BEA WebLogic Server® and WebLogic Express®

Programming WebLogic JDBC

Version 10.0
Revised: March 30, 2007

Contents

6. Using RowSets with WebLogic Server

7. Troubleshooting JDBC

CHAPTER 1

Introduction and Roadmap

This section describes the contents and organization of this guide—*Programming WebLogic JDBC*.

Document Scope and Audience

This document is a resource for software developers and system administrators who develop and support applications that use the Java Database Connectivity (JDBC) API. It also contains information that is useful for business analysts and system architects who are evaluating WebLogic Server. The topics in this document are relevant during the evaluation, design, development, pre-production, and production phases of a software project.

It is assumed that the reader is familiar with Java EE and EJB concepts. This document emphasizes the value-added features provided by WebLogic Server EJBs and key information about how to use WebLogic Server features and facilities to get an EJB application up and running.

Guide to this Document

- This chapter, Chapter 1, “Introduction and Roadmap,” introduces the organization of this guide.
- Chapter 2, “Using WebLogic JDBC in an Application,” which explains how to use a JDBC connection in your application.
- Chapter 3, “Performance Tuning Your JDBC Application,” which describes how to design JDBC connection usage in your applications for the best performance.
- Chapter 4, “Using WebLogic Wrapper Drivers,” which describes how to use some alternative drivers for getting a JDBC connection from a data source.
- Chapter 5, “Using Third-Party Drivers with WebLogic Server,” which describes special programming considerations for third-party drivers in your applications.
- Chapter 6, “Using RowSets with WebLogic Server,” which describes how to use rowsets in your applications.
- Chapter 7, “Troubleshooting JDBC,” which describes some common JDBC problems and solutions.

Related Documentation

This document contains JDBC-specific programming information.

For comprehensive guidelines for developing, deploying, and monitoring WebLogic Server applications, see the following documents:

- *Configuring and Managing WebLogic JDBC* is a guide to JDBC configuration and management for WebLogic Server.
- *Developing Applications with WebLogic Server* is a guide to developing WebLogic Server applications.
- *Deploying Applications to WebLogic Server* is the primary source of information about deploying WebLogic Server applications in development and production environments.

JDBC Samples and Tutorials

In addition to this document, BEA Systems provides a variety of JDBC code samples and tutorials that show JDBC configuration and API use, and provide practical instructions on how to perform key JDBC development tasks.

Avitek Medical Records Application (MedRec) and Tutorials

MedRec is an end-to-end sample Java EE application shipped with WebLogic Server that simulates an independent, centralized medical record management system. The MedRec application provides a framework for patients, doctors, and administrators to manage patient data using a variety of different clients.

MedRec demonstrates WebLogic Server and Java EE features, and highlights BEA-recommended best practices. MedRec is included in the WebLogic Server distribution, and can be accessed from the Start menu on Windows machines. For Linux and other platforms, you can start MedRec from the `WL_HOME\samples\domains\medrec` directory, where `WL_HOME` is the top-level installation directory for WebLogic Platform.

JDBC Examples in the WebLogic Server Distribution

WebLogic Server optionally installs API code examples in `WL_HOME\samples\server\examples\src\examples`, where `WL_HOME` is the top-level directory of your WebLogic Server installation. You can start the examples server, and obtain information about the samples and how to run them from the WebLogic Server Start menu.

Additional JDBC Examples Available for Download

Additional API examples for download at `http://codesamples.projects.dev2dev.bea.com`. These examples are distributed as ZIP files that you can unzip into an existing WebLogic Server samples directory structure.

You build and run the downloadable examples in the same manner as you would an installed WebLogic Server example. See the download pages of individual examples for more information at `https://codesample.projects.dev2dev.bea.com`.

New and Changed JDBC Features in This Release

For a comprehensive listing of the new WebLogic Server features introduced in release, see "What's New in WebLogic Server 10.0" in *Release Notes*.

CHAPTER 2

Using WebLogic JDBC in an Application

You use the WebLogic Server Administration Console to enable, configure, and monitor features of WebLogic Server, including JDBC data sources and multi data sources. You can do the same tasks programmatically using the JMX API and the WebLogic Scripting Tool (WLST). After configuring JDBC connectivity components, you can use them in your applications.

The following sections describe how to use the JDBC connectivity in your applications.

For more information about configuring JDBC data sources and multi data sources, see *Configuring and Managing WebLogic JDBC*.

Getting a Database Connection from a DataSource Object

The following sections provide details about requesting a database connection from a DataSource object - either a data source or a multi data source:

Importing Packages to Access DataSource Objects

To use the DataSource objects in your applications, import the following classes in your client code:

```
import java.sql.*;
import java.util.*;
import javax.naming.*;
```

Obtaining a Client Connection Using a DataSource

To obtain a connection for a JDBC client, use a Java Naming and Directory Interface (JNDI) lookup to locate the DataSource object, as shown in this code fragment.

Note: When using a JDBC connection in a client-side application, the *exact same* JDBC driver classes must be in the `CLASSPATH` on both the server and the client. If the driver classes do not match, you may see `java.rmi.UnmarshalException` exceptions.

```
Context ctx = null;
  Hashtable ht = new Hashtable();
  ht.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.jndi.WLInitialContextFactory");
  ht.put(Context.PROVIDER_URL,
         "t3://hostname:port");

  Connection conn = null;
  Statement stmt = null;
  ResultSet rs = null;

  try {
    ctx = new InitialContext(ht);
    javax.sql.DataSource ds
      = (javax.sql.DataSource) ctx.lookup ("myDataSource");
    conn = ds.getConnection();

   // You can now use the conn object to create
   //  Statements and retrieve result sets:

    stmt = conn.createStatement();
    stmt.execute("select * from someTable");
    rs = stmt.getResultSet();

...
```

```
//Close JDBC objects as soon as possible
    stmt.close();
    stmt=null;

    conn.close();
    conn=null;

 }
  catch (Exception e) {
    // a failure occurred
    log message;
  }
finally {
  try {
    ctx.close();
  } catch (Exception e) {
     log message; }
  try {
    if (rs != null) rs.close();
  } catch (Exception e) {
     log message; }
  try {
    if (stmt != null) stmt.close();
  } catch (Exception e) {
     log message; }
  try {
    if (conn != null) conn.close();
  } catch (Exception e) {
     log message; }
}
```

(Substitute the correct `hostname` and `port` number for your WebLogic Server.)

Note: The code above uses one of several available procedures for obtaining a JNDI context. For more information on JNDI, see Programming WebLogic JNDI.

Possible Exceptions When a Connection Request Fails

The weblogic.jdbc.extensions package includes the following exceptions that can be thrown when an application request fails. Each exception extends `java.sql.SQLException`.

- `ConnectionDeadSQLException`—generated when an application request to get a connection fails because the connection test on the reserved connection failed. This typically happens when the database server is unavailable.
- `ConnectionUnavailableSQLException`—generated when an application request to get a connection fails because there are currently no connections available in the pool to be allocated. This is a transient failure, and is generated if all connections in the pool are currently in use. It can also be thrown when connections are unavailable because they are being tested.
- `PoolDisabledSQLException`—generated when an application request to get a connection fails because the JDBC Data Source has been administratively disabled.
- `PoolLimitSQLException`—generated when an application request to get a connection fails due to a configured threshold of the data source, such as `HighestNumWaiters`, `ConnectionReserveTimeoutSeconds`, and so forth.
- `PoolPermissionsSQLException`—generated when an application request to get a connection fails a (security) authentication or authorization check.

Pooled Connection Limitation

When using pooled connections in a data source, it is possible to execute DBMS-specific SQL code that will alter the database connection properties and that WebLogic Server and the JDBC driver will be unaware of. When the connection is returned to the data source, the characteristics of the connection may not be set back to a valid state. For example, with a Sybase DBMS, if you use a statement such as `"set rowcount 3 select * from y"`, the connection will only ever return a maximum of 3 rows from any subsequent query on this connection. When the connection is returned to the data source and then reused, the next user of the connection will still only get 3 rows returned, even if the table being selected from has 500 rows.

In most cases, there is standard JDBC code that can accomplish the same result. In this example, you could use `setMaxRows()` instead of `set rowcount`. BEA recommends that you use the standard JDBC code instead of the DBMS-specific SQL code. When you use standard JDBC calls to alter the connection, Weblogic Server returns the connection to a standard state when the connection is returned to the data source.

If you use DBMS-specific SQL code that alters the connection, you must set the connection back to an acceptable state before returning the connection to the data source.

Getting a Connection from an Application-Scoped Data Source

To get a connection from an application-scoped data source, see "Getting a Database Connection from a Packaged JDBC Module" in *Configuring and Managing WebLogic JDBC*.

CHAPTER 3

Performance Tuning Your JDBC Application

The following sections explain how to get the best performance from JDBC applications:

WebLogic Performance-Enhancing Features

WebLogic has several features that enhance performance for JDBC applications.

How Pooled Connections Enhance Performance

Establishing a JDBC connection with a DBMS can be very slow. If your application requires database connections that are repeatedly opened and closed, this can become a significant performance issue. Connection pools in WebLogic data sources offer an efficient solution to this problem.

When WebLogic Server starts, connections in the data sources are opened and are available to all clients. When a client closes a connection from a data source, the connection is returned to the pool and becomes available for other clients; the connection itself is not closed. There is little cost to opening and closing pooled connections.

Caching Statements and Data

DBMS access uses considerable resources. If your program reuses prepared or callable statements or accesses frequently used data that can be shared among applications or can persist between connections, you can cache prepared statements or data by using the following:

- Statement Cache for a data source
- Read-Only Entity Beans
- JNDI in a Clustered Environment

Designing Your Application for Best Performance

Most performance gains or losses in a database application is not determined by the application language, but by how the application is designed. The number and location of clients, size and structure of DBMS tables and indexes, and the number and types of queries all affect application performance.

The following are general hints that apply to all DBMSs. It is also important to be familiar with the performance documentation of the specific DBMS that you use in your application.

1. Process as Much Data as Possible Inside the Database

Most serious performance problems in DBMS applications come from moving raw data around needlessly, whether it is across the network or just in and out of cache in the DBMS. A good method for minimizing this waste is to put your logic where the data is—in the DBMS, not in the client —even if the client is running on the same box as the DBMS. In fact, for some DBMSs a fat client and a fat DBMS sharing one CPU is a performance disaster.

Most DBMSs provide stored procedures, an ideal tool for putting your logic where your data is. There is a significant difference in performance between a client that calls a stored procedure to update 10 rows, and another client that fetches those rows, alters them, and sends update statements to save the changes to the DBMS.

Also review the DBMS documentation on managing cache memory in the DBMS. Some DBMSs (Sybase, for example) provide the means to partition the virtual memory allotted to the DBMS, and to guarantee certain objects exclusive use of some fixed areas of cache. This means that an important table or index can be read once from disk and remain available to all clients without having to access the disk again.

2. Use Built-in DBMS Set-based Processing

SQL is a set processing language. DBMSs are designed from the ground up to do set-based processing. Accessing a database one row at a time is, without exception, slower than set-based processing and, on some DBMSs is poorly implemented. For example, it will always be faster to update each of four tables one at a time for all the 100 employees represented in the tables than to alter each table 100 times, once for each employee.

Many complicated processes that were originally thought too complex to do any other way but row-at-a-time have been rewritten using set-based processing, resulting in improved performance. For example, a major payroll application was converted from a huge slow COBOL application to four stored procedures running in series, and what took hours on a multi-CPU machine now takes fifteen minutes with many fewer resources used.

3. Make Your Queries Smart

Frequently customers ask how to tell how many rows will be coming back in a given result set. The only way to find out without fetching all the rows is by issuing the same query using the *count keyword:*

```
SELECT count(*) from myTable, yourTable where ...
```

This returns the number of rows the original query would have returned, assuming no change in relevant data. The actual count may change when the query is issued if other DBMS activity has occurred that alters the relevant data.

Be aware, however, that this is a resource-intensive operation. Depending on the original query, the DBMS may perform nearly as much work to count the rows as it will to send them.

Make your application queries as specific as possible about what data it actually wants. For example, tailor your application to select into temporary tables, returning only the count, and then sending a refined second query to return only a subset of the rows in the temporary table.

Learning to select only the data you really want at the client is crucial. Some applications ported from ISAM (a pre-relational database architecture) will unnecessarily send a query selecting all the rows in a table when only the first few rows are required. Some applications use a 'sort by' clause to get the rows they want to come back first. Database queries like this cause unnecessary degradation of performance.

Proper use of SQL can avoid these performance problems. For example, if you only want data about the top three earners on the payroll, the proper way to make this query is with a correlated subquery. Table 3-1 shows the entire table returned by the SQL statement

```
select * from payroll
```

Table 3-1 Full Results Returned

Name	Salary
Joe	10
Mike	20
Sam	30
Tom	40
Jan	50
Ann	60
Sue	70
Hal	80
May	80

A correlated subquery

```
select p.name, p.salary from payroll p
where 3 >= (select count(*) from payroll pp
where pp.salary >= p.salary);
```

returns a much smaller result, shown in Table 3-2.

Table 3-2 Results from Subquery

Name	Salary
Sue	70
Hal	80
May	80

This query returns only *three rows, with the name and salary of the top three earners.* It scans through the payroll table, and for every row, it goes through the whole payroll table again in an

inner loop to see how many salaries are higher than the current row of the outer scan. This may look complicated, but DBMSs are designed to use SQL efficiently for this type of operation.

4. Make Transactions Single-batch

Whenever possible, collect a set of data operations and submit an update transaction in one statement in the form:

```
BEGIN TRANSACTION
   UPDATE TABLE1...
   INSERT INTO TABLE2
   DELETE TABLE3
COMMIT
```

This approach results in better performance than using separate statements and commits. Even with conditional logic and temporary tables in the batch, it is preferable because the DBMS obtains all the locks necessary on the various rows and tables, and uses and releases them in one step. Using separate statements and commits results in many more client-to-DBMS transmissions and holds the locks in the DBMS for much longer. These locks will block out other clients from accessing this data, and, depending on whether different updates can alter tables in different orders, may cause deadlocks.

Warning: If any individual statement in the preceding transaction fails, due, for instance, to violating a unique key constraint, you should put in conditional SQL logic to detect statement failure and to roll back the transaction rather than commit. If, in the preceding example, the insert failed, most DBMSs return an error message about the failed insert, but behave as if you got the message between the second and third statement, and decided to commit anyway! Microsoft SQL Server offers a connection option enabled by executing the SQL `set xact_abort on`, which automatically rolls back the transaction if any statement fails.

5. Never Have a DBMS Transaction Span User Input

If an application sends a `'BEGIN TRAN'` and some SQL that locks rows or tables for an update, do not write your application so that it must wait on the user to press a key before committing the transaction. That user may go to lunch first and lock up a whole DBMS table until the user returns.

