JDBC® Connection Pooling

1 Introduction

This document provides information intended to help developers provide a connection pooling strategy for applications that must handle connection pooling. First, this document provides an overview of JDBC® connection pooling as specified by the JDBC 3.0 specification. Next, it provides examples of how to use the DataDirect Connection Pool Manager (which is shipped with DataDirect Connect® *for* JDBC and DataDirect SequeLink® *for* JDBC) for your applications. Finally, this document provides an example showing performance benchmarks that demonstrate the performance benefit you can achieve by using connection pooling.

2 Connection Pooling

Establishing JDBC connections is resource-expensive, especially when the JDBC API is used in a middle-tier server environment, such as when DataDirect Connect *for* JDBC or DataDirect SequeLink *for* JDBC is running on a Java®-enabled web server. In this type of environment, performance can be improved significantly when connection pooling is used. *Connection pooling* means that connections are reused rather than created each time a connection is requested. To facilitate connection reuse, a memory cache of database connections, called a *connection pool*, is maintained by a connection pooling module as a layer on top of any standard JDBC driver product.

Connection pooling is performed in the background and does not affect how an application is coded; however, the application must use a DataSource object (an object implementing the DataSource interface) to obtain a connection instead of using the DriverManager class. A class implementing the DataSource interface may or may not provide connection pooling. A DataSource object registers with a JNDI naming service. Once a DataSource object is registered, the application retrieves it from the JNDI naming service in the standard way. For example:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("jdbc/SequeLink");
```

If the DataSource object provides connection pooling, the lookup returns a connection from the pool if one is available. If the DataSource object does not provide connection pooling or if there are no available connections in the pool, the lookup creates a new connection. The application benefits from connection reuse without requiring any code changes. Reused connections from the pool behave the same way as newly created physical connections. The application makes a connection to the database and data access works in the usual way. When the application has finished its work with the connection, the application explicitly closes the connection. For example:

```
Connection con = ds.getConnection("scott", "tiger");
// Do some database activities using the connection...
con.close();
```

The closing event on a pooled connection signals the pooling module to place the connection back in the connection pool for future reuse.

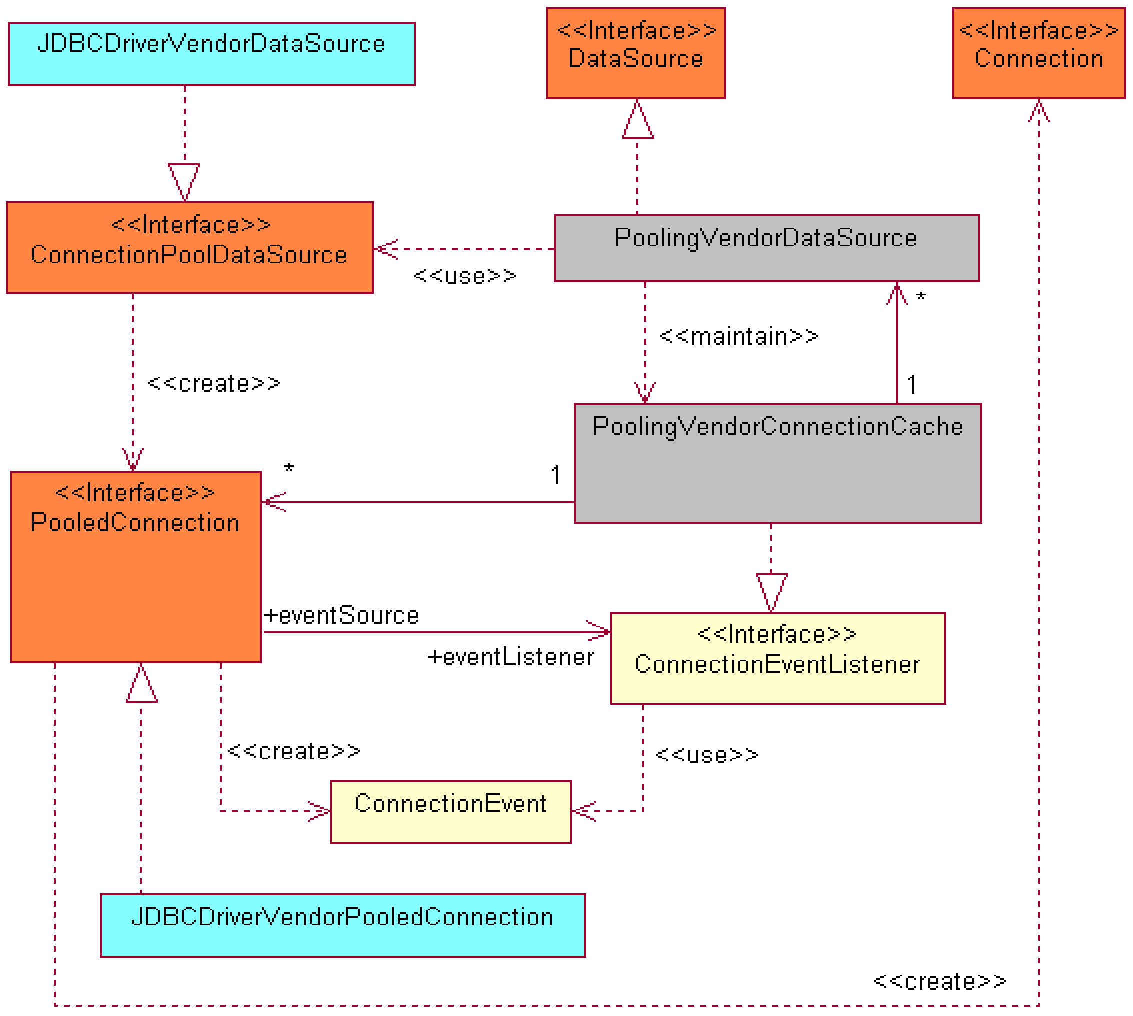
3 JDBC 3.0 Connection Pooling Framework

The JDBC 3.0 API provides a general framework with "hooks" to support connection pooling rather than specifying a particular connection pooling implementation. In this way, third-party vendors or users can implement the specific caching or pooling algorithms that best fit their needs. The JDBC 3.0 API specifies a ConnectionEvent class and the following interfaces as the hooks for any connection pooling implementation:

- ConnectionPoolDataSource
- PooledConnection
- ConnectionEventListener

Figure 1 shows this general framework.

Figure 1. JDBC 3.0 Connection Pooling Architecture


JDBCDriverVendorDataSource
<<Interface>> DataSource
<<Interface>> Connection
<<Interface>> ConnectionPoolDataSource
PoolingVendorDataSource
<<use>>
<<maintain>>
*
1
<<create>>
PoolingVendorConnectionCache
*
1
<<Interface>> PooledConnection
+eventSource
+eventListener
<<Interface>> ConnectionEventListener
<<create>>
<<use>>
ConnectionEvent
JDBCDriverVendorPooledConnection
<<create>>

JDBCDriverVendorDataSource
A JDBC driver vendor must provide a class that implements the standard ConnectionPoolDataSource interface. This interface provides hooks that third-party vendors can use to implement pooling as a layer on top of their JDBC drivers. The ConnectionPoolDataSource interface acts as a "factory" that creates PooledConnection objects.

JDBCDriverVendorPooledConnection
A JDBC driver vendor must provide a class that implements the standard PooledConnection interface. This interface allows third-party vendors to implement pooling on top of their JDBC drivers. A PooledConnection object acts as a "factory" that creates Connection objects. A PooledConnection object maintains the physical connection to the database; the Connection object created by the PooledConnection object is simply a handle to the physical connection.

PoolingVendorDataSource
A third-party vendor must provide a class that implements the DataSource interface. This interface is the entry point that allows interaction with their pooling module. The pooling vendor's class uses the JDBC driver's PooledConnectionDataSource object to create the PooledConnections that the pool manages.

PoolingVendorConnectionCache
The JDBC 3.0 API does not specify the interfaces to be used between the DataSource object and the connection cache. The pooling vendor determines how these components interact. Usually, a connection cache module contains one or multiple classes. In Figure 1, the PoolingVendorConnectionCache class is used as a simple way to convey this concept. The connection cache module should have a class that implements the standard ConnectionEventListener interface. The ConnectionEventListener interface receives ConnectionEvent objects from PooledConnection objects when a connection closes or a connection error occurs. When a connection, created by a PooledConnection, closes, the connection cache module returns the PooledConnection object to the cache.

When an application makes a connection by calling DataSource.getConnection() on a PoolingVendorDataSource object, the PoolingVendorDataSource object performs a lookup in the connection cache to determine if a PooledConnection object is available. If one is available, it is used. If a PooledConnection object is not available, the JDBC driver vendor’s ConnectionPoolDataSource creates a new PooledConnection object. In either case, a PooledConnection object is made available.

The PoolingVendorDataSource object then invokes the PooledConnection.getConnection() method to obtain a Connection object, which it returns to the application to use as a normal connection. Because the JDBC driver vendor implements the PooledConnection interface, the JDBC driver creates the Connection object; however, this Connection object is not a physical connection as in the non-pooling case. The Connection object is a handle to the physical connection maintained by the PooledConnection object.

When the application closes the connection by calling the Connection.close() method, a ConnectionEvent is generated and is passed to the cache module. The cache module returns the PooledConnection object that created the connection to the cache to be reused. The application does not have access to the PooledConnection.close() method. Only the connection pooling module, as part of its clean-up activity, issues the PooledConnection.close() method to actually close the physical connection.

4 Creating a Data Source

This section provides examples on how to create pooled and non-pooled DataSource objects for DataDirect Connect *for* JDBC and DataDirect SequeLink *for* JDBC, and register them to a JNDI naming service.

4.1 Creating a DataDirect Data Source Object

DataDirect Connect® *for* JDBC

This example shows how to create a DataDirect Connect *for* JDBC DataSource object and register it to a JNDI naming service. The DataSource class provided by the DataDirect Connect *for* JDBC drivers is database-dependent. In the following example we use Oracle, so the DataSource class is com.ddtek.jdbcx.oracle.OracleDataSource.

If you want the client application to use:

- A *non-pooled* data source, the application can specify the JNDI name of this data source object as registered in the following code example ("jdbc/ConnectSparkyOracle").
- A *pooled* data source, the application must specify the JNDI name ("jdbc/SparkyOracle") as registered in the code example in the section "4.2 Creating a Data Source Using the DataDirect Connection Pool Manager" on page 8.

