Oracle 10*g* Implicit Connection Caching

As mentioned earlier, starting with Oracle 10*g*, the cache architecture just discussed was deprecated. It has been replaced by a more powerful, JDBC 3.0–compliant implicit connection caching. The highlights of implicit connection caching are

- Transparent access to a connection cache at the data source level.
- Support for connections with different usernames and passwords in the same cache.
- Ability to control cache behavior by defining a set of cache properties. The supported properties include ones that set timeouts, the maximum number of physical connections, and so on.
- Ability to retrieve a connection based on user-defined connection attributes (a feature known as *connection striping*).
- Ability to use callbacks to control cache behavior
  - When a connection is returned to the cache.
  - When a connection has been abandoned.
  - When an application requests a connection that does not exist in the cache.
- The new class `OracleConnectionCacheManager` is provided for administering the connection cache.

With the new cache architecture, you can turn on connection caching simply by invoking `setConnectionCachingEnabled( true )` on an `OracleDataSource` object. After caching is turned on, the first connection request to `OracleDataSource` implicitly creates a connection cache. There is a one-to-one mapping between the `OracleDataSource` object and its implicit connection cache.

Using the Oracle 10*g* Implicit Connection Cache

The following sections discuss the various steps involved in using the implicit connection cache.

Instantiating OracleDataSource

This step should be familiar to you by now:

```
OracleDataSource ods = new OracleDataSource();
ods.setURL ( "jdbc:oracle:thin:@rmenon-pc:1521:ora10g" );
ods.setUser("scott");           // username
ods.setPassword("tiger");       // password
```

Turning the Connection Cache On

You turn on the connection cache by simply invoking `setConnectionCachingEnabled()` on the `OracleDataSource` object:

```
ods.setConnectionCachingEnabled( true );
```

Setting Connection Cache Properties

You can optionally set connection properties listed later in this section by either using the method `setConnectionCacheProperties()` of the `OracleDataSource` object, or using the `OracleConnectionCacheManager` API to create or reinitialize the connection cache as discussed later. For example, the following code sets three properties of the connection cache using the `setConnectionCacheProperties()` method of the `OracleDataSource` object:

```
Properties cacheProperties = new Properties();
cacheProperties.setProperty( "InitialLimit", "2" );
cacheProperties.setProperty( "MinLimit", "3" );
cacheProperties.setProperty( "MaxLimit", "15" );
ods.setConnectionCacheProperties(cacheProperties);
```

By setting connection cache properties, you can control the characteristics of the connection cache.

■ **Caution** In my tests with 10.1.0.2.0, I found that the JDBC driver silently ignores an invalid (or misspelled) property. Thus, you need to be extra careful in spelling these properties while setting them up. Another problem is that, unfortunately, the property names that Oracle chose in many cases are not the same as the ones mentioned in JDBC 3.0 standard, though their meanings may be the same. For example, the JDBC property `InitialPoolSize` means the same thing as the Oracle property `InitialLimit`. This can be confusing.

Let's look at these cache properties in more detail.

Limit Properties

Limit properties control the size of the cache and the number of statements that are cached (see Chapter 13 for details on statement caching), among other things. Table 14-2 lists these properties along with their equivalent JDBC 3.0 standard property (if available), default value, and a brief description.

Table 14-2. *Cache Properties Related to Various Cache Limits Supported by Oracle 10*g *Implicit Connection Cache*

Property	Equivalent JDBC 3.0 Property	Default Value	Description
`InitialLimit`	`initialPoolSize`	0	Determines how many connections are created in the cache when it is created or reinitialized.
`MaxLimit`	`maxPoolSize`	No limit	Sets the maximum number of connections the cache can hold.
`MaxStatementsLimit`	`maxStatements`	0	Sets the maximum number of statements cached by a connection.
`MinLimit`	`minPoolSize`	0	Sets the minimum number of connections the cache is guaranteed to have at all times.
`LowerThresholdLimit`		20% of `maxLimit`	Sets the lower threshold limit on the cache. It is used when a `releaseConnection()` callback is registered with a cached connection. When the number of connections in the cache reaches this limit (`LowerThresholdLimit`), and a request is pending, the cache manager calls this method on the cache connections (instead of waiting for the connection to be freed).

Timeout and Time Interval Properties

These properties determine when the connections in the cache time out or at what interval Oracle checks and enforces the specified cache properties. Table 14-3 lists each of these properties with its default value and the type of connection it impacts (physical or logical). Only the property `PropertyCheckInterval` has a JDBC 3.0–equivalent property (`propertyCycle`).

Table 14-3. *Timeout and Interval Properties of Implicit Connection Cache*

Property	Default Value	Type of Connection Impacted	Description
`InactivityTimeout`	0 (no timeout)	Physical	Sets the maximum time in seconds a physical connection can remain idle in a connection cache. An *idle* connection is one that is not active and does not have a logical handle associated with it.
`TimeToLiveTimeout`	0 (no timeout)	Logical	Sets the maximum time in seconds that a logical connection can remain open (or checked out), after which it is returned to the cache.
`AbandonedConnectionTimeout`	0 (no timeout)	Logical	Sets the maximum time in seconds that a logical connection can remain open (or checked out) without any SQL activity on that connection, after which the logical connection is returned to the cache.
`ConnectionWaitTimeout`	0 (no timeout)	Logical	Comes into play when there is a request for a logical connection, the cache has reached the `MaxLimit`, and all physical connections are in use. This is the number of seconds the cache will wait for one of the physical connections currently in use to become free so that the request can be satisfied. After this timeout expires, the cache returns `null`.
`PropertyCheckInterval`	900 seconds		Sets the time interval in seconds at which the cache manager inspects and enforces all specified cache properties.

Attribute Weight Properties

Attribute weight properties allow you to set weights on certain attributes of a connection in the connection cache. If the property `ClosestConnectionMatch` is set to `true`, then these weights are used to get a "closest" match to the connection you request. We will look at these properties along with this feature in more detail in the section "Using Connection Attributes and Attribute Weights (10*g* Only)."

