27
OCI Connection Pooling

The Java Database Connectivity (JDBC) Oracle Call Interface (OCI) driver connection pooling functionality is part of the JDBC client. This functionality is provided by the `OracleOCIConnectionPool` class.

A JDBC application can have multiple pools at the same time. Multiple pools can correspond to multiple application servers or pools to different data sources. The connection pooling provided by the JDBC OCI driver enables applications to have multiple logical connections, all using a small set of physical connections. Each call on a logical connection gets routed on to the physical connection that is available at the time of call.

This chapter contains the following sections:

> **Note:**
>
> Use OCI connection pooling if you need session multiplexing. Otherwise, Oracle recommends using Universal Connection Pool.

27.1 Background of OCI Driver Connection Pooling

The Oracle JDBC OCI driver provides several transaction monitor capabilities, such as the fine-grained management of Oracle sessions and connections. It is possible for a high-end application server or transaction monitor to multiplex several sessions over fewer physical connections on a call-level basis, thereby achieving a high degree of scalability by pooling of connections and back-end Oracle server processes.

The connection pooling provided by the `OracleOCIConnectionPool` interface simplifies the session/connection separation interface hiding the management of the physical connection pool. The Oracle sessions are the `OracleOCIConnection` objects obtained from `OracleOCIConnectionPool`. The connection pool itself is usually configured with a much smaller shared pool of physical connections, translating to a back-end server pool containing an identical number of dedicated server processes. Note that many more Oracle sessions can be multiplexed over this pool of fewer shared connections and back-end Oracle processes.

27.2 Comparison Between OCI Driver Connection Pooling and Shared Servers

In some ways, what OCI driver connection pooling offers on the middle tier is similar to what shared server processes offer on the back end. OCI driver connection pooling makes a dedicated server instance behaves as a shared instance by managing the session multiplexing logic on the middle tier. Therefore, the pooling of dedicated server processes and incoming connections into the dedicated server processes is controlled by the OCI connection pool on the middle tier.

The main difference between OCI connection pooling and shared servers is that in the case of shared servers, the connection from the client is typically to a dispatcher in the database instance. The dispatcher is responsible for directing the client request to an appropriate shared server. On the other hand, the physical connection from the OCI connection pool is established directly from the middle tier to the Oracle dedicated server process in the back-end server pool.

Note that OCI connection pool is mainly beneficial only if the middle tier is multithreaded. Each thread could maintain a session to the database. The actual connections to the database are maintained by `OracleOCIConnectionPool`, and these connections, including the pool of dedicated database server processes, are shared among all the threads in the middle tier.

27.3 About Defining an OCI Connection Pool

This section describes the following concepts:

- Overview of Creating an OCI Connection Pool
- Importing the oracle.jdbc.pool and oracle.jdbc.oci Packages
- Creating an OCI Connection Pool
- Setting the OCI Connection Pool Parameters
- Checking the OCI Connection Pool Status

27.3.1 Overview of Creating an OCI Connection Pool

An OCI connection pool is created at the beginning of the application. Creating connections from a pool is quite similar to creating connections using the `OracleDataSource` class.

The `oracle.jdbc.pool.OracleOCIConnectionPool` class, which extends the `OracleDataSource` class, is used to create OCI connection pools. From an `OracleOCIConnectionPool` instance, you can obtain logical connection objects. These connection objects are of the `OracleOCIConnection` class type. This class implements the `OracleConnection` interface. The `Statement` objects you create from the `OracleOCIConnection` instance have the same fields and methods as `OracleStatement` objects you create from `OracleConnection` instances.

The following code shows header information for the `OracleOCIConnectionPool` class:

```
/*
   * @param us  ConnectionPool user-id.
   * @param p   ConnectionPool password
   * @param name  logical name of the pool. This needs to be one in the
   *                  tnsnames.ora configuration file.
```

```
     @param config (optional)  Properties of the pool, if the default does not
                suffice. Default connection configuration is min =1, max=1,
                incr=0
                  Please refer setPoolConfig for property names.

                 Since this is optional, pass null if the default configuration
                 suffices.

   * @return
   *
   * Notes: Choose a userid and password that can act as proxy for the users
   *        in the getProxyConnection() method.

            If config is null, then the following default values will take
            effect
            CONNPOOL_MIN_LIMIT = 1
            CONNPOOL_MAX_LIMIT = 1
            CONNPOOL_INCREMENT = 0

*/

public synchronized OracleOCIConnectionPool
  (String       user,    String   password,  String name, Properties config)
  throws SQLException

/*
 * This will use the user-id, password and connection pool name values set
   LATER using the methods setUser, setPassword, setConnectionPoolName.

 * @return
 *
 * Notes:

     No OracleOCIConnection objects can be created on
     this class unless the methods setUser, setPassword, setPoolConfig
     are invoked.
     When invoking the setUser, setPassword later, choose a userid and
     password that can act as proxy for the users
 *   in the getProxyConnection() method.
 */
  public synchronized OracleOCIConnectionPool ()
    throws SQLException
```