```
//************************************************************************
//
// This code creates a DataDirect Connect for JDBC data source and
// registers it to a JNDI naming service. This DataDirect Connect for
// JDBC data source uses the DataSource implementation provided by
// DataDirect Connect for JDBC Drivers.
//
// This data source registers its JNDI name as <jdbc/ConnectSparkyOracle>.
// Client applications using non-pooled connections must perform a lookup
// for this name.
//
//************************************************************************

// From DataDirect Connect for JDBC:
import com.ddtek.jdbcx.oracle.OracleDataSource;

import javax.sql.*;
import java.sql.*;
import javax.naming.*;
import javax.naming.directory.*;
import java.util.Hashtable;

public class OracleDataSourceRegisterJNDI
{
    public static void main(String argv[])
    {
             try {
                    // Set up data source reference data for naming context:
                    // ----------------------------------------------------
                    // Create a class instance that implements the interface
                    // ConnectionPoolDataSource
                    OracleDataSource ds = new OracleDataSource();

                    ds.setDescription(
                           "Oracle on Sparky - Oracle Data Source");
                    ds.setServerName("sparky");
                    ds.setPortNumber(1521);
                    ds.setUser("scott");
                    ds.setPassword("test");

                    // Set up environment for creating initial context
                    Hashtable env = new Hashtable();
                    env.put(Context.INITIAL_CONTEXT_FACTORY,
                           "com.sun.jndi.fscontext.RefFSContextFactory");
                    env.put(Context.PROVIDER_URL, "file:c:\\JDBCDataSource");
                    Context ctx = new InitialContext(env);

                    // Register the data source to JNDI naming service
                    ctx.bind("jdbc/ConnectSparkyOracle", ds);

