22 Statement and Result Set Caching

This chapter describes the benefits and use of Statement caching, an Oracle Java Database Connectivity (JDBC) extension.

> **Note:**
>
> Use statement caching only when you are sure that the table structure remains the same in the database. If you alter the table structure and then reuse a statement that was created and executed before changing the table structure, then you may get an error.

This chapter contains the following sections:

22.1 About Statement Caching

Statement caching improves performance by caching executable statements that are used repeatedly, such as in a loop or in a method that is called repeatedly. Starting from JDBC 3.0, JDBC standards define a statement-caching interface.

Statement caching can do the following:

- Prevent the overhead of repeated cursor creation
- Prevent repeated statement parsing and creation
- Reuse data structures in the client

This section covers the following topics:

> **Note:**
>
> Oracle strongly recommends you use the implicit Statement cache. Oracle JDBC drivers are designed on the assumption that the implicit Statement cache is enabled. So, not using the Statement cache will have a negative impact on performance.

22.1.1 Basics of Statement Caching

Applications use the Statement cache to cache statements associated with a particular physical connection. The cache is associated with an `OracleConnection` object. `OracleConnection` includes methods to enable Statement caching. When you enable Statement caching, a statement object is cached when you call the `close` method.

Because each physical connection has its own cache, multiple caches can exist if you enable Statement caching for multiple physical connections. When you enable Statement caching on a connection cache, the logical connections benefit from the Statement caching that is enabled on the underlying physical connection. If you try to enable Statement caching on a logical connection held by a connection cache, then this will throw an exception.

There are two types of Statement caching: implicit and explicit. Each type of Statement cache can be enabled or disabled independent of the other. You can have either, neither, or both in effect. Both types of Statement caching share a single cache per connection.

22.1.2 Implicit Statement Caching

When you enable implicit Statement caching, JDBC automatically caches the prepared or callable statement when you call the `close` method of this statement object. The prepared and callable statements are cached and retrieved using standard connection object and statement object methods.

Plain statements are not implicitly cached, because implicit Statement caching uses a SQL string as a key and plain statements are created without a SQL string. Therefore, implicit Statement caching applies only to the `OraclePreparedStatement` and `OracleCallableStatement` objects, which are created with a SQL string. You *cannot* use implicit Statement caching with `OracleStatement`. When you create an `OraclePreparedStatement` or `OracleCallableStatement`, the JDBC driver automatically searches the cache for a matching statement. The match criteria are the following:

- The SQL string in the statement must be identical to one in the cache.
- The statement type must be the same, that is, prepared or callable.
- The scrollable type of result sets produced by the statement must be the same, that is, forward-only or scrollable.

If a match is found during the cache search, then the cached statement is returned. If a match is not found, then a new statement is created and returned. In either case, the statement, along with its cursor and state are cached when you call the `close` method of the statement object.

When a cached `OraclePreparedStatement` or `OracleCallableStatement` object is retrieved, the state and data information are automatically reinitialized and reset to default values, while metadata is saved. Statements are removed from the cache to conform to the maximum size using a Least Recently Used (LRU) algorithm.

Note:

The JDBC driver does not clear metadata. However, although metadata is saved for performance reasons, it has no semantic impact. A statement that comes from the implicit cache appears as if it were newly created.

You can prevent a particular statement from being implicitly cached.

Related Topics

- About Using Implicit Statement Caching

22.1.3 Explicit Statement Caching

Explicit Statement caching enables you to cache and retrieve selected prepared and callable statements. Explicit Statement caching relies on a key, an arbitrary Java `String` that you provide.

> **Note:**
>
> Plain statements cannot be cached.

Because explicit Statement caching retains statement data and state as well as metadata, it has a performance edge over implicit Statement caching, which retains only metadata. However, you must be cautious when using this type of caching, because explicit Statement caching saves all three types of information for reuse and you may not be aware of what data and state are retained from prior use of the statements.

Implicit and explicit Statement caching can be differentiated on the following points:

- Retrieving statements

  In the case of implicit Statement caching, you take no special action to retrieve statements from a cache. Instead, whenever you call `prepareStatement` or `prepareCall`, JDBC automatically checks the cache for a matching statement and returns it if found. However, in the case of explicit Statement caching, you use specialized Oracle `WithKey` methods to cache and retrieve statement objects.

