



Red Hat JBoss Data Virtualization 6.2

Development Guide Volume 5: Caching Guide

This guide is intended for developers

Red Hat JBoss Data Virtualization 6.2 Development Guide Volume 5: Caching Guide

This guide is intended for developers

Red Hat Customer Content Services

Legal Notice

Copyright © 2017 Red Hat, Inc.

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](#). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

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

Table of Contents

CHAPTER 1. READ ME	3
1.1. BACK UP YOUR DATA	3
1.2. VARIABLE NAME: EAP_HOME	3
1.3. VARIABLE NAME: MODE	3
1.4. RED HAT DOCUMENTATION SITE	3
CHAPTER 2. SOME KEY DEFINITIONS	4
2.1. RESULT SET CACHING	4
2.2. INTERNAL MATERIALIZATION	4
2.3. EXTERNAL MATERIALIZATION	4
2.4. MATERIALIZED VIEWS	4
2.5. MATERIALIZATION TABLE	4
2.6. NOCACHE OPTION	4
2.7. RESULTS CACHING	4
CHAPTER 3. USING CACHING	5
3.1. CACHING	5
3.2. INTERNAL MATERIALIZATION	5
3.3. EXTERNAL MATERIALIZATION	9
3.4. CODE TABLE CACHING	19
3.5. PROGRAMMATIC CONTROL	21
CHAPTER 4. RESULT SET CACHING	23
4.1. USER QUERY CACHE	23
4.2. PROCEDURE RESULT CACHING	24
4.3. CACHE CONFIGURATION	24
4.4. EXTENSION METADATA	24
4.5. CACHE ADMINISTRATION	25
4.6. CACHING LIMITATIONS	25
4.7. TRANSLATOR RESULT CACHING	25
4.8. CACHE HINTS AND OPTIONS	25
APPENDIX A. REVISION HISTORY	28

CHAPTER 1. READ ME

1.1. BACK UP YOUR DATA



WARNING

Red Hat recommends that you back up your system settings and data before undertaking any of the configuration tasks mentioned in this book.

1.2. VARIABLE NAME: EAP_HOME

EAP_HOME refers to the root directory of the Red Hat JBoss Enterprise Application Platform installation on which JBoss Data Virtualization has been deployed.

1.3. VARIABLE NAME: MODE

MODE will either be `standalone` or `domain` depending on whether JBoss Data Virtualization is running in standalone or domain mode. Substitute one of these whenever you see **MODE** in a file path in this documentation. (You need to set this variable yourself, based on where the product has been installed in your directory structure.)

1.4. RED HAT DOCUMENTATION SITE

Red Hat's official documentation site is available at <https://access.redhat.com/site/documentation/>. There you will find the latest version of every book, including this one.

CHAPTER 2. SOME KEY DEFINITIONS

2.1. RESULT SET CACHING

Result set caching is the caching of the final results of a user query.

2.2. INTERNAL MATERIALIZATION

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

2.3. EXTERNAL MATERIALIZATION

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

2.4. MATERIALIZED VIEWS

Materialized views are the cached contents of a virtual table. The queries written using this virtual table are served from the cached contents instead from original source.

2.5. 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.

2.6. NOCACHE OPTION

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

2.7. RESULTS 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.

CHAPTER 3. USING CACHING

3.1. 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-datafederation` quick start.

3.2. INTERNAL MATERIALIZATION

3.2.1. Use Internal Materialization

Procedure 3.1. Use Internal Materialization

1. Build a VDB using the Teiid Designer for your usecase normally.
2. Identify all the "Virtual Tables", that you think can use caching, then click on the table.
3. In the `Properties` panel, switch the `Materialized` property to "true"



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.

3.2.2. Internal Materialization Usage Steps

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 3.1. Properties

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

Property	Description
<code>teiid_rel:MATVIEW_SCOPE</code>	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_UNCOMMITTED`) 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.

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

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

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.

Using TTL Snapshot Refresh

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`.

```
Auto-refresh Transformation Query
/*+ 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).

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.

Updatable

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 corresponding 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`.

Transformation Query:

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

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.

Secondary Indexes

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 workaroud some of these limitations though.

Function-based indexes are supported but they 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 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.

3.2.3. Internal Materialization Options

By default the results are cached for the life of virtual machine, unless you force the cache refresh through admin-console or by executing a system procedure. However, if you like to invalidate the cache after a pre-defined time add the following cache hint to the transformation SQL that makes up your view. This will set it for 5 mins (300000 milliseconds). (TTL is an abbreviation for time to live.)