             } catch (Exception e) {
                    System.out.println(e);
                    return;
             }
    } // Main
} // class OracleDataSourceRegisterJNDI
```

DataDirect SequeLink®

The following example shows how to create a SequeLink *for* JDBC DataSource object and register it to a JNDI naming service. The DataSource class provided by the DataDirect SequeLink *for* JDBC driver is database-independent; therefore, for all databases, the DataSource class is SequeLinkDataSource.

If you want the client application to use:

- A *non-pooled* connection ("4.1 Creating a DataDirect Data Source Object" on page 5), you must modify this example so that the JNDI entry is registered using the name `jdbc/SparkyOracle`.
- A *pooled* connection, the JNDI entry must map to the DataSource of the DataDirect Connection Pool Manager. Therefore, you must register two data sources:
  - The Connection Pool Manager's Data Source using the example in "4.2 Creating a Data Source Using the DataDirect Connection Pool Manager" on page 8. This process registers the data source using the JNDI entry jdbc/SparkyOracle. The Connection Pool Manager creates physical connections using the JNDI entry jdbc/SequeLinkSparkyOracle.
  - A SequeLink Data Source, using the following example to register the DataSource using the JNDI entry jdbc/SequeLinkSparkyOracle.

```
//***********************************************************************
//
// This code creates a SequeLink for JDBC data source and registers it to a
// JNDI naming service. This SequeLink for JDBC data source uses the
// DataSource implementation provided by the SequeLink for JDBC Driver.
//
// If you want users to use non-pooled connections, you must modify this
// example so that it registers the SequeLink Data Source using the JNDI
// entry <jdbc/SparkyOracle>.
//
// If you want users to use pooled connections, use this example as is
// to register the SequeLink Data Source using the JNDI entry
// <jdbc/SequeLinkSparkyOracle>. Also, use the example in the next section
// to register the Connection Pool Manager's Data Source using the JNDI entry
// <jdbc/SparkyOracle>
//
//***********************************************************************

// From SequeLink for JDBC:
import com.ddtek.jdbcx.sequelink.SequeLinkDataSource;

import javax.sql.*;
import java.sql.*;
import javax.naming.*;
import javax.naming.directory.*;
import java.util.Hashtable;

public class SequeLinkDataSourceRegisterJNDI
{
    public static void main(String argv[])
    {
             try {
                    // Set up data source reference data for naming context:
                    // --------------------------------------------------
                    // Create a class instance that implements the interface
                    // ConnectionPoolDataSource
                    OracleDataSource ds = new SequeLinkDataSource();

                    ds.setDescription(
                           "Oracle on Sparky - SequeLink Data Source");
                    ds.setServerName("sparky");
                    ds.setPortNumber(19996);
                    ds.setUser("scott");
                    ds.setPassword("test");

                    // Set up environment for creating initial context
                    Hashtable env = new Hashtable();
                    env.put(Context.INITIAL_CONTEXT_FACTORY,
                           "com.sun.jndi.fscontext.RefFSContextFactory");
                    env.put(Context.PROVIDER_URL, "file:c:\\JDBCDataSource");
                    Context ctx = new InitialContext(env);

                    // Register the data source to JNDI naming service
                    ctx.bind("jdbc/SequeLinkSparkyOracle", ds);

             } catch (Exception e) {
                    System.out.println(e);
                    return;
             }
    } // Main
} // class SequeLinkDataSourceRegisterJNDI
```

4.2 Creating a Data Source Using the DataDirect Connection Pool Manager

DataDirect Connect® *for* JDBC

The following Java code example creates a data source for DataDirect Connect *for* JDBC and registers it to a JNDI naming service. The PooledConnectionDataSource class is provided by the DataDirect com.ddtek.pool package. In the following code example, the PooledConnectionDataSource object references a pooled DataDirect Connect *for* JDBC data source object. Therefore, the example performs a lookup by setting the DataSourceName attribute to the JNDI name of a registered pooled data source (in this example, jdbc/ConnectSparkyOracle, which is the DataDirect Connect *for* JDBC DataSource object created in section "4.1 Creating a DataDirect Data Source Object" on page 5).

Client applications that use this data source must perform a lookup using the registered JNDI name (jdbc/SparkyOracle in this example).

