid_rel:MATVIEW_PREFER_MEMORY	Same as the pref_mem cache hint option
teiid_rel:MATVIEW_TTL	Same as the ttl cache hint option
teiid_rel:MATVIEW_UPDATABLE	Same as the updatable cache hint option
teiid_rel:MATVIEW_SCOPE	Same as the scope cache hint option

The pref_mem option also applies to internal materialized views. Internal table index pages already have a memory preference, so the perf_mem option indicates that the data pages should prefer memory as well.

All internal materialized view refresh and updates happen atomically. Internal materialized views support READ_COMMITTED (used also for READ_UNCOMMITED) and SERIALIZABLE (used also for REPEATABLE_READ) transaction isolation levels.

Here is a sample Dynamic VDB defining an internal materialization:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<vdb name="sakila" version="1">

    <model name="pg">
        <source name="pg" translator-name="postgresql" connection-jndi-
name="java:/sakila-ds"/>
    </model>

    <model name="sakila" type="VIRTUAL">
    <metadata type="DDL"><![CDATA[
        CREATE VIEW actor (
           actor_id integer,
           first_name varchar(45) NOT NULL,
           last_name varchar(45) NOT NULL,
           last_update timestamp NOT NULL
        ) OPTIONS (materialized true, "teiid_rel:MATVIEW_TTL" 120000,
              "teiid_rel:MATVIEW_PREFER_MEMORY" 'true',
              "teiid_rel:MATVIEW_UPDATABLE" 'true',
              "teiid_rel:MATVIEW_SCOPE" 'vdb')
            AS SELECT actor_id, first_name, last_name, last_update from
pg."public".actor;
     ]>
    </metadata>
    </model>
</vdb>
```

An internal materialized view table is initially in an invalid state (there is no data). If teiid_rel:ALLOW_MATVIEW_MANAGEMENT is not specified, the first user query will trigger an implicit

loading of the data. All other queries against the materialized view will block until the load completes. In some situations administrators may wish to better control when the cache is loaded with a call to SYSADMIN.refreshMatView. The initial load may itself trigger the initial load of dependent materialized views. After the initial load user queries against the materialized view table will only block if it is in an invalid state. The valid state may also be controlled through the SYSADMIN.refreshMatView procedure. This is how you invalidate a refresh:

```
CALL SYSADMIN.refreshMatView(viewname=>'schema.matview', invalidate=>true)
```

Through this, the matview will be refreshed and user queries will block until the refresh is complete (or fails).

If you set the teiid_rel:ALLOW_MATVIEW_MANAGEMENT property to "true", this will trigger the loading when the Virtual Database is deployed.

While the initial load may trigger a transitive loading of dependent materialized views, subsequent refreshes performed with refreshMatView will use dependent materialized view tables if they exist. Only one load may occur at a time. If a load is already in progress when the SYSADMIN.refreshMatView procedure is called, it will return -1 immediately rather than preempting the current load.

The Cache Hint may be used to automatically trigger a full snapshot refresh after a specified time to live (ttl). The ttl starts from the time the table is finished loading. The refresh is equivalent to CALL SYSADMIN.refreshMatView('view name', *), where the invalidation behavior is determined by the vdb property lazy-invalidate.

By default ttl refreshes are invalidating, which will cause other user queries to block while loading. That is once the ttl has expired, the next access will be required to refresh the materialized table in a blocking manner. If you would rather that the ttl is enforced lazily, such that the refresh task is performed asynchronously with the current contents not replaced until the refresh completes, set the vdb property lazy-invalidate=true.