The ValidateConnection Property

If you set this property to `true`, the cache manager tests for validity each connection to be retrieved from the database. The default value is `false`.

Closing a Connection

Once an application is done using a connection, the application closes it using the `close()` method on the connection. There is another variant of this method that we will discuss in the section on "Using Connection Attributes and Attribute Weights (10*g* Only)."

An Example of Using Implicit Connection Caching

The following `DemoImplicitConnectionCaching` class illustrates how implicit connection caching works. First, we declare the class and the `main()` method after importing the required classes:

```
/* This program demonstrates implicit connection caching.
* COMPATIBLITY NOTE: runs successfully 10.1.0.2.0
*/
import java.sql.Connection;
import java.util.Properties;
import oracle.jdbc.pool.OracleDataSource;
import book.util.InputUtil;
class DemoImplicitConnectionCaching
{
  public static void main(String args[]) throws Exception
  {
```

Next, in the `main()` method, we instantiate the `OracleDataSource` object, which will hold our implicit cache:

```
    OracleDataSource ods = new OracleDataSource();
    ods.setURL ( "jdbc:oracle:thin:@rmenon-lap:1521:ora10g" );
    ods.setUser("scott");           // username
    ods.setPassword("tiger");       // password
```

We then enable the implicit caching:

```
    // enable implicit caching
    ods.setConnectionCachingEnabled( true );
```

We set the cache properties with an initial and minimum limit of three connections and a maximum limit of fifteen connections. The cache, when first set up, should pre-establish three connections, and it should never shrink below three connections later. Note that in production code, you should use a properties file to set these properties instead of using the `setProperty()` method in your Java code. This makes it easier to change them during runtime.

```
// set cache properties
Properties cacheProperties = new Properties();
cacheProperties.setProperty( "InitialLimit", "3" );
cacheProperties.setProperty( "MinLimit", "3" );
cacheProperties.setProperty( "MaxLimit", "15" );
ods.setConnectionCacheProperties(cacheProperties);
```

We first establish two connections to the `SCOTT` user, followed by one connection to the `BENCHMARK` user. We calculate and print the time it took to establish each of these connections. We also interject pauses using the `InputUtil.waitTillUserHitsEnter()` method (explained earlier in this chapter) after each connection establishment step. During these pauses, we will examine the database to see the number of physical connections using the query we covered in the section "Connections and Sessions in Oracle."

```
// time the process of establishing first connection
long startTime = System.currentTimeMillis();
Connection conn1 = ods.getConnection("scott", "tiger");
long endTime = System.currentTimeMillis();
System.out.println("It took " + (endTime-startTime) +
  " ms to establish the 1st connection (scott)." );
InputUtil.waitTillUserHitsEnter();
// time the process of establishing second connection
startTime = System.currentTimeMillis();
Connection conn2 = ods.getConnection("scott", "tiger");
endTime = System.currentTimeMillis();
System.out.println("It took " + (endTime-startTime) +
  " ms to establish the 2nd connection (scott)." );
InputUtil.waitTillUserHitsEnter();
// time the process of establishing third connection
startTime = System.currentTimeMillis();
Connection conn3 = ods.getConnection("benchmark", "benchmark");
endTime = System.currentTimeMillis();
System.out.println("It took " + (endTime-startTime) +
  " ms to establish the 3rd connection (benchmark)." );
InputUtil.waitTillUserHitsEnter();
```

At the end of the program, we close all connections, putting a pause after the first `close()` statement:

```
// close all connections
conn1.close();
InputUtil.waitTillUserHitsEnter("After closing the first connection.");
conn2.close();
```

```
    conn3.close();
  }// end of main
}// end of program
```

I ran this program on my machine, and after every pause, I ran the query to detect connections. Let's look at the output and the query results side by side:

```
B:\> java DemoImplicitConnectionCaching
It took 1015 ms to establish the 1st connection (scott).
Press Enter to continue...
```

The query results executed as the SYS user right after the pause are as follows:

```
sys@ORA10G> select s.server, p.spid server_pid, s.username
  2  from v$session s, v$process p
  3  where s.type = 'USER'
  4    and s.username != 'SYS'
  5    and p.addr(+) = s.paddr;
SERVER          SERVER_PID   USERNAME
--------------- ------------ ----------
DEDICATED       2460         SCOTT
DEDICATED       3528         SCOTT
DEDICATED       3288         SCOTT
```

This tells us that the very first time we establish a connection, the implicit connection caching results in the creation of three connections (equal to the parameter InitialLimit that we set earlier). That also explains why it took 1,015 milliseconds to establish the "first" connection. The program output after I pressed Enter follows:

```
It took 0 ms to establish the 2nd connection (scott).
Press Enter to continue...
```

This shows the main benefit of connection caching. Since we already have three connections established, this call gets one of these from the cache and completes really fast. At this time, if we run the query again, we will see the same output as before, since no new connections have been established. Pressing Enter again results in the following output:

```
It took 266 ms to establish the 3rd connection (benchmark).
Press Enter to continue...
```

Even though we have one more physical connection left in the cache, we cannot use it, since this time the request is to establish a connection to the user BENCHMARK. Hence the connection cache has to create a new connection, which took 266 milliseconds. At this time, the cache has four physical connections (three to SCOTT and one to BENCHMARK) established, as shown by our query results:

```
sys@ORA10G> select s.server, p.spid server_pid, s.username
  2  from v$session s, v$process p
  3  where s.type = 'USER'
  4    and s.username != 'SYS'
  5    and p.addr(+) = s.paddr;
```

```
SERVER          SERVER_PID   USERNAME
--------------- ------------ ----------
DEDICATED       2460         SCOTT
DEDICATED       3528         SCOTT
DEDICATED       3288         SCOTT
DEDICATED       2132         BENCHMARK
```

Finally, we press Enter again and see

```
After closing the first connection.
Press Enter to continue...
```

After the first connection is closed, when we execute the preceding query, we get the same results as before (four connections). This is, of course, because closing the logical connection does not result in a closing of the physical connection.

To manage your implicit caches, Oracle provides you with an API in the form of the class `OracleConnectionCacheManager` we'll look at it in the next section.