27.3.2 Importing the oracle.jdbc.pool and oracle.jdbc.oci Packages

Before you create an OCI connection pool, import the following to have Oracle OCI connection pooling functionality:

```
import oracle.jdbc.pool.*;
import oracle.jdbc.oci.*;
```

27.3.3 Creating an OCI Connection Pool

The following code show how you create an instance of the `OracleOCIConnectionPool` class called `cpool`:

```
OracleOCIConnectionPool cpool = new OracleOCIConnectionPool
    ("HR", "hr", "jdbc:oracle:oci:@(description=(address=(host=
    localhost)(protocol=tcp)(port=5221))(connect_data=(INSTANCE_NAME=orcl)))",
    poolConfig);
```

`poolConfig` is a set of properties that specify the connection pool. If `poolConfig` is null, then the default values are used. For example, consider the following:

- `poolConfig.put (OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT, "4");`
- `poolConfig.put (OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT, "10");`
- `poolConfig.put (OracleOCIConnectionPool.CONNPOOL_INCREMENT, "2");`

As an alternative to the constructor call, you can create an instance of the `OracleOCIConnectionPool` class using individual methods to specify the user, password, and connection string.

```
OracleOCIConnectionPool cpool = new OracleOCIConnectionPool ( );
cpool.setUser("HR");
cpool.setPassword("hr");
cpool.setURL("jdbc:oracle:oci:@(description=(address=(host=
    localhost)(protocol=tcp)(port=5221))(connect_data=(INSTANCE_NAME=orcl)))");
cpool.setPoolConfig(poolConfig);  // In case you want to specify a different
                                  // configuration other than the default
                                  // values.
```

27.3.4 Setting the OCI Connection Pool Parameters

The connection pool configuration is determined by the following `OracleOCIConnectionPool` class attributes:

- `CONNPOOL_MIN_LIMIT`

  Specifies the minimum number of physical connections that can be maintained by the pool.

- `CONNPOOL_MAX_LIMIT`

  Specifies the maximum number of physical connections that can be maintained by the pool.

- `CONNPOOL_INCREMENT`

  Specifies the incremental number of physical connections to be opened when all the existing ones are busy and a call needs one more connection; the increment is done only when the total number of open physical connections is less than the maximum number that can be opened in that pool.

- `CONNPOOL_TIMEOUT`

  Specifies how much time must pass before an idle physical connection is disconnected; this does not affect a logical connection.

- `CONNPOOL_NOWAIT`

  Specifies, if enabled, that an error is returned if a call needs a physical connection while the maximum number of connections in the pool are busy. If disabled, a call waits until a connection is available. Once this attribute is set to `true`, it cannot be reset to `false`.

You can configure all of these attributes dynamically. Therefore, an application has the flexibility of reading the current load, that is number of open connections and number of busy connections, and adjusting these attributes appropriately, using the `setPoolConfig` method.

> **Note:**
>
> The default values for the `CONNPOOL_MIN_LIMIT`, `CONNPOOL_MAX_LIMIT`, and `CONNPOOL_INCREMENT` parameters are `1`, `1`, and `0`, respectively.