```
/*+ cache(ttl:3600000) */ select t.col, t1.col from t, t1 where t.id =
t1.id
```

The resulting materialized view will be reloaded every hour (3600000 milliseconds). It has these limitations:

- The automatic ttl refresh may not be suitable for complex loading scenarios as nested materialized views will be used by the refresh query.
- The non-managed ttl refresh is performed lazily, that is it is only trigger by using the table after the ttl has expired. For infrequently used tables with long load times, this means that data may be used well past the intended ttl.

In advanced use-cases the cache hint may also be used to mark an internal materialized view as updatable. An updatable internal materialized view may use the SYSADMIN.refreshMatViewRow procedure to update a single row in the materialized table. If the source row exists, the materialized view table row will be updated. If the source row does not exist, the correpsonding materialized row will be deleted. To be updatable the materialized view must have a single column primary key. Composite keys are not yet supported by SYSADMIN.refreshMatViewRow. Here is a sample transformation query:

```
/*+ cache(updatable) */ select t.col, t1.col from t, t1 where t.id = t1.id
```

Here is the update SQL:

```
CALL SYSADMIN.refreshMatViewRow(viewname=>'schema.matview', key=>5)
```

Given that the schema.matview defines an integer column col as its primary key, the update will check the live source(s) for the row values.

The update query will not use dependent materialized view tables, so care should be taken to ensure that getting a single row from this transformation query performs well. See the Reference Guide for information on controlling dependent joins, which may be applicable to increasing the performance of retrieving a single row. The refresh query does use nested caches, so this refresh method should be used with caution.

When the updatable option is not specified, accessing the materialized view table is more efficient because modifications do not need to be considered. Therefore, only specify the updatable option if row based incremental updates are needed. Even when performing row updates, full snapshot refreshes may be needed to ensure consistency.

The EventDistributor also exposes the updateMatViewRow as a lower level API for Programmatic Control - care should be taken when using this update method.

Internal materialized view tables will automatically create non-unique indexes for each unique constraint and index defined on the materialized view. These indexes are created as non-unique even for unique constraints since the materialized table is not intended as an enforcement point for data integrity and when updatable the table may not be consistent with underlying values and thus unable to satisfy constraints. The primary key (if it exists) of the view will automatically be part of the covered columns for the index.

The secondary indexes are always created as trees - bitmap or hash indexes are not supported. Teiid's metadata for indexes is currently limited. We are not currently able to capture additional information, sort direction, additional columns to cover, etc. You may workaround some of these limitations though.

- Function based index are supported, but can only be specified through DDL metadata. If you are not using DDL metadata, consider adding another column to the view that projects the function expression, then place an index on that new column. Queries to the view will need to be modified as appropriate though to make use of the new column/index.

- If additional covered columns are needed, they may simply be added to the index columns. This however is only applicable to comparable types. Adding additional columns will increase the amount of space used by the index, but may allow its usage to result in higher performance when only the covered columns are used and the main table is not consulted.

Each member in a cluster maintains its own copy of each materialized table and associated indexes. An attempt is made to ensure each member receives the same full refresh events as the others. Full consistency for updatable materialized views however is not guaranteed. Periodic full refreshes of updatable materialized view tables helps ensure consistency among members.

2.6. CODE TABLE CACHING

Red Hat JBoss Data Virtualization provides a short cut to creating an internal materialized view table via the lookup function.

The lookup function provides a way to accelerate getting a value out of a table when a key value is provided. The function automatically caches all of the key/return pairs for the referenced table. This caching is performed on demand, but will proactively load the results to other members in a cluster. Subsequent lookups against the same table using the same key and return columns will use the cached information.

This caching solution is appropriate for integration of "reference data" with transactional or operational data. Reference data is usually static and small data sets that are used frequently. Examples are ISO country codes, state codes, and different types of financial instrument identifiers.

This caching mechanism is automatically invoked when the lookup scalar function is used. The lookup function returns a scalar value, so it may be used anywhere an expression is expected. Each time this function is called with a unique combination of referenced table, return column, and key column (the first three arguments to the function). Here is a lookup for country codes:

```
   lookup('ISOCountryCodes', 'CountryCode', 'CountryName', 'United
States')
```

Code table caching does have some limitations:

- The use of the lookup function automatically performs caching; there is no option to use the lookup function and not perform caching.
- No mechanism is provided to refresh code tables
- Only a single key/return column is cached - values will not be session/user specific.

The lookup function is a shortcut to create an internal materialized view with an appropriate primary key. In many situations, it may be better to directly create the analogous materialized view rather than to use a code table.