The OracleConnectionCacheManager Class

`OracleConnectionCacheManager` provides methods for the middle tier to centrally manage one or more connection caches that share a JVM. Each cache is given a unique name (implicitly or explicitly). The `OracleConnectionCacheManager` class also provides information about the cache, such as number of physical connections that are in use and the number of available connections.

The following sections describe some of the more commonly used methods that this class provides, with short descriptions. For a complete list of supported methods, please refer to *Oracle Database JDBC Developer's Guide and Reference* (for 10*g*).

createCache()

Using `createCache()`, you can create a connection cache with a given `DataSource` object and a `Properties` object. It also allows you to give a meaningful name to the cache, which is useful when you are managing multiple caches in the middle tier. The second variant listed generates a name for the cache internally.

```
public void createCache(String cacheName, javax.sql.DataSource datasource,
  java.util.Properties cacheProperties );
public void createCache(javax.sql.DataSource datasource,
   java.util.Properties cacheProperties );
```

removeCache()

This method waits `timeout` number of seconds for the in-use logical connections to be closed before removing the cache.

```
public void removeCache(String cacheName, int timeout);
```

reinitializeCache()

This method allows you to reinitialize the cache with the new set of properties. This is useful in dynamically configuring the cache based on runtime load changes and so forth.

```
public void reinitializeCache(String cacheName, java.util.properties
  cacheProperties)
```

■ **Caution** Invoking `reinitializeCache()` will close all in-use connections.

---

enableCache() and disableCache()

These two methods enable or disable a given cache. When the cache is disabled, in-use connections will work as usual, but no new connections will be served out from the cache.

```
public void enableCache(String cacheName);
public void disableCache(String cacheName);
```

getCacheProperties()

This method gets the cache properties for the specified cache.

```
public java.util.Properties getCacheProperties(String cacheName)
```

getNumberOfAvailableConnections()

This method gets the number of connections in the connection cache that are available for use.

```
public int getNumberOfAvailableConnections(String cacheName)
```

getNumberOfActiveConnections()

This method gets the number of in-use connections at a given point of time for a given cache.

```
public int getNumberOfActiveConnections(String cacheName)
```

setConnectionPoolDataSource()

This method sets the connection pool data source for the cache. All properties are derived from this data source.

```
public void setConnectionPoolDataSource(String cacheName,
  ConnectionPoolDataSource cpds)
```

An Example of Using the OracleConnectionCacheManager API

Let's look at the program DemoOracleConnectionCacheManager, which illustrates using some of the methods of the OracleConnectionCacheManager class. First, we import the classes and set up OracleDataSource as usual:

```
/* This program demonstrates using the Oracle connection cache manager API.
* COMPATIBLITY NOTE: runs successfully against 10.1.0.2.0
*/
import java.sql.Connection;
import java.util.Properties;
import oracle.jdbc.pool.OracleDataSource;
import oracle.jdbc.pool.OracleConnectionCacheManager;
class DemoOracleConnectionCacheManager
{
  public static void main(String args[]) throws Exception
  {
    OracleDataSource ods = new OracleDataSource();
    ods.setURL ( "jdbc:oracle:thin:@rmenon-lap:1521:ora10g" );
    ods.setUser("scott");           // username
    ods.setPassword("tiger");       // password
```

We then enable implicit connection caching:

```
    // enable implicit caching
    ods.setConnectionCachingEnabled( true );
```

Next, we set the connection cache properties and print them out:

```
    // set cache properties (use a properties file in production code.)
    Properties cacheProperties = new Properties();
    cacheProperties.setProperty( "InitialLimit", "2" );
    cacheProperties.setProperty( "MinLimit", "3" );
    cacheProperties.setProperty( "MaxLimit", "15" );
    ods.setConnectionCacheProperties(cacheProperties);
    System.out.println("Connection Cache Properties: ");
    System.out.println("\tInitialLimit: 2");
    System.out.println("\tMinLimit: 3");
    System.out.println("\tMaxLimit: 15");
```

We create the connection cache and explicitly give it a name (CONNECTION_CACHE_NAME is a constant defined later in the file):

```
    // create the connection cache
    OracleConnectionCacheManager occm =
      OracleConnectionCacheManager.getConnectionCacheManagerInstance();
    occm.createCache( CONNECTION_CACHE_NAME, ods, cacheProperties );
    System.out.println( "Just after creating the cache, " +
      "active connections: " +
      occm.getNumberOfActiveConnections( CONNECTION_CACHE_NAME )  +
        ", available connections: " +
      occm.getNumberOfAvailableConnections( CONNECTION_CACHE_NAME ) );
```

We print out the number of in-use (or active) connections and available connections:

```
System.out.println( "Just after creating the cache, " +
  "active connections: " +
  occm.getNumberOfActiveConnections( CONNECTION_CACHE_NAME )  +
     ", available connections: " + occm.getNumberOfAvailableConnections(
      CONNECTION_CACHE_NAME ));
```

Next, we get a connection and close it, printing the active and available connections after each step:

```
// get first connection
Connection conn1 = ods.getConnection("scott", "tiger");
System.out.println( "After getting first connection from cache, " +
  "active connections: " +
  occm.getNumberOfActiveConnections( CONNECTION_CACHE_NAME )  +
  ", available connections: " + occm.getNumberOfAvailableConnections(
  CONNECTION_CACHE_NAME ) );
conn1.close();
System.out.println( "After closing first connection, " +
  "active connections: " +
  occm.getNumberOfActiveConnections( CONNECTION_CACHE_NAME )  +
  ", available connections: " + occm.getNumberOfAvailableConnections(
   CONNECTION_CACHE_NAME ));
```