```
/*+ cache(ttl:300000) */ select * from Source.UpdateProduct
```

After doing this, deploy the VDB to the server. The next time you use the virtual table directly or indirectly by issuing a user query against this VDB, the virtual table contents will be cached. The cached contents will be used for any future queries.

There are few different ways to load or refresh the contents of the Internal Materialized tables:

- Loading of materialized contents is automatic when the user query is executed like the one below. (It is assumed for this example that you used the cache hint mentioned above.)

```
SELECT * FROM Products.UpdateProduct
```

This will retain the contents of the materialization table for 300000 milliseconds (5 minutes). After the defined cache time expires, any user query like above will see that the cache time has expired and will start a new asynchronous job to load the contents of the view into materialization cache.



IMPORTANT

In this example, the user query started a new refresh/load job. As a default, this invalidates the current results unless the vdb property *lazy-invalidate* is set to `true`. For details about this property, see [Using TTL Snapshot Refresh](#) in [Section 3.2.2, “Internal Materialization Usage Steps”](#). If this property is set, while the load/refresh job is running, the user queries continue to return the expired results. When the refresh/reload job completes, then the old results will be flushed and new results will be used for any future user requests.

The main idea here is not to block user queries while the refresh job is running. This is mainly due to the fact that you can not predict how long a refresh job going to run. This is "implicit" materialization loading, if your use case does not allow this kind of refresh then use below techniques for "explicit" materialization load.

- To start a load/refresh job on internal materialized view immediately, launch the "admin-console", go to the **Runtime** tab, select **Virtual Databases**, select your VDB, click the **Caching** tab, **Materialized Tables**, select your view and click **Reload**.

When you choose to invalidate, then any user queries that come in against this materialization table will be blocked until the cache contents are fully loaded. If the invalidation is not chosen then the behaviour will be the same as that seen in the first option above.

- You can also issue SQL command `"EXEC SYSADMIN.refreshMatView(viewname=>'Portfolio.UpdateTable', invalidate=>true)"` from any JDBC query tool or through Admin Shell to clear the contents of materialization table and start the refresh of contents immediately. The same invalidation rules apply here as in the above.

This third option has the advantage of allowing you to automate the refresh process via scripting it, so you can run it at frequent intervals.

3.3. EXTERNAL MATERIALIZATION

3.3.1. Use External Materialization

Procedure 3.2. Use External Materialization

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.

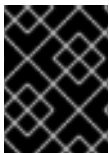
12. Select the type of the database and DDL file name and click **Finish**.

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 Teiid 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.

3.3.2. External Materialization Usage Steps

Create materialized views and corresponding physical materialized target tables in Designer. 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**.

Generate the DDL for your physical model materialization target tables. This can be done by selecting the model, right clicking, then choosing **Export-Metadate Modeling-Data Definition Language (DDL) File**. This script can be used to create the desired schema for your

materialization target on whatever source you choose.

Determine a load and refresh strategy. With the schema created the simplest approach is to load the data. The load can even be done through Teiid with this command: `insert into target_table select * from matview option nocache matview`

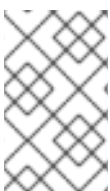
That however may be too simplistic because your index creation may perform better if deferred until after the table has been created. Also full snapshot refreshes are best done to a staging table then swapping it for the existing physical table to ensure that the refresh does not impact user queries 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, Teiid 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 3.2. Extension Properties

Property	Description	Optional	Default
teiid_rel:ALLOW_MATVIEW_MANAGEMENT	Allow Teiid 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
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, Teiid 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

Property	Description	Optional	Default
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
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 above properties 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>
```

With the Dynamic VDB and above table extension properties, you can set up an external materialization for an embedded server.