```
  SELECT (SELECT CountryCode From MatISOCountryCodes WHERE CountryName =
tbl.CountryName) as cc FROM tbl
```

Here MatISOCountryCodes is a view selecting from ISOCountryCodes that has been marked as materialized and has a primary key and index on CountryName. The scalar subquery will use the index to lookup the country code for each country name in tbl.

Here are some reasons why you should use a materialized view:

- More control of the possible return columns. Code tables will create a materialized view for each key/value pair. If there are multiple return columns it would be better to have a single materialized view.
- Proper materialized views have built-in system procedure/table support.
- More control via the cache hint.
- The ability to use OPTION NOCACHE.
- There is almost no performance difference.

2.7. CREATE A MATERIALIZED VIEW FOR CODE TABLE CACHING

Procedure 2.1. Create a Materialized View for Code Table Caching

1. Create a view selecting the appropriate columns from the desired table. In general, this view may have an arbitrarily complicated transformation query.
2. Designate the appropriate column(s) as the primary key. Additional indexes can be added if needed.
3. Set the materialized property to true.
4. Add a cache hint to the transformation query. To mimic the behavior of the implicit internal materialized view created by the lookup function, use the Hints and Options /*+ cache(pref_mem) */ to indicate that the table data pages should prefer to remain in memory.

Result

Just as with the lookup function, the materialized view table will be created on first use and reused subsequently.

2.8. PROGRAMMATIC CONTROL

Red Hat JBoss Data Virtualization exposes a bean that implements the org.teiid.events.EventDistributor interface. You can find it in JNDI under the name teiid/event-distributor-factory. The EventDistributor exposes methods like **`dataModification`** (which affects result set caching) or **`updateMatViewRow`** (which affects internal materialization) to alert the Teiid engine that the underlying source data has been modified. These operations, which work cluster wide will invalidate the cache entries appropriately and reload the new cache contents.

NOTE

If your source system has any built-in change data capture facilities that can scrape logs, install triggers and so forth to capture data change events, they can be captured and propagated to the Red Hat JBoss Data Virtualization engine through a pojo bean/MDB/Session bean.

This code shows how you can use the **`EventDistributor`** interface in their own code that is deployed in the same JBoss EAP virtual machine using a Pojo/MDB/Session Bean:

```
   public class ChanageDataCapture {

    public void invalidate() {
        InitialContext ic = new InitialContext();
        EventDistributor ed =
((EventDistributorFactory)ic.lookup("teiid/event-distributor-
factory")).getEventDistributor();

        // this below line indicates that Customer table in the "model-
name" schema has been changed.
        // this result in cache reload.
        ed.dataModification("vdb-name", "version", "model-name",
"Customer");
    }
}
```

IMPORTANT

The EventDistributor interface also exposes many methods that can be used to update the costing information on your source models for optimized query planning. Note that these values are volatile and will be lost during a cluster re-start, as there is no repository to persist.

CHAPTER 3. RESULT SET CACHING

3.1. USER QUERY CACHE

User query result set caching will cache result sets based on an exact match of the incoming SQL string and PreparedStatement parameter values if present. Caching only applies to SELECT, set query, and stored procedure execution statements; it does not apply to SELECT INTO statements, or INSERT, UPDATE, or DELETE statements.

End users or client applications explicitly state whether to use result set caching. Do this by setting the JDBC ResultSetCacheMode execution property to true (by default it is set to false).

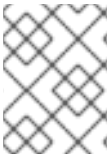
```
  Properties info = new Properties();

...

info.setProperty("ResultSetCacheMode", "true");

Connection conn = DriverManager.getConnection(url, info);
```

Alternatively, add a Cache Hint to the query.



NOTE

Note that if either of these mechanisms are used, DV must also have result set caching enabled (it is so by default).

The most basic form of the cache hint, /*+ cache */, is sufficient to inform the engine that the results of the non-update command must be cached.