We then get three connections (so that we go beyond the initial minimum of two connections), and close one connection, printing the number of active and available connections after each step:

```
// get 3 connections to go beyond the InitialMinimum limit
Connection conn2 = ods.getConnection("scott", "tiger");
Connection conn3 = ods.getConnection("scott", "tiger");
Connection conn4 = ods.getConnection("scott", "tiger");
System.out.println( "After getting 3 connections, " +
  "active connections: " +
  occm.getNumberOfActiveConnections( CONNECTION_CACHE_NAME )  +
  ", available connections: " +   occm.getNumberOfAvailableConnections(
  CONNECTION_CACHE_NAME ) );
// close one connection - the number of connections should not
// go below 3 since we set a MinLimit value of 3.
conn2.close();
System.out.println( "After closing 1 connection, " +
  "active connections: " + occm.getNumberOfActiveConnections(
  CONNECTION_CACHE_NAME )  + ", available connections: " +
  occm.getNumberOfAvailableConnections( CONNECTION_CACHE_NAME ) );
```

We close the remaining two connections:

```
conn3.close();
conn4.close();
```

Just as an experiment, before ending the program, we disable the cache and try to retrieve a connection from it:

```
    // what happens if we disable cache and try to get a connection?
    occm.disableCache( CONNECTION_CACHE_NAME );
    Connection conn5 = ods.getConnection("scott", "tiger");
    conn5.close();
  }// end of main
  private static final String CONNECTION_CACHE_NAME = "myConnectionCache";
}// end of program
```

The output of this program is as follows:

```
B:\> java DemoOracleConnectionCacheManager
Connection Cache Properties:
        InitialLimit: 2
        MinLimit: 3
        MaxLimit: 15
Just after creating the cache, active connections: 0, available connections: 2
After getting first connection from cache, active connections: 1,
available connections: 1
After closing first connection, active connections: 0, available connections: 2
After getting 3 connections, active connections: 3, available connections: 0
After closing 1 connection, active connections: 2, available connections: 1
Exception in thread "main" java.sql.SQLException: Connection Cache with this
Cache Name is Disabled
        ... <-- trimmed to save space -->
```

In particular, note that the number of connections (available plus active) does not go below the minimum limit of three that we set, even though we close all but two connections, as shown by the last line of the output. Also, if we try to get a connection from a disabled cache, we get an exception as expected.

Let's now look at how we can use connection attributes and attribute weights in Oracle 10*g*.

Using Connection Attributes and Attribute Weights (10*g* Only)

A new feature of Oracle 10*g* JDBC drivers is that you can tag a connection with a label of your choice and use the tag to retrieve the same connection on which the tag was previously set from the connection cache. This feature is also known as *connection striping*.

Typically, you will use connection striping to change the state of the connection (say, setting its transaction isolation level) and then tag it. The next time you retrieve the connection using the tag, its state need not be reinitialized. Thus, you can create "stripes" of connections in your cache, each of which have its state set once to cater to the requirements of different applications sharing the cache.

Applying Connection Attributes on a Connection

We can apply a connection attribute to a connection in the cache in two ways. The first method is to invoke `applyConnectionAttributes( java.util.Properties connectionAttribute)` on the connection object to set the attributes. Later, when we want to retrieve it, we invoke `getConnection( java.util.Properties connectionAttributes)` on the connection object. Let's look at an example.

First, we get the connection from the cache (assume `ods` is initialized properly):

```
OracleConnection conn1 = (OracleConnection)
  ods.getConnection("scott", "tiger");
```

Then, we set the transaction isolation level of the connection to serializable (see the section "Transaction Isolation Levels" of Chapter 4 for details on transaction isolations):

```
conn1.setTransactionIsolation( Connection.TRANSACTION_SERIALIZABLE );
```

Now, we would like to remember this "attribute" of connection throughout its life in the cache, meaning we would like to retrieve a connection with transaction isolation level set to serializable later on. So we would mark this connection object with our own attribute value pair constants (`TXN_ISOLATION` and `SERIALIZABLE` are constants defined by us):

```
Properties connectionAttributes = new Properties();
connectionAttributes.setProperty( TXN_ISOLATION, SERIALIZABLE );
conn1.applyConnectionAttributes( connectionAttributes );
```

After using the connection, we close it:

```
conn1.close();
```

Later, we retrieve the same connection back from the cache using the attribute that we set:

```
conn1 = (OracleConnection) ods.getConnection( connectionAttributes );
```

Notice that we use the overloaded version of `getConnection()` that takes in a set of attributes to get a matching connection. Also note that once we tag a connection, we need to retrieve the connection always using the same tag. If we don't specify a tag (e.g., if we use the `getConnection()` method without any parameters), we'll get access to only the "untagged" set of connections. This is the intended behavior, since we don't want applications to run into each other's connection states. For example, an application shouldn't inadvertently get a connection with the transaction isolation set to serializable by some other application or module sharing the same cache.

The second method to set these attributes involves using the `close( java.util.Properties )` method on the connection. This method will override any attributes that we may have set on the same connection using the previous `applyConnectionAttributes()` method.

■ **Note** For some reason, Oracle treats the "autocommit" attribute of `Connection` as special, in that you can't retain the state of autocommit by setting it to `false` and tagging the connection. Although this looks bad, it turns out that autocommit doesn't involve a round-trip to the database because of the way Oracle JDBC drivers implement it, so tagging it wouldn't have resulted in a performance gain anyway. However, I found this special treatment of autocommit somewhat confusing.

Attribute Weights and the ClosestConnectionMatch Property

Attributes-based connection retrieval can be further refined by specifying attribute weights for each attribute. An attribute weight is a positive integer: the higher the weight, the higher the priority when a match is made for retrieving a connection. When performing a match, the connection cache tries to return a connection that matches all the attributes specified in the `getConnection( Properties connectionAttributes)` invocation. If an exact match is not found, and if `ClosestConnectionMatch` is set to `true`, then the cache tries to return the connection with the maximum number of matching attributes. If there is a tie here as well, then the connection cache returns the connection whose attributes have the highest combined weight.

The attribute weights should usually be specified based on how expensive it is to reconstruct a connection back to its intended state.

The last topic of this chapter is OCI driver connection pooling in 10*g*, which is an OCI driver–specific connection pooling implementation that offers some advantages over the standard connection pooling. This feature is also available in 9*i*.

