Oracle® Universal Connection Pool for JDBC

Developer's Guide,

11*g* Release 1 (11.1.0.7.0)

E10788-01

August 2008

Oracle Universal Connection Pool for JDBC Developer's Guide, 11*g* Release 1 (11.1.0.7.0)

E10788-01

Primary Author: Joseph Ruzzi

Contributing Author:

Contributor: Rajkumar Irudayaraj

Contents

Preface

The Oracle Universal Connection Pool (UCP) for JDBC is a full-featured connection pool for managing database connections. Java applications that are database-intensive use the connection pool to improve performance and better utilize system resources.

The instructions in this guide detail how to use the UCP for JDBC API and cover a wide range of use cases. The guide does not provide detailed information about using Oracle JDBC Drivers, Oracle Database, or SQL except as required to understand UCP for JDBC.

Audience

This guide is primarily written for Application Developers and System Architects who want to learn how to use UCP for JDBC to create and manage database connections for their Java applications. Users must be familiar with Java and JDBC to use this guide. Knowledge of Oracle Database concepts (such as Oracle RAC and ONS) is required when using some UCP for JDBC features.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at `http://www.oracle.com/accessibility/`.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, 7 days a week. For TTY support, call 800.446.2398. Outside the United States, call +1.407.458.2479.

Related Documents

For more information about using Java with the Oracle Database, see the following documents in the Oracle Database documentation set:

- *Oracle Database JDBC Developer's Guide and Reference*
- *Oracle Database 2 Day + Java Developer's Guide*
- *Oracle Database Java Developer's Guide*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 Introduction

The following sections are included in this chapter:

- Connection Pool Overview
- Universal Connection Pool for JDBC Overview

Connection Pool Overview

A connection pool is a cache of database connection objects. The objects represent physical database connections that can be used by an application to connect to a database. At run time, the application requests a connection from the pool. If the pool contains a connection that can satisfy the request, it returns the connection to the application. If no connections are found, a new connection is created and returned to the application. The application uses the connection to perform some work on the database and then returns the object back to the pool. The connection is then available for the next connection request.

Connection pools promote the reuse of connection objects and reduce the number of times that connection objects are created. Connection pools significantly improve performance for database-intensive applications because creating connection objects is costly both in terms of time and resources. Tasks such as network communication, reading connection strings, authentication, transaction enlistment, and memory allocation all contribute to the amount of time and resources it takes to create a connection object. In addition, because the connections are already created, the application waits less time to get the connection.

Connection pools often provide properties that are used to optimize a pool's performance. The properties control behaviors such as the minimum and maximum number of connections allowed in the pool or the amount of time a connection can remain idle before it is returned to the pool. The best configured connection pools balance quick response times with the memory spent maintaining connections in the pool. It is often necessary to try different settings until the best balance is achieved for a specific application.

Benefits of Using Connection Pools

Applications that are database-intensive generally benefit the most from connection pools. As a policy, applications should use a connection pool whenever database usage is known to affect application performance.

Connection pools provide the following benefits:

- Reduces the number of times new connection objects are created.
- Promotes connection object reuse.

- Quickens the process of getting a connection.
- Reduces the amount of effort required to manually manage connection objects.
- Minimizes the number of stale connections.
- Controls the amount of resources spent on maintaining connections.

Universal Connection Pool for JDBC Overview

UCP for JDBC provides a connection pool implementation for caching JDBC connections. Java applications that are database-intensive use the connection pool to improve performance and better utilize system resources.

A UCP JDBC connection pool can use any JDBC driver to create physical connections that are then maintained by the pool. The pool can be configured and provides a full set of properties that are used to optimize pool behavior based on the performance and availability requirements of an application. For more advanced applications, UCP for JDBC provides a pool manager that can be used to manage a pool instance.

The pool also leverages many high availability and performance features available through an Oracle Real Application Clusters (RAC) database. These features include Fast Connection Failover (FCF), run-time connection load balancing, and connection affinity.

Conceptual Architecture

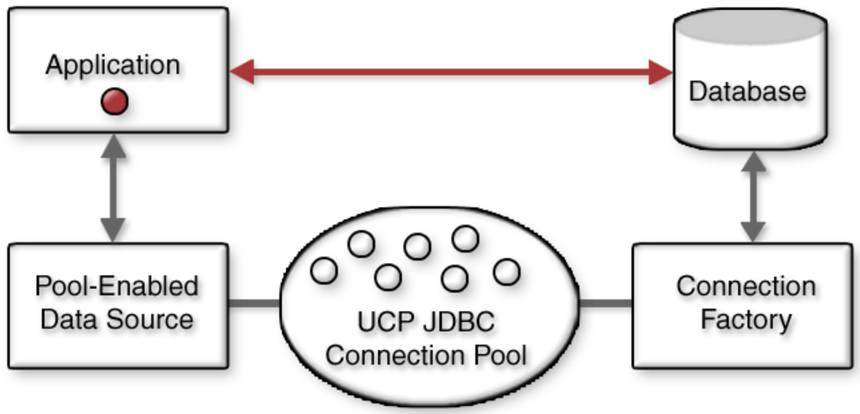
Applications use a UCP for JDBC pool-enabled data source to get connections from a UCP JDBC connection pool instance. The `PoolDataSource` data source is used for getting regular connections (`java.sql.Connection`), and the `PoolXADataSource` data source is used for getting XA connections (`javax.sql.XAConnection`). The same pool features are included in both XA and non-XA UCP JDBC connection pools.

The pool-enabled data source relies on a connection factory class to create the physical connections that are maintained by the pool. An application can choose to use any factory class that is capable of creating `Connection` or `XAConnection` objects. The pool-enabled data sources provide a method for setting the connection factory class, as well as methods for setting the database URL and database credentials that are used by the factory class to connect to a database.

Applications borrow a connection handle from the pool to perform work on a database. Once the work is completed, the connection is closed and the connection handle is returned to pool and is available to be used again. Figure 1–1 below shows the conceptual view of the interaction between an application and a UCP JDBC connection pool.

See Chapter 3, "Getting Database Connections," for more information on using pool-enabled data sources and borrowing database connections.

Figure 1–1 Conceptual View of a UCP JDBC Connection Pool



Connection Pool Properties

UCP JDBC Connection pool properties are configured through methods available on the pool-enabled data source. The pool properties are used to control the pool size, handle stale connections, and make autonomous decisions about how long connections can remain borrowed before they are returned to the pool. The optimal settings for the pool properties depend on the application and hardware resources. Typically, there is a trade-off between the time it takes for an application to get a connection versus the amount of memory it takes to maintain a certain pool size. In many cases, experimentation is required to find the optimal balance to achieve the desired performance for a specific application.

See Chapter 4, "Optimizing Connection Pool Behavior," for more information on setting connection pool properties.

Connection Pool Manager

UCP for JDBC includes a connection pool manager that is used by applications that require administrative control over a connection pool. The manager is used to explicitly control the lifecycle of a pool and to perform maintenance on a pool. The manager also provides the opportunity for an application to expose the pool and its manageability through an administrative console.

See Chapter 6, "Using the Connection Pool Manager," for more information on explicitly controlling a connection pool.

High Availability and Performance Scenarios

A UCP JDBC connection pool provides many features that are used to ensure high connection availability and performance. Many of these features, such as refreshing a pool or validating connections, are generic and work across driver and database implementations. Some of these features, such as Fast Connection Failover, run-time connection load balancing, and connection affinity, require the use of an Oracle JDBC driver and an Oracle RAC database.

See Chapter 7, "Using Oracle RAC Features," for more information on using Oracle RAC features.

2 Getting Started

The following sections are included in this chapter:

Requirements

UCP for JDBC has the following design-time and run-time requirements:

- JRE 1.5 or higher
- A JDBC diver or a connection factory class capable of returning a `java.sql.Connection` and `javax.sql.XAConnection` object

  Oracle drivers from releases 10.1 or higher are supported. Advanced Oracle Database features, such as Oracle RAC and Fast Connection Failover, require the Oracle Notification Service library (`ons.jar`) that is included with the Oracle Client software.
- The `ucp.jar` library must be included in an application's classpath.
- A database that supports SQL. Advanced features, such as Oracle RAC and Fast Connection Failover, require an Oracle Database.

Basic Connection Steps

UCP for JDBC provides a pool-enabled data source that is used by applications to borrow connections from a UCP JDBC connection pool. A connection pool is not explicitly defined for the most basic use case. Instead, a default connection pool is implicitly created when the connection is borrowed.

The following steps describe how to get a connection from a UCP for JDBC pool-enabled data source in order to access a database. The complete example is provided in Example 2–1, "Basic Connection Example":

1. Use the UCP for JDBC data source factory (`oracle.ucp.jdbc.PoolDataSourceFactory`) to get an instance of a pool-enabled data source using the `getPoolDataSource` method. The data source instance must be of the type `PoolDataSource`. For example:

   ```
   PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();
   ```

2. Set the connection properties that are required to get a physical connection to a database. These properties are set on the data source instance and include: the URL, the user name, and password to connect to the database and the connection factory used to get the physical connection. These properties are specific to a JDBC driver and database. For example:

   ```
   pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
   pds.setURL("jdbc:oracle:thin:@//localhost:1521/XE");
   pds.setUser("<user>");
   pds.setPassword("<password>");
   ```

3. Set any pool properties in order to override the connection pool's default behavior. the pool properties are set on the data source instance. For example:

   ```
   pds.setInitialPoolSize(5);
   ```

4. Get a connection using the data source instance. The returned connection is a logical handle to a physical connection in the data source's connection pool. For example:

   ```
   Connection conn = pds.getConnection();
   ```

5. Use the connection to perform some work on the database:

   ```
   Statement stmt = conn.createStatement ();
   stmt.execute("SELECT * FROM foo");
   ```

6. Close the connection and return it to the pool.

   ```
   conn.close();
   ```

Basic Connection Example

The following example is a program that connects to a database to do some work and then exits. The example is simple and in some cases not very practical; however, it does demonstrate the basic steps required to get a connection from a UCP for JDBC pooled-enabled data source in order to access a database.