```
  ...
PreparedStatement ps = connection.prepareStatement("/*+ cache */ select
col from t where col2 = ?");
ps.setInt(1, 5);
ps.execute();
...
```

The results will be cached with the default ttl and use the SQL string and the parameter value as part of the cache key. The pref_mem and ttl options of the cache hint may also be used for result set cache queries. If a cache hint is not specified, then the default time to live of the result set caching configuration will be used.

Here is a more advanced example:

```
  /*+ cache(pref_mem ttl:60000) */ select col from t
```

In this example the memory preference has been enabled and the time to live is set to 60000 milliseconds (1 minute). The time-to-live for an entry is actually treated as its maximum age and the entry may be purged sooner if the maximum number of cache entries has been reached.

IMPORTANT

Each query is re-checked for authorization using the current user's permissions, regardless of whether or not the results have been cached.

3.2. PROCEDURE RESULT CACHING

Cached virtual procedure results are used automatically when a matching set of parameter values is detected for the same procedure execution. Usage of the cached results may be bypassed when used with the OPTION NOCACHE clause.

To indicate that a virtual procedure is to be cached, its definition must include a Cache Hint.

```
/*+ cache */
BEGIN
    ...
END
```

Results will be cached with the default ttl.

The pref_mem and ttl options of the cache hint may also be used for procedure caching.

Procedure results cache keys include the input parameter values. To prevent one procedure from filling the cache, at most 256 cache keys may be created per procedure per VDB.

A cached procedure will always produce all of its results prior to allowing those results to be consumed and placed in the cache. This differs from normal procedure execution which in some situations allows the returned results to be consumed in a streaming manner.

3.3. CACHE CONFIGURATION

By default, result set caching is enabled with 1024 maximum entries with a maximum entry age of two hours. There are actually two caches configured with these settings. One cache holds results that are specific to sessions and is local to each Red Hat JBoss Data Virtualization instance. The other cache holds VDB scoped results and can be replicated.

You can also override the default maximum entry age via the Cache Hint.

Result set caching is not limited to memory. There is no explicit limit on the size of the results that can be cached. Cached results are primarily stored in the BufferManager and are subject to its configuration, including the restriction of maximum buffer space.

IMPORTANT

While the result data is not held in memory, cache keys - including parameter values - may be held in memory. Thus the cache should not be given an unlimited maximum size.

Result set cache entries can be invalidated by data change events. The max-staleness setting determines how long an entry will remain in the case after one of the tables that contributed to the results has been changed.

3.4. EXTENSION METADATA

You can use the extension metadata property data-ttl as a model property or on a source table to indicate a default TTL. A negative value means no TTL, 0 means do not cache, and a positive number indicates the time to live in milliseconds. If no TTL is specified on the table, then the schema will be checked. The TTL for the cache entry will be taken as the least positive value among all TTLs. Thus setting this value as a model property can quickly disable any caching against a particular source.

Here is an example that shows you how to set the property in the **vdb.xml**:

```
<vdb name="vdbname" version="1">
    <model name="Customers">
        <property name="teiid_rel:data-ttl" value="0"/>
        ...</para>
```

3.5. CACHE ADMINISTRATION

Clear the cache by using the AdminAPI's clearCache method. (The expected cache key is "QUERY_SERVICE_RESULT_SET_CACHE".)