OCI Connection Pooling

OCI connection pooling allows you to exploit *session multiplexing*, a mechanism in which multiple sessions are created using a low number of physical connections. Recall that in Oracle you can have more than one session on the same physical connection. OCI connection pooling provides better scalability over implicit connection caching, since fewer physical connections are required to support the same number of sessions. As its name suggests, OCI connection pooling requires you to have the JDBC OCI client installed and your environment set up accordingly as explained in Chapter 3.

■ **Note** Although in this section we cover only the Oracle 10*g* implementation of OCI connection pooling, this feature is also available in Oracle9*i*.

To use an OCI connection pool in your JDBC application, you need to take the following steps:

1. Create an OCI connection pool.
2. Configure the OCI connection pool properties.
3. Retrieve a connection from the OCI connection pool.

The following `DemoOCIConnectionPooling` class looks at these steps in detail. We begin by importing the classes and declaring the `main()` method:

```
/*
* This program demonstrates explicit statement caching.
* COMPATIBLITY NOTE:
*   runs successfully against 9.2.0.1.0 and 10.1.0.2.0
*/
import java.sql.Connection;
import java.util.Properties;
import oracle.jdbc.pool.OracleOCIConnectionPool;
import book.util.InputUtil;
class DemoOCIConnectionPooling
{
  public static void main(String args[]) throws Exception
  {
```

Creating an OCI Connection Pool

You create an OCI connection pool by initializing an `OracleOCIConnectionPool` object (`OracleOCIConnectionPool` extends from the now familiar `OracleDataSource` class) as follows:

```
    String tnsAlias = "(DESCRIPTION = (ADDRESS_LIST = (ADDRESS =
(PROTOCOL = TCP)(HOST = rmenon-lap)(PORT = 1521))) (CONNECT_DATA =
(SERVER = DEDICATED) (SERVICE_NAME = ora10g.us.oracle.com)))";
    OracleOCIConnectionPool ods = new OracleOCIConnectionPool();
    ods.setURL ( "jdbc:oracle:oci:@"+ tnsAlias );
    ods.setUser("scott");           // username
    ods.setPassword("tiger");       // password
```

Notice how we use the OCI driver–style connection parameters.

Configuring the OCI Connection Pool Properties

The next step is to configure the pool properties that dictate how the pool behaves. Table 14-4 lists the various pool properties we can set and their meanings. Note that all these attributes can be configured dynamically.

Table 14-4. *Oracle OCI Connection Pool Configuration Properties*

Property	Default	Constraints	Getter Method	Description
`OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT`	1	Is mandatory; must be a positive integer	`getMinLimit()`	Specifies the minimum number of physical connections in the pool.
`OracleOCIConnectionPool.CONNPOOL_INCREMENT`	0	Is mandatory; must be a positive integer	`getConnectionIncrement()`	Specifies the number of additional physical connections to be opened when a request for a connection is pending and all available physical connections are in use.
`OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT`	1	Is mandatory; must be >(`CONNPOOL_MIN_LIMIT` +`CONNPOOL_INCREMENT`)	`getMaxLimit()`	Specifies the maximum number of physical connections in the pool.
`OracleOCIConnectionPool.CONNPOOL_TIMEOUT`	0	Must be a positive integer	`getTimeout()`	Specifies the number of seconds after which an idle physical connection is disconnected.
`OracleOCIConnectionPool.CONNPOOL_NOWAIT`	`false`	Must be `true` or `false`	`getNoWait()`	When specified, this property implies that the connection pool should return an error if all connections in the pool are busy and another request for a connection comes in.

Continuing the definition of our class, we set the OCI connection pool properties using the `setPoolConfig()` method of the `OracleOCIConnectionPool` class:

```
Properties cacheProperties = new Properties();
cacheProperties.setProperty(OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT, "2" );
cacheProperties.setProperty(OracleOCIConnectionPool.CONNPOOL_INCREMENT, "1" );
cacheProperties.setProperty(OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT, "10" );
ods.setPoolConfig( cacheProperties );
System.out.println("Min Limit: 2");
System.out.println("Max Limit: 10");
System.out.println("Increment : 1");
System.out.println("pool size:" + ods.getPoolSize())
```

The cache gets created when the we invoke the preceding `setPoolConfig()` method.

Retrieving a Connection from the OCI Connection Pool

Once the pool is configured, we can retrieve a connection using the standard `getConnection()` method on the `OracleOCIConnectionPool` object (recall that it extends the `OracleDataSource` class):

```
Connection conn = oocp.getConnection("scott", "tiger");
```

Analyzing Connections and Sessions When OCI Connection Pooling Is in Use

The behavior of OCI connection pooling depends on whether your program is multithreaded or not. For a single-threaded program, the OCI connection pool sets up a connection cache equal to the configured property `OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT`. After that, every connection request results in a *session* being created using one of these cached connections. No new connections are created in the cache above `OracleOCIConnectionPool.CONNPOOL_➥ MIN_LIMIT`. But new sessions are created as requested on top of the connections created initially. As you will learn, this is a case where you can see more than one session being created on top of a connection.

For a multithreaded program, the OCI connection pool sets up a connection cache equal to the configured property `OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT`. After that, every connection request from a new thread results in a new physical connection being created.

Let's look at each of these cases now, beginning with the case of a single-threaded program.

Analyzing OCI Connection Pooling in a Single-Threaded Program

The program `AnalyzeOCIConnPoolSingleThread` described in this section takes as input the number of sessions to open using the OCI connection pool. It has the now familiar pauses introduced for us to run our query listing physical connections and sessions. Let's look at the program piecemeal, starting with the imports and the declaration of the `main()` method:

```
/*
* This program demonstrates use of OCI connection pooling in a single-threaded
program.
* COMPATIBLITY NOTE: tested against 10.1.0.2.0.
```

```
*/
import java.sql.Connection;
import java.util.Properties;
import oracle.jdbc.pool.OracleDataSource;
import oracle.jdbc.pool.OracleOCIConnectionPool;
import book.util.InputUtil;
class AnalyzeOCIConnPoolSingleThread
{
  public static void main(String args[]) throws Exception
  {
```