If you require user input to form or complete a transaction, use optimistic locking. Briefly, optimistic locking employs timestamps and triggers in queries and updates. Queries select data

with timestamp values and prepare a transaction based on that data, without locking the data in a transaction.

When an update transaction is finally defined by the user input, it is sent as a single submission that includes timestamped safeguards to make sure the data is the same as originally fetched. A successful transaction automatically updates the relevant timestamps for changed data. If an interceding update from another client has altered data on which the current transaction is based, the timestamps change, and the current transaction is rejected. Most of the time, no relevant data has been changed so transactions usually succeed. When a transaction fails, the application can refetch the updated data to present to the user to reform the transaction if desired.

6. Use In-place Updates

Changing a data row in place is much faster than moving a row, which may be required if the update requires more space than the table design can accommodate. If you design your rows to have the space they need initially, updates will be faster, although the table may require more disk space. Because disk space is cheap, using a little more of it can be a worthwhile investment to improve performance.

7. Keep Operational Data Sets Small

Some applications store operational data in the same table as historical data. Over time and with accumulation of this historical data, all operational queries have to read through lots of useless (on a day-to-day basis) data to get to the more current data. Move non-current data to other tables and do joins to these tables for the rarer historical queries. If this can't be done, index and cluster your table so that the most frequently used data is logically and physically localized.

8. Use Pipelining and Parallelism

DBMSs are designed to work best when very busy with lots of different things to do. The worst way to use a DBMS is as dumb file storage for one big single-threaded application. If you can design your application and data to support lots of parallel processes working on easily distinguished subsets of the work, your application will be much faster. If there are multiple steps to processing, try to design your application so that subsequent steps can start working on the portion of data that any prior process has finished, instead of having to wait until the prior process is complete. This may not always be possible, but you can dramatically improve performance by designing your program with this in mind.

CHAPTER 4

Using WebLogic Wrapper Drivers

BEA recommends that you use DataSource objects to get database connections in new applications. DataSource objects (WebLogic data sources and multi data sources), along with the JNDI tree, provide access to pooled connections in a data source for database connectivity. The WebLogic wrapper drivers are deprecated. For existing or legacy applications that use the JDBC 1.x API, you can use the WebLogic wrapper drivers to get database connectivity.

The following sections describe how to use WebLogic wrapper drivers with WebLogic Server:

Using the WebLogic RMI Driver (Deprecated)

RMI driver clients make their connection to the DBMS by looking up the DataSource object. This lookup is accomplished by using a Java Naming and Directory Service (JNDI) lookup, or by directly calling WebLogic Server which performs the JNDI lookup on behalf of the client.

The RMI driver replaces the functionality of both the WebLogic t3 driver (deprecated) and the Pool driver, and uses the Java standard Remote Method Invocation (RMI) to connect to WebLogic Server rather than the proprietary t3 protocol.

Because the details of the RMI implementation are taken care of automatically by the driver, a knowledge of RMI is not required to use the WebLogic JDBC/RMI driver.

Setting Up WebLogic Server to Use the WebLogic RMI Driver

The RMI driver is accessible through DataSource objects, which are created in the Administration Console. You should create DataSource objects in your WebLogic Server configuration before you use the RMI driver in your applications.

Sample Client Code for Using the RMI Driver

The following code samples show how to use the RMI driver to get and use a database connection from a WebLogic Server data source.

Import the Required Packages

Before you can use the RMI driver to get and use a database connection, you must import the following packages:

```
javax.sql.DataSource
java.sql.*
java.util.*
javax.naming.*
```

Get the Database Connection

The WebLogic JDBC/RMI client obtains its connection to a DBMS from the DataSource object that you defined in the Administration Console. There are two ways the client can obtain a DataSource object:

- Using a JNDI lookup. This is the preferred and most direct procedure.
- Passing the DataSource name to the RMI driver with the `Driver.connect()`method. In this case, WebLogic Server performs the JNDI look up on behalf of the client.

Using a JNDI Lookup to Obtain the Connection

To access the WebLogic RMI driver using JNDI, obtain a context from the JNDI tree by looking up the name of your DataSource object. For example, to access a DataSource called "`myDataSource`" that is defined in Administration Console:

```
Context ctx = null;
  Hashtable ht = new Hashtable();
  ht.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.jndi.WLInitialContextFactory");
```

```
  ht.put(Context.PROVIDER_URL,
         "t3://hostname:port");
  try {
    ctx = new InitialContext(ht);
    javax.sql.DataSource ds
      = (javax.sql.DataSource) ctx.lookup ("myDataSource");
   java.sql.Connection conn = ds.getConnection();
   // You can now use the conn object to create
   //  a Statement object to execute
   //  SQL statements and process result sets:
   Statement stmt = conn.createStatement();
   stmt.execute("select * from someTable");
   ResultSet rs = stmt.getResultSet();
   // Do not forget to close the statement and connection objects
   //  when you are finished:
 }
  catch (Exception e) {
    // a failure occurred
    log message;
  }
} finally {
  try {
    ctx.close();
  } catch (Exception e) {
     log message; }
  try {
    if (rs != null) rs.close();
  } catch (Exception e) {
     log message; }
  try {
    if (stmt != null) stmt.close();
  } catch (Exception e) {
     log message; }
  try {
    if (conn != null) conn.close();
  } catch (Exception e) {
```

```
      log message; }
}
```

(Where *hostname* is the name of the machine running your WebLogic Server and *port* is the port number where that machine is listening for connection requests.)

In this example a *Hashtable* object is used to pass the parameters required for the JNDI lookup. There are other ways to perform a JNDI lookup. For more information, see *Programming WebLogic JNDI*.

Notice that the JNDI lookup is wrapped in a `try/catch` block in order to catch a failed look up and also that the context is closed in a `finally` block.

Using Only the WebLogic RMI Driver to Obtain a Database Connection

Instead of looking up a DataSource object to get a database connection, you can access WebLogic Server using the `Driver.connect()` method, in which case the JDBC/RMI driver performs the JNDI lookup. To access the WebLogic Server, pass the parameters defining the URL of your WebLogic Server and the name of the DataSource object to the `Driver.connect()` method. For example, to access a DataSource called "myDataSource" as defined in the Administration Console:

```
java.sql.Driver myDriver = (java.sql.Driver)
  Class.forName("weblogic.jdbc.rmi.Driver").newInstance();

String url ="jdbc:weblogic:rmi";

java.util.Properties props = new java.util.Properties();
props.put("weblogic.server.url", "t3://hostname:port");
props.put("weblogic.jdbc.datasource", "myDataSource");

java.sql.Connection conn = myDriver.connect(url, props);
```

(Where *hostname* is the name of the machine running your WebLogic Server and *port* is the port number where that machine is listening for connection requests.)

You can also define the following properties which will be used to set the JNDI user information:

- `weblogic.user`—specifies a username
- `weblogic.credential`—specifies the password for the `weblogic.user`.

Row Caching with the WebLogic RMI Driver

Row caching is a WebLogic Server JDBC feature that improves the performance of your application. Normally, when a client calls `ResultSet.next()`, WebLogic Server fetches a single row from the DBMS and transmits it to the client JVM. With row caching enabled, a single call to `ResultSet.next()` retrieves multiple DBMS rows, and caches them in client memory. By reducing the number of trips across the wire to retrieve data, row caching improves performance.

Note: WebLogic Server will not perform row caching when the client and WebLogic Server are in the same JVM.

You can enable and disable row caching and set the number of rows fetched per `ResultSet.next()` call with the data source attributes Row Prefetch Enabled and Row Prefetch Size, respectively. You set data source attributes via the Administration Console. To enable row caching and to set the row prefetch size attribute for a data source, follow these steps:

1. If you have not already done so, in the Change Center of the Administration Console, click Lock & Edit.
2. In the Domain Structure tree, expand Services > JDBC, then select Data Sources.
3. On the Summary of Data Sources page, click the data source name.
4. Select the Configuration: General tab and then do the following:.
   a. Select the Row Prefetch Enabled check box.
   b. In Row Prefetch Size, type the number of rows you want to cache for each `ResultSet.next()` call.
5. Click Save.
6. To activate these changes, in the Change Center of the Administration Console, click Activate Changes.

See the JDBC Data Source: Configuration: General page in the *Administration Console Online Help*.

Important Limitations for Row Caching with the WebLogic RMI Driver

Keep the following limitations in mind if you intend to implement row caching with the RMI driver:

- WebLogic Server only performs row caching if the result set type is both TYPE_FORWARD_ONLY and CONCUR_READ_ONLY.
- Certain data types in a result set may disable caching for that result set. These include the following:
  - LONGVARCHAR/LONGVARBINARY
  - NULL
  - BLOB/CLOB
  - ARRAY
  - REF
  - STRUCT
  - JAVA_OBJECT
- Certain ResultSet methods are not supported if row caching is enabled and active for that result set. Most pertain to streaming data, scrollable result sets or data types not supported for row caching. These include the following:
  - `getAsciiStream()`
  - `getUnicodeStream()`
  - `getBinaryStream()`
  - `getCharacterStream()`
  - `isBeforeLast()`
  - `isAfterLast()`
  - `isFirst()`
  - `isLast()`
  - `getRow()`
  - `getObject (Map)`
  - `getRef()`
  - `getBlob()/getClob()`
  - `getArray()`
  - `getDate()`
  - `getTime()`
  - `getTimestamp()`

Limitations When Using Global Transactions

Populating a RowSet in a global transaction may fail with "Fetch Out Of Sequency" exception. For example:

1. When the RMI call returns, the global transaction is suspended automatically by the server instance.
2. The JDBC driver invalidates the pending ResultSet object to release the system resources.
3. The client tries to read data from the invalidated ResultSet.
4. A "Fetch Out Of Sequency" exception is thrown if that data has not been prefetched. Since the number of rows prefetched is vendor specific, you may or may not encounter this issue, especially when working with one or two rows.

If you encounter this exception, make sure to populate the RowSet on the server side and then serialize it back to the client.

Using the WebLogic JTS Driver (Deprecated)

The Java Transaction Services or JTS driver is a server-side Java Database Connectivity (JDBC) driver that provides access to both data sources and global transactions from applications running in WebLogic Server. Connections to a database are made from a data source and use a JDBC driver in WebLogic Server to connect to the Database Management System (DBMS) on behalf of your application. Your application uses the JTS driver to access a connection from the data source.

WebLogic Server also uses the JTS driver internally when a connection from a data source that uses a non-XA JDBC driver participates in a global transaction (Logging Last Resource and Emulate Two-Phase Commit). This behavior enables a non-XA resource to emulate XA and participate in a two-phase commit transaction. See “Transaction Options” in *Configuring and Managing WebLogic JDBC*.

Once a transaction begins, all database operations in an execute thread that get their connection from the *same data source* share the *same connection* from that data source. These operations can be made through services such as Enterprise JavaBeans (EJB) or Java Messaging Service (JMS), or by directly sending SQL statements using standard JDBC calls. All of these operations will, by default, share the same connection and participate in the same transaction.When the transaction is committed or rolled back, the connection is returned to the pool.

Although Java clients may not register the JTS driver themselves, they may participate in transactions via Remote Method Invocation (RMI). You can begin a transaction in a thread on a client and then have the client call a remote RMI object. The database operations executed by the remote object become part of the transaction that was begun on the client. When the remote object is returned back to the calling client, you can then commit or roll back the transaction. The database operations executed by the remote objects must all use the same data source to be part of the same transaction.

For the JTS driver and your application to participate in a global transaction, the application must call `conn = myDriver.connect("jdbc:weblogic:jts", props);` within a global transaction. After the transaction completes (gets committed or rolled back), WebLogic Server puts the connection back in the data source. If you want to use a connection for another global transaction, the application must call `conn = myDriver.connect("jdbc:weblogic:jts", props);` again within a new global transaction.

Sample Client Code for Using the JTS Driver

To use the JTS driver, you must first use the Administration Console to create a data source in WebLogic Server.

This explanation demonstrates creating and using a JTS transaction from a server-side application and uses a data source named "`myDataSource`."

1. Import the following classes:

```
import javax.transaction.UserTransaction;
import java.sql.*;
import javax.naming.*;
import java.util.*;
import weblogic.jndi.*;
```

2. Establish the transaction by using the `UserTransaction` class. You can look up this class on the JNDI tree. The `UserTransaction` class controls the transaction on the current execute thread. Note that this class does not represent the transaction itself. The actual context for the transaction is associated with the current execute thread.

```
Context ctx = null;
Hashtable env = new Hashtable();

env.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.jndi.WLInitialContextFactory");

// Parameters for the WebLogic Server.
// Substitute the correct hostname, port number
```

```
// user name, and password for your environment:
env.put(Context.PROVIDER_URL, "t3://localhost:7001");
env.put(Context.SECURITY_PRINCIPAL, “Fred”);
env.put(Context.SECURITY_CREDENTIALS, “secret”);

ctx = new InitialContext(env);

UserTransaction tx = (UserTransaction)
  ctx.lookup("javax.transaction.UserTransaction");
```

3. Start a transaction on the current thread:

   ```
   // Start the global transaction before getting a connection
   tx.begin();
   ```

4. Load the JTS driver:

   ```
   Driver myDriver = (Driver)
    Class.forName("weblogic.jdbc.jts.Driver").newInstance();
   ```

5. Get a connection from the data source:

   ```
   Properties props = new Properties();
   props.put("connectionPoolID", "myDataSource");

   conn = myDriver.connect("jdbc:weblogic:jts", props);
   ```

6. Execute your database operations. These operations may be made by any service that uses a database connection, including EJB, JMS, and standard JDBC statements. These operations must use the JTS driver to access the same data source as the transaction begun in step 3 in order to participate in that transaction.

   If the additional database operations using the JTS driver use a *different data source* than the one specified in step 5, an exception will be thrown when you try to commit or roll back the transaction.

7. Close your connection objects. Note that closing the connections does not commit the transaction nor return the connection to the pool:

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

8. Complete the transaction by either committing the transaction or rolling it back. In the case of a commit, the JTS driver commits all the transactions on all connection objects in the current thread and returns the connection to the pool.

   ```
   tx.commit();

   // or:
   ```

```
tx.rollback();
```

Using the WebLogic Pool Driver (Deprecated)

The WebLogic Pool driver enables utilization of data sources from server-side applications such as HTTP servlets or EJBs. For information about using the Pool driver, see "Accessing Databases" in Accessing Databases in *Developing Web Applications, Servlets, and JSPs for WebLogic Server*.

CHAPTER 5

Using Third-Party Drivers with WebLogic Server

The following sections describe how to set up and use third-party JDBC drivers:

Getting a Connection with Your Third-Party Driver

The following sections describe how to get a database connection using a third-party, Type 4 driver, such as the Oracle Thin Driver. BEA recommends you use DataSource objects and a JNDI lookup to establish your connection.

Using Data Sources with a Third-Party Driver

First, you create the data source using the Administration Console, then establish a connection using a JNDI Lookup.