Example 2–1 Basic Connection Example

```
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import oracle.ucp.jdbc.PoolDataSourceFactory;
import oracle.ucp.jdbc.PoolDataSource;

public class BasicConnectionExample {
   public static void main(String args[]) throws SQLException {
      try
      {
         //Create pool-enabled data source instance.

         PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

         //set the connection properties on the data source.

         pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
         pds.setURL("jdbc:oracle:thin:@//localhost:1521/XE");
         pds.setUser("<user>");
```

```
         pds.setPassword("<password>");

         //Override any pool properties.

         pds.setInitialPoolSize(5);

         //Get a database connection from the datasource.

         Connection conn = pds.getConnection();

         System.out.println("\nConnection obtained from " +
          "UniversalConnectionPool\n");

         //do some work with the connection.
         //Statement stmt = conn.createStatement ();
         //stmt.execute("select * from foo");

         //Close the Connection.

         conn.close();
         conn=null;

         System.out.println("Connection returned to the " +
          "UniversalConnectionPool\n");

      }
      catch(SQLException e)
      {
         System.out.println("BasicConnectionExample - " +
          "main()-SQLException occurred : "
              + e.getMessage());
      }
   }
}
```

UCP for JDBC API Overview

The following section provides a quick overview of the most commonly used packages of the UCP for JDBC API. Refer to the *Oracle Universal Connection Pool Java API Reference* for complete details on the API.

oracle.ucp.jdbc

This package includes various interfaces and classes that are used by applications to work with JDBC connections and a connection pool. Among the interfaces found in this package, the `PoolDataSource` and `PoolXADataSource` data source interfaces are used by an application to get connections as well as get and set connection pool properties. Data source instances implementing these two interfaces automatically create a connection pool.

oracle.ucp.admin

This package includes interfaces for using a connection pool manager as well as MBeans that allow users to access connection pool and the connection pool manager operations and attributes using JMX operations. Among the interfaces, the `UniversalConnectionPoolManager` interface provides methods for creating and maintaining connection pool instances.

oracle.ucp

This package includes both required and optional callback interfaces that are used to implement connection pool features. For example, the `ConnectionAffinityCallback` interface is used to create a callback that enables or disables connection affinity and can also be used to customize connection affinity behavior. As another example, the abandon connection feature returns a borrowed connection to the pool after it has been inactive for a specified amount of time. The `AbandonedConnectionTimeoutCallback` interface can be used to customize whether or not a connection should be returned to the pool after the abandonment timeout occurs and also allows a connection to be processed before it is returned to the pool.

Setting Up Logging

UCP for JDBC leverages the JDK logging facility (`java.util.logging`). Logging is not enabled by default and must be configured in order to print log messages. Logging can be configured using a log configuration file as well as through API-level configuration.

Note: The default log level is `null`. This ensures that a parent logger's log level is used by default.

Using a Logging Properties File

Logging can be configured using a properties file. The location of the properties file must be set as a Java property for the logging configuration file property. For example:

```
java -Djava.util.logging.config.file=log.properties
```

The logging properties file defines the handler to use for writing logs, the formatter to use for formatting logs, a default log level, as well as log levels for specific packages or classes. For example:

```
handlers = java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level = ALL
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

oracle.ucp.level = FINEST
oracle.ucp.jdbc.PoolDataSource = WARNING
```

A custom formatter is included with UCP for JDBC and can be entered as the value for the formatter property. For example:

```
java.util.logging.ConsoleHandler.formatter = oracle.ucp.util.logging.UCPFormatter
```

Using UCP for JDBC and JDK API

Logging can be dynamically configured though either the UCP for JDBC API or the JDK API. When using the UCP for JDBC API, logging is configured using a connection pool manager. When using the JDK, logging is configured using the `java.util.logging` implementation.

The following example demonstrates using the UCP for JDBC API to configure logging:

```
UniversalConnectionPoolManager mgr = UniversalConnectionPoolManagerImpl.
getUniversalConnectionPoolManager();
```

```
mgr.setLogLevel(Level.FINE);
```

The following example demonstrate using the JDK logging implementation directly:

```
Logger.getLogger("oracle.ucp").setLevel(Level.FINEST);
Logger.getLogger("oracle.ucp.jdbc.PoolDataSource").setLevel(Level.FINEST);
```

Supported Log Levels

The following list describes each of the log levels that are supported for JDBC. Levels lower than `FINE` produce output that may not be meaningful to users. Levels lower than `FINER` will produce very large volumes of output.

- `INTERNAL_ERROR` – Internal Errors
- `SEVERE` – SQL Exceptions
- `WARNING` – SQL Warnings and other invisible problems
- `INFO` – Public events such as connection attempts or Oracle RAC events
- `CONFIG` – SQL statements
- `FINE` – Public APIs
- `TRACE_10` – Internal events
- `FINER` – Internal APIs
- `TRACE_20` – Internal debug
- `TRACE_30` – High volume internal APIs
- `FINEST` – High volume internal debug

3 Getting Database Connections

The following sections are included in this chapter:

Borrowing Connections

An application borrows connections using a pool-enabled data source. The UCP for JDBC API provides two pool-enabled data sources; one for borrowing regular connections; and one for borrowing XA connections. These data sources provide access to UCP JDBC connection pool functionality and include a set of `getConnection` methods that are used to borrow connections. The same pool features are included in both XA and non-XA UCP JDBC connection pools.

UCP JDBC connection pools maintain both available connections and borrowed connections. A connection is reused from the pool if an application requests to borrow a connection that matches an available connection. A new connection is created if no available connection in the pool match the requested connection. The number of available connections and borrowed connections are subject to pool properties that control pool size, timeouts, and validation rules.

> **Note:** The instructions in this section use a pool-enabled data source to implicitly create and start a connection pool. See Chapter 6, "Using the Connection Pool Manager"for instructions on using the connection pool manager to explicitly create a connection pool.

Using the Pool-Enabled Data Source

UCP for JDBC provides a pool-enabled data source (`oracle.ucp.jdbc.PoolDataSource`) that is used to get connections to a database. The `oracle.ucp.jdbc.PoolDataSourceFactory` factory class provides a `getPoolDataSource()` method that creates the pool-enabled data source instance. For example:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();
```

The pool-enabled data source requires a connection factory class in order to get an actual physical connection. The connection factory is typically provided as part of a JDBC driver and can be a data source itself. A UCP JDBC connection pool can use any JDBC driver to create physical connections that are then maintained by the pool. The `setConnectionFactoryClassName(String)` method is used to define the connection factory for the pool-enabled data source instance. The following example uses Oracle's `oracle.jdbc.pool.OracleDataSource` connection factory class included with the JDBC driver. If you are using a different vendor's JDBC driver, refer to the vendor's documentation for an appropriate connection factory class.

```
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
```

In addition to the connection factory class, a pool-enabled data source requires the URL, user name, and password that is used to connect to a database. A pool-enabled data source instance includes methods to set each of these properties. The following example uses an Oracle JDBC Thin driver syntax. If you are using a different vendor's JDBC driver, refer to the vendor's documentation for the appropriate URL syntax to use.

```
pds.setURL("jdbc:oracle:thin:@//localhost:1521/XE");
pds.setUser("user");
pds.setPassword("password");
```

Note: See the *Oracle Database JDBC Developer's Guide and Reference* for detailed Oracle URL syntax usage.

Lastly, a pool-enabled data source provides a set of `getConnection` methods. The methods include:

- `getConnection()` – Returns a connection that is associated with the user name and password that was used to connect to the database.
- `getConnection(String username, String password)` – Returns a connection that is associated with the given user name and password.
- `getConnection(java.util.Properties labels)` – Returns a connection that matches a given label. See Chapter 5, "Labeling Connections," for detailed information on using connection labels.
- `getConnection(String username, String password, java.util.Properties labels)` – Returns a connection that is associated with a given user name and password and that matches a given label. See Chapter 5, "Labeling Connections," for detailed information on using connection labels.

An application uses the `getConnection` methods to borrow a connection handle from the pool that is of the type `java.sql.Connection`. If a connection handle is already in the pool that matches the requested connection (same URL, user name, and password) then it is returned to the application; or else, a new connection is created and a new connection handle is returned to the application. An example for both Oracle and MySQL are provided.

Oracle Example

The following example demonstrates borrowing a connection when using Oracle's JDBC Thin driver:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();
```

```
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL("jdbc:oracle:thin:@//localhost:1521/XE");
pds.setUser("<user>");
pds.setPassword("<password>");

Connection conn = pds.getConnection();
```

MySQL Example

The following example demonstrates borrowing a connection when using MySQL's Connector/J JDBC driver:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionFactoryClassName("com.mysql.jdbc.jdbc2.optional.
   MysqlDataSource");
pds.setURL("jdbc:mysql://host:3306/dbname");
pds.setUser("<user>");
pds.setPassword("<password>");

Connection conn = pds.getConnection();
```

Using the Pool-Enabled XA Data Source

UCP for JDBC provides a pool-enabled XA data source (`oracle.ucp.jdbc.PoolXADataSource`) that is used to get XA connections that can be enlisted in a distributed transaction. UCP JDBC XA pools have the same features as non-XA UCP JDBC pools. The `oracle.ucp.jdbc.PoolDataSourceFactory` factory class provides a `getPoolXADataSource()` method that creates the pool-enabled XA data source instance. For example:

```
PoolXADataSource  pds = PoolDataSourceFactory.getPoolXADataSource();
```

A pool-enabled XA data source instance, like a non-XA data source instance, requires the connection factory, URL, user name, and password in order to get an actual physical connection. These properties are set in the same way as a non-XA data source instance (see above). However, an XA-specific connection factory class is required to get XA connections. The XA connection factory is typically provided as part of a JDBC driver and can be a data source itself. The following example uses Oracle's `oracle.jdbc.xa.client.OracleXADataSource` XA connection factory class included with the JDBC driver. If a different vendor's JDBC driver is used, refer to the vendor's documentation for an appropriate XA connection factory class.

```
pds.setConnectionFactoryClassName("oracle.jdbc.xa.client.OracleXADataSource");
pds.setURL("jdbc:oracle:thin:@//localhost:1521/XE");
pds.setUser("user");
pds.setPassword("password");
```

Lastly, a pool-enabled XA data source provides a set of `getXAConnection` methods that are used to borrow a connection handle from the pool that is of the type `javax.sql.XAConnection`. The `getXAConnection` methods are the same as the `getConnection` methods previously described. The following example demonstrates borrowing an XA connection.