- Providing key

  Implicit Statement caching uses the SQL string of a prepared or callable statement as the key, requiring no action on your part. In contrast, explicit Statement caching requires you to provide a Java `String`, which it uses as the key.

- Returning statements

  During implicit Statement caching, if the JDBC driver cannot find a statement in cache, then it will automatically create one. However, during explicit Statement caching, if the JDBC driver cannot find a matching statement in cache, then it will return a `null` value.

Table 22-1 compares the different methods employed in implicit and explicit Statement caching.

Table 22-1 Comparing Methods Used in Statement Caching

Type of Caching	Allocate	Insert Into Cache	Retrieve From Cache
Implicit	`prepareStatement` `prepareCall`	`close`	`prepareStatement` `prepareCall`
Explicit	`createStatement` `prepareStatement` `prepareCall`	`closeWithKey`	`getStatementWithKey` `getCallWithKey`

22.2 About Using Statement Caching

This section discusses the following topics:

- About Enabling and Disabling Statement Caching
- About Closing a Cached Statement
- About Using Implicit Statement Caching
- About Using Explicit Statement Caching

22.2.1 About Enabling and Disabling Statement Caching

When using the `OracleConnection` API, implicit and explicit Statement caching can be enabled or disabled independent of one other. You can have either, neither, or both of them in effect.

Enabling Implicit Statement Caching

There are two ways to enable implicit Statement caching. The first method enables Statement caching on a nonpooled physical connection, where you need to explicitly specify the Statement size for every connection, using the `setStatementCacheSize` method. The second method enables Statement caching on a pooled logical connection. Each connection in the pool has its own Statement cache with the same maximum size that can be specified by setting the `MaxStatementsLimit` property.

Method 1

Perform the following steps:

- Call the `OracleDataSource.setImplicitCachingEnabled(true)` method on the connection to set the `OracleDataSource` property `implicitCachingEnabled` to `true`. For example:

```
OracleDataSource ods =  new OracleDataSource();
...
ods.setImplicitCachingEnabled(true);
...
```

- Call the `OracleConnection.setStatementCacheSize` method on the physical connection. The argument you supply is the maximum number of statements in the cache. For example, the following code specifies a cache size of ten statements:

```
((OracleConnection)conn).setStatementCacheSize(10);
```

Method 2

Perform the following steps:

- Set the `OracleDataSource` properties `implicitCachingEnabled` and `connectionCachingEnabled` to `true`. For example:

```
OracleDataSource ods =  new OracleDataSource();
...
ods.setConnectionCachingEnabled( true );
ods.setImplicitCachingEnabled( true );
...
```

- Set the `MaxStatementsLimit` property to a positive integer on the connection cache, when using the connection cache. For example:

```
    Properties cacheProps = new Properties();
    ...
    cacheProps.put( "MaxStatementsLimit", "50" );
```

To determine whether implicit caching is enabled, call `getImplicitCachingEnabled`, which returns `true` if implicit caching is enabled, `false` otherwise.

> **Note:**
>
> Enabling Statement caching enables both implicit and explicit Statement caching.

Disabling Implicit Statement Caching

Disable implicit Statement caching by calling `setImplicitCachingEnabled(false)` on the connection or by setting the `ImplicitCachingEnabled` property to `false`.

Enabling Explicit Statement Caching

To enable explicit Statement caching you must first set the Statement cache size. For setting the cache size, call `OracleConnection.setStatementCacheSize` method on the physical connection. The argument you supply is the maximum number of statements in the cache. An argument of `0` specifies no caching. To check the cache size, use the `getStatementCacheSize` method in the following way:

```
System.out.println("Stmt Cache size is " +
   ((OracleConnection)conn).getStatementCacheSize());
```

The following code specifies a cache size of ten statements:

```
((OracleConnection)conn).setStatementCacheSize(10);
```

Enable explicit Statement caching by calling `setExplicitCachingEnabled(true)` on the connection.