MATERIALIZED_TABLE is necessary, it can be any name, but its structure should match to materialization view. MATVIEW_STATUS_TABLE is necessary, it should be named as 'status':

```
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 uses a Silent Column Changes function, meaning that the , 'long' type will silently change to 'MEDIUMTEXT'. Therefore, if you execute the schema above in MariaDB, the 'Cardinality' and 'LoadNumber' columns will change to 'bigint'.

If 'teiid_rel:MATVIEW_LOAD_SCRIPT' MATERIALIZED_STAGE_TABLE is necessary, it can have any name, but its structure should match the materialization view.

Create Materialization Views and corresponding physical materialized target table, staging table, status table, it should be manually created in a model of Dynamic VDB, the model type should be 'VIRTUAL'.

The TransactionManager is necessary except when you add the ConnectionFactory and Translator to the Embedded Server. Here is the configuration code for it:

```
EmbeddedServer server = new EmbeddedServer();
...
server.addConnectionFactory("name", Object);
...
server.addTranslator("name", ExecutionFactory);
...
EmbeddedConfiguration config = new EmbeddedConfiguration();
config.setTransactionManager(EmbeddedHelper.getTransactionManager());
server.start(config);
...
server.deployVDB("matView-vdb.xml");
```

The following steps show a sample Dynamic VDB with Materialization view in Embedded Server.

Run this schema script against a H2 Database before deploy Dynamic VDB

```
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)
);

CREATE TABLE h2_test_mat(
    product_id integer,
    SYMBOL varchar(16)
);

CREATE TABLE mat_test_staging(
    product_id integer,
    SYMBOL varchar(16)
);

CREATE TABLE PRODUCT (
    ID integer,
    SYMBOL varchar(16),
    COMPANY_NAME varchar(256),
    CONSTRAINT PRODUCT_PK PRIMARY KEY(ID)
);

INSERT INTO PRODUCT (ID,SYMBOL,COMPANY_NAME)
VALUES(100,'IBM','International Business Machines Corporation');
INSERT INTO PRODUCT (ID,SYMBOL,COMPANY_NAME) VALUES(101,'DELL','Dell
Computer Corporation');
INSERT INTO PRODUCT (ID,SYMBOL,COMPANY_NAME) VALUES(102,'HPQ','Hewlett-
Packard Company');
INSERT INTO PRODUCT (ID,SYMBOL,COMPANY_NAME) VALUES(103,'GE','General
Electric Company');
INSERT INTO PRODUCT (ID,SYMBOL,COMPANY_NAME) VALUES(104,'SAP','SAP AG');
INSERT INTO PRODUCT (ID,SYMBOL,COMPANY_NAME) VALUES(105,'TM','Toyota Motor
Corporation');

```

Create the Dynamic VDB with DDL content:

```

<metadata type="DDL">
CREATE VIEW MatView (
    product_id integer,
    symbol string
)OPTIONS (MATERIALIZED 'TRUE', UPDATABLE 'TRUE',
    MATERIALIZED_TABLE 'Accounts.h2_test_mat',
    "teiid_rel:MATVIEW_TTL" 20000,
    "teiid_rel:MATVIEW_BEFORE_LOAD_SCRIPT" 'execute
accounts.native(''truncate table mat_test_staging'');',
    "teiid_rel:MATVIEW_AFTER_LOAD_SCRIPT" 'execute
accounts.native(''ALTER TABLE h2_test_mat RENAME TO
h2_test_mat_temp'');execute accounts.native(''ALTER TABLE mat_test_staging
RENAME TO h2_test_mat'');execute accounts.native(''ALTER TABLE
h2_test_mat_temp RENAME TO mat_test_staging'');',
    "teiid_rel:ON_VDB_DROP_SCRIPT" 'DELETE FROM
Accounts.status WHERE Name='''MatView''' AND schemaname = '''Stocks''',
    "teiid_rel:MATERIALIZED_STAGE_TABLE"
'Accounts.mat_test_staging',

```

```

        "teiid_rel:ALLOW_MATVIEW_MANAGEMENT" 'true',
        "teiid_rel:MATVIEW_STATUS_TABLE" 'status',
        "teiid_rel:MATVIEW_SHARE_SCOPE" 'NONE',
        "teiid_rel:MATVIEW_ONERROR_ACTION" 'THROW_EXCEPTION')
    AS
    SELECT A.ID, A.symbol FROM Accounts.PRODUCT AS A;
</metadata>
</model>

```

To deploy the Dynamic VDB to the Embedded Server, run this code:

```

Thread.currentThread().sleep(10 * 1000); // wait loadMatView finish
executeQuery(conn, "select * from MatView");
executeUpdate(conn, "INSERT INTO PRODUCT (ID,SYMBOL,COMPANY_NAME)
VALUES(2000,'RHT','Red Hat Inc')");