The function _getNumOfSessionsToOpen(), defined at the end of the program, returns the number of sessions requested as the first command-line parameter value when running the program.

```
    int numOfSessionsToOpen = _getNumOfSessionsToOpen( args );
```

We set up the TNS alias and configure the pool with connection properties to connect to the user SCOTT:

```
    String tnsAlias = "(DESCRIPTION = (ADDRESS_LIST = (ADDRESS =
(PROTOCOL = TCP)(HOST = rmenon-lap)(PORT = 1521))) (CONNECT_DATA =
(SERVER = DEDICATED)(SERVICE_NAME = ora10g.us.oracle.com)))";
    OracleOCIConnectionPool oocp = new OracleOCIConnectionPool();
    oocp.setURL ( "jdbc:oracle:oci:@"+ tnsAlias );
    oocp.setUser("scott");           // username
    oocp.setPassword("tiger");       // password
```

Next we set the pool configuration properties and print them out. We also print out the time it takes to set up the initial pool (which happens when we invoke setPoolConfig()).

```
    // set pool config properties
    Properties poolConfigProperties = new Properties();
    poolConfigProperties.setProperty(
      OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT, "3" );
    poolConfigProperties.setProperty(
      OracleOCIConnectionPool.CONNPOOL_INCREMENT, "1" );
    poolConfigProperties.setProperty(
      OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT, "20" );
    long startTime = System.currentTimeMillis();
    oocp.setPoolConfig( poolConfigProperties );
    long endTime = System.currentTimeMillis();
    System.out.println("It took " + (endTime-startTime) +
      " ms to establish initial pool size of "+ oocp.getPoolSize() +
      " connections." );
    //print config properties
    System.out.println( "min Limit: " + oocp.getMinLimit() );
    System.out.println( "max Limit: " + oocp.getMaxLimit() );
    System.out.println( "connection increment : " + oocp.getConnectionIncrement() );
    System.out.println( "timeout: " + oocp.getTimeout() );
    System.out.println( "nowait: " + oocp.getNoWait() );
```

We can find out the number of physical connections opened by invoking the `getPoolSize()` method as follows:

```
System.out.println( "num of physical connections: " + oocp.getPoolSize() );
```

Next, we create the number of sessions specified at the command line to the user `SCOTT`, with a pause before and after. We also measure the time it takes to establish the sessions.

```
InputUtil.waitTillUserHitsEnter( "before establishing scott connections");
for( int i=0; i < numOfSessionsToOpen; i++ )
{
  // time the process of establishing a connection
  startTime = System.currentTimeMillis();
  scottConnections[i] = oocp.getConnection("scott", "tiger");
  endTime = System.currentTimeMillis();
  System.out.println("It took " + (endTime-startTime) +
    " ms to establish session # " + (i+1) + " (scott)." );
  System.out.println( "num of physical connections: "   + oocp.getPoolSize() );
}
InputUtil.waitTillUserHitsEnter();
```

We create and time the same number of sessions for the user `BENCHMARK`, with a pause at the end:

```
Connection[] benchmarkConnections = new Connection[ numOfSessionsToOpen ];
for( int i=0; i < numOfSessionsToOpen; i++ )
{
  // time the process of establishing a connection
  startTime = System.currentTimeMillis();
  benchmarkConnections[i] = oocp.getConnection( "benchmark", "benchmark");
  endTime = System.currentTimeMillis();
  System.out.println("It took " + (endTime-startTime) +
    " ms to establish the session # " + (i+1) + " (benchmark)." );
  System.out.println( "num of physical connections: " + oocp.getPoolSize() );
}
InputUtil.waitTillUserHitsEnter();
```

Finally, we close all sessions and define the method `_getNumOfSessionsToOpen()` we invoked earlier to end the program:

```
// close all connections (or sessions)
for( int i=0; i < numOfSessionsToOpen; i++ )
{
  if( benchmarkConnections[i] != null )
    benchmarkConnections[i].close();
  if( scottConnections[i] != null )
    scottConnections[i].close();
}
}// end of main
private static int _getNumOfSessionsToOpen( String[] args )
{
```

```
    int numOfSessionsToOpen = 3; //by default open 3 sessions
    if( args.length == 1 )
    {
      numOfSessionsToOpen = Integer.parseInt( args[0] );
    }
    System.out.println( "Num of sessions to open for scott and benchmark each = "
      + numOfSessionsToOpen);
    return numOfSessionsToOpen;
  }
}// end of program
```

Let's now look at the program output, and also discuss the output of the query that lists opened physical connections and sessions during the programmed pauses. Consider the case where we request that six sessions be opened each for the SCOTT and BENCHMARK users. The first few lines of program output follow:

```
B:\>java AnalyzeOCIConnPoolSingleThread 6
Num of sessions to open for scott and benchmark each = 6
It took 781 ms to establish initial pool size of 3 connections.
min Limit: 3
max Limit: 20
connection increment : 1
timeout: 0
nowait: false
num of physical connections: 3
before establishing scott connections
Press Enter to continue...
```

Note from the output that it takes 781 milliseconds to establish three physical connections when we use OCI connection pooling. This is slightly better than the 1,015 milliseconds we saw when we used the implicit connection cache. The difference remains even if we use the OCI driver with the implicit connection cache. We also get the number of physical connections by invoking `getPoolSize()` on the OCI connection pool variable. Our query for listing connections and sessions confirms this:

```
sys@ORA10G> select s.program, s.server, p.spid server_pid, s.username
  2  from v$session s, v$process p
  3  where s.type = 'USER'
  4    and s.username != 'SYS'
  5    and p.addr(+) = s.paddr;

PROGRAM                        SERVER          SERVER_PID   USERNAME
------------------------------ --------------- ------------ ----------
java.exe                       DEDICATED       3260         SCOTT
java.exe                       DEDICATED       2472         SCOTT
java.exe                       DEDICATED       1896         SCOTT
```