To determine whether explicit caching is enabled, call `getExplicitCachingEnabled`, which returns `true` if explicit caching is enabled, `false` otherwise.

> **Note:**
>
> - You enable implicit and explicit caching for a particular physical connection independently. Therefore, it is possible to do Statement caching both implicitly and explicitly during the same session.
> - Implicit and explicit Statement caching share the *same* cache. Remember this when you set the statement cache size.

Disabling Explicit Statement Caching

Disable explicit Statement caching by calling `setExplicitCachingEnabled(false)`. Disabling caching or closing the cache purges the cache. The following example disables explicit Statement caching:

```
((OracleConnection)conn).setExplicitCachingEnabled(false);
```

22.2.2 About Closing a Cached Statement

Perform the following to close a Statement and assure that it is not returned to the cache:

In J2SE 5.0

- Disable caching for that statement

  ```
  stmt.setDisableStmtCaching(true);
  ```

- Call the `close` method of the statement object

  ```
  stmt.close();
  ```

In JSE 6.0

```
stmt.setPoolable(false);
stmt.close();
```

Physically Closing a Cached Statement

With implicit Statement caching enabled, you cannot physically close statements manually. The `close` method of a statement object caches the statement instead of closing it. The statement is physically closed automatically under one of following three conditions:

- When the associated connection is closed
- When the cache reaches its size limit and the least recently used statement object is preempted from cache by the LRU algorithm
- If you call the `close` method on a statement for which Statement caching is disabled

22.2.3 About Using Implicit Statement Caching

Once you enable implicit Statement caching, by default, all prepared and callable statements are automatically cached. Implicit Statement caching includes the following steps:

1. Enable implicit Statement caching.
2. Allocate a statement using one of the standard methods.
3. Disable implicit Statement caching for any particular statement you do not want to cache. This is an optional step.
4. Cache the statement using the `close` method.
5. Retrieve the implicitly cached statement by calling the appropriate standard prepare method.

Allocating a Statement for Implicit Caching

To allocate a statement for implicit Statement caching, use either the `prepareStatement` or `prepareCall` method as you would typically.

The following code allocates a new statement object called `pstmt`:

```
PreparedStatement pstmt = conn.prepareStatement
   ("UPDATE emp SET ename = ? WHERE rowid = ?");
```

Disabling Implicit Statement Caching for a Particular Statement

With implicit Statement caching enabled for a connection, by default, all callable and prepared statements of that connection are automatically cached. To prevent a particular callable or prepared statement from being implicitly cached, use the `setDisableStmtCaching` method of the statement object. You can manage cache space by calling the `setDisableStmtCaching` method on any infrequently used statement.

The following code disables implicit Statement caching for `pstmt`:

```
PreparedStatement pstmt = conn.prepareStatement("SELECT 1 from DUAL");
((OraclePreparedStatement)pstmt).setDisableStmtCaching(true);
pstmt.close ();
```

Note:

If you are using JSE 6, then you can disable Statement caching by using the standard JDBC 4.0 method `setPoolable`:

```
PreparedStatement.setPoolable(false);
```

Use the following to check whether the `Statement` object is poolable:

```
Statement.isPoolable();
```

Implicitly Caching a Statement

To cache an allocated statement, call the `close` method of the statement object. When you call the `close` method on an `OraclePreparedStatement` or `OracleCallableStatement` object, the JDBC driver automatically puts this statement in cache, unless you have disabled caching for this statement.

The following code caches the `pstmt` statement:

```
pstmt.close ();
```

Retrieving an Implicitly Cached Statement

To retrieve an implicitly cached statement, call either the `prepareStatement` or `prepareCall` method, depending on the statement type.

The following code retrieves `pstmt` from cache using the `prepareStatement` method:

```
pstmt = conn.prepareStatement ("UPDATE emp SET ename = ? WHERE rowid = ?");
```

22.2.3.1 Methods Used in Statement Allocation and Implicit Statement Caching

Table 22-2 describes the methods used to allocate statements and retrieve implicitly cached statements.

