31 Continuous Query Notification

This chapter describes how the Continuous Query Notification feature works.

This chapter describes the following topics:

31.1 Overview of Continuous Query Notification

Generally, a middle-tier data cache duplicates some data from the back-end database server, so that it can avoid redundant queries to the database. However, this is efficient only when the data rarely changes in the database. The data cache has to be updated or invalidated when the data changes in the database.

Starting from 11*g* Release 1, Oracle JDBC drivers provide support for the Continuous Query Notification feature of Oracle Database. Using this functionality, multitier systems can take advantage of the Continuous Query Notification feature to maintain a data cache as up-to-date as possible, by receiving invalidation events from the JDBC drivers.

The JDBC drivers can register SQL queries with the database and receive notifications in response to the following:

- DML or DDL changes on the objects associated with the queries
- DML or DDL changes that affect the result set

The notifications are published when the DML or DDL transaction commits (changes made in a local transaction do not generate any event until they are committed).

To use Oracle JDBC driver support for Continuous Query Notification, perform the following:

1. Registration: You first need to create a registration.
2. Query association: After you have created a registration, you can associate SQL queries with it. These queries are part of the registration.
3. Notification: Notifications are created in response to changes in tables or result set. Oracle database communicates these notifications to the JDBC drivers through a dedicated network connection and JDBC drivers convert these notifications to Java events.

Also, you need to grant the `CHANGE NOTIFICATION` privilege to the user. For example, if you connect to the database using the `HR` user name, then you need to run the following command in the database:

```
grant change notification to HR;
```

31.2 Overview of Client Initiated Continuous Query Notification

Starting from Oracle Database Release 19c, the JDBC Thin driver supports the Client Initiated Continuous Query Notification feature. In this case, the client application initiates a connection to the database server for receiving notifications.

A client application first initiates a new database connection before creating a Continuous Query Notification registration. When the application creates a registration, the JDBC driver internally starts a new thread and creates a new connection with the database server. The database server then uses this new connection to send change notifications to the client.

By default, this feature is disabled for an on-premise database. You must set the `OracleConnection.DCN_CLIENT_INIT_CONNECTION` to true for enabling this feature.

See Also:

Continuous Query Notification Registration Options

31.3 Creating a Registration

Creating a CQN registration is a one-time process and is done outside the currently used transaction. The API for creating a registration in the server is executed in its own transaction and is committed immediately.

You need a JDBC connection to create a registration. However, the registration is not attached to the connection. You can close the connection after creating a registration, and the registration survives. In an Oracle RAC environment, a registration is a persistent entity that exists on all nodes. The registration exists in the Database. So, even if a node goes down, the registration continues to exist and is notified when the tables change.

There are two ways to create a registration:

- The JDBC-style of registration: Use the JDBC driver to create a registration on the server. The JDBC driver launches a new thread that listens to notifications from the server (through a dedicated channel) and converts these notification messages into Java events. The driver then notifies all the listeners registered with this registration.
- The PL/SQL-style of registration: If you want a PL/SQL stored procedure to handle the notifications, then create a PL/SQL-style registration. As in the JDBC-style of registration, the JDBC drivers enable you to attach statements (queries) to this registration. However the JDBC drivers do not get notifications from the server because the notifications are handled by the PL/SQL stored procedure.

Note:

This approach is useful only for nonmultithreaded languages, such as PHP.

There is no way to remove one particular object (table) from an existing registration. A workaround would be to either create a new registration without this object or ignore the events that are related to this object.

You can use the `registerDatabaseChangeNotification` method of the `oracle.jdbc.OracleConnection` interface to create a JDBC-style of registration. You can set certain registration options through the `options` parameter of this method. The "Continuous Query Notification Registration Options" table in the following section lists some of the registration options that can be set. To set these options, use the `java.util.Properties` object. These options are defined in the `oracle.jdbc.OracleConnection` interface. The registration options have a direct impact on the notification events that the JDBC drivers will create. The example (at the end of this chapter) illustrates how to use the Continuous Query Notification feature.