```
//************************************************************************
//
// This code creates a data source and registers it to a JNDI naming
// service. This data source uses the PooledConnectionDataSource
// implementation provided by the DataDirect com.ddtek.pool package.
//
// This data source refers to a previously registered pooled data source.
//
// This data source registers its name as <jdbc/SparkyOracle>.
// Client applications using pooling must perform a lookup for this name.
//
//************************************************************************

// From the DataDirect connection pooling package:
import com.ddtek.pool.PooledConnectionDataSource;

import javax.sql.*;
import java.sql.*;
import javax.naming.*;
import javax.naming.directory.*;
import java.util.Hashtable;

public class PoolMgrDataSourceRegisterJNDI
{
    public static void main(String argv[])
    {
             try {
             // Set up data source reference data for naming context:
             // ----------------------------------------------------
                    // Create a pooling manager's class instance that implements
                    // the interface DataSource
                    PooledConnectionDataSource ds = new PooledConnectionDataSource();

                    ds.setDescription("Sparky Oracle - Oracle Data Source");

                    // Refer to a previously registered pooled data source to access
                    // a ConnectionPoolDataSource object
                    ds.setDataSourceName("jdbc/ConnectSparkyOracle");

                    // The pool manager will be initiated with 5 physical connections
                    ds.setInitialPoolSize(5);
                    // The pool maintenance thread will make sure that there are 5
                    // physical connections available
                    ds.setMinPoolSize(5);

                    // The pool maintenance thread will check that there are no more
                    // than 10 physical connections available
                    ds.setMaxPoolSize(10);

                    // The pool maintenance thread will wake up and check the pool
                    // every 20 seconds
                    ds.setPropertyCycle(20);

                    // The pool maintenance thread will remove physical connections
                    // that are inactive for more than 300 seconds
                    ds.setMaxIdleTime(300);

                    // Set tracing off since we choose not to see output listing
                    // of activities on a connection
                    ds.setTracing(false);

                    // Set up environment for creating initial context
                    Hashtable env = new Hashtable();
                    env.put(Context.INITIAL_CONTEXT_FACTORY,
                           "com.sun.jndi.fscontext.RefFSContextFactory");
                    env.put(Context.PROVIDER_URL, "file:c:\\JDBCDataSource");
                    Context ctx = new InitialContext(env);

                    // Register the data source to JNDI naming service
                    // for application to use
                    ctx.bind("jdbc/SparkyOracle", ds);

             } catch (Exception e) {
                    System.out.println(e);
                    return;
             }

    } // Main
} // class PoolMgrDataSourceRegisterJNDI
```

DataDirect SequeLink®

The following Java code example creates a data source for JDBC and registers it to a JNDI naming service. The PooledConnectionDataSource class is provided by the DataDirect com.ddtek.pool package. In the following code example, the PooledConnectionDataSource object references a JDBC data source object. The example performs a lookup by setting the DataSourceName attribute to the JNDI name of a registered pooled data source (in this example, jdbc/SequeLinkSparkyOracle, which is the JDBC DataSource object created in section "4.1 Creating a DataDirect Data Source Object" on page 5).

Client applications that use this data source must perform a lookup using the registered JNDI name (jdbc/SparkyOracle in this example).

```
//***********************************************************************
//
// This code creates a data source and registers it to a JNDI naming
// service. This data source uses the PooledConnectionDataSource
// implementation provided by the DataDirect com.ddtek.pool package.
//
// This data source refers to a previously registered pooled data source.
//
// This data source registers its name as <jdbc/SparkyOracle>.
// Client applications using pooling must perform a lookup for this name.
//
//***********************************************************************

// From the DataDirect connection pooling package:
import com.ddtek.pool.PooledConnectionDataSource;

import javax.sql.*;
import java.sql.*;
import javax.naming.*;
import javax.naming.directory.*;
import java.util.Hashtable;

public class PoolMgrDataSourceRegisterJNDI
{
    public static void main(String argv[])
    {
             try {
                    // Set up data source reference data for naming context:
                    // ----------------------------------------------------
                    // Create a pooling manager's class instance that implements
                    // the interface DataSource
                    PooledConnectionDataSource ds = new PooledConnectionDataSource();

                    ds.setDescription("Sparky Oracle - Oracle Data Source");