Using a JNDI Lookup to Obtain the Connection

To access the data source using JNDI, obtain a Context from the JNDI tree by providing the URL of your server, and then use that context object to perform a lookup using the DataSource Name.

For example, to access a DataSource called "`myDataSource`" that is defined in the Administration Console:

Listing 5-1 Using a JNDI Lookup to Obtain a Connection

```
Context ctx = null;
  Hashtable ht = new Hashtable();
  ht.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.jndi.WLInitialContextFactory");
  ht.put(Context.PROVIDER_URL,
         "t3://hostname:port");

  Connection conn = null;
  Statement stmt = null;
  ResultSet rs = null;

  try {
    ctx = new InitialContext(ht);
    javax.sql.DataSource ds
      = (javax.sql.DataSource) ctx.lookup ("myDataSource");
    conn = ds.getConnection();

   // You can now use the conn object to create
   //  Statements and retrieve result sets:

    stmt = conn.createStatement();
    stmt.execute("select * from someTable");
    rs = stmt.getResultSet();

...

//Close JDBC objects as soon as possible
    stmt.close();
    stmt=null;

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

```
 }
  catch (Exception e) {
    // a failure occurred
    log message;
  }
finally {
  try {
    ctx.close();
  } catch (Exception e) {
     log message; }
  try {
    if (rs != null) rs.close();
  } catch (Exception e) {
     log message; }
  try {
    if (stmt != null) stmt.close();
  } catch (Exception e) {
     log message; }
  try {
    if (conn != null) conn.close();
  } catch (Exception e) {
     log message; }
}
```

(Where *hostname* is the name of the machine running your WebLogic Server and `port` is the port number where that machine is listening for connection requests.)

In this example a *Hashtable* object is used to pass the parameters required for the JNDI lookup. There are other ways to perform a JNDI lookup. For more information, see Programming WebLogic JNDI.

Notice that the JNDI lookup is wrapped in a `try/catch` block in order to catch a failed look up and also that the context is closed in a `finally` block.

Getting a Physical Connection from a Data Source

Note: BEA strongly discourages directly accessing a physical JDBC connection except for when it is absolutely required. See "Getting a Physical Connection from a Data Source" on page 5-4.

Standard practice is to cast a connection to the generic JDBC connection (a wrapped physical connection) provided by WebLogic Server. This allows the server instance to manage the connection for the connection pool, enable connection pool features, and maintain the quality of connections provided to applications. Occasionally, a DBMS vendor may provide extra non-standard JDBC-related classes that require direct access of the physical connection (the actual vendor JDBC connection). To directly access a physical connection in a connection pool, you must cast the connection using `getVendorConnection`.

The following sections provide information on getting a physical connection:

Opening a Connection

To get a physical database connection, you first get a connection from a connection pool as described in "Using a JNDI Lookup to Obtain the Connection" on page 5-2, then do one of the following:

- Implicitly pass the physical connection (using `getVendorConnection`) within a method that requires the physical connection.
- Cast the connection as a `WLConnection` and call `getVendorConnection`.

Always limit direct access of physical database connections to vendor-specific calls. For all other situations, use the generic JDBC connection provided by WebLogic Server. Sample code to open a connection for vendor-specific calls is provided in Listing 5-2.

Listing 5-2 Code Sample to Open a Connection for Vendor-specific Calls

```
//Import this additional class and any vendor packages
//you may need.
import weblogic.jdbc.extensions.WLConnection
```

```
.
.
.
myJdbcMethod()
{
  // Connections from a connection pool should always be
  // method-level variables, never class or instance methods.
  Connection conn = null;

   try {
     ctx = new InitialContext(ht);
     // Look up the data source on the JNDI tree and request
     // a connection.
     javax.sql.DataSource ds
        = (javax.sql.DataSource) ctx.lookup ("myDataSource");

     // Always get a pooled connection in a try block where it is
     // used completely and is closed if necessary in the finally
     // block.
     conn = ds.getConnection();

     // You can now cast the conn object to a WLConnection
     // interface and then get the underlying physical connection.

     java.sql.Connection vendorConn =
       ((WLConnection)conn).getVendorConnection();
     // do not close vendorConn

     // You could also cast the vendorConn object to a vendor
     // interface, such as:
     // oracle.jdbc.OracleConnection vendorConn = (OracleConnection)
     // ((WLConnection)conn).getVendorConnection()

     // If you have a vendor-specific method that requires the
     // physical connection, it is best not to obtain or retain
     // the physical connection, but simply pass it implicitly
     // where needed, eg:
//vendor.special.methodNeedingConnection(((WLConnection)conn)).getVendorCo
nnection());
```

Closing a Connection

When you are finished with your JDBC work, you should close the logical connection to get it back into the pool. When you are done with the physical connection:

- Close any objects you have obtained from the connection.
- Do not close the physical connection. Set the physical connection to null.

You determine how a connection closes by setting the value of the `Remove Infected Connections Enabled` property in the administration console. See the JDBC Data Source: Configuration: Connection Pool page in the *Administration Console Help* or see "JDBCConnectionPoolParamsBean" in the WebLogic Server MBean Reference for more details about these options

Note: The `Remove Infected Connections Enabled` property applies only to applications that explicitly call `getVendorConnection`.

Listing 5-3 Sample Code to Close a Connection for Vendor-specific Calls

```
// As soon as you are finished with vendor-specific calls,
    // nullify the reference to the connection.
    // Do not keep it or close it.
    // Never use the vendor connection for generic JDBC.
    // Use the logical (pooled) connection for standard JDBC.
    vendorConn = null;

    ... do all the JDBC needed for the whole method...

    // close the logical (pooled) connection to return it to
    // the connection pool, and nullify the reference.
    conn.close();
    conn = null;
  }

  catch (Exception e)
  {
    // Handle the exception.
  }
  finally
{
    // For safety, check whether the logical (pooled) connection
```

```
    // was closed.
    // Always close the logical (pooled) connection as the
    // first step in the finally block.
    if (conn != null) try {conn.close();} catch (Exception ignore){}
  }
}
```

Remove Infected Connections Enabled is True

When `Remove infected Connections Enabled=false` (default value) and you close the logical connection, the server instance discards the underlying physical connection and creates a new connection to replace it. This action ensures that the pool can guarantee to the next user that they are the sole user of the pool connection. This configuration provides a simple and safe way to close a connection. However, there is a performance loss because:

- The physical connection is replaced with a new database connection in the connection pool, which uses resources on both the application server and the database server.
- The statement cache for the original connection is closed and a new cache is opened for the new connection. Therefore, the performance gains from using the statement cache are lost.

Remove Infected Connections Enabled is False

Note: Use `Remove infected Connections Enabled=false` only if you are sure that the exposed physical connection will never be retained or reused after the logical connection is closed.

When `Remove infected Connections Enabled=false` and you close the logical connection, the server instance simply returns the physical connection to the connection pool for reuse. Although this configuration minimizes performance losses, the server instance does not guarantee the quality of the connection or to effectively manage the connection after the logical connection is closed. You must make sure that the connection is suitable for reuse by other applications before it is returned to the connection pool.

Limitations for Using a Physical Connection

BEA strongly discourages using a physical connection instead of a logical connection from a connection pool. However, if you must use a physical connection, for example, to create a STRUCT, consider the following costs and limitations:

- The physical connection can only be used in server-side code.
- When you use a physical connection, you lose all of the connection management benefits that WebLogic Server offer, such as error handling and statement caching.
- You should use the physical connection only for the vendor-specific methods or classes that require it. Do not use the physical connection for generic JDBC, such as creating statements or transactional calls.

Using Vendor Extensions to JDBC Interfaces

Some database vendors provide additional proprietary methods for working with data from a database that uses their DBMS. These methods extend the standard JDBC interfaces. In previous releases of WebLogic Server, only specific JDBC extensions for a few vendors were supported. The current release of WebLogic Server supports all extension methods exposed as a public interface in the vendor's JDBC driver.

If the driver vendor does not expose the methods you need in a public interface, you should submit a request to the vendor to expose the methods in a public interface. WebLogic Server does provide support for extension methods in the Oracle Thin Driver for ARRAYs, STRUCTs, and REFs, even though the extension methods are not exposed in a public interface. See "Using Oracle Extensions with the Oracle Thin Driver" on page 5-11.

In general, WebLogic Server supports using vendor extensions in server-side code. To use vendor extensions in client-side code, the object type or data type must be serializable. Exceptions to this are the following object types:

- CLOB
- BLOB
- InputStream
- OutputStream

WebLogic Server handles de-serialization for these object types so they can be used in client-side code.

Note: There are interoperability limitations when using different versions of WebLogic Server clients and servers. See "Support for Vendor Extensions Between Versions of WebLogic Server Clients and Servers" on page 5-28.

To use the extension methods exposed in the JDBC driver, you must include these steps in your application code:

- Import the driver interfaces from the JDBC driver used to create connections in the data source.
- Get a connection from the data source.
- Cast the connection object as the vendor's connection interface.
- Use the vendor extensions as described in the vendor's documentation.

The following sections provide details in code examples. For information about specific extension methods for a particular JDBC driver, refer to the documentation from the JDBC driver vendor.

Sample Code for Accessing Vendor Extensions to JDBC Interfaces

The following code examples use extension methods available in the Oracle Thin driver to illustrate how to use vendor extensions to JDBC. You can adapt these examples to fit methods exposed in your JDBC driver.

Import Packages to Access Vendor Extensions

Import the interfaces from the JDBC driver used to create the connection in the data source. This example uses interfaces from the Oracle Thin Driver.

```
import java.sql.*;
import java.util.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.sql.DataSource;
import oracle.jdbc.*;

// Import driver interfaces. The driver must be the same driver
// used to create the database connection in the data source.
```

Get a Connection

Establish the database connection using JNDI, DataSource and data source objects. For information, see "Using a JNDI Lookup to Obtain the Connection" on page 5-2.

```
// Get a valid DataSource object for a data source.
// Here we assume that getDataSource() takes
```

```
// care of those details.
javax.sql.DataSource ds = getDataSource(args);

// get a java.sql.Connection object from the DataSource
java.sql.Connection conn = ds.getConnection();
```

Cast the Connection as a Vendor Connection

Now that you have the connection, you can cast it as a vendor connection. This example uses the OracleConnection interface from the Oracle Thin Driver.

```
orConn = (oracle.jdbc.OracleConnection)conn;
// This replaces the deprecated process of casting the connection
// to a weblogic.jdbc.vendor.oracle.OracleConnection. For example:
// orConn = (weblogic.jdbc.vendor.oracle.OracleConnection)conn;
```

Use Vendor Extensions

The following code fragment shows how to use the Oracle Row Prefetch method available from the Oracle Thin driver.

Listing 5-4 Using a Vendor Extension

```
// Cast to OracleConnection and retrieve the
// default row prefetch value for this connection.

int default_prefetch =
   ((oracle.jdbc.OracleConnection)conn).getDefaultRowPrefetch();
// This replaces the deprecated process of casting the connection
// to a weblogic.jdbc.vendor.oracle.OracleConnection. For example:
// ((weblogic.jdbc.vendor.oracle.OracleConnection)conn).
//       getDefaultRowPrefetch();

System.out.println("Default row prefetch
    is " + default_prefetch);

java.sql.Statement stmt = conn.createStatement();

// Cast to OracleStatement and set the row prefetch
// value for this statement. Note that this
// prefetch value applies to the connection between
// WebLogic Server and the database.
```

```
   ((oracle.jdbc.OracleStatement)stmt).setRowPrefetch(20);
// This replaces the deprecated process of casting the
// statement to a weblogic.jdbc.vendor.oracle.OracleStatement.
// For example:
// ((weblogic.jdbc.vendor.oracle.OracleStatement)stmt).
//       setRowPrefetch(20);
   // Perform a normal sql query and process the results...
   String query = "select empno,ename from emp";
   java.sql.ResultSet rs = stmt.executeQuery(query);
   while(rs.next()) {
      java.math.BigDecimal empno = rs.getBigDecimal(1);
      String ename = rs.getString(2);
      System.out.println(empno + "\t" + ename);
   }
   rs.close();
   stmt.close();
   conn.close();
   conn = null;
 }
```

Using Oracle Extensions with the Oracle Thin Driver

For most extensions that Oracle provides, you can use the standard technique as described in "Using Vendor Extensions to JDBC Interfaces" on page 5-8. However, the Oracle Thin driver does not provide public interfaces for its extension methods in the following classes:

- `oracle.sql.ARRAY`
- `oracle.sql.STRUCT`
- `oracle.sql.REF`
- `oracle.sql.BLOB`
- `oracle.sql.CLOB`

WebLogic Server provides its own interfaces to access the extension methods for those classes:

- `weblogic.jdbc.vendor.oracle.OracleArray`

- `weblogic.jdbc.vendor.oracle.OracleStruct`
- `weblogic.jdbc.vendor.oracle.OracleRef`
- `weblogic.jdbc.vendor.oracle.OracleThinBlob`
- `weblogic.jdbc.vendor.oracle.OracleThinClob`

The following sections provide code samples for using the WebLogic Server interfaces for Oracle extensions. For a list of supported methods, see "Tables of Oracle Extension Interfaces and Supported Methods" on page 5-29. For more information, please refer to the Oracle documentation.

Note: You can use this process to use any of the WebLogic Server interfaces for Oracle extensions listed in the "Tables of Oracle Extension Interfaces and Supported Methods" on page 5-29. However, all but the interfaces listed above are deprecated and will be removed in a future release of WebLogic Server.

Limitations When Using Oracle JDBC Extensions

Please note the following limitations when using Oracle extensions to JDBC interfaces:

- You can use Oracle extensions for ARRAYs, REFs, and STRUCTs in server-side applications that use the same JVM as the server only. You cannot use Oracle extensions for ARRAYs, REFs, and STRUCTs in remote client applications.
- You cannot create ARRAYs, REFs, and STRUCTs in your applications. You can only retrieve existing ARRAY, REF, and STRUCT objects from a database. To create these objects in your applications, you must use a non-standard Oracle descriptor object, which is not supported in WebLogic Server.
- There are interoperability limitations when using different versions of WebLogic Server clients and servers. See "Support for Vendor Extensions Between Versions of WebLogic Server Clients and Servers" on page 5-28.

Sample Code for Accessing Oracle Extensions to JDBC Interfaces

The following code examples show how to access the WebLogic Server interfaces for Oracle extensions that are not available as public interfaces, including interfaces for:

- ARRAYs—See "Programming with ARRAYs" on page 5-13.
- STRUCTS—See "Programming with STRUCTs" on page 5-15.

- REFs—See "Programming with REFs" on page 5-20.
- BLOBs and CLOBs—See "Programming with BLOBs and CLOBs" on page 5-25.

If you selected the option to install server examples with WebLogic Server, see the JDBC examples for more code examples, typically at *WL_HOME*\samples\server\src\examples\jdbc, where *WL_HOME* is the folder where you installed WebLogic Server.

Programming with ARRAYs

In your WebLogic Server server-side applications, you can materialize an Oracle Collection (a SQL ARRAY) in a result set or from a callable statement as a Java array.

To use ARRAYs in WebLogic Server applications:

1. Import the required classes.
2. Get a connection and then create a statement for the connection.
3. Get the ARRAY using a result set or a callable statement.
4. Use the ARRAY as either a `java.sql.Array` or a `weblogic.jdbc.vendor.oracle.OracleArray`.
5. Use the standard Java methods (when used as a `java.sql.Array`) or Oracle extension methods (when cast as a `weblogic.jdbc.vendor.oracle.OracleArray`) to work with the data.

The following sections provide more details for these actions.

Note: You can use ARRAYs in server-side applications only. You cannot use ARRAYs in remote client applications.

Import Packages to Access Oracle Extensions

Import the Oracle interfaces used in this example. The OracleArray interface is counterpart to `oracle.sql.ARRAY` and can be used in the same way as the Oracle interface when using the methods supported by WebLogic Server.