Table 22-2 Methods Used in Statement Allocation and Implicit Statement Caching

Method	Functionality for Implicit Statement Caching
`prepareStatement`	Performs a cache search that either finds and returns the desired cached `OraclePreparedStatement` object or allocates a new `OraclePreparedStatement` object if a match is not found
`prepareCall`	Performs a cache search that either finds and returns the desired cached `OracleCallableStatement` object or allocates a new `OracleCallableStatement` object if a match is not found

Example 22-1 provides a sample code that shows how to enable implicit statement caching.

Example 22-1 Using Implicit Statement Cache

```
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Properties;
import javax.sql.DataSource;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.pool.OracleDataSource;
public class TestJdbc
{
  /**
   * Get a Connection, prepare a statement, execute a query, fetch the results, close
the connection.
   * @param ods the DataSource used to get the connection.
   */
  private static void doSQL( DataSource ods ) throws SQLException
  {
    final String SQL = "select username from all_users";
    OracleConnection  conn = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    try
    {
      conn = (OracleConnection) ods.getConnection();
      System.out.println( "Connection:" + conn );
      System.out.println( "Connection getImplicitCachingEnabled:" +
conn.getImplicitCachingEnabled() );
      System.out.println( "Connection getStatementCacheSize:" +
conn.getStatementCacheSize() );
      ps = conn.prepareStatement( SQL );
      System.out.println( "PreparedStatement:" + ps );
      rs = ps.executeQuery();
      while ( rs.next() )
      {
        String owner = rs.getString( 1 );
        System.out.println( owner );
      }
    }
    finally
    {
      if ( rs != null )
      {
        rs.close();
      }
      if ( ps != null )
      {
        ps.close();
      conn.close();
    }
  }
  }
  public static void main( String[] args )
  {
    try
    {
      OracleDataSource ods =  new OracleDataSource();
      ods.setDriverType( "thin" );
      ods.setServerName( "localhost" );
      ods.setPortNumber( 5221 );
      ods.setServiceName( "orcl" );
      ods.setUser( "HR" );
      ods.setPassword( "hr" );
      ods.setConnectionCachingEnabled( true );
      ods.setImplicitCachingEnabled( true );
      Properties cacheProps = new Properties();
      cacheProps.put( "InitialLimit", "1" );
      cacheProps.put( "MinLimit", "1" );
      cacheProps.put( "MaxLimit", "5" );
      cacheProps.put( "MaxStatementsLimit", "50" );
      ods.setConnectionCacheProperties( cacheProps );
      System.out.println( "DataSource getImplicitCachingEnabled: " +
ods.getImplicitCachingEnabled() );
      for ( int i = 0; i < 5; i++ )
      {
        doSQL( ods );
      }
    }
    catch ( Exception ex )
    {
      ex.printStackTrace();
    }
  }
}
```

22.2.4 About Using Explicit Statement Caching

A prepared or callable statement can be explicitly cached when you enable explicit Statement caching. Explicit Statement caching includes the following steps:

1. Enable explicit Statement caching.
2. Allocate a statement using one of the standard methods.
3. Explicitly cache the statement by closing it with a key, using the `closeWithKey` method.
4. Retrieve the explicitly cached statement by calling the appropriate Oracle WithKey method, specifying the appropriate key.
5. Re-cache an open, explicitly cached statement by closing it again with the `closeWithKey` method. Each time a cached statement is closed, it is re-cached with its key.

Allocating a Statement for Explicit Caching

To allocate a statement for explicit Statement caching, use either the `createStatement`, `prepareStatement`, or `prepareCall` method as you would typically.

The following code allocates a new statement object called `pstmt`:

```
PreparedStatement pstmt =
   conn.prepareStatement ("UPDATE emp SET ename = ? WHERE rowid = ?");
```

Explicitly Caching a Statement

To explicitly cache an allocated statement, call the `closeWithKey` method of the statement object, specifying a key. The key is an arbitrary Java `String` that you provide. The `closeWithKey` method caches a statement as is. This means the data, state, and metadata are retained and not cleared.

The following code caches the `pstmt` statement with the key `"mykey"`:

```
((OraclePreparedStatement)pstmt).closeWithKey ("mykey");
```

Retrieving an Explicitly Cached Statement

To recall an explicitly cached statement, call either the `getStatementWithKey` or `getCallWithKey` methods depending on the statement type.