                    // Refer to a previously registered pooled data source to access
                    // a ConnectionPoolDataSource object
                    ds.setDataSourceName("jdbc/SequeLinkSparkyOracle");

                    // The pool manager will be initiated with 5 physical connections
                    ds.setInitialPoolSize(5);

                    // The pool maintenance thread will make sure that there are
                    // at least 5 physical connections available
                    ds.setMinPoolSize(5);

                    // The pool maintenance thread will check that there are no more
                    // than 10 physical connections available
                    ds.setMaxPoolSize(10);

                    // The pool maintenance thread will wake up and check the pool
                    // every 20 seconds
                    ds.setPropertyCycle(20);

                    // The pool maintenance thread will remove physical connections
                    // that are inactive for more than 300 seconds
                    ds.setMaxIdleTime(300);

                    // Set tracing off since we choose not to see output listing
                    // of activities on a connection
                    ds.setTracing(false);
                    // Set up environment for creating initial context
                    Hashtable env = new Hashtable();
                    env.put(Context.INITIAL_CONTEXT_FACTORY,
                          "com.sun.jndi.fscontext.RefFSContextFactory");
                    env.put(Context.PROVIDER_URL, "file:c:\\JDBCDataSource");
                    Context ctx = new InitialContext(env);

                    // Register the data source to JNDI naming service
                    // for application to use
                    ctx.bind("jdbc/SparkyOracle", ds);

             } catch (Exception e) {
                    System.out.println(e);
                    return;
             }

    } // Main
} // class PoolMgrDataSourceRegisterJNDI
```

5 Connecting to a Data Source

Whether connection pooling is used does not affect application code. It does not require any code changes to the application because the application performs a lookup on a JNDI name of a previously registered data source. If the data source specifies a connection pooling implementation during JNDI registration (as described in section "4.2 Creating a Data Source Using the DataDirect Connection Pool Manager" on page 8), the client application benefits from faster connections through connection pooling.

DataDirect Connect® *for* JDBC

The following example shows code that can be used to look up and use a JNDI-registered data source for connections. You specify the JNDI lookup name for the data source you created (as described in section "4.2 Creating a Data Source Using the DataDirect Connection Pool Manager" on page 8).

```
//*********************************************************************
//
// Test program to look up and use a JNDI-registered data source.
//
// To run the program, specify the JNDI lookup name for the
// command-line argument, for example:
//
//           java  TestDataSourceApp  JNDI_lookup_name
//
//*********************************************************************
import javax.sql.*;
import java.sql.*;
import javax.naming.*;
import java.util.Hashtable;

public class TestDataSourceApp
{
    public static void main(String argv[])
    {
             String strJNDILookupName = "";
```

```
// Get the JNDI lookup name for a data source
int nArgv = argv.length;
if (nArgv != 1) {
      // User does not specify a JNDI lookup name for a data source,
      System.out.println(
            "Please specify a JNDI name for your data source");
      System.exit(0);
} else {
      strJNDILookupName = argv[0];
}

DataSource ds = null;
Connection con = null;
Context ctx = null;
Hashtable env = null;

long nStartTime, nStopTime, nElapsedTime;

// Set up environment for creating InitialContext object
env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.sun.jndi.fscontext.RefFSContextFactory");
env.put(Context.PROVIDER_URL, "file:c:\\JDBCDataSource");

try {
      // Retrieve the DataSource object that bound to the logical
      // lookup JNDI name
      ctx = new InitialContext(env);
      ds = (DataSource) ctx.lookup(strJNDILookupName);
} catch (NamingException eName) {
      System.out.println("Error looking up " +
            strJNDILookupName + ": " +eName);
      System.exit(0);
}

int numOfTest = 4;
int [] nCount = {100, 100, 1000, 3000};

for (int i = 0; i < numOfTest; i ++) {
      // Log the start time
      nStartTime = System.currentTimeMillis();
      for (int j = 1; j <= nCount[i]; j++) {
            // Get Database Connection
            try {
                  con = ds.getConnection("scott", "tiger");
                  // Do something with the connection
                  // ...