```
import java.sql.*;
import java.util.*;
import javax.naming.Context;
import javax.naming.InitialContext;
```

```
import javax.sql.DataSource;
import weblogic.jdbc.vendor.oracle.*;
```

Establish the Connection

Establish the database connection using JNDI and DataSource objects. For information, see "Using a JNDI Lookup to Obtain the Connection" on page 5-2.

```
// Get a valid DataSource object.
// Here we assume that getDataSource() takes
// care of those details.
javax.sql.DataSource ds = getDataSource(args);

// get a java.sql.Connection object from the DataSource
java.sql.Connection conn = ds.getConnection();
```

Getting an ARRAY

You can use the `getArray()` methods for a callable statement or a result set to get a Java array. You can then use the array as a `java.sql.array` to use standard `java.sql.array` methods, or you can cast the array as a `weblogic.jdbc.vendor.oracle.OracleArray` to use the Oracle extension methods for an array.

The following example shows how to get a `java.sql.array` from a result set that contains an ARRAY. In the example, the query returns a result set that contains an object column—an ARRAY of test scores for a student.

```
try {
  conn = getConnection(url);
  stmt = conn.createStatement();
  String sql = "select * from students";
//Get the result set
  rs = stmt.executeQuery(sql);
  while(rs.next()) {
    BigDecimal id = rs.getBigDecimal("student_id");
    String name   = rs.getString("name");
    log("ArraysDAO.getStudents() -- Id = "+id.toString()+", Student =
"+name);
//Get the array from the result set
    Array scoreArray  = rs.getArray("test_scores");
    String[] scores = (String[])scoreArray.getArray();
```

```
    for (int i = 0; i < scores.length; i++) {
      log("   Test"+(i+1)+" = "+scores[i]);
    }
  }
```

Updating ARRAYs in the Database

To update an ARRAY in a database, you can Follow these steps:

1. Create an array in the database using PL/SQL, if the array you want to update does not already exist in the database.
2. Get the ARRAY using a result set or a callable statement.
3. Work with the array in your Java application as either a `java.sql.Array` or a `weblogic.jdbc.vendor.oracle.OracleArray`.
4. Update the array in the database using the `setArray()` method for a prepared statement or a callable statement. For example:

```
String sqlUpdate = "UPDATE SCOTT." + tableName + " SET col1 = ?";
conn  = ds.getConnection();
pstmt = conn.prepareStatement(sqlUpdate);
pstmt.setArray(1, array);
pstmt.executeUpdate();
```

Using Oracle Array Extension Methods

To use the Oracle extension methods for an ARRAY, you must first cast the array as a `weblogic.jdbc.vendor.oracle.OracleArray`. You can then make calls to the Oracle extension methods for ARRAYs. For example:

```
   oracle.sql.Datum[] oracleArray = null;
   oracleArray =
      ((weblogic.jdbc.vendor.oracle.OracleArray)scoreArray).getOracleArray
();
   String sqltype = null
   sqltype = oracleArray.getSQLTypeName()
```

Programming with STRUCTs

In your WebLogic Server applications, you can access and manipulate *objects* from an Oracle database. When you retrieve objects from an Oracle database, you can cast them as either custom Java objects or as STRUCTs (`java.sql.struct` or

`weblogic.jdbc.vendor.oracle.OracleStruct`). A STRUCT is a loosely typed data type for structured data which takes the place of custom classes in your applications. The STRUCT interface in the JDBC API includes several methods for manipulating the attribute values in a STRUCT. Oracle extends the STRUCT interface with several additional methods. WebLogic Server implements all of the standard methods and most of the Oracle extensions.

Note: Please note the following limitations when using STRUCTs:

- STRUCTs are supported for use with Oracle only. To use STRUCTs in your applications, you must use the Oracle Thin Driver to communicate with the database, typically through a data source.
- You can use STRUCTs in server-side applications only. You cannot use STRUCTs in client applications.

To use STRUCTs in WebLogic Server applications:

1. Import the required classes. (See "Import Packages to Access Oracle Extensions" on page 5-13.)
2. Get a connection. (See "Establish the Connection" on page 5-14.)
3. Use `getObject` to get the STRUCT.
4. Cast the STRUCT as a STRUCT, either `java.sql.Struct` (to use standard methods) or `weblogic.jdbc.vendor.oracle.OracleStruct` (to use standard and Oracle extension methods).
5. Use the standard or Oracle extension methods to work with the data.

The following sections provide more details for steps 3 through 5.

Getting a STRUCT

To get a database object as a STRUCT, you can use a query to create a result set and then use the `getObject` method to get the STRUCT from the result set. You then cast the STRUCT as a `java.sql.Struct` so you can use the standard Java methods. For example:

```
conn = ds.getConnection();
stmt = conn.createStatement();
rs = stmt.executeQuery("select * from people");
struct = (java.sql.Struct)(rs.getObject(2));
Object[] attrs = ((java.sql.Struct)struct).getAttributes();
```

WebLogic Server supports all of the JDBC API methods for STRUCTs:

- `getAttributes()`
- `getAttributes(java.util.Dictionary map)`
- `getSQLTypeName()`

Oracle supports the standard methods as well as the Oracle extensions. Therefore, when you cast a STRUCT as a `weblogic.jdbc.vendor.oracle.OracleStruct`, you can use both the standard and extension methods.

Using OracleStruct Extension Methods

To use the Oracle extension methods for a STRUCT, you must cast the `java.sql.Struct` (or the original `getObject` result) as a `weblogic.jdbc.vendor.oracle.OracleStruct`. For example:

```
java.sql.Struct struct =
   (weblogic.jdbc.vendor.oracle.OracleStruct)(rs.getObject(2));
```

WebLogic Server supports the following Oracle extensions:

- `getDescriptor()`
- `getOracleAttributes()`
- `getAutoBuffering()`
- `setAutoBuffering(boolean)`

Getting STRUCT Attributes

To get the value for an individual attribute in a STRUCT, you can use the standard JDBC API methods `getAttributes()` and `getAttributes(java.util.Dictionary map)`, or you can use the Oracle extension method `getOracleAttributes()`.

To use the standard method, you can create a result set, get a STRUCT from the result set, and then use the `getAttributes()` method. The method returns an array of ordered attributes. You can assign the attributes from the STRUCT (object in the database) to an object in the application, including Java language types. You can then manipulate the attributes individually. For example:

```
conn = ds.getConnection();
stmt = conn.createStatement();
rs = stmt.executeQuery("select * from people");
```

```
//The third column uses an object data type.
//Use getObject() to assign the object to an array of values.

struct = (java.sql.Struct)(rs.getObject(2));

Object[] attrs = ((java.sql.Struct)struct).getAttributes();

String address = attrs[1];
```

In the preceding example, the third column in the `people` table uses an object data type. The example shows how to assign the results from the `getObject` method to a Java object that contains an array of values, and then use individual values in the array as necessary.

You can also use the `getAttributes(java.util.Dictionary map)` method to get the attributes from a STRUCT. When you use this method, you must provide a hash table to map the data types in the Oracle object to Java language data types. For example:

```
java.util.Hashtable map = new java.util.Hashtable();

map.put("NUMBER", Class.forName("java.lang.Integer"));

map.put("VARCHAR", Class.forName("java.lang.String"));

Object[] attrs = ((java.sql.Struct)struct).getAttributes(map);

String address = attrs[1];
```

You can also use the Oracle extension method `getOracleAttributes()` to get the attributes for a STRUCT. You must first cast the STRUCT as a `weblogic.jdbc.vendor.oracle.OracleStruct`. This method returns a datum array of `oracle.sql.Datum` objects. For example:

```
oracle.sql.Datum[] attrs =
   ((weblogic.jdbc.vendor.oracle.OracleStruct)struct).getOracleAttribut
es();

oracle.sql.STRUCT address = (oracle.sql.STRUCT) attrs[1];

Object address_attrs[] = address.getAttributes();
```

The preceding example includes a nested STRUCT. That is, the second attribute in the datum array returned is another STRUCT.

Using STRUCTs to Update Objects in the Database

To update an object in the database using a STRUCT, you can use the `setObject` method in a prepared statement. For example:

```
conn  = ds.getConnection();
stmt = conn.createStatement();
ps = conn.prepareStatement ("UPDATE SCHEMA.people SET EMPLNAME = ?,
   EMPID = ? where EMPID = 101");
ps.setString (1, "Smith");
ps.setObject (2, struct);
ps.executeUpdate();
```

WebLogic Server supports all three versions of the `setObject` method.

Creating Objects in the Database

STRUCTs are typically used to materialize database objects in your Java application in place of custom Java classes that map to the database objects. In WebLogic Server applications, you cannot create STRUCTs that transfer to the database. However, you can use statements to create objects in the database that you can then retrieve and manipulate in your application. For example:

```
conn = ds.getConnection();
stmt = conn.createStatement();
cmd = "create type ob as object (ob1 int, ob2 int)"
stmt.execute(cmd);
cmd = "create table t1 of type ob";
stmt.execute(cmd);
cmd = "insert into t1 values (5, 5)"
stmt.execute(cmd);
```

Note: You cannot create STRUCTs in your applications. You can only retrieve existing objects from a database and cast them as STRUCTs. To create STRUCT objects in your applications, you must use a non-standard Oracle STRUCT descriptor object, which is not supported in WebLogic Server.

Automatic Buffering for STRUCT Attributes

To enhance the performance of your WebLogic Server applications that use STRUCTs, you can toggle automatic buffering with the `setAutoBuffering(boolean)` method. When automatic

buffering is set to `true`, the `weblogic.jdbc.vendor.oracle.OracleStruct` object keeps a local copy of all the attributes in the STRUCT in their converted form (materialized from SQL to Java language objects). When your application accesses the STRUCT again, the system does not have to convert the data again.

Note: Buffering the converted attributes my cause your application to use an excessive amount of memory. Consider potential memory usage when deciding to enable or disable automatic buffering.

The following example shows how to activate automatic buffering:

```
((weblogic.jdbc.vendor.oracle.OracleStruct)struct).setAutoBuffering(true);
```

You can also use the `getAutoBuffering()` method to determine the automatic buffering mode.

Programming with REFs

A REF is a logical pointer to a row object. When you retrieve a REF, you are actually getting a pointer to a value in another table. The REF target must be a row in an object table. You can use a REF to examine or update the object it refers to. You can also change a REF so that it points to a different object of the same object type or assign it a null value.

Note: Please note the following limitations when using REFs:

- REFs are supported for use with Oracle only. To use REFs in your applications, you must use the Oracle Thin Driver to communicate with the database, typically through a data source.
- You can use REFs in server-side applications only.

To use REFs in WebLogic Server applications, follow these steps:

1. Import the required classes. (See "Import Packages to Access Oracle Extensions" on page 5-13.)
2. Get a database connection. (See "Establish the Connection" on page 5-14.)
3. Get the REF using a result set or a callable statement.
4. Cast the result as a STRUCT or as a Java object. You can then manipulate data using STRUCT methods or methods for the Java object.

You can also create and update a REF in the database.

The following sections describe these steps 3 and 4 in greater detail.

Getting a REF

To get a REF in an application, you can use a query to create a result set and then use the `getRef` method to get the REF from the result set. You then cast the REF as a `java.sql.Ref` so you can use the built-in Java method. For example:

```
conn = ds.getConnection();
stmt = conn.createStatement();
rs = stmt.executeQuery("SELECT ref (s) FROM t1 s where s.ob1=5");
rs.next();
//Cast as a java.sql.Ref and get REF
ref = (java.sql.Ref) rs.getRef(1);
```

Note that the WHERE clause in the preceding example uses dot notation to specify the attribute in the referenced object.

After you cast the REF as a `java.sql.Ref`, you can use the Java API method `getBaseTypeName`, the only JDBC 2.0 standard method for REFs.

When you get a REF, you actually get a pointer to a value in an object table. To get or manipulate REF values, you must use the Oracle extensions, which are only available when you cast the `sql.java.Ref` as a `weblogic.jdbc.vendor.oracle.OracleRef`.

Using OracleRef Extension Methods

In order to use the Oracle extension methods for REFs, you must cast the REF as an Oracle REF. For example:

```
oracle.sql.StructDescriptor desc =
   ((weblogic.jdbc.vendor.oracle.OracleRef)ref).getDescriptor();
```

WebLogic Server supports the following Oracle extensions:

- `getDescriptor()`
- `getSTRUCT()`
- `getValue()`
- `getValue(dictionary)`
- `setValue(object)`

Getting a Value

Oracle provides two versions of the `getValue()` method—one that takes no parameters and one that requires a hash table for mapping return types. When you use either version of the `getValue()` method to get the value of an attribute in a REF, the method returns a either a STRUCT or a Java object.

The example below shows how to use the `getValue()` method without parameters. In this example, the REF is cast as an oracle.sql.STRUCT. You can then use the STRUCT methods to manipulate the value, as illustrated with the `getAttributes()` method.