If you retrieve a statement with a specified key, then the JDBC driver searches the cache for the statement, based on the specified key. If a match is found, then the matching statement is returned along with its state, data, and metadata. This information is as it was when the statement was last closed. If a match is not found, then the JDBC driver returns `null`.

The following code recalls `pstmt` from cache using the `"mykey"` key with the `getStatementWithKey` method. Recall that the `pstmt` statement object was cached with the `"mykey"` key.

```
pstmt = ((OracleConnection)conn).getStatementWithKey ("mykey");
```

If you call the `creationState` method on the `pstmt` statement object, then the method returns `EXPLICIT`.

Note:

When you retrieve an explicitly cached statement, ensure that you use the method that is appropriate for your statement type when specifying the key. For example, if you used the `prepareStatement` method to allocate a statement, then use the `getStatementWithKey` method to retrieve that statement from cache. The JDBC driver does *not* verify the type of statement it is returning.

22.2.4.1 Methods Used to Retrieve Explicitly Cached Statements

Table 22-3 describes the methods used to retrieve explicitly cached statements.

Table 22-3 Methods Used to Retrieve Explicitly Cached Statements

Method	Functionality for Explicit Statement Caching
`getStatementWithKey`	Specifies the key needed to retrieve a prepared statement from cache
`getCallWithKey`	Specifies the key needed to retrieve a callable statement from cache

22.3 About Reusing Statements Objects

The JDBC 3.0 specification introduces the feature of statement pooling that enables an application to reuse a `PreparedStatement` object in the same way as it uses a `Connection` object. The `PreparedStatement` objects can be reused by multiple logical connections in a transparent manner.

This section covers the following topics:

- About Using a Pooled Statement
- About Closing a Pooled Statement

> **Note:**
>
> The Oracle JDBC Drivers use implicit statement caching to support statement pooling.

22.3.1 About Using a Pooled Statement

An application can find out whether a data source supports statement pooling by calling the `isPoolable` method from the `Statement` interface. If the return value is `true`, then the application knows that the `PreparedStatement` object is being pooled. The application can also request a statement to be pooled or not pooled by using the `setPoolable` method from the `Statement` interface.

Reusing of pooled statement should be completely transparent to the application, that is, the application code should remain the same whether a `PreparedStatement` object participates in statement pooling or not. If an application closes a `PreparedStatement` object, it must still call `Connection.prepareStatement` method in order to reuse it.

> **Note:**
>
> An application has no direct control over how statements are pooled. A pool of statements is associated with a `PooledConnection` object, whose behavior is determined by the properties of the `ConnectionPoolDataSource` object that produced it.

22.3.2 About Closing a Pooled Statement

An application closes a pooled statement exactly the same way it closes a nonpooled statement. Once a statement is closed, whether is it pooled or nonpooled, it is no longer available for use by the application and an attempt to reuse it causes an exception to be thrown. The only difference visible is that an application cannot directly close a physical statement that is being pooled. This is done by the pool manager. The method `PooledConnection.closeAll` closes all of the statements open on a given physical connection, which releases the resources associated with those statements.

The following methods can close a pooled statement:

- `close`

  This `java.sql.Statement` interface method is called by an application. If the statement is being pooled, then it closes the logical statement used by the application but does not close the physical statement being pooled.
- `close`

  This `java.sql.Connection` interface method is called by an application. This method acts differently depending upon whether the connection using the statement is being pooled or not:
  - Nonpooled connection

    This method closes the physical connection and all statements created by that connection. This is necessary because the garbage collection mechanism is unable to detect when externally managed resources can be released.
  - Pooled connection

    This method closes the logical connection and the logical statements it returned, but leaves open the underlying `PooledConnection` object and any associated pooled statements
- `PooledConnection.closeAll`

  This method is called by the connection pool manager to close all of the physical statements being pooled by the `PooledConnection` object

22.4 About Result Set Caching

Your applications sometime send repetitive queries to the database. To improve the response time of repetitive queries, results of queries, query fragments, and PL/SQL functions can be cached in memory. A result cache stores the results of queries shared across all sessions. When these queries are executed repeatedly, the results are retrieved directly from the cache memory.