```
   connectAsAdmin()
clearCache("QUERY_SERVICE_RESULT_SET_CACHE")
...
```

3.6. CACHING LIMITATIONS

- XML, BLOB, CLOB, and OBJECT type cannot be used as part of the cache key for prepared statement of procedure cache keys.
- The exact SQL string, including the cache hint if present, must match the cached entry for the results to be reused. This allows cache usage to skip parsing and resolving for faster responses.
- Result set caching is transactional by default using the NON_XA transaction mode. To use full XA support, change the configuration to use NON_DURABLE_XA.
- Clearing the results cache clears all cache entries for all VDBs.

3.7. TRANSLATOR RESULT CACHING

Translators can contribute cache entries into the result set cache by using the CacheDirective object. The resulting cache entries behave just as if they were created by a user query.

3.8. CACHE HINTS AND OPTIONS

A query cache hint can be used in the following ways:

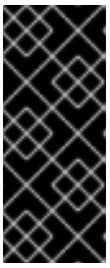
- Indicate that a user query is eligible for result set caching and set the cache entry memory preference, time to live and so forth.
- Set the materialized view memory preference, time to live, or updatablity.

- Indicate that a virtual procedure should be cachable and set the cache entry memory preference, time to live and so on

```
      /*+ cache[([pref_mem] [ttl:n] [updatable])] [scope:
(session|user|vdb)] */ sql ...
```

- The cache hint should appear at the beginning of the SQL. It will not have any affect on INSERT/UPDATE/DELETE statements or INSTEAD OF TRIGGERS.

- pref_mem- if present indicates that the cached results should prefer to remain in memory. The results may still be paged out based upon memory pressure.



IMPORTANT

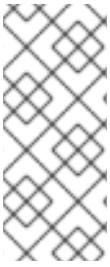
Care should be taken to not over use the pref_mem option. The memory preference is implemented with Java soft references. While soft references are effective at preventing out of memory conditions. Too much memory held by soft references can limit the effective working memory. Consult your JVM options for clearing soft references if you need to tune their behavior.

- ttl:n- if present n indicates the time to live value in milliseconds. The default value for result set caching is the default expiration for the corresponding Infinispan cache. There is no default time to live for materialized views.

- updatable- if present indicates that the cached results can be updated. This defaults to false for materialized views and to true for result set cache entries.

- scope- There are three different cache scopes: session - cached only for current session, user - cached for any session by the current user, vdb - cached for any user connected to the same vdb. For cached queries the presense of the scope overrides the computed scope. Materialized views on the other hand default to the vdb scope. For materialized views explicitly setting the session or user scopes will result in a non-replicated session scoped materialized view.

The pref_mem, ttl, updatable, and scope values for a materialized view may also be set via extension properties on the view (by using the teiid_rel namespace with MATVIEW_PREFER_MEMORY, MATVIEW_TTL, MATVIEW_UPDATABLE, and MATVIEW_SCOPE respectively). If both are present, the use of an extension property supersedes the usage of the cache hint.



NOTE

The form of the query hint must be matched exactly for the hint to be effective. For a user query if the hint is not specified correctly, e.g. /*+ cach(pref_mem) */, it will not be used by the engine nor will there be an informational log. It is currently recommended that you verify in your testing that the user command in the query plan has retained the proper hint.

Individual queries may override the use of cached results by specifying OPTION NOCACHE on the query. 0 or more fully qualified view or procedure names may be specified to exclude using their cached results. If no names are specified, cached results will not be used transitively. In this case, no cached results will be used at all:

```
SELECT * from vg1, vg2, vg3 WHERE … OPTION NOCACHE
```

In this case, only the vg1 and vg3 caches will be skipped. vg2 or any cached results nested under vg1 and vg3 will be used:

```
SELECT * from vg1, vg2, vg3 WHERE … OPTION NOCACHE vg1, vg3
```

OPTION NOCACHE may be specified in procedure or view definitions. In that way, transformations can specify to always use real-time data obtained directly from sources.

APPENDIX A. REVISION HISTORY

Revision 6.4.0-20 **Thu Jun 06 2017** **David Le Sage**

Updated for 6.4.