                  // Close Database Connection
                  if (con != null) con.close();
            } catch (SQLException eCon) {
                  System.out.println("Error getting a connection: " + eCon);
                  System.exit(0);
            } // try getConnection
      } // for j loop

      // Log the end time
      nStopTime = System.currentTimeMillis();

      // Compute elapsed time
      nElapsedTime = nStopTime - nStartTime;
      System.out.println("Test number " + i + ": looping " +
            nCount[i] + " times");
      System.out.println("Elapsed Time: " + nElapsedTime + "\n");
} // for i loop
```

```
             // All done
             System.exit(0);

    } // Main
} // TestDataSourceApp
```

NOTE: The DataDirect Connect *for* JDBC DataSource object class implements the DataSource interface for non-pooling in addition to ConnectionPoolDataSource for pooling. To use a non-pooling data source, use the JNDI name registered in the example code in section "4.1 Creating a DataDirect Data Source Object" on page 5 and run the TestDataSourceApp. For example:

```
java TestDataSourceApp jdbc/ConnectSparkyOracle
```

DataDirect SequeLink® *for* JDBC

The following example shows code that can be used to look up and use a JNDI-registered data source for connections. You specify the JNDI lookup name for the data source you created (as described in section "4.2 Creating a Data Source Using the DataDirect Connection Pool Manager" on page 8).

```
//*********************************************************************
//
// Test program to look up and use a JNDI-registered data source.
//
// To run the program, specify the JNDI lookup name for the
// command-line argument, for example:
//
//             java  TestDataSourceApp  JNDI_lookup_name
//
//*********************************************************************
import javax.sql.*;
import java.sql.*;
import javax.naming.*;
import java.util.Hashtable;

public class TestDataSourceApp
{
    public static void main(String argv[])
    {
             String str JNDILookupName = "jdbc/SparkyOracle";

             // Hard-code the JNDI entry, the application does not need to change

             DataSource ds = null;
             Connection con = null;
             Context ctx = null;
             Hashtable env = null;

             long nStartTime, nStopTime, nElapsedTime;

             // Set up environment for creating InitialContext object
             env = new Hashtable();
             env.put(Context.INITIAL_CONTEXT_FACTORY,
                    "com.sun.jndi.fscontext.RefFSContextFactory");
             env.put(Context.PROVIDER_URL, "file:c:\\JDBCDataSource");
```

```
            try {
                  // Retrieve the DataSource object that bound to the logical
                  // lookup JNDI name
                  ctx = new InitialContext(env);
                  ds = (DataSource) ctx.lookup(strJNDILookupName);
            } catch (NamingException eName) {
                  System.out.println("Error looking up " +
                        strJNDILookupName + ": " +eName);
                  System.exit(0);
            }

            int numOfTest = 4;
            int [] nCount = {100, 100, 1000, 3000};

            for (int i = 0; i < numOfTest; i ++) {
                  // Log the start time
                  nStartTime = System.currentTimeMillis();
                  for (int j = 1; j <= nCount[i]; j++) {
                        // Get Database Connection
                        try {
                              con = ds.getConnection("scott", "tiger");
                              // Do something with the connection
                              // ...

                              // Close Database Connection
                              if (con != null) con.close();
                        } catch (SQLException eCon) {
                              System.out.println("Error getting a connection: " + eCon);
                              System.exit(0);
                        } // try getConnection
                  } // for j loop

                  // Log the end time
                  nStopTime = System.currentTimeMillis();

                  // Compute elapsed time
                  nElapsedTime = nStopTime - nStartTime;
                  System.out.println("Test number " + i + ": looping " +
                        nCount[i] + " times");
                  System.out.println("Elapsed Time: " + nElapsedTime + "\n");
            } // for i loop

            // All done
            System.exit(0);