Note:

If a result set is very large, then it may not be cached due to size restrictions.

You must annotate a query or query fragment with a result cache hint to indicate that results are to be stored in the query result cache.

The query result set can be cached in the following ways:

- Server-Side Result Set Cache
- Client-Side Result Set Cache

Note:

- The server-side and client result set caches are most useful for read-only or read-mostly data. They may reduce performance for queries with highly dynamic results.
- Both server-side and client result set caches use memory. So, caching very large result sets can cause performance problems.

22.4.1 Server-Side Result Set Cache

Support for server-side Result Set caching has been introduced for both JDBC Thin and JDBC Oracle Call Interface (OCI) drivers since Oracle Database 11*g* Release 1. The server-side result cache is used to cache the results of the current queries, query fragments, and PL/SQL functions in memory and then to use the cached results in future executions of the query, query fragment, or PL/SQL function. The cached results reside in the result cache memory portion of the SGA. A cached result is automatically invalidated whenever a database object used in its creation is successfully modified. The server-side caching can be of the following two types:

- SQL Query Result Cache
- PL/SQL Function Result Cache

See Also:

- *Oracle Database Performance Tuning Guide* for more information about SQL Query Result Cache
- *Oracle Database PL/SQL Language Reference* for more information about PL/SQL Function Result Cache

22.4.2 Client-Side Result Set Cache

Client-side result set cache feature enables client-side caching of SQL query result sets in client memory. In this way, the applications can use client memory to take advantage of the client-side result set cache to improve response times of repetitive queries.

This section covers the following topics:

- Enabling the Client-Side Result Set Cache
- Benefits of Client-Side Result Set Cache
- Usage Guidelines in JDBC

22.4.2.1 Enabling the Client-Side Result Set Cache

Oracle Database Release 23ai supports client-side result set cache in the JDBC thin driver. You can use the new `oracle.jdbc.enableQueryResultCache` connection property for enabling this feature. The default value of this property is `true`, which means that this feature is enabled by default. You can disable this feature by setting the property to `false`.

See Also:

Oracle Call Interface Programmer's Guide

For using this feature, you must set the following database initialization parameters in the following way:

```
CLIENT_RESULT_CACHE_SIZE=100M
CLIENT_RESULT_CACHE_LAG=1000
```

This value of the `CLIENT_RESULT_CACHE_SIZE` parameter controls how much memory the thin driver can use for its cache.

A read-only or read-mostly table can then be annoted and its data can be cached on the driver. For example, `RESULT_CACHE(MODE FORCE)`.

You can also use a SQL hint `/*+RESULT_CACHE */` for identifying queries that are eligible for caching.

See Also:

Oracle Database JDBC Java API Reference

22.4.2.2 Benefits of Client-Side Result Set Cache

The benefits of the client-side result set cache are the following:

- The client-side result set cache is completely transparent to the applications and its cache of result set data is kept consistent with any session or database changes that affect its result set.
- Table annotation makes client-side result set work transparently to the JDBC applications. Otherwise, you must use a hint to enable it. The cache hit avoids the execution of the query and roundtrip to the server to get the result sets. This can result in huge performance savings for server resources, for example, server CPU and server I/O.

  See Also:

  Table Annotations and SQL Hints

- The result cache on the client is per-process, so multiple client sessions can simultaneously use matching cached result sets.
- The result cache on the client minimizes the need for each application to have its own custom result set cache.
- The result cache on the client uses the client memory that is cheaper than server memory.

22.4.2.3 Usage Guidelines in JDBC

You can enable result set caching in the following three ways:

- The RESULT_CACHE_MODE Parameter
- The RESULT_CACHE_INTEGRITY Parameter
- Table Annotations
- SQL Hints

Note:

- You must use JDBC statement caching or cache statements at the application level when using the client-side result set cache.
- The SQL hints take precedence over the session parameter `RESULT_CACHE_MODE` and table annotations. The table annotation `FORCE` takes precedence over session parameter.
- If you set the value of the `RESULT_CACHE_INTEGRITY` parameter to `TRUSTED`, then the Database honors the setting of the `RESULT_CACHE_MODE` parameter and the specified hints, and considers queries using possibly non-deterministic constructs as candidates for result caching.