```
PoolXADataSource  pds = PoolDataSourceFactory.getPoolXADataSource();
```

```
pds.setConnectionFactoryClassName("oracle.jdbc.xa.client.OracleXADataSource");
pds.setURL("jdbc:oracle:thin:@//localhost:1521/XE");
pds.setUser("<user>");
pds.setPassword("<password>");

XAConnection conn = pds.getXAConnection();
```

Setting Connection Properties

Oracle's connection factories support properties that configure connections with specific features. UCP for JDBC pool-enabled data sources provide the `setConnectionProperties(Properties)` method, which is used to set properties on a given connection factory. The following example demonstrates setting connection properties for Oracle's JDBC driver. If you are using a different vendor's JDBC driver, refer to their documentation to check whether setting properties in this manner is supported and what properties are available:

```
Properties connProps = new Properties();
connProps.put("fixedString", false);
connProps.put("remarksReporting", false);
connProps.put("restrictGetTables", false);
connProps.put("includeSynonyms", false);
connProps.put("defaultNChar", false);
connProps.put("AccumulateBatchResult", false);

pds.setConnectionProperties(connProps);
```

The UCP JDBC connection pool does not remove connections that are already created if `setConnectionProperties` is called after the pool is created and in use.

Note: See the *Oracle Database JDBC Developer's Guide and Reference* for a detailed list of supported properties.

Using JNDI to Borrow a Connection

A connection can be borrowed from a connection pool by performing a JNDI look up for a pool-enabled data source and then calling `getConnection()` on the returned object. The pool-enabled data source must first be bound to a JNDI context and a logical name. This assumes that an application includes a Service Provider Interface (SPI) implementation for a naming and directory service where object references can be registered and located.

The following example uses Sun's file system JNDI service provider, which can be downloaded from the JNDI software download page:

`http://java.sun.com/products/jndi/downloads/index.html`

The example demonstrates creating an initial context and then performing a lookup for a pool-enabled data source that is bound to the name `MyPooledDataSource`. The object returned is then used to borrow a connection from the connection pool.

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
   "com.sun.jndi.fscontext.RefFSContextFactory");
env.put(Context.PROVIDER_URL, "file:/tmp");

ctx = new InitialContext(env);
```

```
PoolDataSource jpds = (PoolDataSource)ctx.lookup(MyPooledDataSource);
Connection conn = jpds.getConnection();
```

In the example, `MyPoolDataSource` must be bound to the context. For example:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL("jdbc:oracle:thin:@//localhost:1521/XE");
pds.setUser("<user>");
pds.setPassword("<password>");

ctx.bind(MyPooledDataSource, pds);
```

Setting Connection Pool Properties

UCP JDBC connection pools are configured using connection pool properties. The properties have `get` and `set` methods that are available through a pool-enabled data source instance. The methods are a convenient way to programmatically configure a pool. If no pool properties are set, then a connection pool uses default property values.

The following example demonstrates configuring connection pool properties. The example sets the connection pool name and the maximum/minimum number of connections allowed in the pool. See Chapter 4, "Optimizing Connection Pool Behavior," for a detailed description of all the supported properties as well as their default values.

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionPoolName("JDBC_UCP");
pds.setMinPoolSize(4);
pds.setMaxPoolSize(20);
```

UCP JDBC connection pool properties may be set in any order and can be dynamically changed at runtime. For example, `setMaxPoolSize` could be changed at any time and the pool recognizes the new value and adapts accordingly.

Validating Connections

Connections can be validated using pool properties when the connection is borrowed, and also programmatically using the `ValidConnection` interface. Both approaches are detailed in this section. Invalid connections can affect application performance and availability.

Validate When Borrowing

A connection can be validated by executing an SQL statement on a connection when the connection is borrowed from the connection pool. Two connection pool properties are used in conjunction in order to enable connection validation:

- `setValidateConnectionOnBorrow(boolean)` – Specifies whether or not connections are validated when the connection is borrowed from the connection pool. The method enables validation for every connection that is borrowed from the pool. A value of `false` means no validation is performed. The default value is `false`.

- `setSQLForValidateConnection(String)` – Specifies the SQL statement that is executed on a connection when it is borrowed from the pool.

Note: The `setSQLForValidateConnection` property is not recommended when using an Oracle JDBC driver. UCP for JDBC performs an internal ping when using an Oracle JDBC driver. The mechanism is faster than executing an SQL statement and is overridden if this property is set. Instead, set the `setValidateConnectionOnBorrow` property to `true` and do not include the `setSQLForValidateConnection` property.

The following example demonstrates validating a connection when borrowing the connection from the pool. The example uses MySQL's Connector/J JDBC driver:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionFactoryClassName("com.mysql.jdbc.jdbc2.optional.
   MysqlDataSource");
pds.setURL("jdbc:mysql://host:3306/mysql");
pds.setUser("<user>");
pds.setPassword("<password>");

pds.setValidateConnectionOnBorrow(true);
pds.setSQLForValidateConnection("select * from mysql.user");

Connection conn = pds.getConnection();
```

Checking If a Connection Is Valid

The `oracle.ucp.jdbc.ValidConnection` interface provides two methods: `isValid` and `setInvalid`. The `isValid` method returns whether or not a connection is usable and the `setInvalid` method is used to indicate that a connection should be removed from the pool instance. See "Removing Connections From the Pool" on page 3-7 for more information on using the `setInvalid` method.

The `isValid` method is used to check if a connection is still usable after an SQL exception has been thrown. This method can be used at any time to check if a borrowed connection is valid. The method is particularly useful in combination with a retry mechanism, such as the Fast Connection Failover actions that are triggered after a RAC-down event. See Chapter 7, "Using Oracle RAC Features," for more information on Fast Connection Failover.

Note: The `isValid` method checks with the pool instance and Oracle JDBC driver to determine whether a connection is still valid. The `isValid` method results in a roundtrip to the database only if both the pool and the driver report that a connection is still valid. The roundtrip is used to check for database failures that are not immediately discovered by the pool or the driver.

The `isValid` method is also helpful when used in conjunction with the connection timeout and connection harvesting features. These features may return a connection to the pool while a connection is still held by an application. In such cases, the `isValid` method returns `false`, allowing the application to get a new connection.

The following example demonstrates using the `isValid` method:

```
try
{
  conn = poolDataSouorce.getConnection
  ...
}
catch (SQLException sqlexc)
{
   if (conn == null || !((ValidConnection) conn).isValid())

   // take the appropriate action

...
conn.close
}
```

Returning Borrowed Connections

Borrowed connections that are no longer being used should be returned to the pool so that they can be available for the next connection request. The `close` method is used to close connections and automatically returns the connections to the pool. The `close` method does not physically remove the connection from the pool.

Borrowed connections that are not closed will remain borrowed; subsequent requests for a connection result in a new connection being created if no connections are available. This behavior can cause many connections to be created and can affect system performance.

The following example demonstrates closing a connection and returning it to the pool:

```
Connection conn = pds.getConnection();

//do some work with the connection.

conn.close();
conn=null;
```

Removing Connections From the Pool

The `setInvalid` method of the `ValidConnection` interface indicates that a connection should be removed from the connection pool when it is closed. The method is typically used when a connection is no longer usable, such as after an exception or if the `isValid` method of the `ValidConnection` interface returns `false`. The method can also be used if an application deems the state on a connection to be bad. The following example demonstrates using the `setInvalid` method to close and remove a connection from the pool:

```
Connection conn = pds.getConnection();
...

((ValidConnection) conn).setInvalid();
...

conn.close();
conn=null;
```

Third-Party Integration

Third-party products, such as middleware platforms or frameworks, can use UCP to provide connection pooling functionality for their applications and services. UCP integration includes the same connection pool features that are available to stand-alone applications and offers the same tight integration with the Oracle Database.

Two data source classes are available as integration points with UCP: `PoolDataSourceImpl` for non-XA connection pools and `PoolXADataSourceImpl` for XA connection pools. Both classes are located in the `oracle.ucp.jdbc` package. These classes are implementations of the `PoolDataSource` and `PoolXADataSource` interfaces, respectively, and contain default constructors. For more information on the implementation classes refer to the *Oracle Universal Connection Pool Java API Reference*.

These implementations explicitly create connection pool instances and can return connections. For example:

```
PoolXADataSource pds = new PoolXADataSourceImpl();

pds.setConnectionFactoryClassName("oracle.jdbc.xa.client.OracleXADataSource");
pds.setURL("jdbc:oracle:thin:@//localhost:1521/XE");
pds.setUser("user");
pds.setPassword("password");

XAConnection conn = pds.getXAConnection();
```

Third-party products can instantiate these data source implementation classes. In addition, the methods of these interfaces follow the JavaBean design pattern and can be used to set connection pool properties on the class using reflection. For example, a UCP data source that uses an Oralce JDBC connection factory and database might be defined as follows and loaded into a JNDI registry:

```
<data-sources>
   <data-source
      name="UCPDataSource"
      jndi-name="jdbc/UCP_DS"
      data-source-class="oracle.ucp.jdbc.PoolDataSourceImpl">
      <property name="ConnectionFactoryClassName"
                value="oracle.jdbc.pool.OracleDataSource"/>
      <property name="URL" value="jdbc:oracle:thin:@//localhost:1521:oracle"/>
      <property name="User" value"user"/>
      <property name="Password" value="password"/>
      <property name="ConnectionPoolName" value="MyPool"/>
      <property name="MinPoolSize" value="5"/>
      <property name="MaxPoolSize" value="50"/>
   </data-source>
</data-sources>
```

When using reflection, the name attribute matches (case sensitive) the name of the setter method used to set the property. An application could then use the data source as follows:

```
Connection connection = null;
try {
   InitialContext context = new InitialContext();
   DataSource ds = (DataSource) context.lookup( "jdbc/UCP_DS" );
   connection = ds.getConnection();
   ...
```

4 Optimizing Connection Pool Behavior

The following sections are included in this chapter:

Overview of Optimizing Connection Pools

This chapter provides instructions for setting connection pool properties in order to optimize pooling behavior. Upon creation, UCP JDBC connection pools are pre-configured with a default setup. The default setup provides a general, all-purpose connection pool. However, different applications may have different database connection requirements and may want to modify the default behavior of the connection pool. Behaviors, such as pool size and connection timeouts can be configured and can improve overall connection pool performance as well as connection availability. In many cases, the best way to tune a connection pool for a specific application is to try different property combinations using different values until optimal performance and throughput is achieved.

Setting Connection Pool Properties

Connection pool properties are set either when getting a connection through a pool-enabled data source or when creating a connection pool using the connection pool manager.

The following example demonstrates setting connection pool properties though a pool-enabled data source:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionPoolName("JDBC_UCP");
pds.setMinPoolSize(4);
pds.setMaxPoolSize(20);
...
```

The following example demonstrates setting connection pool properties when creating a connection pool using the connection pool manager:

```
UniversalConnectionPoolManager mgr = UniversalConnectionPoolManagerImpl.
getUniversalConnectionPoolManager();
```

```
pds.setConnectionPoolName("JDBC_UCP");
pds.setMinPoolSize(4);
pds.setMaxPoolSize(20);
...

mgr.createConnectionPool(pds);
```

> **Tip:** UCP JDBC connection pool properties may be set in any order and can be dynamically changed at runtime. For example, `setMaxPoolSize` could be changed after the pool is created and the pool recognizes the new value and adapts accordingly.