The `registerDatabaseChangeNotification` method creates a new database change registration in the database server with the given options. It returns a `DatabaseChangeRegistration` object, which can then be used to associate a statement with this registration. It also opens a listener socket that will be used by the database to send notifications.

Note:

If a listener socket (created by a different registration) exists, then this socket is used by the new database change registration as well.

31.3.1 Continuous Query Notification Registration Options

The following table lists the Continuous Query Notification Registration Options:

Table 31-1 Continuous Query Notification Registration Options

Option	Description
`DCN_IGNORE_DELETEOP`	If set to `true`, `DELETE` operations will not generate any database change event.
`DCN_IGNORE_INSERTOP`	If set to `true`, `INSERT` operations will not generate any database change event.
`DCN_IGNORE_UPDATEOP`	If set to `true`, `UPDATE` operations will not generate any database change event.
`DCN_NOTIFY_CHANGELAG`	Specifies the number of transactions by which the client is willing to lag behind. **Note:** If this option is set to any value other than `0`, then `ROWID` level granularity of information will not be available in the events, even if the `DCN_NOTIFY_ROWIDS` option is set to `true`.
`DCN_NOTIFY_ROWIDS`	Database change events will include row-level details, such as operation type and `ROWID`.
`DCN_QUERY_CHANGE_NOTIFICATION`	Activates query change notification instead of object change notification. **Note:** This option is available only when running against an 11.0 database.
`DCN_CLIENT_INIT_CONNECTION`	Specifies Client Initiated Continuous Query Notification, where the client application initiates a database connection, which the server uses to send change notifications to the client.
`NTF_LOCAL_HOST`	Specifies the IP address of the computer that will receive the notifications from the server.
`NTF_LOCAL_TCP_PORT`	Specifies the TCP port that the driver should use for the listener socket.
`NTF_QOS_PURGE_ON_NTFN`	Specifies if the registration should be expunged on the first notification event.
`NTF_QOS_RELIABLE`	Specifies whether or not to make the notifications persistent, which comes at a performance cost.

Table 31-1 (Cont.) Continuous Query Notification Registration Options

Option	Description
`NTF_TIMEOUT`	Specifies the time in seconds after which the registration will be automatically expunged by the database.

If there exists a registration, then you can also use the `getDatabaseChangeRegistration` method to map the existing registration with a new `DatabaseChangeRegistration` object. This method is particularly useful if you have created a registration using PL/SQL and want to associate a statement with it.

31.4 Associating a Query with a Registration

After you have created a registration or mapped to an existing registration, you can associate a query with it. Like creating a registration, associating a query with a registration is a one-time process and is done outside of the currently used registration. The query will be associated even if the local transaction is rolled back.

You can associate a query with registration using the `setDatabaseChangeRegistration` method defined in the `OracleStatement` class. This method takes a `DatabaseChangeRegistration` object as parameter. The following code snippet illustrates how to associate a query with a registration:

```
...
// conn is an OracleConnection object.
// prop is a Properties object containing the registration options.
DatabaseChangeRegistration dcr = conn.registerDatabaseChangeNotifictaion(prop);
...
Statement stmt = conn.createStatement();
// associating the query with the registration
((OracleStatement)stmt).setDatabaseChangeRegistration(dcr);
// any query that will be executed with the 'stmt' object will be associated with
// the registration 'dcr' until 'stmt' is closed or
// '((OracleStatement)stmt).setDatabaseChangeRegistration(null);' is executed.
...
```

31.5 Notifying Database Change Events

To receive Continuous Query Notifications, attach a listener to the registration. When a database change event occurs, the database server notifies the JDBC driver. The driver then constructs a new Java event, identifies the registration to be notified, and notifies the listeners attached to the registration. The event contains the object ID of the database object that has changed and the type of operation that caused the change. Depending on the registration options, the event may also contain row-level detail information. The listener code can then use the event to make decisions about the data cache.