```
oracle.sql.STRUCT student1 =
   (oracle.sql.STRUCT)((weblogic.jdbc.vendor.oracle.OracleRef)ref).getV
alue ();

Object attributes[] = student1.getAttributes();
```

You can also use the `getValue(dictionary)` method to get the value for a REF. You must provide a hash table to map data types in each attribute of the REF to Java language data types. For example:

```
java.util.Hashtable map = new java.util.Hashtable();

map.put("VARCHAR", Class.forName("java.lang.String"));

map.put("NUMBER", Class.forName("java.lang.Integer"));

oracle.sql.STRUCT result = (oracle.sql.STRUCT)
   ((weblogic.jdbc.vendor.oracle.OracleRef)ref).getValue (map);
```

Updating REF Values

When you update a REF, you can do any of the following:

- Change the value in the underlying table with the `setValue(object)` method.
- Change the location to which the REF points with a prepared statement or a callable statement.
- Set the value of the REF to null.

To use the `setValue(object)` method to update a REF value, you create an object with the new values for the REF, and then pass the object as a parameter of the `setValue` method. For example:

```
STUDENT s1 = new STUDENT();
```

```
   s1.setName("Terry Green");
   s1.setAge(20);
((weblogic.jdbc.vendor.oracle.OracleRef)ref).setValue(s1);
```

When you update the value for a REF with the `setValue(object)` method, you actually update the value in the table to which the REF points.

To update the *location* to which a REF points using a prepared statement, you can follow these basic steps:

1. Get a REF that points to the new location. You use this REF to replace the value of another REF.
2. Create a string for the SQL command to replace the location of an existing REF with the value of the new REF.
3. Create and execute a prepared statement.

For example:

```
   try {
   conn = ds.getConnection();
   stmt = conn.createStatement();
   //Get the REF.
   rs = stmt.executeQuery("SELECT ref (s) FROM t1 s where s.ob1=5");
   rs.next();
   ref = (java.sql.Ref) rs.getRef(1); //cast the REF as a java.sql.Ref
   }
   //Create and execute the prepared statement.
   String sqlUpdate = "update t3 s2 set col = ? where s2.col.ob1=20";
   pstmt = conn.prepareStatement(sqlUpdate);
   pstmt.setRef(1, ref);
   pstmt.executeUpdate();
```

To use a callable statement to update the location to which a REF points, you prepare the stored procedure, set any IN parameters and register any OUT parameters, and then execute the statement. The stored procedure updates the REF value, which is actually a location. For example:

```
conn = ds.getConnection();
stmt = conn.createStatement();
rs = stmt.executeQuery("SELECT ref (s) FROM t1 s where s.ob1=5");
rs.next();
ref1 = (java.sql.Ref) rs.getRef(1);
// Prepare the stored procedure
sql = "{call SP1 (?, ?)}";
cstmt = conn.prepareCall(sql);
// Set IN and register OUT params
cstmt.setRef(1, ref1);
cstmt.registerOutParameter(2, getRefType(), "USER.OB");
// Execute
cstmt.execute();
```

Creating a REF in the Database

You cannot create REF objects in your JDBC application—you can only retrieve existing REF objects from the database. However, you can create a REF in the database using statements or prepared statements. For example:

```
conn = ds.getConnection();
stmt = conn.createStatement();
cmd = "create type ob as object (ob1 int, ob2 int)"
stmt.execute(cmd);
cmd = "create table t1 of type ob";
stmt.execute(cmd);
cmd = "insert into t1 values (5, 5)"
stmt.execute(cmd);
cmd = "create table t2 (col ref ob)";
stmt.execute(cmd);
```

```
cmd = "insert into t2 select ref(p) from t1 where p.ob1=5";
stmt.execute(cmd);
```

The preceding example creates an object type (`ob`), a table (`t1`) of that object type, a table (`t2`) with a REF column that can point to instances of `ob` objects, and inserts a REF into the REF column. The REF points to a row in `t1` where the value in the first column is `5`.

Programming with BLOBs and CLOBs

This section contains sample code that demonstrates how to access the OracleBlob interface. You can use the syntax of this example for the OracleBlob interface, when using methods supported by WebLogic Server. See "Tables of Oracle Extension Interfaces and Supported Methods" on page 5-29.

Note: When working with BLOBs and CLOBs (referred to as "LOBs"), you must take transaction boundaries into account; for example, direct all read/writes to a particular LOB within a transaction. For additional information, refer to Oracle documentation about "LOB Locators and Transaction Boundaries" at the Oracle Web site at `http://www.oracle.com`.

Query to Select BLOB Locator from the DBMS

The BLOB Locator, or handle, is a reference to an Oracle Thin Driver BLOB:

```
String selectBlob = "select blobCol from myTable where blobKey = 666"
```

Declare the WebLogic Server java.sql Objects

The following code presumes the Connection is already established:

```
ResultSet rs = null;
Statement myStatement = null;
java.sql.Blob myRegularBlob = null;
java.io.OutputStream os = null;
```

Begin SQL Exception Block

In this try catch block, you get the BLOB locator and access the Oracle BLOB extension.

```
try {
     // get our BLOB locator..
```

```
      myStatement = myConnect.createStatement();
      rs = myStatement.executeQuery(selectBlob);
      while (rs.next()) {
        myRegularBlob = rs.getBlob("blobCol");
}
      // Access the underlying Oracle extension functionality for
      // writing. Cast to the OracleThinBlob interface to access
      // the Oracle method.
      os = ((OracleThinBlob)myRegularBlob).getBinaryOutputStream();
      ...
      } catch (SQLException sqe) {
        System.out.println("ERROR(general SQE): " +
          sqe.getMessage());
      }
```

Once you cast to the `Oracle.ThinBlob` interface, you can access the BEA supported methods.

Updating a CLOB Value Using a Prepared Statement

If you use a prepared statement to update a CLOB and the new value is shorter than the previous value, the CLOB will retain the characters that were not specifically replaced during the update. For example, if the current value of a CLOB is `abcdefghij` and you update the CLOB using a prepared statement with `zxyw`, the value in the CLOB is updated to `zxywefghij`. To correct values updated with a prepared statement, you should use the `dbms_lob.trim` procedure to remove the excess characters left after the update. See the Oracle documentation for more information about the `dbms_lob.trim` procedure.

Programming with Oracle Virtual Private Databases

An Oracle Virtual Private Database (VPD) is an aggregation of server-enforced, application-defined fine-grained access control, combined with a secure application context in the Oracle 9i database server. To use VPDs in your WebLogic Server application, you would typically do the following:

1. Create a JDBC data source in your WebLogic Server configuration that uses either the Oracle Thin driver or the Oracle OCI driver. See "Using WebLogic JDBC in an Application" on page 2-1 or "Create JDBC data sources" in the *Administration Console Online Help*.

Note: If you are using an XA-enabled version of the JDBC driver, you must set `KeepXAConnTillTxComplete=true`. See "JDBC Data Source: Configuration: Connection Pool" in the *Administration Console Online Help*.

2. Do the following in your application:

```
import weblogic.jdbc.extensions.WLConnection

// get a connection from a WLS JDBC data source
Connection conn = ds.getConnection();

// Get the underlying vendor connection object
oracle.jdbc.OracleConnection orConn = (oracle.jdbc.OracleConnection)
   (((WLConnection)conn).getVendorConnection());

// Set CLIENT_IDENTIFIER (which will be accessible from
// USERENV naming context on the database server side)
orConn.setClientIdentifier(clientId);

/* perform application specific work, preferably using conn instead of
orConn */

// clean up connection before returning to WLS JDBC data source
orConn.clearClientIdentifier(clientId);

// As soon as you are finished with vendor-specific calls,
// nullify the reference to the physical connection.
orConn = null;

// close the pooled connection
conn.close();
```

Note: This code uses an underlying physical connection from a pooled (logical) connection. See "Getting a Physical Connection from a Data Source" on page 5-4 for usage guidelines.

Oracle VPD with WebLogic Server

WebLogic Server provides support for the `oracle.jdbc.OracleConnection.setClientIdentitfier` and `oracle.jdbc.OracleConnection.clearClientIndentifier` methods without using the underlying physical connection from a pooled connection. To use VPDs in your WebLogic Server application, you would typically do the following:

```
import weblogic.jdbc.vendor.oracle.OracleConnection;
```

```
   // get a connection from a WLS JDBC data source
   Connection conn = ds.getConnection();

   // cast to the Oracle extension and set CLIENT_IDENTIFIER
   // (which will be accessible from USERENV naming context on
   //  the database server side)
      ((weblogic.jdbc.vendor.oracle.OracleConnection)conn).setClientIdenti
fier(clientId);

   /* perform application specific work */

   // clean up connection before returning to WLS JDBC data source
   ((OracleConnection)conn).clearClientIdentifier(clientId);