Controlling the Pool Size

UCP JDBC connection pools include a set of properties that are used to control the size of the pool. The properties allow the number of connections in the pool to increase and decrease as demand increases and decreases. This dynamic behavior helps conserve system resources that are otherwise lost on maintaining unnecessary connections.

Setting the Initial Pool Size

The initial pool size property specifies the number of available connections that are created when the connection pool is initially created or re-initialized. This property is typically used to reduce the ramp-up time incurred by priming the pool to its optimal size.

A value of `0` indicates that no connections are pre-created. The default value is `0`. The following example demonstrates configuring an initial pool size:

```
pds.setInitialPoolSize(5);
```

If the initial pool size property is greater than the maximum pool size property, then only the maximum number of connections are initialized.

If the initial pool size property is less than the minimum pool size property, then only the initial number of connections are initialized and maintained until enough connections are created to meet the minimum pool size value.

Setting the Minimum Pool Size

The minimum pool size property specifies the minimum amount of available and borrowed connections that a pool maintains. A connection pool always tries to return to the minimum pool size specified unless the minimum amount has yet to be reached. For example, if the minimum limit is set to `10` and only 2 connections are ever created and borrowed, then the number of connections maintained by the pool remains at `2`.

This property allows the number of connections in the pool to decrease as demand decreases. At the same time, the property ensures that system resources are not wasted on maintaining connections that are unnecessary.

The default value is `0`. The following example demonstrates configuring a minimum pool size:

```
pds.setMinPoolSize(2);
```

Setting the Maximum Pool Size

The maximum pool size property specifies the maximum number of available and borrowed (in use) connections that a pool maintains. If the maximum number of connections are borrowed, no connections will be available until a connection is returned to the pool.

This property allows the number of connections in the pool to increase as demand increases. At the same time, the property ensures that the pool doesn't grow to the point of exhausting a system's resources, which ultimately affects an application's performance and availability.

A value of `0` indicates that no connections are maintained by the pool. An attempt to get a connection results in an exception. The default value is to allow the pool to continue to create connections up to `Interger.MAX_VALUE` (2147483647 by default). The following example demonstrates configuring a maximum pool size:

```
pds.setMaxPoolSize(100);
```

Controlling Stale Connections

Stale connections are connections that remain either available or borrowed, but are no longer being used. Stale connections that remain borrowed may affect connection availability. In addition, stale connections may impact system resources that are otherwise wasted on maintaining unused connections for extended periods of time. The pool properties discussed in this section are used to control stale connections.

Note: It is good practice to close all connections that are no longer required by an application. Closing connections helps minimize the number of stale connections that remain borrowed.

Setting Connection Reuse

The connection reuse feature allows connections to be gracefully closed and removed from a connection pool after a specific amount of time or after the connection has been used a specific number of times. This feature saves system resources that are otherwise wasted on maintaining unusable connections.

Setting the Maximum Connection Reuse Time

The maximum connection reuse time allows connections to be gracefully closed and removed from the pool after a connection has been in use for a specific amount of time. The timer for this property starts when a connection is physically created. Borrowed connections are closed only after they are returned to the pool and the reuse time has been exceeded.

This feature is typically used when a firewall exists between the pool tier and the database tier and is setup to block connections based on time restrictions. The blocked connections remain in the pool even though they are unusable. In such scenarios, the connection reuse time is set to a smaller value than the firewall timeout policy.

Note: The maximum connection reuse time is different from the time-to-live connection timeout. The time-to-live connection timeout starts when a connection is borrowed from the pool; while, the maximum connection reuse time starts when the connection is physically created. In addition, with a time-to-live timeout, a connection is closed and returned to the pool for reuse if the timeout expires during the borrowed period. With maximum connection reuse time, a connection is closed and discarded from the pool after the timeout expires. See Setting the Time-To-Live Connection Timeout.

The maximum connection reuse time value represents seconds. A value of `0` indicates that this feature is disabled. The default value is `0`. The following example demonstrates configuring a maximum connection reuse time:

```
pds.setMaxConnectionReuseTime(300);
```

Setting the Maximum Connection Reuse Count

The maximum connection reuse count allows connections to be gracefully closed and removed from the connection pool after a connection has been borrowed a specific number of times. This property is typically used to periodically recycle connections in order to eliminate issues such as memory leaks.

A value of `0` indicates that this feature is disabled. The default value is `0`. The following example demonstrates configuring a maximum connection reuse count:

```
pds.setMaxConnectionReuseCount(100);
```

Setting the Abandon Connection Timeout

The abandoned connection timeout allows borrowed connections to be reclaimed back into the connection pool after a connection has not been used for a specific amount of time. Abandonment is determined by monitoring calls to the database. This timeout feature helps maximize connection reuse and conserves system resources that are otherwise lost on maintaining borrowed connections that are no longer in use.

Note: UCP for JDBC either cancels or rolls back connections that have local transactions pending before reclaiming connections for reuse.

The abandoned connection timeout value represents seconds. A value of `0` indicates that the feature is disabled. The default value is set to `0`. The following example demonstrates configuring an abandoned connection timeout:

```
pds.setAbandonConnectionTimeout(10);
```

Setting the Time-To-Live Connection Timeout

The time-to-live connection timeout allows borrowed connections to remain borrowed for a specific amount of time before the connection is reclaimed by the pool. This timeout feature helps maximize connection reuse and helps conserve systems resources that are otherwise lost on maintaining connections longer than their expected usage.

Note: UCP for JDBC either cancels or rolls back connections that have local transactions pending before reclaiming connections for reuse.

The time-to-live connection timeout value represents seconds. A value of `0` indicates that the feature is disabled. The default value is set to `0`. The following example demonstrates configuring a time-to-live connection timeout:

```
pds.setTimeToLiveConnectionTimeout(18000)
```

Setting the Connection Wait Timeout

The connection wait timeout specifies how long an application request waits to obtain a connection if there are no longer any connections in the pool. A connection pool runs out of connections if all connections in the pool are being used (borrowed) and if the pool size has reached it's maximum connection capacity as specified by the maximum pool size property. The request receives an SQL exception if the timeout value is reached. The application can then retry getting a connection. This timeout feature improves overall application usability by minimizing the amount of time an application is blocked and provides the ability to implement a graceful recovery.

The connection wait timeout value represents seconds. A value of `0` indicates that the feature is disabled. The default value is set to `3` seconds. The following example demonstrates configuring a connection wait timeout:

```
pds.setConnectionWaitTimeout(10);
```

Setting the Inactive Connection Timeout

The inactive connection timeout specifies how long an available connection can remain idle before it is closed and removed from the pool. This timeout property is only applicable to available connections and does not affect borrowed connections. This property helps conserve resources that are otherwise lost on maintaining connections that are no longer being used. The inactive connection timeout (together with the maximum pool size) allows a connection pool to grow and shrink as application load changes.

The inactive connection timeout value represents seconds. A value of `0` indicates that the feature is disabled. The default value is set to `0`. The following example demonstrates configuring an inactive connection timeout:

```
pds.setInactiveConnectionTimeout(60);
```

Setting the Timeout Check Interval

The timeout check interval property controls how frequently the timeout properties (abandoned connection timeout, time-to-live connection timeout, and inactive connection timeout) are enforced. Connections that have timed-out are reclaimed when the timeout check cycle runs. This means that a connection may not actually be reclaimed by the pool at the moment that the connection times-out. The lag time between the connection timeout and actually reclaiming the connection may be considerable depending on the size of the timeout check interval.

The timeout check interval property represents seconds. The default value is set to `30`. The following example demonstrates configuring a property check interval:

```
pds.setTimoutCheckInterval(60);
```

Harvesting Connections

The connection harvesting feature allows a specified number of borrowed connections to be reclaimed when the connection pool reaches a specified number of available connections. This feature helps ensure that a certain number of connections are always available in the pool and helps maximize performance. The feature is particularly useful if an application caches connection handles. Caching is typically performed for performance reasons because it minimizes re-initialization of state necessary for connections to participate in a transaction.

For example, a connection is borrowed from the pool, initialized with necessary session state, and then held in a context object. Holding connections in this manner may cause the connection pool to run out of available connections. The connection harvest feature reclaims the borrowed connections, if appropriate, and allows the connections to be reused.

Connection harvesting is controlled using the `HarvestableConnection` interface and configured/enabled using two pool properties: Connection Harvest Trigger Count and Connection Harvest Maximum Count. The interface and properties are used together when implementing the connection harvest feature.

Setting Whether a Connection is Harvestable

The `setConnectionHarvestable(boolean)` method of the `oralce.ucp.jdbc.HarvestableConnection` interface controls whether or not a connection will be harvested. This method is used as a locking mechanism when connection harvesting is enabled. For example, the method is set to `false` on a connection when the connection is being used within a transaction and must not be harvested. After the transaction completes, the method is set to `true` on the connection and the connection can be harvested if required.

Note: All connections are harvestable, by default, when the connection harvest feature is enabled. If the feature is enabled, the `setConnectionHarvestable` method should always be used to explicitly control whether a connection is harvestable.