Note:

The listener code must not slow down the JDBC notification mechanism. If the code is time-consuming, for example, if it refreshes the data cache by querying the database, then it needs to be executed within its own thread.

You can attach a listener to a registration using the `addListener` method. The following code snippet illustrates how to attach a listener to a registration:

```
...
// conn is an OracleConnection object.
// prop is a Properties object containing the registration options.
DatabaseChangeRegistration dcr = conn.registerDatabaseChangeNotifictaion(prop);
...
// Attach the listener to the registration.
// Note: DCNListener is a custom listener and not a predefined or standard
// lsiener
DCNListener list = new DCNListener();
dcr.addListener(list);
...
```

31.6 Deleting a Registration

You need to explicitly unregister a registration to delete it from the server and release the resources in the driver.

You can unregister a registration using a connection different from one that was used for creating it. To unregister a registration, you can use the `unregisterDatabaseChangeNotification` method defined in `oracle.jdbc.OracleConnection`.

You must pass the `DatabaseChangeRegistration` object as a parameter to this method. This method deletes the registration from the server and the driver and closes the listener socket.

If the registration was created outside of JDBC, say using PL/SQL, then you must pass the registration ID instead of the `DatabaseChangeRegistration` object. The method will delete the registration from the server, however, it does not free any resources in the driver.

The example in this section demonstrates how to use the Continuous Query Notification feature. In this example, the `HR` user is connecting to the database. Therefore in the database you need to grant the following privilege to the user:

```
grant change notification to HR;
```

This code will also work with Oracle Database 10*g* Release 2 (10.2). This code uses table registration. That is, when you register a `SELECT` query, what you register is the name of the tables involved and not the query itself. In other words, you might select one single row of a table and if another row is updated, you will be notified although the result of your query has not changed.

In this example, if you leave the registration open instead of closing it, then the Continuous Query Notification thread continues to run. Now if you run a DML query that changes the `HR.DEPARTMENTS` table and commit it, say from SQL*Plus, then the Java program prints the notification.