The `setPoolConfig` method is used to configure OCI connection pool properties. The following is a typical example of how the `OracleOCIConnectionPool` class attributes can be set:

```
...
java.util.Properties p  = new java.util.Properties( );
p.put (OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT, "1");
p.put (OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT, "5");
p.put (OracleOCIConnectionPool.CONNPOOL_INCREMENT, "2");
p.put (OracleOCIConnectionPool.CONNPOOL_TIMEOUT, "10");
p.put (OracleOCIConnectionPool.CONNPOOL_NOWAIT, "true");
cpool.setPoolConfig(p);
...
```

Observe the following rules when setting these attributes:

- `CONNPOOL_MIN_LIMIT`, `CONNPOOL_MAX_LIMIT`, and `CONNPOOL_INCREMENT` are mandatory.
- `CONNPOOL_MIN_LIMIT` must be a value greater than zero.
- `CONNPOOL_MAX_LIMIT` must be a value greater than or equal to `CONNPOOL_MIN_LIMIT` plus `CONNPOOL_INCREMENT`.
- `CONNPOOL_INCREMENT` must be a value greater than or equal to zero.
- `CONNPOOL_TIMEOUT` must be a value greater than zero.
- `CONNPOOL_NOWAIT` must be `true` or `false`.

> **See Also:**
>
> *Oracle Call Interface Programmer's Guide*

27.3.5 Checking the OCI Connection Pool Status

To check the status of the connection pool, use the following methods from the `OracleOCIConnectionPool` class:

- `int getMinLimit()`

  Retrieves the minimum number of physical connections that can be maintained by the pool.

- `int getMaxLimit()`

  Retrieves the maximum number of physical connections that can be maintained by the pool.

- `int getConnectionIncrement()`

  Retrieves the incremental number of physical connections to be opened when all the existing ones are busy and a call needs a connection.

- `int getTimeout()`

  Retrieves the specified time (in seconds) that a physical connection in a pool can remain idle before it is disconnected; the age of a connection is based on the Least Recently Used (LRU) algorithm.

- `String getNoWait()`

  Retrieves if the `NOWAIT` property is enabled. It returns a string of "`true`" or "`false`".

- `int getPoolSize()`

  Retrieves the number of physical connections that are open. This should be used only as an estimate and for statistical analysis.

- `int getActiveSize()`

  Retrieves the number of physical connections that are open and busy. This should be used only as an estimate and for statistical analysis.

- `boolean isPoolCreated()`

  Retrieves if the pool has been created. The pool is actually created when `OracleOCIConnection(user, password, url, poolConfig)` is called or when `setUser`, `setPassword`, and `setURL` has been done after calling `OracleOCIConnection()`.

27.4 About Connecting to an OCI Connection Pool

The `OracleOCIConnectionPool` class, through a `getConnection` method call, creates an instance of the `OracleOCIConnection` class. This instance represents a connection.

Because the `OracleOCIConnection` class extends `OracleConnection` class, it has the functionality of this class too. Close the `OracleOCIConnection` objects once the user session is over, otherwise, they are closed when the pool instance is closed.

There are two ways of calling `getConnection`:

- `OracleConnection getConnection()`

  If you do not supply the user name and password, then the default user name and password used for the creation of the connection pool are used while creating the connection objects.

- `OracleConnection getConnection(String user, String password)`

  If you this method, you will get a logical connection identified with the specified user name and password, which can be different from that used for pool creation.

The following code shows the signatures of the overloaded `getConnection` method:

```
public synchronized OracleConnection getConnection( )
                    throws SQLException

/*
 * For getting a connection to the database.
 *
 * @param us  Connection user-id
 * @param p   Connection password
 * @return     connection object
 */
public synchronized OracleConnection getConnection(String us, String p)
throws SQLException
```

As an enhancement to `OracleConnection`, the following new method is added into `OracleOCIConnection` as a way to change the password for the user:

```
void passwordChange (String user, String oldPassword, String newPassword)
```