The following example demonstrates using the `setConnectionHarvestable` method to indicate that a connection is not harvestable when the connection harvest feature attempts to harvest connections:

```
Connection conn = pds.getConnection();

((HarvestableConnection) conn).setConnectionHarvestable(false);
```

Setting the Harvest Trigger Count

The connection harvest trigger count specifies the available connection threshold that triggers connection harvesting. For example, if the connection harvest trigger count is set to 10, then connection harvesting is triggered when the number of available connections in the pool drops to 10.

A value of `Interger.MAX_VALUE` (2147483647 by default) indicates that connection harvesting is disabled. The default value is `Interger.MAX_VALUE`.

The following example demonstrates enabling connection harvesting by configuring a connection harvest trigger count.

```
pds.setConnectionHarvestTriggerCount(2);
```

Setting the Harvest Maximum Count

The connection harvest maximum count property specifies how many borrowed connections should be returned to the pool once the harvest trigger count has been reached. The number of connections actually harvested may be anywhere from 0 to the connection harvest maximum count value. Least recently used connections are harvested first which allows very active user sessions to keep their connections the most.

The harvest maximum count value can range from `0` to the maximum connection property value. The default value is `1`. An SQLException is thrown if an out-of-range value is specified.

The following example demonstrates configuring a connection harvest maximum count.

```
pds.setConnectionHarvestMaxCount(5);
```

Caching SQL Statements

Statement caching makes working with statements more efficient. Statement caching improves performance by caching executable statements that are used repeatedly and makes it unnecessary for programmers to explicitly reuse prepared statements. Statement caching eliminates overhead due to repeated cursor creation, repeated statement parsing and creation and reduces overhead of communication between applications and the database. Statement caching and reuse is transparent to an application. Each statement cache is associated with a physical connection. That is, each physical connection will have its own statement cache.

The match criteria for cached statements are as follows:

- The SQL string in the statement must be the same (case-sensitive) to one in the cache.
- The statement type must be the same (`prepared` or `callable`) to the one in the cache.
- The scrollable type of result sets produced by the statement must be the same (`forward-only` or `scrollable`) as the one in the cache.

Statement caching is implemented and enabled differently depending on the JDBC driver vendor. The instructions in this section are specific to Oracle's JDBC driver. Statement caching on other vendors' drivers can be configured by setting a connection property on a connection factory. See "Setting Connection Properties" on page 3-4 for information on setting connection properties. In addition, refer to the JDBC vendor's documentation to determine whether statement caching is supported and if it can be set as a connection property. UCP for JDBC does support JDBC 4.0 (JDK16) APIs to enable statement pooling if a JDBC vendor supports it.

Enabling Statement Caching

The maximum number of statements property specifies the number of statements to cache for each connection. The property only applies to the Oracle JDBC driver. If the property is not set, or if it is set to `0`, then statement caching is disabled. By default, statement caching is disabled. When statement caching is enabled, a statement cache is associated with each physical connection maintained by the connection pool. A single statement cache is not shared across all physical connections.

The following example demonstrates enabling statement caching:

```
pds.setMaxStatements(10);
```

Determining the Statement Cache Size

The cache size should be set to the number of distinct statements the application issues to the database. If the number of statements that an application issues to the database is unknown, use the JDBC performance metrics to assist with determining the statement cache size.

Statement Cache Size Resource Issues

Each connection is associated with its own statement cache. Statements held in a connection's statement cache may hold on to database resources. It is possible that the number of opened connections combined with the number of cached statements for each connection could exceed the limit of open cursors allowed for the database. This issue may be avoided by reducing the number of statements allowed in the cache, or by increasing the limit of open cursors allowed by the database.

5 Labeling Connections

The following sections are included in this chapter:

Labeling Connections Overview

Applications often initialize connections retrieved from a connection pool before using the connection. The initialization varies and could include simple state re-initialization that requires method calls within the application code or database operations that require round trips over the network. The cost of such initialization may be significant.

Labeling connections allows an application to attach arbitrary name/value pairs to a connection. The application can request a connection with the desired label from the connection pool. By associating particular labels with particular connection states, an application can retrieve an already initialized connection from the pool and avoid the time and cost of re-initialization. The connection labeling feature does not imposes any meaning on user-defined keys or values; the meaning of user-defined keys and values is defined solely by the application.

Some of the examples for connection labeling include, role, NLS language settings, transaction isolation levels, stored procedure calls, or any other state initialization that is expensive and necessary on the connection before work can be executed by the resource.

Connection labeling is application-driven and requires the use of two interfaces. The `oracle.ucp.jdbc.LabelableConnection` interface is used to apply and remove connection labels, as well as retrieve labels that have been set on a connection. The `oracle.ucp.ConnectionLabelingCallback` interface is used to create a labeling callback that determines whether or not a connection with a requested label already exists. If no connections exist, the interface allows current connections to be configured as required. The methods of these interfaces are described in detail throughout this chapter.

Implementing a Labeling Callback

A labeling callback is used to define how the connection pool selects labeled connections and allows the selected connection to be configured before returning it to an application. Applications that use the connection labeling feature must provide a callback implementation.

A labeling callback is used when a labeled connection is requested but there are no connections in the pool that match the requested labels. The callback determines which connection requires the least amount of work in order to be re-configured to match the requested label and then allows the connection's labels to be updated before returning the connection to the application.

Creating a Labeling Callback

To create a labeling callback, an application implements the `oracle.ucp.ConnectionLabelingCallback` interface. One callback is created per connection pool. The interface provides two methods as shown below:

```
public int cost(Properties requestedLabels, Properties currentLabels);

public boolean configure(Properties requestedLabels, Connection conn);
```

- `cost` – This method projects the cost of configuring connections considering label-matching differences. Upon a connection request, the connection pool uses this method to select a connection with the least configuration cost.
- `configure` – This method is called by the connection pool on the selected connection before returning it to the application. The method is used to set the state of the connection and apply or remove any labels to/from the connection.

The connection pool iterates over each connection available in the pool. For each connection, it calls the `cost` method. The result of the `cost` method is an `integer` which represents an estimate of the cost required to reconfigure the connection to the required state. The larger the value, the costlier it is to reconfigure the connection. The connection pool always returns connections with the lowest cost value. The algorithm is as follows:

- If the `cost` method returns `0` for a connection, the connection is a match. The connection pool calls `configure` on the connection found and returns the connection.
- If the `cost` method returns a value greater than `0`, then the connection pool iterates until it finds a connection with a cost value of `0` or runs out of available connections.
- If the pool has iterated through all available connections and the lowest cost of a connection is `Integer.MAX_VALUE` (2147483647 by default), then no connection in the pool is able to satisfy the connection request. The pool creates and returns a new connection. If the pool has reached the maximum pool size (it cannot create a new connection), then the pool either throws an SQL exception or waits if the connection wait timeout attribute is specified.
- If the pool has iterated through all available connections and the lowest cost of a connection is less than `Integer.MAX_VALUE`, then the `configure` method is called on the connection and the connection is returned. If multiple connections are less than `Integer.MAX_VALUE`, the connection with the lowest cost is returned.

Note: A cost of `0` does not imply that `requestedLabels` equals `currentLabels`.

An Example Labeling Callback

The following example demonstrates a simple labeling callback implementation that implements both the `cost` and `configure` methods. The callback is used to find a labeled connection that is initialized with a specific transaction isolation level.

```
class MyConnectionLabelingCallback
  implements ConnectionLabelingCallback {

  public MyConnectionLabelingCallback()
  {
  }

  public int cost(Properties reqLabels, Properties currentLabels)
  {
    // Case 1: exact match
    if (reqLabels.equals(currentLabels))
    {
      System.out.println("## Exact match found!! ##");
      return 0;
    }

    // Case 2: some labels match with no unmatched labels
    String iso1 = (String) reqLabels.get("TRANSACTION_ISOLATION");
    String iso2 = (String) currentLabels.get("TRANSACTION_ISOLATION");
    boolean match =
      (iso1 != null && iso2 != null && iso1.equalsIgnoreCase(iso2));
    Set rKeys = reqLabels.keySet();
    Set cKeys = currentLabels.keySet();
    if (match && rKeys.containsAll(cKeys))
    {
      System.out.println("## Partial match found!! ##");
      return 10;
    }

    // No label matches to application's preference.
    // Do not choose this connection.
    System.out.println("## No match found!! ##");
    return Integer.MAX_VALUE;
  }

  public boolean configure(Properties reqLabels, Object conn)
  {
    try
    {
      String isoStr = (String) reqLabels.get("TRANSACTION_ISOLATION");
      ((Connection)conn).setTransactionIsolation(Integer.valueOf(isoStr));
      LabelableConnection lconn = (LabelableConnection) conn;

      // Find the unmatched labels on this connection
      Properties unmatchedLabels =
       lconn.getUnmatchedConnectionLabels(reqLabels);

      // Apply each label <key,value> in unmatchedLabels to conn
      for (Map.Entry<Object, Object> label : unmatchedLabels.entrySet())
      {
        String key = (String) label.getKey();
        String value = (String) label.getValue();
        lconn.applyConnectionLabel(key, value);
      }
    }
    catch (Exception exc)
    {
      return false;
    }
    return true;
  }
}
```

Registering a Labeling Callback

A pool-enabled data source provides the `registerConnectionLabelingCallback(ConnectionLabelingCallback callback)` method for registering labeling callbacks. Only one callback may be registered on a connection pool. The following example demonstrates registering a labeling callback that is implemented in the `MyConnectionLabelingCallback` class:

```
MyConnectionLabelingCallback callback = new MyConnectionLabelingCallback();
pds.registerConnectionLabelingCallback( callback );
```

Removing a Labeling Callback

A pool-enabled data source provides the `removeConnectionLabelingCallback()` method for removing a labeling callback. The following example demonstrates removing a labeling callback.

```
pds.removeConnectionLabelingCallback( callback );
```

Applying Connection Labels

Labels are applied on a borrowed connection using the `applyConnectionLabel` method from the `LabelableConnection` interface. This method is typically called from the `configure` method of the labeling callback. Any number of connection labels may be cumulatively applied on a borrowed connection. Each time a label is applied to a connection, the supplied key/value pair is added to the collection of labels already applied to the connection. Only the last applied value is retained for any given key.

Note: A labeling callback must be registered on the connection pool before a label can be applied on a borrowed connection; otherwise, an exception is thrown. See "Implementing a Labeling Callback" on page 5-2.