Related Topics

- Statement and Result Set Caching

22.4.2.3.1 The RESULT_CACHE_MODE Parameter

You can use the `RESULT_CACHE_MODE` parameter to decide the result cache mode across tables in your queries.

Use this clause with the `ALTER SESSION` and `ALTER SYSTEM` statements, or inside the server parameter file (`init.ora`) to determine result caching. You can set the `RESULT_CACHE_MODE` parameter to control whether the SQL query result cache is used for all queries, or only for the queries that are annotated with the result cache hint using SQL hints or table annotations.

See Also:

RESULT_CACHE_MODE

22.4.2.3.2 The RESULT_CACHE_INTEGRITY Parameter

You can use the `RESULT_CACHE_INTEGRITY` parameter to specify whether the result cache must consider queries using non-deterministic constructs, such as PL/SQL functions, because those are not declared as deterministic. You can use this parameter to enforce explicit declaration of objects as deterministic, prior to their consideration in the result cache.

See Also:

- Setting Result Cache Integrity
- RESULT_CACHE_INTEGRITY

22.4.2.3.3 Table Annotations

You can use table annotations to enable result caching without making changes to the code. The `ALTER TABLE` and `CREATE TABLE` statements enable you to annotate tables with result cache mode. The syntax is:

```
CREATE|ALTER TABLE [<schema>.]<table> ... [RESULT_CACHE (MODE {FORCE|DEFAULT})]
```

Following example shows how to use table annotations with `CREATE TABLE` statements:

```
CREATE TABLE foo (a NUMBER, b VARCHAR2(20)) RESULT_CACHE (MODE FORCE);
```

Following example shows how to use table annotations with `ALTER TABLE` statements:

```
ALTER TABLE foo RESULT_CACHE (MODE DEFAULT);
```

22.4.2.3.4 SQL Hints

You can use SQL hints to specify the queries to be cached by annotating the queries with a `/*+ result_cache */` or /*+ no_result_cache */ hint.

For example, look at the following code snippet:

```
String  query  = "select /*+ result_cache */ * from employees where employee_id < : 1";
   ((oracle.jdbc.OracleConnection)conn).setImplicitCachingEnabled(true);
   ((oracle.jdbc.OracleConnection)conn).setStatementCacheSize(10);
   PreparedStatement  pstmt;
   ResultSet rs;

    for (int j = 0 ; j < 10 ; j++ )
    {
       pstmt  = conn.prepareStatement (query);
       pstmt.setInt(1,7500);
       rs  = pstmt.executeQuery();
         while (rs.next( ) )
        {     // see the values  }
          rs.close;
          pstmt.close( ) ;
        }
    }
```

In the preceding example, the client result cache hint `/*+ result_cache */` is annotated to the actual query, that is, `select * from employees where employee_id < : 1`. So, the first execution of the query goes to the database and the result set is cached for the remaining nine executions of the query. This improves the performance of your application significantly. This is primarily useful for read-only data.

Following are some more examples of SQL hints. All the following examples assume that the `departments` table is annotated for result caching by using the following command:

```
ALTER TABLE departments result_cache (MODE FORCE);
```

Examples

- SELECT * FROM employees

  The result set will not be cached.
- SELECT * FROM departments

  The result set will be cached.

- SELECT /*+ result_cache */ employee_id FROM employees

  The result set will be cached.
- SELECT /*+ no_result_cache */ department_id FROM departments

  The result set will not be cached.
- SELECT /*+ result_cache */ * FROM departments

  The result set will be cached though query hint is not necessary.
- SELECT e.first_name FROM employees e, departments d WHERE e.department_id = d.department_id

  The result set will not be cached because neither is a query hint available nor are all the tables annotated as `FORCE`.

Note:

For information about usage guidelines, Client cache consistency, Deployment Time settings, Client cache Statistics, Validation of client result cache, and OCI Client Result Cache and Server Result Cache, refer to the *Oracle Call Interface Programmer's Guide*.