27.5 Sample Code for OCI Connection Pooling

The following code illustrates the use of OCI connection pooling in a sample application:

```
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Properties;
import oracle.jdbc.OracleDriver;
import oracle.jdbc.pool.OracleOCIConnectionPool;

public class conPoolAppl extends Thread
{
  public static final String query = "SELECT object_name FROM all_objects WHERE rownum <
300";
  static public void main(String args[]) throws SQLException
  {
    int _maxCount = 10;
    Connection []conn = new Connection[_maxCount];
    try
    {
      String s = null;   //System.getProperty ("JDBC_URL");
      String url = "jdbc:oracle:oci8:@localhost";
      OracleOCIConnectionPool cpool = new OracleOCIConnectionPool("HR", "hr", url, null);

      // Print out the default configuration for the OracleOCIConnectionPool
      System.out.println ("-- The default configuration for the OracleOCIConnectionPool
--");
      displayPoolConfig(cpool);

      //Set up the initial pool configuration
      Properties p1  = new Properties();
      p1.put (OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT, Integer.toString(1));
      p1.put (OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT, Integer.toString(_maxCount));
      p1.put (OracleOCIConnectionPool.CONNPOOL_INCREMENT, Integer.toString(1));

      // Enable the initial configuration
      cpool.setPoolConfig(p1);

      Thread []t = new Thread[_maxCount];
      for (int i = 0; i < _maxCount; ++i)
      {
        conn[i] = cpool.getConnection("HR", "hr");
        if ( conn[i] == null )
        {
          System.out.println("Unable to create connection.");
          return;
        }
        t[i] = new conPoolAppl (i, conn[i]);
        t[i].start ();
        //displayPoolConfig(cpool);
      }

      ((conPoolAppl)t[0]).startAllThreads ();
```

```
    try
    {
      Thread.sleep (200);
    }
    catch (Exception ea) {}

    displayPoolConfig(cpool);
    for (int i = 0; i < _maxCount; ++i)
      t[i].join ();
  }
  catch(Exception ex)
  {
    System.out.println("Error: " + ex);
    ex.printStackTrace ();
    return;
  }
  finally
  {
    for (int i = 0; i < _maxCount; ++i)
      if (conn[i] != null)
      conn[i].close ();
  }
} //end of main

private Connection m_conn;
private static boolean m_startThread = false;
private int m_threadId;

public conPoolAppl (int i, Connection conn)
{
  m_threadId = i;
  m_conn = conn;
}

public void startAllThreads ()
{
  m_startThread = true;
}

public void run ()
{
  while (!m_startThread) Thread.yield ();
  try
  {
    doQuery (m_conn);
  }
  catch (SQLException ea)
  {
    System.out.println ("*** Thread id: " + m_threadId);
    ea.printStackTrace ();
  }
} // end of run

private static void doQuery (Connection conn) throws SQLException
{
  PreparedStatement pstmt = null;
  ResultSet rs = null;
  try
  {
    pstmt = conn.prepareStatement (query);
    rs = pstmt.executeQuery ();
    while (rs.next ())
```

```
        {
          //System.out.println ("Object name: " +rs.getString (1));
        }
      }
      catch (Exception ea)
      {
        System.out.println ("Error during execution: " +ea);
        ea.printStackTrace ();
      }
      finally
      {
        if (rs != null)
          rs.close ();
        if (pstmt != null)
          pstmt.close ();
        if (conn != null)
          conn.close ();
      }
    }  // end of doQuery (Connection)

    // Display the current status of the OracleOCIConnectionPool
    private static void displayPoolConfig (OracleOCIConnectionPool cpool) throws
SQLException
    {
      System.out.println (" Min poolsize Limit: " + cpool.getMinLimit());
      System.out.println (" Max poolsize Limit: " + cpool.getMaxLimit());
    /*
      System.out.println (" Connection Increment: " + cpool.getConnectionIncrement());
      System.out.println (" NoWait: " + cpool.getNoWait());
      System.out.println (" Timeout: " + cpool.getTimeout());
    */
      System.out.println (" PoolSize: " + cpool.getPoolSize());
      System.out.println (" ActiveSize: " + cpool.getActiveSize());
    }

}  // end of class conPoolAppl
```

27.6 Statement Handling and Caching

Statement caching is supported with `OracleOCIConnectionPool`. The caching improves performance by not having to open, parse, and close cursors. When `OracleOCIConnection.prepareStatement` ("*a_SQL_query*") is processed, the statement cache is searched for a statement that matches the SQL query. If a match is found, then you can reuse the `Statement` object instead of incurring the cost of creating another `Statement` object. The cache size can be dynamically increased or decreased. The default cache size is zero.

> **Note:**
>
> The `OracleStatement` object created from `OracleOCIConnection` has the same behavior as one that is created from `OracleConnection`.

27.7 JNDI and the OCI Connection Pool

The Java Naming and Directory Interface (JNDI) feature makes the properties of a Java object persist, therefore these properties can be used to construct a new instance of the object, such

as cloning the object. The benefit is that the old object can be freed, and at a later time a new object with exactly the same properties can be created. The `InitialContext.bind` method makes the properties persist, either on file or in a database, while the `InitialContext.lookup` method retrieves the properties from the persistent store and creates a new object with these properties.

`OracleOCIConnectionPool` objects can be bound and looked up using the JNDI feature. No new interface calls in `OracleOCIConnectionPool` are necessary.