The following example demonstrates initializing a connection with a transaction isolation level and then applying a label to the connection:

```
String pname = "property1";
String pvalue = "value";
Connection conn = pds.getConnection();
```

```
// itialize the connection as required.

conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);

((LabelableConnection) conn).applyConnectionLabel(pname, pvalue);
```

In order to remove a given key from the set of connection labels applied, apply a label with the key to be removed and a `null` value. This may be used to clear a particular key/value pair from the set of connection labels.

Borrowing Labeled Connections

A pool-enabled data source provides two `getConnection` methods that are used to borrow a labeled connection from the pool. The methods are shown below:

```
public Connection getConnection(java.util.Properties labels )
   throws SQLException;

public Connection getConnection( String user, String password,
                                 java.util.Properties labels )
   throws SQLException;
```

The methods require that the label be passed to the `getConnection` method as a `Properties` object. The following example demonstrates getting a connection with the label `property1, value`.

```
String pname = "property1";
String pvalue = "value";
Properties label = new Properties();
label.setProperty(pname, pvalue);

Connection conn = pds.getConnection(label);
```

Checking Unmatched Labels

A connection may have multiple labels that each uniquely identifies the connection based on some desired criteria. The `getUnmatchedConnectionLabels` method is used to verify which connection labels matched from the requested labels and which did not. The method is used after a connection with multiple labels is borrowed from the connection pool and is typically used by a labeling callback. The following example demonstrates checking for unmatched labels.

```
String pname = "property1";
String pvalue = "value";
Properties label = new Properties();
label.setProperty(pname, pvalue);

Connecion conn = pds.getConnection(label);
Properties unmatched = ((LabelableConnection)
   connection).getUnmatchedConnectionLabels (label);
```

Removing a Connection Label

The `removeConnectionLabel` method is used to remove a label from a connection. This method is used after a labeled connection is borrowed from the connection pool. The following example demonstrates removing a connection label.

```
String pname = "property1";
String pvalue = "value";
Properties label = new Properties();
label.setProperty(pname, pvalue);
Connection conn = pds.getConnection(label);
((LabelableConnection) conn).removeConnectionLabel(pname);
```

6 Using the Connection Pool Manager

The following sections are included in this chapter:

Connection Pool Manager Overview

Applications use a connection pool manager to explicitly create and manage UCP JDBC connection pools. Applications use the manager because it offers full lifecycle control, such as creating, starting, stopping, and destroying a connection pool. Applications also use the manager to perform routine maintenance on the connection pool, such as refreshing, recycling, and purging connections in a pool. Lastly, applications use the connection pool manager because it offers a centralized integration point for administrative tools and consoles.

Creating a Connection Pool Manager

A connection pool manager is an instance of the `UniversalConnectionPoolManager` interface, which is located in the `oracle.ucp.admin` package. The manager is a Singleton instance that is used to manage multiple connection pools per JVM. The interface includes methods for interacting with a connection pool manager. UCP for JDBC includes an implementation that is used to get a connection pool manager instance. The following example demonstrates creating a connection pool manager instance using the implementation:

```
UniversalConnectionPoolManager mgr = UniversalConnectionPoolManagerImpl.
getUniversalConnectionPoolManager();
```

Controlling the Lifecycle of a Connection Pool

Applications use the connection pool manager to explicitly control the lifecycle of connection pools. The manager is used to create, start, stop, and destroy connection pools. Lifecycle methods are included as part of the `UniversalConnectionPoolManager` interface.

Understanding Lifecycle States

A connection pool's lifecycle states affects what manager operations can be performed on a connection pool. Applications that explicitly control a pool's lifecycle must ensure that the manager's operations are used only when the pool is in an appropriate state. Lifecycle constraints are discussed throughout this section.

The following list describes a pool's lifecycle states:

- Starting – Indicates that the connection pool's start method has been called and it is in the process of starting up.
- Running – Indicates that the connection pool has been started and is ready to give out connections.
- Stopping – Indicates that the connection pool is in the process of stopping.
- Stopped – Indicates that the connection pool is stopped.
- Failed – Indicates that the connection pool has encountered failures during starting, stopping, or execution.

Creating a Connection Pool

The manager's `CreateConnectionPool` method creates and registers a connection pool. The manager uses a connection pool adapter to create the pool and relies on a pool-enabled data source to configure the pool's properties. A connection pool name must be defined as part of the configuration and provides a way to refer to specific pools when interacting with the manager. A connection pool name must be unique and cannot be used by more than one connection pool.

The following example demonstrates creating a connection pool instance when using a manager:

```
UniversalConnectionPoolManager mgr = UniversalConnectionPoolManagerImpl.
getUniversalConnectionPoolManager();

PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();
pds.setConnectionPoolName("mgr_pool");
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL("jdbc:oracle:thin:@//localhost:1521/XE");
pds.setUser("<user>");
pds.setPassword("<password>");

mgr.createConnectionPool((UniversalConnectionPoolAdapter)pds);
```

It is good practice to use the `createConnectionPool` method to explicitly create a connection pool when using a pool manager. However, an application does not have to use the manager to create a pool in order for the pool to be managed. A pool that is implicitly created (that is, automatically created when using a pool-enabled data source) and configured with a pool name is automatically registered and managed by the pool manager.

Note: The manager throws an exception if a connection pool already exists with the same name. An application must not implicitly start a connection pool before using the `createConnectionPool` method to explicitly create the same pool.

Starting a Connection Pool

The manager's `startConnectionPool` method starts a connection pool using the pool name as a parameter to determine which pool to start. The pool name is defined as a pool property on a pool-enabled data source.

The following example demonstrates starting a connection pool:

```
mgr.startConnectionPool("mgr_pool");
```

An application must always create a connection pool using the manager's `createConnectionPool` method prior to starting the pool. In addition, a lifecycle state exception occurs if an application attempts to start a pool that has been previously started or if the pool is in a state other than stopped or failed.

Stopping a Connection Pool

The manager's `stopConnectionPool` method stops a connection pool using the pool name as a parameter to determine which pool to stop. The pool name is defined as a pool property on the pool-enabled data source. Stopping a connection pool closes all available and borrowed connections.

The following example demonstrates stopping a connection pool:

```
mgr.stopConnectionPool("mgr_pool");
```

An application can use the manager to stop a connection pool that was started implicitly or explicitly. An error occurs if an application attempts to stop a pool that does not exist or if the pool is in a state other than started or starting.

Destroying a Connection Pool

The manager's `destroyConnectionPool` method stops a connection pool and removes it from the connection pool manager. A pool name is used as a parameter to determine which pool to destroy. The pool name is defined as a pool property on the pool-enabled data source.

The following example demonstrates destroying a connection pool:

```
mgr.destroyConnectionPool("mgr_pool");
```

An application cannot start a connection pool that has been destroyed and must explicitly create and start a new connection pool.

Performing Maintenance on a Connection Pool

Applications use the connection pool manager to perform maintenance on a connection pool. Maintenance includes refreshing, recycling, and purging a connection pool. The maintenance methods are included as part of the `UniversalConnectionPoolManager` interface.

Maintenance is typically performed to remove and replace invalid connections and ensures a high availability of valid connections. Invalid connections typically cannot be used to connect to a database but are still maintained by the pool. These connections waste system resources and directly affect a pool's maximum connection limit. Ultimately, too many invalid connections negatively affects an applications performance.

Note: Applications can check whether or not a connection is valid when borrowing the connection from the pool. See "Validating Connections" on page 3-5 for detailed information. If an application consistently has a high number of invalid connections, additional testing should be performed to determine the cause.

Refreshing a Connection Pool

Refreshing a connection pool replaces every connection in the pool with a new connection. Any connections that are currently borrowed are marked for removal and refreshed after the connection is returned to the pool. The manager's `refreshConnectionPool` method refreshes a connection pool using the pool name as a parameter to determine which pool to refresh. The pool name is defined as a pool property on the pool-enabled data source.

The following example demonstrates refreshing a connection pool:

```
mgr.refreshConnectionPool("mgr_pool");
```

Recycling a Connection Pool

Recycling a connection pool replaces only invalid connection in the pool with a new connection and does not replace borrowed connections. The manager's `recycleConnectionPool` method recycles a connection pool using the pool name as a parameter to determine which pool to recycle. The pool name is defined as a pool property on the pool-enabled data source.

The `setSQLForValidateConnection` property must be set when using non-Oracle drivers. UCP for JDBC uses this property to determine whether or not a connection is valid before recycling the connection. See "Validating Connections" for more information on using the `setSQLForValidateConnection` property.

The following example demonstrates recycling a connection pool:

```
mgr.recycleConnectionPool("mgr_pool");
```

Purging a Connection Pool

Purging a connection pool removes every connection (available and borrowed) from the connection pool and leaves the connection pool empty. Subsequent requests for a connection result in a new connection being created. The manager's `purgeConnectionPool` method purges a connection pool using the pool name as a parameter to determine which pool to purge. The pool name is defined as a pool property on the pool-enabled data source.

The following example demonstrates purging a connection pool:

```
mgr.purgeConnectionPool("mgr_pool");
```

Note: Connection pool properties, such as `minPoolSize` and `initialPoolSize`, may not be enforced after a connection pool is purged.

---

7 Using Oracle RAC Features

The following sections are included in this chapter:

Overview of Oracle RAC Features

UCP JDBC connection pools provide a tight integration with various Oracle Real Application Clusters (RAC) Database features. The features include Fast Connection Failover (FCF), Run-Time Connection Load Balancing, and Connection Affinity. These features require the use of an Oracle JDBC driver, Oracle RAC database, and the Oracle Notification Service library (`ons.jar`) that is included with the Oracle Client software. For those new to these technologies, refer to the *Oracle Real Application Clusters Administration and Deployment Guide* and the *Oracle Database JDBC Developer's Guide and Reference*.

Applications use Oracle RAC features to maximize connection performance and availability and to mitigate down-time due to connection problems. Applications have different availability and performance requirements and should implement Oracle RAC features accordingly.

Generic High Availability and Performance Features

The UCP for JDBC APIs and connection pool properties include many high availability and performance features that do not require an Oracle RAC database. These features work well with both Oracle and non-Oracle connections and are discussed throughout this guide. For example: validating connections on borrow; setting timeout properties; setting maximum reuse properties; and connection pool manager operations are all used to ensure a high-level of connection availability and optimal performance.