   // close the connection
   conn.close();
```

Support for Vendor Extensions Between Versions of WebLogic Server Clients and Servers

Because the way WebLogic Server supports vendor JDBC extensions was changed in WebLogic Server 8.1, interoperability between versions of client and servers is affected.

When a WebLogic Server 8.1 or later client interacts with a WebLogic Server 7.0 or earlier server, Oracle extensions are not supported. When the client application tries to cast the JDBC objects to the Oracle extension interfaces, it will get a `ClassCastException`. However, when a WebLogic Server 7.0 or earlier client interacts with a WebLogic Server 8.1 or later server, Oracle extensions *are* supported.

This applies to the following Oracle extension interfaces:

- `weblogic.jdbc.vendor.oracle.OracleConnection`
- `weblogic.jdbc.vendor.oracle.OracleStatement`
- `weblogic.jdbc.vendor.oracle.OraclePreparedStatement`
- `weblogic.jdbc.vendor.oracle.OracleCallableStatement`
- `weblogic.jdbc.vendor.oracle.OracleResultSet`
- `weblogic.jdbc.vendor.oracle.OracleThinBlob`
- `weblogic.jdbc.vendor.oracle.OracleThinClob`
- `weblogic.jdbc.vendor.oracle.OracleArray`

- `weblogic.jdbc.vendor.oracle.OracleRef`
- `weblogic.jdbc.vendor.oracle.OracleStruct`

Note: Standard JDBC interfaces are supported regardless of the client or server version.

Tables of Oracle Extension Interfaces and Supported Methods

In previous releases of WebLogic Server, only the JDBC extensions listed in the following tables were supported. The current release of WebLogic Server supports most extension methods exposed as a public interface in the vendor's JDBC driver. See "Using Vendor Extensions to JDBC Interfaces" on page 5-8 for instructions for using vendor extensions. Because the new internal mechanism for supporting vendor extensions does not rely on the previous implementation, several interfaces are no longer needed and are deprecated. These interfaces will be removed in a future release of WebLogic Server. See Table 5-1. BEA encourages you to use the alternative interface listed in the table.

Table 5-1 Deprecated Interfaces for Oracle JDBC Extensions

Deprecated Interface (supported in WebLogic Server 7.0 and earlier)	Instead, use this interface from Oracle (supported in WebLogic Server version 8.1 and later)
`weblogic.jdbc.vendor.oracle.OracleConnection`	`oracle.jdbc.OracleConnection`
`weblogic.jdbc.vendor.oracle.OracleStatement`	`oracle.jdbc.OracleStatement`
`weblogic.jdbc.vendor.oracle.OracleCallableStatement`	`oracle.jdbc.OracleCallableStatement`
`weblogic.jdbc.vendor.oracle.OraclePreparedStatement`	`oracle.jdbc.OraclePreparedStatement`
`weblogic.jdbc.vendor.oracle.OracleResultSet`	`oracle.jdbc.OracleResultSet`

The interfaces listed in Table 5-2 are still valid because Oracle does not provide interfaces to access these extension methods.

Table 5-2 Oracle Interfaces with Continued Support in WebLogic Server

Oracle Interface
`weblogic.jdbc.vendor.oracle.OracleArray`
`weblogic.jdbc.vendor.oracle.OracleRef`
`weblogic.jdbc.vendor.oracle.OracleStruct`
`weblogic.jdbc.vendor.oracle.OracleThinClob`
`weblogic.jdbc.vendor.oracle.OracleThinBlob`

The following tables describe the Oracle interfaces and supported methods you use with the Oracle Thin Driver (or another driver that supports these methods) to extend the standard JDBC (`java.sql.*`) interfaces.

Table 5-3 OracleConnection Interface

Extends	Method Signature

`OracleConnection` extends `java.sql.Connection` (This interface is **deprecated**. See Table 5-1.)	`void clearClientIdentifier(String s) throws java.sql.SQLException;`
	`boolean getAutoClose() throws java.sql.SQLException;`
	`String getDatabaseProductVersion() throws java.sql.SQLException;`
	`String getProtocolType() throws java.sql.SQLException;`
	`String getURL() throws java.sql.SQLException;`
	`String getUserName() throws java.sql.SQLException;`
	`boolean getBigEndian() throws java.sql.SQLException;`
	`boolean getDefaultAutoRefetch() throws java.sql.SQLException;`
	`boolean getIncludeSynonyms() throws java.sql.SQLException;`
	`boolean getRemarksReporting() throws java.sql.SQLException;`
	`boolean getReportRemarks() throws java.sql.SQLException;`

Table 5-3 OracleConnection Interface

Extends	Method Signature

`OracleConnection` extends `java.sql.Connection` (continued) (This interface is **deprecated**. See Table 5-1.)	`boolean getRestrictGetTables()`
	`throws java.sql.SQLException;`
	`boolean getUsingXAFlag()`
	`throws java.sql.SQLException;`
	`boolean getXAErrorFlag()`
	`throws java.sql.SQLException;`
	`boolean isCompatibleTo816()`
	`throws java.sql.SQLException;`
	`(Deprecated)`
	`byte[] getFDO(boolean b)`
	`throws java.sql.SQLException;`
	`int getDefaultExecuteBatch() throws`
	`java.sql.SQLException;`
	`int getDefaultRowPrefetch()`
	`throws java.sql.SQLException;`
	`int getStmtCacheSize()`
	`throws java.sql.SQLException;`
	`java.util.Properties getDBAccessProperties()`
	`throws java.sql.SQLException;`
	`short getDbCsId() throws java.sql.SQLException;`
	`short getJdbcCsId() throws java.sql.SQLException;`
	`short getStructAttrCsId()`
	`throws java.sql.SQLException;`
	`short getVersionNumber()`
	`throws java.sql.SQLException;`
	`void archive(int i, int j, String s)`
	`throws java.sql.SQLException;`

Table 5-3 OracleConnection Interface

Extends	Method Signature

`OracleConnection` extends `java.sql.Connection` (continued) (This interface is **deprecated**. See Table 5-1.)	`void close_statements() throws java.sql.SQLException;`
	`void initUserName() throws java.sql.SQLException;`
	`void logicalClose() throws java.sql.SQLException;`
	`void needLine() throws java.sql.SQLException;`
	`void printState() throws java.sql.SQLException;`
	`void registerSQLType(String s, String t) throws java.sql.SQLException;`
	`void releaseLine() throws java.sql.SQLException;`
	`void removeAllDescriptor() throws java.sql.SQLException;`
	`void removeDescriptor(String s) throws java.sql.SQLException;`
	`void setAutoClose(boolean on) throws java.sql.SQLException;`
	`void setClientIdentifier(String s) throws java.sql.SQLException;`
	`void clearClientIdentifier(String s) throws java.sql.SQLException;`
	`void setDefaultAutoRefetch(boolean b) throws java.sql.SQLException;`
	`void setDefaultExecuteBatch(int i) throws java.sql.SQLException;`
	`void setDefaultRowPrefetch(int i) throws java.sql.SQLException;`
	`void setFDO(byte[] b) throws java.sql.SQLException;`
	`void setIncludeSynonyms(boolean b) throws java.sql.SQLException;`

Table 5-3 OracleConnection Interface

Extends	Method Signature

`OracleConnection` extends `java.sql.Connection` (continued)
(This interface is **deprecated**. See Table 5-1.)
	`void setPhysicalStatus(boolean b) throws java.sql.SQLException;`
	`void setRemarksReporting(boolean b) throws java.sql.SQLException;`
	`void setRestrictGetTables(boolean b) throws java.sql.SQLException;`
	`void setStmtCacheSize(int i) throws java.sql.SQLException;`
	`void setStmtCacheSize(int i, boolean b) throws java.sql.SQLException;`
	`void setUsingXAFlag(boolean b) throws java.sql.SQLException;`
	`void setXAErrorFlag(boolean b) throws java.sql.SQLException;`
	`void shutdown(int i) throws java.sql.SQLException;`
	`void startup(String s, int i) throws java.sql.SQLException;`

Table 5-4 OracleStatement Interface

Extends	Method Signature

`OracleStatement` extends `java.sql.statement` (This interface is **deprecated**. See Table 5-1.)	`String getOriginalSql()`
	`  throws java.sql.SQLException;`
	`String getRevisedSql()`
	`  throws java.sql.SQLException;`
	`  (**Deprecated** in Oracle 8.1.7, removed in Oracle 9i.)`
	`boolean getAutoRefetch()`
	`  throws java.sql.SQLException;`
	`boolean is_value_null(boolean b, int i)`
	`  throws java.sql.SQLException;`
	`byte getSqlKind()`
	`  throws java.sql.SQLException;`
	`int creationState()`
	`  throws java.sql.SQLException;`
	`int getAutoRollback()`
	`  throws java.sql.SQLException;`
	`  (**Deprecated**)`
	`int getRowPrefetch()`
	`  throws java.sql.SQLException;`
	`int getWaitOption()`
	`  throws java.sql.SQLException;`
	`  (**Deprecated**)`
	`int sendBatch()`
	`  throws java.sql.SQLException;`

Table 5-4 OracleStatement Interface

Extends	Method Signature

`OracleStatement` extends `java.sql.statement` (continued) (This interface is **deprecated**. See Table 5-1.)	`void clearDefines()`
	`  throws java.sql.SQLException;`
	`void defineColumnType(int i, int j)`
	`  throws java.sql.SQLException;`
	`void defineColumnType(int i, int j, String s)`
	`  throws java.sql.SQLException;`
	`void defineColumnType(int i, int j, int k)`
	`  throws java.sql.SQLException;`
	`void describe()`
	`  throws java.sql.SQLException;`
	`void setAutoRefetch(boolean b)`
	`  throws java.sql.SQLException;`
	`void setAutoRollback(int i)`
	`  throws java.sql.SQLException;`
	`  (Deprecated)`
	`void setRowPrefetch(int i)`
	`  throws java.sql.SQLException;`
	`void setWaitOption(int i)`
	`  throws java.sql.SQLException;`
	`  (Deprecated)`

Table 5-5 OracleResultSet Interface

Extends	Method Signature

`OracleResultSet` extends `java.sql.ResultSet` (This interface is **deprecated**. See Table 5-1.)	`boolean getAutoRefetch() throws java.sql.SQLException;`
	
	`int getFirstUserColumnIndex()`
	`throws java.sql.SQLException;`
	
	`void closeStatementOnClose()`
	`throws java.sql.SQLException;`
	
	`void setAutoRefetch(boolean b)`
	`throws java.sql.SQLException;`
	
	`java.sql.ResultSet getCursor(int n)`
	`throws java.sql.SQLException;`
	
	`java.sql.ResultSet getCURSOR(String s)`
	`throws java.sql.SQLException;`

.

Table 5-6 OracleCallableStatement Interface

Extends	Method Signature

`OracleCallableStatement` extends `java.sql.CallableStatement` (This interface is **deprecated**. See Table 5-1.)	`void clearParameters()`
	`  throws java.sql.SQLException;`
	`void registerIndexTableOutParameter(int i,`
	`  int j, int k, int l)`
	`  throws java.sql.SQLException;`
	`void registerOutParameter`
	`  (int i, int j, int k, int l)`
	`  throws java.sql.SQLException;`
	`java.sql.ResultSet getCursor(int i)`
	`  throws java.sql.SQLException;`
	`java.io.InputStream getAsciiStream(int i)`
	`  throws java.sql.SQLException;`
	`java.io.InputStream getBinaryStream(int i)`
	`  throws java.sql.SQLException;`
	`java.io.InputStream getUnicodeStream(int i)`
	`  throws java.sql.SQLException;`

.

Table 5-7 OraclePreparedStatement Interface

Extends	Method Signature

`OraclePreparedStatement` extends `OracleStatement` and `java.sql.PreparedStatement`
(This interface is **deprecated**. See Table 5-1.)
	`int getExecuteBatch()`
	`throws java.sql.SQLException;`
	`void defineParameterType(int i, int j, int k)`
	`throws java.sql.SQLException;`
	`void setDisableStmtCaching(boolean b)`
	`throws java.sql.SQLException;`
	`void setExecuteBatch(int i)`
	`throws java.sql.SQLException;`
	`void setFixedCHAR(int i, String s)`
	`throws java.sql.SQLException;`
	`void setInternalBytes(int i, byte[] b, int j)`
	`throws java.sql.SQLException;`

Table 5-8 OracleArray Interface

Extends	Method Signature

`OracleArray` extends `java.sql.Array`	`public ArrayDescriptor getDescriptor() throws java.sql.SQLException;`
	`public Datum[] getOracleArray() throws SQLException;`
	`public Datum[] getOracleArray(long l, int i) throws SQLException;`
	`public String getSQLTypeName() throws java.sql.SQLException;`
	`public int length() throws java.sql.SQLException;`
	`public double[] getDoubleArray() throws java.sql.SQLException;`
	`public double[] getDoubleArray(long l, int i) throws java.sql.SQLException;`
	`public float[] getFloatArray() throws java.sql.SQLException;`
	`public float[] getFloatArray(long l, int i) throws java.sql.SQLException;`
	`public int[] getIntArray() throws java.sql.SQLException;`
	`public int[] getIntArray(long l, int i) throws java.sql.SQLException;`
	`public long[] getLongArray() throws java.sql.SQLException;`
	`public long[] getLongArray(long l, int i) throws java.sql.SQLException;`

Table 5-8 OracleArray Interface

Extends	Method Signature

`OracleArray` extends `java.sql.Array` (continued)	`public short[] getShortArray()`
	`  throws java.sql.SQLException;`
	
	`public short[] getShortArray(long l, int i)`
	`  throws java.sql.SQLException;`
	
	`public void setAutoBuffering(boolean flag)`
	`  throws java.sql.SQLException;`
	
	`public void setAutoIndexing(boolean flag)`
	`  throws java.sql.SQLException;`
	
	`public boolean getAutoBuffering()`
	`  throws java.sql.SQLException;`
	
	`public boolean getAutoIndexing()`
	`  throws java.sql.SQLException;`
	
	`public void setAutoIndexing(boolean flag, int i)`
	`  throws java.sql.SQLException;`

Table 5-9 OracleStruct Interface

Extends	Method Signature

`OracleStruct` extends `java.sql.Struct`	`public Object[] getAttributes()`
	`  throws java.sql.SQLException;`
	`public Object[] getAttributes(java.util.Dictionary map)`
	`  throws java.sql.SQLException;`
	`public Datum[] getOracleAttributes()`
	`  throws java.sql.SQLException;`
	`public oracle.sql.StructDescriptor getDescriptor()`
	`  throws java.sql.SQLException;`
	`public String getSQLTypeName()`
	`  throws java.sql.SQLException;`
	`public void setAutoBuffering(boolean flag)`
	`  throws java.sql.SQLException;`
	`public boolean getAutoBuffering()`
	`  throws java.sql.SQLException;`

Table 5-10 OracleRef Interface

Extends	Method Signature
OracleRef extends java.sql.Ref	public String getBaseTypeName() throws SQLException;
	public oracle.sql.StructDescriptor getDescriptor() throws SQLException;
	public oracle.sql.STRUCT getSTRUCT() throws SQLException;
	public Object getValue() throws SQLException;
	public Object getValue(Map map) throws SQLException;
	public void setValue(Object obj) throws SQLException;

Table 5-11 OracleThinBlob Interface

Extends	Method Signature
OracleThinBlob extends java.sql.Blob	int getBufferSize()throws java.sql.Exception
	int getChunkSize()throws java.sql.Exception
	int putBytes(long, int, byte[])throws java.sql.Exception
	int getBinaryOutputStream()throws java.sql.Exception

Table 5-12 OracleThinClob Interface

Extends	Method Signature

`OracleThinClob` extends `java.sql.Clob`	`public OutputStream getAsciiOutputStream() throws java.sql.Exception;`
	`public Writer getCharacterOutputStream() throws java.sql.Exception;`
	`public int getBufferSize() throws java.sql.Exception;`
	`public int getChunkSize() throws java.sql.Exception;`
	`public char[] getChars(long l, int i) throws java.sql.Exception;`
	`public int putChars(long start, char myChars[]) throws java.sql.Exception;`
	`public int putString(long l, String s) throws java.sql.Exception;`

CHAPTER 6

Using RowSets with WebLogic Server

The following sections describe characteristics and usage of WebLogic RowSets:

About RowSets

WebLogic Server includes an implementation of Java RowSets according to the specifications indicated in JSR-114. See the Sun Web site (http://java.sun.com/products/jdbc/download.html) for details about the specification. The WebLogic rowset implementation also includes extensions to the RowSets specification. These extensions make RowSets more useful in your applications.

A rowset is an extension of a Java ResultSet. Like a ResultSet, a rowset is a Java object that holds tabular data. However, a rowset adds significant flexibility to ResultSet features and reduces or eliminates some ResultSet limitations.

Types of RowSets

The WebLogic Server implementation of rowsets includes the following rowset types and utilities:

Standard RowSet Types:

- CachedRowSets
- FilteredRowSets
- WebRowSets
- JoinRowSets
- JDBCRowSets

WebLogic RowSet Extensions:

- WLCachedRowSets
- SharedRowSets
- SortedRowSets

- SQLPredicate, a SQL-Style RowSet Filter

Programming with RowSets

The Weblogic Server rowset implementation was designed with the expectation that you would work with a rowset in the following set of steps:

1. Create and configure the rowset — define the query, database connection, and other properties.
2. Populate the rowset with data — specify query parameters and execute the query.
3. Optionally, work with rowset metadata.
4. Optionally set the filter or sorter for the rowset.
5. Manipulate the data in the rowset — insert, update, and delete.
6. Synchronize data changes from the rowset to the database.

After synchronizing changes, the process can repeat starting with step 2 or 3, depending on the way your application is designed. See "Reusing a WebLogic RowSet After Completing a Transaction" on page 6-17.

The WebLogic rowset implementation includes a lifecycle framework that prevents a rowset object from getting into an unhealthy condition. Internally, WebLogic Server sets a lifecycle stage for the rowset as the rowset moves through the process described above. To reduce the risk of data loss, WebLogic Server limits the operations you can do on the rowset depending on the lifecycle stage of the rowset. For example, when the rowset is in the *Updating* stage, you can only call `updateXXX()` methods, such as `updateString()` and `updateInt()`, on the rowset until you call `updateRow()` to complete the update phase.

Some important notes:

- If you have pending changes, you cannot re-populate, filter, or sort the rowset. WebLogic Server prevents these operations on the rowset when the rowset data has changed but the changes have not been synchronized with the database to prevent the accidental loss of data changes.
- There is no implicit movement of the cursor! You must explicitly move the cursor from row to row.

- Rowset lifecycle stage is an internal process. There are no public APIs to access it. You cannot set the lifecycle stage. When you call acceptChanges() or restoreOriginal(), WebLogic Server rests the lifecycle stage of the rowset so you can begin again.

Note: When using a rowset in a client-side application, the *exact same* JDBC driver classes must be in the `CLASSPATH` on both the server and the client. If the driver classes do not match, you may see `java.rmi.UnmarshalException` exceptions.

See the comments in Listing 6-1 for an illustration of the lifecycle stages for a rowset from when it is created to when data changes are synchronized with the database.

CachedRowSets

The following sections describe using standard CachedRowSets with WebLogic Server:

Also see "WLCachedRowSets" on page 6-34 for information about using WebLogic extensions to the standard CachedRowSet object.

Characteristics

A CachedRowSet is a disconnected `ResultSet` object. Data in a CachedRowSet is stored in memory. CachedRowSets from the WebLogic Server implementation have the following characteristics:

- Can be used to insert, update, or delete data.
- Are serializable, so they can be passed to various application components, including thin clients and wireless devices.
- Include transaction handling to enable rowset reuse. See "Reusing a WebLogic RowSet After Completing a Transaction" on page 6-17.
- Use an optimistic concurrency control for synchronizing data changes in the rowset with the database.
- Use a SyncResolver object from a SyncProvider exception to resolve conflicts between data changes in the rowset and the database. See "Handling SyncProviderExceptions with a SyncResolver" on page 6-28.

Special Programming Considerations and Limitations for CachedRowSets

When designing your application, consider the following information:

- Entire RowSet Query Results Stored in Memory
- Data Contention

Entire RowSet Query Results Stored in Memory

Because a CachedRowSet does not hold a connection to the database, it must hold the entire query results in memory. If the query result is very large, you may see performance degradation or out-of-memory errors. For large data sets, a ResultSet may be more appropriate because it keeps a connection to the database, so it can hold partial query results in memory and return to the database for additional rows as needed.

Data Contention

CachedRowSets are most suitable for use with data that is not likely to be updated by another process between when the rowset is populated and when data changes in the rowset are synchronized with the database. Database changes during that period will cause data contention. See "Handling SyncProviderExceptions with a SyncResolver" on page 6-28 for more information about detecting and handling data contention.

Code Example

Listing 6-1 shows the basic workflow of a CachedRowSet. It includes comments that describe each major operation and its corresponding rowset lifecycle stage. Following the code example is a more detailed explanation of each of the major sections of the example.

Listing 6-1 Cached RowSet Code Example

```
import javax.sql.rowset.CachedRowSet;
import weblogic.jdbc.rowset.RowSetFactory;

public class CachedRowSetDemo {

public static void main (String[] args) {

//DESIGNING lifecycle stage - Create the rowset and set properties
  try {
    //Create a RowSetFactory instance.
    RowSetFactory rsfact = RowSetFactory.newInstance();
    CachedRowSet rs = rsfact.newCachedRowSet();
    //Set database access through a DataSource.
    rs.setDataSourceName(examples-dataSource-demoPool);
    //See "Database Connection Options" on page 6-9 for more options.
    //Set query command
    rs.setCommand("SELECT ID, FIRST_NAME, MIDDLE_NAME, LAST_NAME,
    PHONE, EMAIL FROM PHYSICIAN WHERE ID>?");