After we press Enter, the program shows the following output after creating six sessions for the user SCOTT:

```
It took 78 ms to establish session # 1 (scott).
num of physical connections: 3
It took 16 ms to establish session # 2 (scott).
num of physical connections: 3
It took 0 ms to establish session # 3 (scott).
num of physical connections: 3
It took 15 ms to establish session # 4 (scott).
num of physical connections: 3
It took 16 ms to establish session # 5 (scott).
num of physical connections: 3
It took 0 ms to establish session # 6 (scott).
num of physical connections: 3
Press Enter to continue...
```

At this point, we have established six *sessions* to SCOTT user. The second-to-last line of our output confirms that we have only three physical connections at this point. Our query results are as follows:

```
sys@ORA10G> /

PROGRAM                        SERVER          SERVER_PID   USERNAME
------------------------------ --------------- ------------ ----------
java.exe                       DEDICATED       3260         SCOTT
java.exe                       DEDICATED       2472         SCOTT
java.exe                       DEDICATED       1896         SCOTT
java.exe                       PSEUDO                       SCOTT
java.exe                       PSEUDO                       SCOTT
java.exe                       PSEUDO                       SCOTT
java.exe                       PSEUDO                       SCOTT
java.exe                       PSEUDO                       SCOTT
java.exe                       PSEUDO                       SCOTT
```

We were expecting six rows, but there are nine rows shown by the query! Careful examination shows that only three of them have a server_pid column—those three rows correspond to actual physical connections created by Oracle so far. The remaining six rows that correspond to a null value for the server_pid column (and also have a value of PSEUDO under the server column) are sessions created by Oracle on top of the three physical connections.

When we press Enter once more, we get the following output:

```
It took 78 ms to establish the session # 1 (benchmark).
num of physical connections: 3
It took 16 ms to establish the session # 2 (benchmark).
num of physical connections: 3
It took 0 ms to establish the session # 3 (benchmark).
num of physical connections: 3
It took 15 ms to establish the session # 4 (benchmark).
num of physical connections: 3
It took 16 ms to establish the session # 5 (benchmark).
num of physical connections: 3
```

```
It took 0 ms to establish the session # 6 (benchmark).
num of physical connections: 3
Press Enter to continue...
```

Notice that it took 78 milliseconds to establish a new benchmark session, though the connection pool had all physical connections authenticated as the user SCOTT. This is pretty good compared to the 266 milliseconds we saw when using implicit cache to establish a new connection and session. This is because in this case we are creating a session on an already existing physical connection. Our query confirms that there are three connections, six SCOTT sessions, and six BENCHMARK sessions in the database at this point:

```
sys@ORA10G> /

PROGRAM                        SERVER          SERVER_PID   USERNAME
------------------------------ --------------- ------------ ----------
java.exe                       DEDICATED       3260         SCOTT
java.exe                       DEDICATED       2472         SCOTT
java.exe                       DEDICATED       1896         SCOTT
java.exe                       PSEUDO                       SCOTT
java.exe                       PSEUDO                       BENCHMARK
java.exe                       PSEUDO                       BENCHMARK
java.exe                       PSEUDO                       BENCHMARK
java.exe                       PSEUDO                       BENCHMARK
java.exe                       PSEUDO                       SCOTT
java.exe                       PSEUDO                       SCOTT
java.exe                       PSEUDO                       BENCHMARK
java.exe                       PSEUDO                       SCOTT
java.exe                       PSEUDO                       SCOTT
java.exe                       PSEUDO                       BENCHMARK
java.exe                       PSEUDO                       SCOTT

12 rows selected.
```

We can end the program by pressing Enter once more.

If you try to run the program with an increasing number of sessions using the command-line parameter, you will find that the number of physical connections remains the same, no matter how many sessions you create. As mentioned, this is because in a single-threaded program, the OCI connection pool creates physical connections only at the beginning based on the OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT parameter. Of course, at some point you would run out of resources—there can be only so many sessions created on three physical connections. For example, on my PC, I was able to create 65 sessions each for SCOTT and BENCHMARK on three physical connections. When I tried to bump the number to 66, I got the following exception:

```
Exception in thread "main" java.sql.SQLException: ORA-00604: error occurred at
ecursive SQL level 1
ORA-04031: unable to allocate 4012 bytes of shared memory (
"large pool","unknown  object","session heap","bind var buf")
… <-- trimmed to save space -->
```

The interesting thing is that you can dynamically set this minimum limit by simply passing in the appropriately modified `Properties` object to the `setPoolConfig()` method at runtime, thus controlling the actual number of physical connections used in setting up the sessions in your pool.

What happens if the program is multithreaded? We cover that in the next section.

Analyzing OCI Connection Pooling in a Multithreaded Program

To analyze the case of a multithreaded program, we will first look at the `WorkerThread` class, which executes a query after getting the connection. Since it is a multithreaded program, we can't introduce pauses easily, so we make each worker thread execute a query that we know will take some time to get the output of. In our case, the query is `select object_name from all_objects`. After the necessary imports, the class `WorkerThread` begins with a constructor that takes a connection pool (assumed to be initialized by the calling program) and a thread number.

```
import book.util.JDBCUtil;
class WorkerThread extends Thread
{
  WorkerThread( OracleOCIConnectionPool ociConnPool, int _threadNumber )
  throws Exception
  {
    super();
    this._ociConnPool = ociConnPool;
    this._threadNumber = _threadNumber;
  }
```

The `run()` method of the `WorkerThread` class gets a connection to `SCOTT` if the thread number is even; otherwise, it gets a connection to `BENCHMARK`.

```
  public void run()
  {
    Connection conn = null;
    Statement stmt = null;
    ResultSet rset = null;
    try
    {
      if( _threadNumber % 2 == 0 )
      {
        System.out.println( "connecting as scott" );
        conn = _ociConnPool.getConnection("scott", "tiger");
      }
      else
      {
        System.out.println( "connecting as benchmark" );
        conn = _ociConnPool.getConnection("benchmark", "benchmark");
      }
```

We then execute our query and end the `WorkerThread` class:

```
      pstmt = conn.prepareStatement( "select owner from all_objects" );
      rset = pstmt.executeQuery();
      while( rset.next() )
      {
        rset.getString(1);
      }
    }
    catch (Exception e )
    {
      e.printStackTrace();
    }
    finally
    {
      JDBCUtil.close( rset );
      JDBCUtil.close( pstmt );
      JDBCUtil.close( conn );
    }
  } // end of run
  private OracleOCIConnectionPool _ociConnPool;
  private int _threadNumber = -1;
} // end of class
```