Note: Generic high availability and performance features work slightly better when using Oracle connections because UCP for JDBC leverages Oracle JDBC internal APIs.

Using Fast Connection Failover

The Fast Connection Failover (FCF) feature is an Oracle RAC/Fast Application Notification (FAN) client implemented through the connection pool. The feature requires the use of an Oracle JDBC driver and an Oracle RAC database. This section only describes the steps that an application must perform when using FCF. For more information on setting up an Oracle RAC database, see the *Oracle Real Application Clusters Administration and Deployment Guide* or consult an Oracle database administrator. For general information about fast connection failover, see the Fast Connection Failover chapter in the *Oracle Database JDBC Developer's Guide and Reference*.

FCF manages pooled connections for high availability and provides the following features:

- FCF supports unplanned outages. Dead connections are rapidly detected and then the connections are aborted and removed from the pool. Connection removal relies on abort to rapidly sever socket connections in order to prevent hangs. Borrowed and in-use connections are interrupted only for unplanned outages.
- FCF supports planned outages. Borrowed or in-use connections are not interrupted and closed until work is done and control of the connection is returned to the pool.
- FCF encapsulates fatal connection errors and exceptions into the `isValid` API for robust and efficient retries. See "Checking If a Connection Is Valid" on page 3-6 for more information on using this API.
- FCF recognizes new nodes that join an Oracle RAC cluster and places new connections on that node appropriately in order to deliver maximum quality of service to applications at run-time. This facilitates middle-tier integration of Oracle RAC node joins and work-request routing from the application tier.
- FCF distributes runtime work requests to all active Oracle RAC instances.

Unplanned Shutdown Scenarios

FCF supports unplanned shutdown scenarios by detecting and removing stale connections to an Oracle RAC cluster. Stale connections include connections that do not have a service available on any instance in an Oracle RAC cluster due to service-down and node-down events. Borrowed connections and available connections that are stale are detected, and their network connection is severed before removing them from the pool. These removed connections are not replaced by the pool. Instead, the application must retry connections before performing any work with a connection.

Note: Borrowed connections are immediately aborted and closed during unplanned shutdown scenarios. Any on-going transactions immediately receive an exception.

Planned Shutdown Scenarios

FCF supports planned shutdown scenarios where an Oracle RAC service can be gracefully shutdown. In such scenarios, stale borrowed connections are marked and will only be aborted and removed after they are returned to the pool. Any on-going transactions do not see any difference and proceed to complete.

The primary difference between unplanned and planned shutdown scenarios is how borrowed connections are handled. Stale connections that are idle in the pool (not borrowed) are removed in the same manner as the unplanned shutdown scenario.

Oracle RAC Instance Rejoin and New Instance Scenarios

FCF supports scenarios where an Oracle RAC cluster adds instances that provide a service of interest. The instance may be new to the cluster or may have been restarted after a down event. In both cases, UCP for JDBC recognizes the new instance and creates connections to the node as required.

Example Fast Connection Failover Configuration

The following example demonstrates a connection pool that uses the FCF feature. FCF is configured through a pool-enabled data source. The example includes enabling FCF, configuring the Oracle Notification Service (ONS) and configuring a connection URL. These topics are discussed after the example.

Example 7–1 Fast Connection Failover Configuration Example

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionPoolName("FCFSamplePool");
pds.setFastConnectionFailoverEnabled(true);
pds.setONSConfiguration("nodes=racnode1:4200,racnode2:4200\nwalletfile=
/oracle11/onswalletfile");
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL("jdbc:oracle:thin@(DESCRIPTION=
   (LOAD_BALANCE=on)
   (ADDRESS=(PROTOCOL=TCP)(HOST=racnode1) (PORT=1521))
   (ADDRESS=(PROTOCOL=TCP)(HOST=racnode2) (PORT=1521))
   (CONNECT_DATA=(SERVICE_NAME=service_name)))");
...
```

The `isValid` method of the `oracle.ucp.jdbc.ValidConnection` interface is typically used in conjunction with the FCF feature and is used to check if a borrowed connection is still usable after an SQL exception has been thrown due to a Oracle RAC down event. For example:

```
try
{
  conn = pds.getConnection
  ...
}
catch (SQLException sqlexc)
{
   if (conn == null || !((ValidConnection) conn).isValid())

   // take the appropriate action

...
conn.close
}
```

For more information on the `ValidConnection` interface, see "Checking If a Connection Is Valid" on page 3-6.

Enabling Fast Connection Failover

The FCF pool property is used to enable and disable FCF. FCF is disabled by default. The following example demonstrates enabling FCF as shown in Example 7–1.

```
pds.setFastConnectionFailoverEnabled(true);
```

Note: FCF must also be enabled to use run-time connection load balancing and connection affinity. These features are discussed later in this chapter.

Configuring ONS

FCF relies on the Oracle Notification Service (ONS) to propagate database events between the connection pool and the Oracle RAC database. At run-time, the connection pool must be able to setup an ONS environment. ONS (`ons.jar`) is included as part of the Oracle Client software. ONS can be configured using either remote configuration or client-side ONS daemon configuration. Remote configuration is the preferred configuration for standalone client applications.

Remote Configuration

UCP for JDBC supports remote configuration of ONS through the `SetONSConfiguration` pool property. The ONS property value is a string that closely resembles the content of an `ons.config` file. The string contains a list of `name=value` pairs separated by a new line character (`\n`). The name can be: `nodes`, `walletfile`, or `walletpassword`. The parameter string should at least specify the ONS configuration nodes attribute as a list of `host:port` pairs separated by a comma. SSL would be used when the `walletfile` attribute is specified as an Oracle wallet file.

The following example demonstrates an ONS configuration string as shown in Example 7–1:

```
...
pds.setONSConfiguration("nodes=racnode1:4200,racnode2:4200\nwalletfile=
/oracle11/onswalletfile");
...
```

Applications that use remote configuration must set the `oracle.ons.oraclehome` system property to the location of `ORACLE_HOME` before starting the application. For example:

```
java -Doracle.ons.oraclehome=$ORACLE_HOME ...
```

Note: The parameters in the configuration string must match those for the Oracle RAC database. In addition, the `setONSConfiguration` property is only used for standalone Java applications. When using Oracle Application Server, ONS should be configured using procedures that are applicable to the server.

Client-Side Daemon Configuration

Client-Side ONS daemon configuration is typical of applications that run on a middle tier server such as the Oracle Application Server. Clients in this scenario directly configure ONS by updating the `ons.config` file. The location of the file may be different depending on the platform. The following example demonstrates an `ons.config` file for Example 7–1:

```
localport=4100
remoteport=4200
nodes=racnode1:4200,racnode2:4200
walletfile=/oracle11/onswalletfile
```

- `localport`: The port that ONS binds to on the localhost interface to talk to local clients.
- `remoteport`: the port that ONS binds to on all interfaces for talking to other ONS daemons.

The ONS utility (`onsctl`) can be used to start, stop, ping, and refresh ONS and can also be used to debug ONS. ONS must be refreshed after updating the `ons.config` file.

For more information on setting up ONS, refer to the following links:

- Oracle Application Server 10g Fast Connection Failover Configuration Guide
- The Fast Connection Failover chapter in the *Oracle Database JDBC Developer's Guide and Reference*

Configuring the Connection URL

A connection factory's connection URL must use the service name syntax when using FCF. The service name is used to map the connection pool to the service. In addition, the factory class must be an Oracle factory class. The following example demonstrates configuring the connection URL as shown in Example 7–1:

```
...
pds.setConnectionFactoryClassName("oracle.jdbc.pool.OracleDataSource");
pds.setURL("jdbc:oracle:thin@//host:port/service_name");
...
```

Note: An exception is thrown if a service identifier (SID) is specified for the connection URL when FCF is enabled.

The following examples demonstrate valid connection URL syntax when connecting to an Oracle RAC database. Examples for both the Oracle JDBC thin and Oracle OCI driver are included. Notice that the URL can be used to explicitly enable load balancing among Oracle RAC nodes:

Valid Connection URL Usage

```
pds.setURL("jdbc:oracle:thin@//host:port/service_name");

pds.setURL("jdbc:oracle:thin@//cluster-alias:port/service_name");

pds.setURL("jdbc:oracle:thin:@(DESCRIPTION=
   (LOAD_BALANCE=on)
   (ADDRESS=(PROTOCOL=TCP)(HOST=host1)(PORT=1521))
   (ADDRESS=(PROTOCOL=TCP)(HOST=host2)(PORT=1521))
   (CONNECT_DATA=(SERVICE_NAME=service_name)))");

pds.setURL("jdbc:oracle:thin:@(DESCRIPTION=
   (ADDRESS=(PROTOCOL=TCP)(HOST=cluster_alias) (PORT=1521))
   (CONNECT_DATA=(SERVICE_NAME=service_name)))");

pds.setURL("jdbc:oracle:oci:@TNS_ALIAS");

pds.setURL("jdbc:oracle:oci:@(DESCRIPTION=
   (LOAD_BALANCE=on)
   (ADDRESS=(PROTOCOL=TCP)(HOST=host1) (PORT=1521))
   (ADDRESS=(PROTOCOL=TCP)(HOST=host2)(PORT=1521))
   (CONNECT_DATA=(SERVICE_NAME=service_name)))");

pds.setURL("jdbc:oracle:oci:@(DESCRIPTION=
   (ADDRESS=(PROTOCOL=TCP)(HOST=cluster_alias) (PORT=1521))
   (CONNECT_DATA=(SERVICE_NAME=service_name)))");
```

Using Run-Time Connection Load Balancing

UCP JDBC connection pools leverage the load balancing functionality provided by an Oracle RAC database. Run-time connection load balancing requires the use of an Oracle JDBC driver and an Oracle RAC database. For more information on setting up an Oracle RAC database, see the *Oracle Real Application Clusters Administration and Deployment Guide* or consult an Oracle database administrator.

Run-time connection load balancing is useful when:

- traditional balancing of workload is not optimal;
- requests must be routed to make optimal use of resources in a clustered database;
- capacity within the cluster differs and is expected to change over time;
- the need to avoid sending work to slow, hung, and dead nodes is required.

UCP for JDBC uses the Oracle RAC Load Balancing Advisory. The advisory is used to balance work across Oracle RAC instances and is used to determine which instances offer the best performance. Applications transparently receive connections from instances that offer the best performance. Connection requests are quickly diverted from instances that have slowed, are not responding, or that have failed.

Run-time connection load balancing provides the following benefits:

- Manages pooled connections for high performance and scalability
- Receives continuous recommendations on the percentage of work to route to database instances
- Adjusts distribution of work based on different back-end node capacities such as CPU capacity or response time
- Reacts quickly to changes in cluster reconfiguration, application workload, overworked nodes, or hangs
- Receives metrics from the Oracle RAC Load Balance Advisory. Connections to well performing instances are used most often. New and unused connections to under-performing instances will gravitate away over time. When distribution metrics are not received, connection are selected using a random choice.

Setting Up Run-Time Connection Load Balancing

Run-time connection load balancing requires that FCF is enabled and configured properly. See "Using Fast Connection Failover" on page 7-2 for detailed instructions on setting up FCF.

In addition, the Oracle RAC Load Balancing Advisory must be configured with service-level goals for each service for which load balancing is enabled. The Oracle RAC Load Balancing Advisory may be configured for `SERVICE_TIME` or `THROUGHPUT`. The connection load balancing goal should be set to `SHORT`. For example:

```
EXECUTE DBMS_SERVICE.MODIFY_SERVICE (service_name => 'sjob' -, goal =>
   DBMS_SERVICE.GOAL_THROUGHPUT -, clb_goal => DBMS_SERVICE.CLB_GOAL_SHORT);
```

Or

```
EXECUTE DBMS_SERVICE.MODIFY_SERVICE (service_name => 'sjob' -, goal =>
   DBMS_SERVICE.GOAL_SERVICE_TIME -, clb_goal => DBMS_SERVICE.CLB_GOAL_SHORT);
```

The Load Balancing Advisory goal can also be set by calling the `DBMS_SERVICE.create_service`. See the Introduction to Automatic Workload Management chapter in the *Oracle Real Application Clusters Administration and Deployment Guide*. In particular, refer to the "Load Balancing Advisory" section.

Using Connection Affinity

UCP JDBC connection pools leverage affinity functionality provided by an Oracle RAC database. Connection affinity requires the use of an Oracle JDBC driver and an Oracle RAC database version 11.1.0.6 or higher. For more information on setting up an Oracle RAC database, see the *Oracle Real Application Clusters Administration and Deployment Guide* or consult an Oracle database administrator.

Connection affinity is a performance feature that allows a connection pool to select connections that are directed at a specific Oracle RAC instance. The pool uses run-time connection load balancing (if configured) to select an Oracle RAC instance to create the first connection and then subsequent connections are created with an affinity to the same instance.

Note: Affinity is only a hint. A connection pool will select a new Oracle RAC instance for connections if a desired instance is not found.

UCP JDBC connection pools support two types of connection affinity: transaction-based affinity and Web session affinity.

Transaction-Based Affinity

Transaction-based affinity is an affinity to an Oracle RAC instance that can be released by either the client application or a failure event. Applications typically use this type of affinity when long-lived affinity to an Oracle RAC instance is desired or when the cost (in terms of performance) of being redirected to a new Oracle RAC instance is high. Distributed transactions are a good example of transaction-based affinity. XA connections that are enlisted in a distributed transaction keep an affinity to the Oracle RAC instance for the duration of the transaction. In this case, an application would incur a significant performance cost if a connection is redirect to a different Oracle RAC instance during the distributed transaction.

Web Session Affinity

Web session affinity is an affinity to an Oracle RAC instance that can be released by either the instance, a client application, or a failure event. The Oracle RAC instance uses a hint to communicate to a connection pool whether affinity has been enabled or disabled on the instance. An Oracle RAC instance may disable affinity based on many factors, such as performance or load. If an Oracle RAC instance can no longer support affinity, the connections in the pool are refreshed to use a new instance and affinity is established once again.

Applications typically use this type of affinity when short-lived affinity to an Oracle RAC instance is expected or if the cost (in terms of performance) of being redirected to a new Oracle RAC instance is minimal. For example, a mail client session might use Web session affinity to an Oracle RAC instance to increase performance and is relatively unaffected if a connection is redirected to a different instance.

Setting Up Connection Affinity

Connection affinity is set up as follows:

- Enable FCF. See "Using Fast Connection Failover" on page 7-2.
- Enable run-time connection load balancing. See "Using Run-Time Connection Load Balancing" on page 7-6.
- Create a connection affinity callback.
- Register the callback.

Note: Transaction-based affinity is strictly scoped between the application/middle-tier and UCP for JDBC; therefore, transaction-based affinity only requires that the `setFastConnectionFailoverEnabled` property be set to `true` and does not require complete FCF configuration.

In addition, transaction-based affinity does not technically require run-time connection load balancing. However, it can help with performance and is usually enabled regardless. If run-time connection load balancing is not enabled, the connection pool randomly picks connections.

Creating a Connection Affinity Callback

Connection affinity requires the use of a callback. The callback is an implementation of the `ConnectionAffinityCallback` interface which is located in the `oracle.ucp` package. The callback is used by the connection pool to establish and retrieve a connection affinity context and is also used to set the affinity policy type (transaction-based or Web session).

The following example demonstrates setting an affinity policy in a callback implementation. The example also demonstrates manually setting an affinity context. typically, the connection pool sets the affinity context inside an application. However, the ability to manually set an affinity context is provided for applications that want to customize affinity behavior and control the affinity context directly.

```
public class AffinityCallbackSample
   implements ConnectionAffinityCallback {

   Object appAffinityContext = null;
   ConnectionAffinityCallback.AffinityPolicy affinityPolicy =
   ConnectionAffinityCallback.AffinityPolicy.TRANSACTION_BASED_AFFINITY;

   //For Web session affinity, use WEBSESSION_BASED_AFFINITY;

   public void setAffinityPolicy(AffinityPolicy policy)
   {
      affinityPolicy = policy;
   }

   public AffinityPolicy getAffinityPolicy()
   {
      return affinityPolicy;
   }

   public boolean setConnectionAffinityContext(Object affCxt)
   {
      synchronized (lockObj)
      {
         appAffinityContext = affCxt;
      }
      return true;
   }

   public Object getConnectionAffinityContext()
   {
      synchronized (lockObj)
      {
         return appAffinityContext;
      }
   }
}
```

Registering a Connection Affinity Callback

A connection affinity callback is registered on a connection pool using the `registerConnectionAffinityCallback` method. The callback is registered when creating the connection pool. Only one callback can be registered per connection pool.

The following example demonstrates registering a connection affinity callback implementation:

```
ConnectionAffinityCallback callback = new MyCallback();

PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionPoolName("AffinitySamplePool");
pds.registerConnectionAffinityCallback(callback);
...
```

Removing a Connection Affinity Callback

A connection affinity callback is removed from a connection pool using the `removeConnectionAffinityCallback` method. For example:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();

pds.setConnectionPoolName("AffinitySamplePool");
pds.removeConnectionAffinityCallback();
```

```
...
```

Diagnostics and Statistics for Oracle RAC Features

UCP for JDBC provides a set of Oracle RAC run-time statistics that are used to determine how well a connection pool is utilizing Oracle RAC features and are also used to help determine whether the connection pool has been configured properly to use the Oracle RAC features. The statistics report FCF processing information, run-time connection load balance success/failure rate, and affinity context success/failure rate.

The `OracleJDBCConnectionPoolStatistics` interface that is located in the `oracle.ucp.jdbc.oracle` package provides methods that are used to query the connection pool for Oracle RAC statistics. The methods of this interface can be called from a pool-enabled and pool-enabled XA data source using the data source's `getStatistics` method. For example:

```
PoolDataSource  pds = PoolDataSourceFactory.getPoolDataSource();
...

Long rclbS = ((OracleJDBCConnectionPoolStatistics)pds.getStatistics()).
   getSuccessfulRCLBBasedBorrowCount();
System.out.println("The RCLB success rate is "+rclbS+".");
```

The data source's `getStatistics` method can also be called by itself and returns all connection pool statistics as a `String` and includes the Oracle RAC statistics.

Fast Connection Failover Statistics

The `getFCFProcessingInfo` method provides information on recent FCF attempts in the form of a `String`. The FCF information is typically used to help diagnose FCF problems. The information includes the outcome of each FCF attempt (successful or failed), the relevant Oracle RAC instances, the number of connections that were cleaned up, the exception that triggered the FCF attempt failure, and more. The following example demonstrates using the `getFCFProcessingInfo` method:

```
Sting fcfInfo = ((OracleJDBCConnectionPoolStatistics)pds.getStatistics()).
   getFCFProcessingInfo();
System.out.println("The FCF information: "+fcfInfo+".");
```

Run-Time Connection Load Balance Statistics

The run-time connection load balance statistics are used to determine if a connection pool is effectively utilizing the Oracle RAC database's run-time connection load balancing feature. The statistics report how many requests successfully used the run-time connection load balancing algorithms and how many requests failed to use the algorithms. The `getSuccessfulRCLBBasedBorrowCount` method and the `getFailedRCLBBasedBorrowCount` method, respectively, are used to get the statistics. The following example demonstrates using the `getFailedRCLBBasedBorrowCount` method:

```
Long rclbF = ((OracleJDBCConnectionPoolStatistics)pds.getStatistics()).
   getFailedRCLBBasedBorrowCount();
System.out.println("The RCLB failure rate is: "+rclbF+".");
```

A high failure rate may indicate that the RAC Load Balancing Advisory or connection pool is not configured properly.

Connection Affinity Statistics

The connection affinity statistics are used to determine if a connection pools is effectively utilizing connection affinity. The statistics report the number of borrow requests that succeeded in matching the affinity context and how many requests failed to match the affinity context. The `getSuccessfulAffinityBasedBorrowCount` method and the `getFailedAffinityBasedBorrowCount` method, respectively, are used to get the statistics. The following example demonstrates using the `getFailedAffinityBasedBorrowCount` method:

```
Long affF = ((OracleJDBCConnectionPoolStatistics)pds.getStatistics()).
   getFailedAffinityBasedBorrowCount();
System.out.println("The connection affinity failure rate is: "+affF+".");
```

Index