Thread.currentThread().sleep(30 * 1000); // make sure MatView be updated

executeQuery(conn, "select * from MatView");
executeUpdate(conn, "DELETE FROM PRODUCT WHERE ID = 2000");

Thread.currentThread().sleep(30 * 1000); // make sure MatView be updated

executeQuery(conn, "select * from MatView");

```

The test result is the MatView can catch the updated of underlying source table 'PRODUCT'.

3.3.3. External Materialization Options

You must undertake every option in external materialization mode manually.

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 SquirrelL 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**

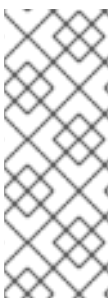
Here is how you would create an AdminShell script to automatically load the materialized table:

```

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 provide 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.

3.3.4. Set Up a Staging Table

Procedure 3.3. Set Up a Staging Table

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 table materialized table that has been created, and click on "Name In Source" property.
10. Create two identical tables, one with "mv_" prefix and another with "st_" prefix. ("mv_" will stand for materialized view and "st_" will stand for the staging table.)
11. Make sure that the "Materialized Table" property is set to the table with "mv_" prefix. Also make sure that both these tables have their "Name in Source" edited and renamed with respective prefixed name.
12. Create the DDL file and the database. (The difference now is there are two identical tables for each materialized view.)
13. Load the contents into staging table using this query (substitute with the names of your own tables): **INSERT INTO mv_view.ST_UpdateProduct SELECT * FROM Portfolio.UpdateProduct OPTION NOCACHE**

14. Once the table is loaded, use the database's "RENAME" table command to rename the staging table (`st_XXX`) to the materialized table (`mv_XXX`). Here is the syntax to do this with MySQL:
ALTER TABLE ST_UpdateProduct RENAME MV_UpdateProduct

The reason you need to run the "rename" command is that the staging table can be used for loading the data (except for the very first time you use it). Meanwhile, materialized table will serve the user queries. Once the staging is loaded, rename will switch the identity of staged with the materialized such that any future queries will be against newly loaded contents. You can keep repeating this cycle in regular intervals and never serve empty results or wait for results to load. To populate the results via an ETL tool, you can do so too in this step by disregarding the above SQL command.

NOTE

You can script this entire process so that it runs at a regular interval. This example code shows you how to script it if you are using the MySQL database. (Make sure you provide the MySQL jdbc driver in the AdminShell classpath before running the script):

```
// connect to mysql first
mysql =
Sql.newInstance("jdbc:mysql://host_name:port/dbname",
"username","password", "com.mysql.jdbc.Driver")
mysql.execute("TRUNCATE TABLE ST_UpdateProducts");

// use the Teiid connection and load the staging
table
sql=connect(${url}, ${user}, ${password});
sql.execute("INSERT INTO mv_view.st_UpdateProduct
SELECT * FROM Portfolio.UpdateProduct OPTION NOCACHE");
sql.close();

// issue the rename command, so that materialization
takes into effect
mysql.execute("ALTER TABLE ST_UpdateProduct RENAME
MV_UpdateProduct");
mysql.close();

// create any indexes you need to
```

Add the script to your cron tab or Windows Scheduler and set it to run at a regular interval to keep your data fresh.

3.4. CODE TABLE CACHING

3.4.1. 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.

3.4.2. Create a Materialized View for Code Table Caching

Procedure 3.4. 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.

3.5. 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 can be propagated to the Teiid engine through a pojo bean/MDB/Session Bean deployed in the Red Hat JBoss EAP engine.

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 ChangeDataCapture {

    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 4. RESULT SET CACHING

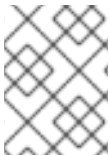
4.1. USER QUERY CACHE

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.

4.2. 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.

4.3. 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 Teiid 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.

4.4. 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>
```

4.5. 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")
...
```

4.6. 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.

4.7. 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.

4.8. CACHE HINTS AND OPTIONS

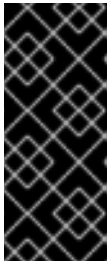
A query cache hint can be used in the following ways:

- 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 updatability.

- 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 6.2.0-24

Updated for 6.2.

Thu Dec 10 2015

David Le Sage