We set up the OCI connection pool and invoke the `WorkerThread` program from the class `AnalyzeOCIConnPoolMultiThread` as follows:

```
/** This program demonstrates the use of OCI connection pooling in a
    multithreaded program.
* COMPATIBLITY NOTE: tested against 10.1.0.2.0.
*/
import java.util.Properties;
import oracle.jdbc.pool.OracleOCIConnectionPool;
public class AnalyzeOCIConnPoolMultiThread
{
  public static void main( String [] args ) throws Exception
  {
    _numOfSessionsToOpen = _getNumOfSessionsToOpen( args );
    String tnsAlias = "(DESCRIPTION = (ADDRESS_LIST = (ADDRESS = (PROTOCOL =
 TCP)(HOST = rmenon-lap)(PORT = 1521))) (CONNECT_DATA = (SERVER = DEDICATED)
 (SERVICE_NAME = ora10g.us.oracle.com)))";
    OracleOCIConnectionPool cpool = new OracleOCIConnectionPool();
    cpool.setURL ( "jdbc:oracle:oci:@"+ tnsAlias );
    cpool.setUser("scott");           // username
    cpool.setPassword("tiger");       // password
    Properties poolConfigProps = new Properties( ) ;
```

```
    poolConfigProps.put(OracleOCIConnectionPool.CONNPOOL_MIN_LIMIT, "2") ;
    poolConfigProps.put(OracleOCIConnectionPool.CONNPOOL_INCREMENT, "1") ;
    poolConfigProps.put(OracleOCIConnectionPool.CONNPOOL_MAX_LIMIT, "20") ;
    cpool.setPoolConfig(poolConfigProps);
    System.out.println ("Min poolsize Limit = " + cpool.getMinLimit());
    System.out.println ("Max poolsize Limit = " + cpool.getMaxLimit());
    System.out.println ("Connection Increment = " + cpool.getConnectionIncrement());
```

Up until this point, this program is the same as the single-threaded program, `AnalyzeOCI➥ConnPoolSingleThread`, which we saw in the previous section. After this, we create the number of threads as specified by the command-line parameter and start them.

```
    Thread [] threads = new Thread[ _numOfSessionsToOpen ];
    for( int i = 0; i<threads.length; i++ )
    {
      (threads[i] = new WorkerThread( cpool, i )).start();
    }
```

Finally, we wait for all threads to finish in a loop. This ends the `main()` program, which is followed by the definition of the `_getNumOfSessionsToOpen()` method at the end:

```
    // wait until all threads are done.
    for( int i = 0; i<threads.length; i++ )
    {
      threads[i].join();
    }

    cpool.close();
  }// end of main

  private static int _getNumOfSessionsToOpen( String[] args )
  {
    int numOfSessionsToOpen = 6; //by default open 6 sessions
    if( args.length == 1 )
    {
      numOfSessionsToOpen = Integer.parseInt( args[0] );
    }
    System.out.println("Total number of sessions to open for " +
      "scott and benchmark = " + numOfSessionsToOpen);

    return numOfSessionsToOpen;
  }

  private static int _numOfSessionsToOpen;
}
```

We will run the program and then examine the database while the program is running. For some reason, the only reliable results our original query shows are the number of connections and the number of sessions. I modified the query to print out this information (note that the following query works only for the case in which we know that sessions are being independently created on top of connections, as is the case here). The program output is as follows:

```
B:\> java AnalyzeOCIConnPoolMultiThread 12
Total number of sessions to open for scott and benchmark = 12
Min poolsize Limit = 2
Max poolsize Limit = 20
Connection Increment = 1
connecting as scott
connecting as benchmark
connecting as scott
connecting as benchmark
connecting as scott
connecting as benchmark
connecting as scott
connecting as benchmark
connecting as scott
connecting as benchmark
connecting as scott
connecting as benchmark
```

The program opens six sessions for `SCOTT` and six more for `BENCHMARK`.

When I ran the modified query and executed it immediately again and again, it gave the following results:

```
sys@ORA10G> select num_of_conns, (conns_plus_sess -num_of_conns) as num_of_seons
  2  from
  3  (
  4    select count(*) conns_plus_sess,
  5           count( distinct p.spid) num_of_conns
  6    from v$session s, v$process p
  7    where s.type != 'BACKGROUND'
  8      and s.username != 'SYS'
  9      and p.addr(+) = s.paddr
 10  );

NUM_OF_CONNS NUM_OF_SESSIONS
------------ ---------------
           8              12
sys@ORA10G> /
NUM_OF_CONNS NUM_OF_SESSIONS
------------ ---------------
           8              12
sys@ORA10G> /
```

```
NUM_OF_CONNS NUM_OF_SESSIONS
------------ ---------------
           12               12
<-- after some time -->
sys@ORA10G> /
NUM_OF_CONNS NUM_OF_SESSIONS
------------ ---------------
           12               10
<-- after some time -->
sys@ORA10G> /
NUM_OF_CONNS NUM_OF_SESSIONS
------------ ---------------
           0                0
```

This shows that the program had eight connections at the point I executed the query for the first time and 12 sessions. Ultimately, we had 12 sessions created on top of 12 connections. We can also see a stage where the number of sessions goes down to 10, while the number of connections open remains at 12. This experiment proves that each thread had a separate physical connection, in the case of multithreaded programs using OCI connection pooling.

Summary

In this chapter, you learned the difference between connections and sessions in Oracle. You learned why connection pooling is necessary, and you distinguished between connection pooling and caching. You examined how Oracle9*i* implements the connection pooling framework and provides a sample connection caching implementation. As you saw, in Oracle 10*g*, the Oracle9*i* connection caching has been deprecated and replaced by the more powerful implicit connection caching. Finally, you took a look at how Oracle's OCI connection pooling improves scalability and performance by creating lightweight sessions on top of a low number of physical connections.