      } // Main
} // TestDataSourceApp
```

NOTE: The DataDirect SequeLink *for* JDBC DataSource object class implements the DataSource interface for non-pooling in addition to ConnectionPoolDataSource for pooling. To use non-pooled connections, modify the example in "4.1 Creating a DataDirect Data Source Object" on page 5 so that it registers the SequeLink Data Source using the JNDI entry

```
jdbc/SparkyOracle
```

You can then run the TestDataSourceApp without any modification:

```
java TestDataSourceApp
```

6. Closing the Connection Pool

To ensure that the connection pool is closed correctly when an application stops running, the application must notify the DataDirect Connection Pool Manager when it stops. If an application runs on JRE 1.3 or higher, notification occurs automatically when the application stops running. If an application runs on JRE 1.2, the application must explicitly notify the pool manager when it stops using the PooledConnectionDataSource.close method as shown in the following code:

```
if (ds instanceof com.ddtek.pool.PooledConnectionDataSource){
    com.ddtek.pool.PooledConnectionDataSource pcds =
      (com.ddtek.pool.PooledConnectionDataSource) ds;
pcds.close();
            }
```

The PooledConnectionDataSource.close method also can be used to explicitly close the connection pool while the application is running. For example, if changes are made to the pool configuration using a pool management tool, the PooledConnectionDataSource.close method can be used to force the connection pool to close and re-create the pool using the new configuration values.

7 Performance Benchmarks

We used the sample application provided in section "5 Connecting to a Data Source" on page 11 to open a connection to Oracle9i using a DataDirect Connect *for* JDBC 3.2 Oracle driver. We closed the connection at 100, 100, 1000, and 3000 iterations. We ran this sample application on a single Pentium IV 2 GHz machine with 512 MB RAM connected to an Oracle9i Server (Pentium IV 2 GHz machine with 512 MB RAM). The total elapsed time for each run was measured when connection pooling was used and was measured again when connection pooling was not used as shown in the following table:

	100 Iterations	**100 Iterations**	**1000 Iterations**	**3000 Iterations**
Pooling	547 ms	<10 ms	47 ms	31 ms[1]
Non-Pooling	4859 ms	4453 ms	43625 ms	134375 ms

When connection pooling was used, the first connection took the longest time because a new physical connection had to be created and the pool manager had to be initialized. Once the connection existed, the physical connection was placed in the pool and was reused to create a handle for each subsequent connection. You can see this by comparing the first connection (the first 100 iterations) with its subsequent connections.

NOTE: In our connection pooling example, all subsequent connections were reused because they were used for the same user and pool cleanup had not occurred.

Now, compare the pooling results at each iteration checkpoint to the non-pooling results. Clearly, connection pooling represents a significant improvement in performance.

[1] Note that the time for the 3000 iteration pooled case is faster than the 1000 iteration pooled case. This is because the Just In Time (JIT) compiler took effect at this point. If the JIT compiler is disabled, the time for the 3000 iteration pooled case increases to 94 ms, while the time for the other pooled cases remains the same.

8 Conclusion

Connection pooling provides a significant improvement on performance by reusing connections rather than creating a new connection for each connection request, without requiring changes in your JDBC application code.

FOR MORE INFORMATION

800-876-3101

info@datadirect.com

Worldwide Sales

Belgium (French)	0800 12 045
Belgium (Dutch)	0800 12 046
France	0800 911 454
Germany	0800 181 78 76
Japan	0120.20.9613
Netherlands	0800 022 0524
United Kingdom	0800 169 19 07
United States	800 876 3101

DataDirect Technologies is focused on data access, enabling software developers at both packaged software vendors and in corporate IT departments to create better applications faster. DataDirect Technologies offers the most comprehensive, proven line of data connectivity components available anywhere. Developers worldwide depend on DataDirect Technologies to connect their applications to an unparalleled range of data sources using standards-based interfaces such as ODBC, JDBC and ADO.NET, as well as cutting-edge XML query technologies. More than 250 leading independent software vendors and thousands of enterprises rely on DataDirect Technologies to simplify and streamline data connectivity. DataDirect Technologies is an operating company of Progress Software Corporation (Nasdaq: PRGS).

www.datadirect.com