Example 31-1 Continuous Query Notification

```
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Properties;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.OracleDriver;
import oracle.jdbc.OracleStatement;
import oracle.jdbc.dcn.DatabaseChangeEvent;
import oracle.jdbc.dcn.DatabaseChangeListener;
import oracle.jdbc.dcn.DatabaseChangeRegistration;
```

```
public class DBChangeNotification
{
  static final String USERNAME= "HR";
  static final String PASSWORD= "hr";
  static String URL;

  public static void main(String[] argv)
  {
    if(argv.length < 1)
    {
      System.out.println("Error: You need to provide the URL in the first argument.");
      System.out.println("  For example: > java -classpath .:ojdbc11.jar 
DBChangeNotification \"jdbc:oracle:thin:
@(DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=yourhost.yourdomain.com)(PORT=5221))
(CONNECT_DATA=
(SERVICE_NAME=orcl)))\"");

      System.exit(1);
    }
    URL = argv[0];
    DBChangeNotification demo = new DBChangeNotification();
    try
    {
      demo.run();
    }
    catch(SQLException mainSQLException )
    {
      mainSQLException.printStackTrace();
    }
  }

  void run() throws SQLException
  {
    OracleConnection conn = connect();

    // first step: create a registration on the server:
    Properties prop = new Properties();

    // if connected through the VPN, you need to provide the TCP address of the client.
    // For example:
    // prop.setProperty(OracleConnection.NTF_LOCAL_HOST,"14.14.13.12");

    // Ask the server to send the ROWIDs as part of the DCN events (small performance
    // cost):
    prop.setProperty(OracleConnection.DCN_NOTIFY_ROWIDS,"true");
//
//Set the DCN_QUERY_CHANGE_NOTIFICATION option for query registration with finer 
granularity.
 prop.setProperty(OracleConnection.DCN_QUERY_CHANGE_NOTIFICATION,"true");

    // The following operation does a roundtrip to the database to create a new
    // registration for DCN. It sends the client address (ip address and port) that
    // the server will use to connect to the client and send the notification
    // when necessary. Note that for now the registration is empty (we haven't registered
    // any table). This also opens a new thread in the drivers. This thread will be
    // dedicated to DCN (accept connection to the server and dispatch the events to
    // the listeners).
    DatabaseChangeRegistration dcr = conn.registerDatabaseChangeNotification(prop);

    try
    {
      // add the listenerr:
      DCNDemoListener list = new DCNDemoListener(this);
      dcr.addListener(list);

      // second step: add objects in the registration:
      Statement stmt = conn.createStatement();
      // associate the statement with the registration:
      ((OracleStatement)stmt).setDatabaseChangeRegistration(dcr);
      ResultSet rs = stmt.executeQuery("select * from dept where deptno='45'");
      while (rs.next())
      {}
      String[] tableNames = dcr.getTables();
      for(int i=0;i<tableNames.length;i++)
        System.out.println(tableNames[i]+" is part of the registration.");
      rs.close();
      stmt.close();
    }
    catch(SQLException ex)
    {
      // if an exception occurs, we need to close the registration in order
      // to interrupt the thread otherwise it will be hanging around.
      if(conn != null)
        conn.unregisterDatabaseChangeNotification(dcr);
      throw ex;
    }
    finally
    {
      try
      {
        // Note that we close the connection!
        conn.close();
      }
      catch(Exception innerex){ innerex.printStackTrace(); }
    }

    synchronized( this )
    {
      // The following code modifies the dept table and commits:
      try
      {
        OracleConnection conn2 = connect();
        conn2.setAutoCommit(false);
        Statement stmt2 = conn2.createStatement();
        stmt2.executeUpdate("insert into dept (deptno,dname) values ('45','cool dept')",
Statement.RETURN_GENERATED_KEYS);
        ResultSet autoGeneratedKey = stmt2.getGeneratedKeys();
        if(autoGeneratedKey.next())
          System.out.println("inserted one row with
ROWID="+autoGeneratedKey.getString(1));
        stmt2.executeUpdate("insert into dept (deptno,dname) values ('50','fun dept')",
Statement.RETURN_GENERATED_KEYS);
        autoGeneratedKey = stmt2.getGeneratedKeys();
        if(autoGeneratedKey.next())
          System.out.println("inserted one row with
ROWID="+autoGeneratedKey.getString(1));
        stmt2.close();
        conn2.commit();
        conn2.close();
      }
      catch(SQLException ex) { ex.printStackTrace(); }

      // wait until we get the event
```

```
        try{ this.wait();} catch( InterruptedException ie ) {}
      }

      // At the end: close the registration (comment out these 3 lines in order
      // to leave the registration open).
      OracleConnection conn3 = connect();
      conn3.unregisterDatabaseChangeNotification(dcr);
      conn3.close();
    }

    /**
     * Creates a connection the database.
     */
    OracleConnection connect() throws SQLException
    {
      OracleDriver dr = new OracleDriver();
      Properties prop = new Properties();
      prop.setProperty("user",DBChangeNotification.USERNAME);
      prop.setProperty("password",DBChangeNotification.PASSWORD);
      return (OracleConnection)dr.connect(DBChangeNotification.URL,prop);
    }
  }
  /**
   * DCN listener: it prints out the event details in stdout.
   */
  class DCNDemoListener implements DatabaseChangeListener
  {
    DBChangeNotification demo;
    DCNDemoListener(DBChangeNotification dem)
    {
      demo = dem;
    }
    public void onDatabaseChangeNotification(DatabaseChangeEvent e)
    {
      Thread t = Thread.currentThread();
      System.out.println("DCNDemoListener: got an event ("+this+" running on thread 
  "+t+")");
      System.out.println(e.toString());
      synchronized( demo ){ demo.notify();}
    }
  }
```