Teiid - Scalable Information Integration

1

Teiid Caching Guide

7.6

Overview

Teiid provides several capabilities for caching data including: materialized views, result set caching, and code table caching. These techniques can be used to significantly improve performance in many situations.

With the exception of external materialized views, the cached data is accessed through the BufferManager. For better performance the BufferManager setting should be adjusted to the memory constraints of your installation. See the Admin Guide for more on parameter tuning.

Results Caching

Teiid provides the capability to cache the results of specific user queries and virtual procedure calls. This caching technique can yield significant performance gains if users of the system submit the same queries or execute the same procedures often.

2.1. Support Summary

- Caching of user query results including XML document model results.
- Caching of virtual procedure results.
- Scoping of results is automatically determined to be VDB/user (replicated) or session level.
- Configurable number of cache entries and time to live.
- Administrative clearing.

2.2. User Interaction

2.2.1. 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. This can be done by setting the JDBC ResultSetCacheMode execution property to true (default false) or by adding a *cache hint* to the query. Note that if either of these mechanisms are used, Teiid must also have result set caching enabled (the default is enabled).

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

Example 2.1. PreparedStatement ResultSet Caching

```
...
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.

Example 2.2. Advanced ResultSet Caching

```
/*+ 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 or 1 minute. The ttl for an entry is actually treated as it's maximum age and the entry may be purged sooner if the maximum number of cache entries has been reached.

Note

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

2.2.2. Procedure Result Cache

Similar to materialized views, 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 with an `OPTION NOCACHE` clause. See the *OPTION NOCACHE* section for more on its usage.

2.3. Cached Virtual Procedure Definition

To indicate that a virtual procedure (only definable by Teiid Designer) should be cached, it's definition should include a *cache hint*.

Example 2.3. Procedure Caching

```
/*+ cache */ CREATE VIRTUAL PROCEDURE
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.

2.4. Cache Configuration

By default result set caching is enabled with 1024 maximum entries with a maximum entry age of 2 hours. There are actually 2 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. See the `<jboss-install>/standalone/configuration/standalone-teiid.xml` config file or the Console's "Runtime Engine Properties" for tuning the configuration. The user may 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 it's configuration - including the restriction of maximum buffer space.

Note

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. See the Developers Guide for further customization.

2.5. Cache Administration

The result set cache can be cleared through the AdminAPI using the `clearCache` method. The expected cache key is "QUERY_SERVICE_RESULT_SET_CACHE".

Example 2.4. Clearing the ResultSet Cache in AdminShell

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

See the Admin Guide for more on using the AdminAPI and AdminShell.

2.6. 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 not transactional. Transactions depend on (and enforce) consistency of data, and cached data is not guaranteed to be consistent with the data store’s data.
- Clearing the results cache clears all cache entries for all VDBs.

Materialized Views

Teiid supports materialized views. Materialized views are just like other views, but their transformations are pre-computed and stored just like a regular table. When queries are issued against the views through the Teiid Server, the cached results are used. This saves the cost of accessing all the underlying data sources and re-computing the view transforms 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 data that is "stale" within some 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.

3.1. Support Summary

- Caching of relational table or view records (pre-computing all transformations)
- Model-based definition of virtual groups to cache (requires Teiid Designer)
- User ability to override use of materialized view cache for specific queries through `OPTION NOCACHE`

3.2. User Interaction

Similar to cached procedures, materialized view tables are used automatically when a query accesses the corresponding view. Usage of the cached results may be bypassed with an `OPTION NOCACHE` clause. See the *OPTION NOCACHE* section for more on its usage.

3.3. Materialized View Definition

Materialized views are defined in Teiid Designer by setting the materialized property on a table or view in a virtual (view) relational model. Setting this property's value to true (the default is false) allows the data generated for this virtual table to be treated as a materialized view.

Note

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

The target materialized table may also be set in the properties. If the value is left blank, the default, then internal materialization will be used. Otherwise for external materialization, the value should

reference the fully qualified name of a table (or possibly view) with the same columns as the materialized view. For most basic scenarios the simplicity of internal materialization makes it the more appealing option.

Reasons to use external materialization

- The cached data needs to be fully durable. Internal materialization should not survive a cluster restart.
- Full control is needed of loading and refresh. Internal materialization does offer several system supported methods for refreshing, but does not give full access to the materialized table.
- Control is needed over the materialized table definition. Internal materialization does support *secondary indexes*, but they cannot be directly controlled. Constraints or other database features cannot be added to internal materialization tables.
- The data volume is large. Internal materialization (and temp tables in general) have memory overhead for each page. A rough guideline is that there can be 100 million rows in all materialized tables across all VDBs for every gigabyte of heap.

Note

Materialized view tables are always scoped to the VDB. If a materialized view definition directly or transitively contains a non-deterministic function call, such as random or hasRole, the resulting table will contain only the initially evaluated values. In most instances you should consider nesting a materialized view without the deterministic results that is joined with relevant non-deterministic values in a parent view.

3.4. External Materialization

External materialized views cache their data in an external database system. External materialized views give the administrator full control over the loading and refresh strategies.

Since the actual physical cache for materialized views is maintained external to the Teiid system, there is no predefined policy for clearing and managing the cache. These policies will be defined and enforced by administrators of the Teiid system.

Typical Usage Steps

1. 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"

2. 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". This script can be used to create the desired schema for your materialization target on whatever source you choose.

3. Determine a load and refresh strategy. With the schema created the most simplistic approach is to just load the data. The load can even be done through Teiid with `insert into target_table select * from matview option nocache`. That however may be too simplistic because your index creation may be more performant 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.

3.5. Internal Materialization

Internal materialization creates Teiid 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 at each Teiid 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 cache hint, when used in the context of an internal materialized view transformation query, provides the ability to fine tune the materialized table. 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.

3.5.1. Loading And Refreshing

An internal materialized view table is initially in an invalid state (there is no data). 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.

Example 3.1. 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.

3.5.1.1. 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', false)`, but performed asynchronously so that user queries do not block on the load.

Example 3.2. 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).

3.5.1.1.1. Limitations

- The automatic ttl refresh is not intended for complex loading scenarios, as nested materialized views will be used by the refresh query.
- The automatic 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.

3.5.1.2. 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 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`.

Example 3.3. Updatable Transformation Query

Transformation Query:

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

Update SQL:

```
CALL SYSADMIN.updateMatViewRow(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.

3.5.2. 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, such as specifying the evaluated expressions, sort direction, additional columns to cover, etc. You may workaround some of these limitations though.

- If a function based index is needed, 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.

3.5.3. Clustering Considerations

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.

Code Table Caching

Teiid provides a short cut to creating an internal materialized view table via the lookup function.

The lookup function provides a way to get a value out of a table when a key value is provided. The function automatically caches all the values in the referenced table for the specified key/value pairs. The cache is created the first time it is used in a particular Teiid process. Subsequent lookups against the same table using the same key and value columns will use the cached information.

This caching solution is appropriate for integration of "reference data" with transactional or operational data. Reference data are static data sets – typically small – which are used very frequently in most enterprise applications. Examples are ISO country codes, state codes, and different types of financial instrument identifiers.

4.1. User Interaction

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, key element, and returned element (the first 3 arguments to the function), the Teiid System caches the entire contents of the table being accessed. Subsequent lookup function uses with the same combination of parameters uses the cached table data.

See the Reference for more information on use of the lookup function.

Example 4.1. Country Code Lookup

```
lookup('ISOCountryCodes', 'CountryName', 'CountryCode', 'US')
```

4.2. 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.

4.3. Materialized View Alternative

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.

Example 4.2. Country Code Lookup Against A Mat View

```
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 or index on CountryName. The scalar subquery will use the index to lookup the country code for each country name in tbl.

Reasons to 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 of the cache hint.
- The ability to use *OPTION NOCACHE*.
- There is almost no performance difference.

Steps to create a materialized view:

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 *cache hint* `/*+ cache(pref_mem) */` to indicate that the table data pages should prefer to remain in memory.

Just as with the lookup function, the materialized view table will be created on first use and reused subsequently. See the *Materialized View Chapter* for more on materialized views.

Hints and Options

5.1. Cache Hint

A query cache hint can be used to:

- Indicate that a user query is eligible for result set caching and set the cache entry memory preference or time to live.
- 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 or time to live.

```
/*+ 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 virtual update procedure definitions.
- *pref_mem* - if present indicates that the cached results should prefer to remain in memory. They are not however required to be memory only.

Note

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.
- *updatable* - if present indicates that the cached results can be updated. This is currently only applicable to materialized views.
- *scope* - if present indicates the override scope of query results. Using this flag, the user can override the computed 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.

5.1.1. Limitations

- The form of the query hint must be matched exactly for the hint to have affect. 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. As a workaround, the query plan may be checked though (see the Client Developers Guide) to see if the user command in the plan has retained the proper hint.

5.2. OPTION NOCACHE

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.

Example 5.1. Full NOCACHE

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

No cached results will be used at all.

Example 5.2. Specific NOCACHE

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

Only the vg1 and vg3 caches will be skipped, vg2 or any cached results nested under vg1 and vg3 will be used.

`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.