   //CONFIGURE QUERY lifecycle operation
    rs.setInt(1, 0);

   //POPULATING lifecycle stage - Execute the command to populate the rowset
    rs.execute();
   }

   //CONFIGURING METADATA - Populate first, then set MetaData,
   //including KeyColumns
    rs.setKeyColumns(new int[] { 1 });
    while (rs.next ()) //NAVIGATING lifecycle stage
      {
      System.out.println ("ID: " +rs.getInt (1));
      System.out.println ("FIRST_NAME: " +rs.getString (2));
```

```
      System.out.println ("MIDDLE_NAME: " +rs.getString (3));
      System.out.println ("LAST_NAME: " +rs.getString (4));
      System.out.println ("PHONE: " +rs.getString (5));
      System.out.println ("EMAIL: " +rs.getString (6));
      }
   }
//Working with data
//Delete rows in the rowset
  try {
  //MANIPULATING lifecycle stage - navigate to a row
  //(manually moving the cursor)
  rs.last();
  rs.deleteRow();
  //Note that the database is not updated yet.
  }
//Update a row in the rowset
  try {
  //MANIPULATING lifecycle stage - navigate to a row
  //(manually moving the cursor)
  rs.first();
  //UPDATING lifecycle stage - call an update() method
  rs.updateString(4, "Francis");
  //MANIPULATING lifecycle stage - finish update
  rs.updateRow();
  //Note that the database is not updated yet.
  }
//INSERTING lifecycle stage - Insert rows in the rowset
  try {
    rs.moveToInsertRow();
    rs.updateInt(1, 104);
    rs.updateString("FIRST_NAME", "Yuri");
    rs.updateString("MIDDLE_NAME", "M");
    rs.updateString("LAST_NAME", "Zhivago");
    rs.updateString("PHONE", "1234567812");
    rs.updateString("EMAIL", "Yuri@poet.com");
    rs.insertRow(); //"Finish Update" action;
```

```
    //MANIPULATING lifecycle stage - navigate to a row
    rs.moveToCurrentRow();
    //Note that the database is not updated yet.
  }
//Send all changes (delete, update, and insert) to the database.
//DESIGNING or POPULATING lifecycle stage - after synchronizing changes
//with the database, lifecycle stage depends on other environment settings.
//See "Reusing a WebLogic RowSet After Completing a Transaction" on
page 6-17.
  try {
    rs.acceptChanges();
    rs.close();
   }
}
```

Importing Classes and Interfaces for a CachedRowSet

For standard RowSets, you must import the following classes:

```
javax.sql.rowset.CachedRowSet;
weblogic.jdbc.rowset.RowSetFactory;
```

Creating a CachedRowSet

Rowsets are created from a factory interface. To create a rowset with WebLogic Server, follow these main steps:

1. Create a RowSetFactory instance, which serves as a factory to create rowset objects for use in your application. You can specify database connection properties in the RowSetFactory so that you can create RowSets with the same database connectivity using fewer lines of code.

   ```
   RowSetFactory rsfact = RowSetFactory.newInstance();
   ```

2. Create a WLCachedRowSet and cast it as a `javax.sql.rowset.CachedRowSet` object. By default, the WebLogic `newCachedRowSet()` RowSetFactory method creates a `WLCachedRowSet` object. You can use it as-is, but if you prefer to use the standard `CachedRowSet` object, you can cast the object as such.

```
CachedRowSet rs = rsfact.newCachedRowSet();
```

Setting CachedRowSet Properties

There are numerous rowset properties, such as concurrency type, data source name, transaction isolation level, and so forth, that you can set to determine the behavior of the rowset. You are required to set only those properties that are needed for your particular use of the rowset. For information about available properties, see the Javadoc for the `javax.sql.rowset.BaseRowSet` class.

Database Connection Options

In most applications, you populate a rowset with data from a database. You can set rowset database connectivity in any of the following ways:

- Automatically with a datasource—You can use the `setDataSourceName()` method to specify the JNDI name of a JDBC datasource. When you call `execute()` and `acceptChanges()`, the rowset gets a database connection from the datasource, uses it, and returns it to the pool of connections in the datasource. This is a preferred method.

  ```
  rs.setDataSourceName(examples-dataSource-demoPool);
  ```

- Manually get a database connection—In your application, you can get a database connection before the rowset needs it, and then pass the connection object as a parameter in the `execute()` and `acceptChanges()` methods. You must also close the connection as necessary.

  ```
  //Lookup DataSource and get a connection
  ctx = new InitialContext(ht);
  javax.sql.DataSource ds = (javax.sql.DataSource) ctx.lookup ("myDS");
  conn = ds.getConnection();

  //Pass the connection to the rowset
  rs.execute(conn);
  ```

  For more information about JDBC data sources, see "Getting a Database Connection from a DataSource Object" on page 2-1.

- Load the JDBC driver for a direct connection—When you load the JDBC driver and set the appropriate properties, the rowset creates a database connection when you call `execute()` and `acceptChanges()`. The rowset closes the connection immediately after it uses it. The rowset does not keep the connection between the `execute()` and `acceptChanges()` method calls.

```
Class.forName("com.pointbase.jdbc.jdbcUniversalDriver");
rs.setUrl("jdbc:pointbase:server://localhost/demo");
rs.setUsername("examples");
rs.setPassword("examples");
rs.execute();
```

- Set connectivity properties in the RowSetFactory—When you set database connection properties in the RowSetFactory, all rowsets created from the RowSetFactory inherit the connectivity properties. The preferred method is to lookup a datasource and then set the datasource property in the RowSetFactory with the `setDataSource()` method.

```
//Lookup DataSource and get a connection
ctx = new InitialContext(ht);
javax.sql.DataSource ds = (javax.sql.DataSource) ctx.lookup ("myDS");

//Set the datasource property on the RowSetFactory
rsfact.setDataSource(ds);
```

Populating a CachedRowSet

Populating a rowset is the act of filling the rowset with rows of data. The source of the data is most commonly a relational database. To populate a rowset with data from a database, you can use either of the following methods:

- Set an SQL command with the `setCommand()` method, then execute the command with the `execute()` method:

```
rs.setCommand("SELECT ID, FIRST_NAME, MIDDLE_NAME, LAST_NAME,
    PHONE, EMAIL FROM PHYSICIAN");
rs.execute();
```

- From an existing result set using the `populate()` method:

```
rs.populate(resultSet);
```

Note: If using a result set that is `ResultSet.TYPE_FORWARD_ONLY` , a SQLException will be thrown if you attempt to populate a row set with the following conditions:

  - If you call `CachedRowset.populate(ResultSet rs)` when the result set cursor is at a position beyond row 1.
  - If you call `CachedRowset.populate(ResultSet rs, int newPosition)` when `newPosition` is less than the current result set cursor position.

Setting CachedRowSet MetaData

In some cases, you may need to set metadata for the rowset in order to synchronize data changes in the rowset with data in the database. See "RowSet MetaData Settings for Database Updates" on page 6-14 for more information.

Working with Data in a CachedRowSet

After you populate the cached rowset with rows of data, you can work with the cached data in much the same way as you would work with data in a result set, except that before your changes are made in the database, you must explicitly call `acceptChanges()`.

Note: Delimiter identifiers may not be used for column or table names in rowsets. Delimiter identifiers are identifiers that need to be enclosed in double quotation marks when appearing in a SQL statement. They include identifiers that are SQL reserved words (e.g., `USER`, `DATE`, etc.) and names that are not identifiers. A valid identifier must start with a letter and contain only letters, numbers, and underscores.

Getting Data from a Row in a RowSet

To get data from a rowset, you use the `getXXX` methods just as you would with a result set. For example:

```
while (rs.next ())
   {
   int id = rs.getInt (1);
   String fname = rs.getString ("FIRST_NAME");
   String mname = rs.getString ("MIDDLE_NAME");
   String lname = rs.getString ("LAST_NAME"));
   }
```

Updating a Row in a RowSet

Data updates typically follow this course of events:

1. Navigate to the row or to an insert row.
2. Change the row with `updateXXX` methods.
3. Complete the operation with `updateRow()` or `insertRow()`.

Note that completing the operation does not synchronize your changes with the database. Changes are made to the rowset only. You must explicitly synchronize your changes by calling

`acceptChanges()`. For details, see "Synchronizing RowSet Changes with the Database" on page 6-13 later in this section.

When working with a rowset, WebLogic Server internally sets the lifecycle stage of the rowset after each operation on the rowset, and then limits further operations you can perform on the rowset based on its current lifecycle stage. After you begin modifying a row with update methods, you must complete the operation with `updateRow()` or `insertRow()` before you can work with data in any other rows, including moving the cursor to another row. See "Programming with RowSets" on page 6-3 for a complete discussion of rowset lifecycle stages and operations allowed for each stage.

To update a row, you move the cursor to the row you want to update, call `updateXXX` methods on individual columns within the row, then call `updateRow()` to complete the operation. For example:

```
rs.first();
rs.updateString(4, "Francis");
rs.updateRow();
```

Note: If you are updating same-named columns from more than one table, you must use the column index number to refer to the column in the update statement.

Inserting a Row in a RowSet

To insert a row, you move the cursor to a new insert row, update the column values within the row, then call `insertRow()` to add the row to the rowset. For example:

```
rs.moveToInsertRow();
rs.updateInt(1, 104);
rs.updateString("FIRST_NAME", "Yuri");
rs.updateString("MIDDLE_NAME", "M");
rs.updateString("LAST_NAME", "Zhivago");
rs.updateString("PHONE", "1234567812");
rs.updateString("EMAIL", "Yuri@poet.com");
rs.insertRow();
rs.moveToCurrentRow();
```

Note that you must explicitly move the cursor after inserting a row. There is no implicit movement of the cursor.

Deleting a Row in a RowSet

To delete a row in the rowset, you move the cursor to the row and call `deleteRow()`. For example:

```
rs.last();
rs.deleteRow();
```

Synchronizing RowSet Changes with the Database

After you make changes to individual rows in a rowset, you call `acceptChanges()` to propagate those changes to the database. For example:

```
rs.acceptChanges();
```

When you call `acceptChanges()`, the rowset connects to the database using the database connection information already used by the rowset (see "Database Connection Options" on page 6-9) or using a connection object passed with the `acceptChanges(connection)` method. You can call `acceptChanges()` after making changes to one row or several rows. Calling `acceptChanges()` after making all changes to the rowset is more efficient because the rowset connects to the database only once.

When using rowsets with WebLogic Server, WebLogic Server internally uses a `weblogic.jdbc.rowset.WLSyncProvider` object to read from and write to the database. The WLSyncProvider uses an optimistic concurrency algorithm for making changes to the database, which means that the design assumes data in the database will not be changed by another process during the time between when a rowset is populated to when rowset data changes are propagated to the database. Before writing changes to the database, the WLSyncProvider compares the data in the database against the original values in the rowset (values read into the rowset when the rowset was created or at the last synchronization). If any values in the database have changed, WebLogic Server throws a `javax.sql.rowset.spi.SyncProviderException` and does not write any changes to the database. You can catch the exception in your application and determine how to proceed. For more information, see "Handling SyncProviderExceptions with a SyncResolver" on page 6-28.

The `WLCachedRowSet` interface, an extension to the standard `CachedRowSet` interface, provides options for selecting an optimistic concurrency policy. See "Optimistic Concurrency Policies" on page 6-37 for more information.

After propagating changes to the database, WebLogic Server changes the lifecycle stage of the rowset to *Designing* or *Populating*, depending on your application environment. In the Designing stage, you must repopulate the rowset before you can use it again; in the Populating stage, you

can use the rowset with its current data. See "Reusing a WebLogic RowSet After Completing a Transaction" on page 6-17 for more details.

If you do not plan to use the rowset again, you should close it with the `close()` method. For example:

```
rs.close();
```

RowSet MetaData Settings for Database Updates

When populating a rowset with an SQL query, the WebLogic rowset implementation uses the `ResultSetMetaData` interface to automatically learn the table and column names of the data in the rowset. In many cases, this is enough information for the rowset to generate the required SQL to write changes back to the database. However, some JDBC drivers do not include table and column metadata for the rows returned by the query. When you attempt to synchronize data changes in the rowset with the database, you will see the following error:

```
java.sql.SQLException: Unable to determine the table name for column: column_name. Please ensure that you've called WLRowSetMetaData.setTableName to set a table name for this column.
```

Without the table name, you can use the rowset for read-only operations only. The rowset cannot issue updates unless the table name is specified programmatically. You may also need to set the primary key columns with the `setKeyColumns()` method. For example:

```
rs.setTableName(PHYSICIAN);
rs.setKeyColumns(new int[] { 1 });
```

See the documentation for the `javax.sql.rowset.CachedRowSet` interface for more details.

WebLogic RowSet Extensions for Working with MetaData

The following sections describe WebLogic rowset extensions that you can use to obtain or set the appropriate metadata for a rowset:

- "executeAndGuessTableName and executeAndGuessTableNameAndPrimaryKeys" on page 6-15
- "Setting Table and Primary Key Information Using the MetaData Interface" on page 6-15
- "Setting the Write Table" on page 6-15

executeAndGuessTableName and executeAndGuessTableNameAndPrimaryKeys

When populating a rowset with an SQL query, you typically use the `execute()` method to run the query and read the data. The `WLCachedRowSet` implementation provides the `executeAndGuessTableName` and `executeAndGuessTableNameAndPrimaryKeys` methods that extend the `execute` method to also determine the associated table metadata.

The `executeAndGuessTableName` method parses the associated SQL and sets the table name for all columns as the first word following the SQL keyword `FROM`.

The `executeAndGuessTableNameAndPrimaryKeys` method parses the SQL command to read the table name. It then uses the `java.sql.DatabaseMetaData` to determine the table's primary keys.

Note: These methods rely on support in the DBMS or JDBC driver. They do not work with all DBMSs or all JDBC drivers.

Setting Table and Primary Key Information Using the MetaData Interface

You can also choose to manually set the table and primary key information using the `WLRowSetMetaData` interface.

```
WLRowSetMetaData metaData = (WLRowSetMetaData) rowSet.getMetaData();

// Sets one table name for all columns
metaData.setTableName("employees");
```

or

```
metaData.setTableName("e_id", "employees");
metaData.setTableName("e_name", "employees");
```

You can also use the `WLRowSetMetaData` interface to identify primary key columns.

```
metaData.setPrimaryKeyColumn("e_id", true);
```

See the Javadoc for `weblogic.jdbc.rowset.WLRowSetMetaData` for more details.

Setting the Write Table

The `WLRowSetMetaData` interface includes the `setWriteTableName` method to indicate the only table that should be updated or deleted. This is typically used when a rowset is populated

with a join from multiple tables, but the rowset should only update one table. Any column that is not from the write table is marked as read-only.

For instance, a rowset might include a join of orders and customers. The write table could be set to orders. If deleteRow were called, it would delete the order row, but not delete the customer row.

Note: JSR-114 provides the `CachedRowSet.setTableName` that provides the same functionality as the WebLogic `CachedRowSetMetaData.setWriteTableName` method. Calling either method marks those columns that do NOT belong to the write table as read-only. WebLogic also provides the `CachedRowSetMetaData.setTableName` method which is used to map which table a column belongs to. When setting the write table using `setTableName`, be careful to implement the method using the appropriate API for your application.

RowSets and Transactions

Most database or JDBC applications use transactions, and RowSets support transactions, including JTA transactions. The common use case is to populate the RowSet in Transaction 1. Transaction 1 commits, and there are no database or application server locks on the underlying data. The RowSet holds the data in-memory, and it can be modified or shipped over the network to a client. When the application wishes to commit the changes to the database, it starts Transaction 2 and calls the RowSet's `acceptChanges` method. It then commits Transaction 2.

Integrating with JTA Global Transactions

The EJB container and the UserTransaction interface start transactions with the JTA transaction manager. The RowSet operations can participate in this transaction. To participate in the JTA transaction, the RowSet *must* use a transaction-aware DataSource (TxDataSource). The DataSource can be configured in the WebLogic Server console.

If an Optimistic conflict or other exception occurs during `acceptChanges`, the RowSet aborts the global JTA transaction. The application will typically re-read the data and process the update again in a new transaction.

Behavior of Rowsets Using Global Transactions

In the case of a failure or rollback, the data is rolled back from the database, but is not rolled back from the rowset. Before proceeding you should do one of the following:

- Call `rowset.refresh` to update the rowset with data from the database.

- Create a new rowset with current data.

Using Local Transactions

If a JTA global transaction is not being used, the RowSet uses a local transaction. It first calls `setAutoCommit(false)` on the connection, then it issues all of the SQL statements, and finally it calls `connection.commit()`. This attempts to commit the local transaction. This method should *not* be used when trying to integrate with a JTA transaction that was started by the EJB or JMS containers.

If an Optimistic conflict or other exception occurs during `acceptChanges`, the RowSet rolls back the local transaction. In this case, none of the SQL issued in `acceptChanges` will commit to the database.

Behavior of Rowsets Using Local Transactions

This section provides information on the behavior of rowsets in failed local transactions. The behavior depends on the type of connection object:

Calling connection.commit

In this situation, the connection object is not created by the rowset and initiates a local transaction by calling `connection.commit`. If the transaction fails or if the connection calls `connection.rollback`, the data is rolled back from the database, but is not rolled back in the rowset. Before proceeding, you must do one of the following:

- Call `rowset.refresh` to update the rowset with data from the database.
- Create a new rowset with current data.

Calling acceptChanges

In this situation, the rowset creates its own connection object and uses it to update the data in rowset by calling `acceptChanges`. In the case of failure or if the rowset calls `connection.rollback`, the data is be rolled back from the rowset and also from the database.

Reusing a WebLogic RowSet After Completing a Transaction

In many cases, after you synchronize changes in the rowset with the database, you may want to continue to use the rowset with its current data, which can improve application performance by reducing the number of database round trips. However, to reuse the rowset and its data,

WebLogic Server needs to make sure that any transaction in which the rowset participates has completed before allowing you to make further changes to the data.

If you use a rowset in a local transaction and if `autocommit=true` is set on the connection object before rowset data changes are synchronized with the database, you can reuse the rowset with its current data after synchronizing the data because the autocommit setting forces the local transaction to complete immediately. WebLogic Server can be sure that the local transaction is complete before any further changes are made to the rowset.

WebLogic Server cannot automatically be sure that all transactions are complete if you use a rowset in either of the following scenarios:

- In a global transaction
- In a local transaction using a connection object with `autocommit=false` to synchronize data changes with the database

With either of these conditions, before you can reuse a rowset with its current data, after calling `acceptChanges()` to synchronize your changes with the database, you must call `javax.sql.rowset.CachedRowSet.commit()` instead of `tx.commit()` or `java.sql.Connection.commit()` to commit the transaction. The `CachedRowSet.commit()` method wraps the `Connection.commit()` method and enables WebLogic Server to ensure that the transaction is complete before allowing changes to the rowset.

FilteredRowSets

The following sections describe using standard FilteredRowSets with WebLogic Server:

FilteredRowSet Characteristics

A FilteredRowSet enables you to work with a subset of cached rows and change the subset of rows while disconnected from the database. A filtered rowset is simply a cached rowset in which only certain rows are available for viewing, navigating, and manipulating. FilteredRowSets have the following characteristics:

- The rows available are determined by a `javax.sql.rowset.Predicate` object supplied by the application and set with the `setFilter()` method.
- The Predicate object must implement the `javax.sql.rowset.Predicate` interface. The Predicate interface includes the `public boolean evaluate(RowSet rs)` method, which evaluates each row in the rowset
  - If the method returns `true`, the row is available and visible.
  - If the method returns `false`, the row is not available or visible.

  See "Setting the Filter for a FilteredRowSet" on page 6-23 for more information.
- WebLogic Server provides the `weblogic.jdbc.rowset.SQLPredicate` class, which is an implementation of the `javax.sql.rowset.Predicate` interface that you can use to define a filter for a FilteredRowSet using SQL-like WHERE clause syntax. See "SQLPredicate, a SQL-Style RowSet Filter" on page 6-36.

Special Programming Considerations

RowSet Filters are Not Cumulative

Current behavior of WebLogic implementation of a FilteredRowSet is that when you set a filter for the second time on a FilteredRowSet, the new filter *replaces* the old filter. JSR-114 is not clear on this point. BEA decided this was the proper way to implement the setFilter method. Sun's reference implementation does not behave the same way. Sun's implementation further filters the filtered rows in the rowset. You can accomplish the same effect by changing the second filter to filter on all necessary criteria.

No Pending Changes Before Setting or Changing a Filter

If you have pending changes in a rowset before you set or change the rowset filter, you must either accept the changes (call `acceptChanges()`) or restore the rowset data to it pre-changed state (call `restoreOriginal()`). WebLogic Server considers navigating within a rowset to be indicative of a possible change and requires you to call either one of these methods before allowing you to change the rowset filter. Note that `acceptChanges()` includes a round-trip to the database, whereas `restoreOriginal()` does not.

FilteredRowSet Code Example

The following example shows how to create a cached rowset and then apply and change a filter using the WebLogic Server SQLPredicate.

Listing 6-2 FilteredRowSet Code Example

```
import javax.sql.rowset.FilteredRowSet;
import weblogic.jdbc.rowset.RowSetFactory;
import weblogic.jdbc.rowset.SQLPredicate;

public class FilteredRowSetDemo {

public static void main (String[] args) {

//DESIGNING lifecycle stage - Create the rowset and set properties
  try {
    //Create a RowSetFactory instance and from the factory,
    //create a WLCachedRowSet and cast it to a FilteredRowSet.
    RowSetFactory rsfact = RowSetFactory.newInstance();
    FilteredRowSet rs = rsfact.newCachedRowSet();
    //Set database access through a DataSource.
    //See "Database Connection Options" on page 6-9 for more options.
    rs.setDataSourceName(examples-dataSource-demoPool);
    rs.setCommand("SELECT ID, FIRST_NAME, MIDDLE_NAME, LAST_NAME,
    PHONE, EMAIL FROM PHYSICIAN WHERE ID>?");

   //CONFIGURE QUERY lifecycle operation - set values for query parameters.
    rs.setInt(1, 0);

   //POPULATING lifecycle stage - Execute the command to populate the rowset
    rs.execute();
  }

//CONFIGURING METADATA - Populate first, then set MetaData, including KeyColumns
  rs.setKeyColumns(new int[] { 1 });
```

```
  while (rs.next ())
//NAVIGATE operations put the rowset in the MANIPULATING lifecycle stage
    {
    System.out.println ("ID: " +rs.getInt (1));
    System.out.println ("FIRST_NAME: " +rs.getString (2));
    System.out.println ("MIDDLE_NAME: " +rs.getString (3));
    System.out.println ("LAST_NAME: " +rs.getString (4));
    System.out.println ("PHONE: " +rs.getString (5));
    System.out.println ("EMAIL: " +rs.getString (6));
    }
  }
//Need to accept changes or call restoreOriginal to put the rowset
//into the DESIGNING or POPULATING stage.
//After navigating, the rowset is in MANIPULATING stage,
//and you cannot change properties in that lifecycle stage.
  rs.restoreOriginal();

//S E T   F I L T E R
//use SQLPredicate class to create a SQLPredicate object,
//then pass the object in the setFilter method to filter the RowSet.

  SQLPredicate filter = new SQLPredicate("ID >= 103");
  rs.setFilter(filter);

  System.out.println("Filtered data: ");
  while (rs.next ())
    {
    System.out.println ("ID: " +rs.getInt (1));
    System.out.println ("FIRST_NAME: " +rs.getString (2));
    System.out.println ("MIDDLE_NAME: " +rs.getString (3));
    System.out.println ("LAST_NAME: " +rs.getString (4));
    System.out.println ("PHONE: " +rs.getString (5));
    System.out.println ("EMAIL: " +rs.getString (6));
    System.out.println (" ");
    }

//Need to accept changes or call restoreOriginal to put the rowset
//into the DESIGNING or POPULATING lifecycle stage.
//After navigating, the rowset is in MANIPULATING stage,
//and you cannot change properties in that lifecycle stage.
  rs.restoreOriginal();

//C H A N G I N G   F I L T E R

  SQLPredicate filter2 = new SQLPredicate("ID <= 103");
  rs.setFilter(filter2);

  System.out.println("Filtered data: ");
  while (rs.next ())
    {
```

```
    System.out.println ("ID: " +rs.getInt (1));
    System.out.println ("FIRST_NAME: " +rs.getString (2));
    System.out.println ("MIDDLE_NAME: " +rs.getString (3));
    System.out.println ("LAST_NAME: " +rs.getString (4));
    System.out.println ("PHONE: " +rs.getString (5));
    System.out.println ("EMAIL: " +rs.getString (6));
    System.out.println (" ");
    }

//Need to accept changes or call restoreOriginal to put the rowset
//into the DESIGNING or POPULATING lifecycle stage.
//After navigating, the rowset is in MANIPULATING stage,
//and you cannot change properties in that lifecycle stage.
  rs.restoreOriginal();

//R E M O V I N G   F I L T E R

  rs.setFilter(null);
  while (rs.next ())
    {
    System.out.println ("ID: " +rs.getInt (1));
    System.out.println ("FIRST_NAME: " +rs.getString (2));
    System.out.println ("MIDDLE_NAME: " +rs.getString (3));
    System.out.println ("LAST_NAME: " +rs.getString (4));
    System.out.println ("PHONE: " +rs.getString (5));
    System.out.println ("EMAIL: " +rs.getString (6));
    System.out.println (" ");
    }

  rs.close();
  }
}
```

Importing Classes and Interfaces for FilteredRowSets

For standard FilteredRowSets, you must import the following classes:

```
javax.sql.rowset.FilteredRowSet;
weblogic.jdbc.rowset.RowSetFactory;
```

The preceding code example also uses the `weblogic.jdbc.rowset.SQLPredicate` class to create a filter. In your application, you can use the `weblogic.jdbc.rowset.SQLPredicate` class or you can create your own filter class. See "Setting the Filter for a FilteredRowSet" on page 6-23 for more information.

Creating a FilteredRowSet

Rowsets are created from a factory interface. To create a FilteredRowSet with WebLogic Server, follow these main steps:

1. Create a RowSetFactory instance, which serves as a factory to create rowset objects for use in your application. For example:

   ```
   RowSetFactory rsfact = RowSetFactory.newInstance();
   ```

2. Create a WLCachedRowSet and cast it as a `javax.sql.rowset.FilteredRowSet` object. By default, the WebLogic `newCachedRowSet()` RowSetFactory method creates a `WLCachedRowSet` object. You can use it as-is, but if you prefer to use the standard `FilteredRowSet` object, you can cast the object as such. For example:

   ```
   FilteredRowSet rs = rsfact.newCachedRowSet();
   ```

Setting FilteredRowSet Properties

Property options for a FilteredRowSet are the same as those for a CachedRowSet. See "Setting CachedRowSet Properties" on page 6-9.

Database Connection Options for a FilteredRowSet

Database connection options for a FilteredRowSet are the same as those for a CachedRowSet. See "Database Connection Options" on page 6-9.

Populating a FilteredRowSet

Data population options for a FilteredRowSet are the same as those for a CachedRowSet. See "Populating a CachedRowSet" on page 6-10.

Setting FilteredRowSet MetaData

In some cases, you may need to set metadata for the rowset in order to synchronize data changes in the rowset with data in the database. See "RowSet MetaData Settings for Database Updates" on page 6-14 for more information.

Setting the Filter for a FilteredRowSet

To filter the rows in a FilteredRowSet, you must call the `setFilter` method and pass a predicate (filter) object as a parameter of the method. The predicate object is an instance of a class that

implements the `javax.sql.rowset.Predicate` interface. With the WebLogic implementation of FilteredRowSets, you can define your own filter or use an instance of the `weblogic.jdbc.rowset.SQLPredicate` class.

User-Defined RowSet Filter

When defining the filter for a FilteredRowSet, you follow these main steps:

1. Define a class that implements the `javax.sql.rowset.Predicate` interface with the filtering *behavior* you plan to use, such as limiting displayed rows to rows with a value in a particular column. For example, you may want to limit displayed rows based on a range of values for the ID column. The class you define would include logic to filter values for the ID column
2. Create an instance of the class (a filter) to specify the filtering *criteria* that you want to use. For example, you may want to see only rows with values in the ID column between 100 and 199.
3. Call `rowset.setFilter()` and pass the class as a parameter of the method.

Listing 6-3 shows an example of a class that implements the `javax.sql.rowset.Predicate` interface. This example shows a class that enables you to create a filter that evaluates a case-insensitive version of the value in a column. Listing 6-4 shows code to create an instance of the class, which determines the filter criteria, and then set the filter object as the filter for a FilteredRowSet.

Listing 6-3 Filter Class that Implements javax.sql.rowset.Predicate

```
package examples.jdbc.rowsets;

import javax.sql.rowset.Predicate;
import javax.sql.rowset.CachedRowSet;
import javax.sql.RowSet;
import java.sql.SQLException;

public class SearchPredicate implements Predicate, java.io.Serializable {
  private boolean DEBUG = false;
  private String col = null;
  private String criteria = null;

  //Constructor to create case-insensitive column - value comparison.
  public SearchPredicate(String col, String criteria) {
    this.col = col;
```

```
    this.criteria = criteria;
  }

  public boolean evaluate(RowSet rs)  {
    CachedRowSet crs = (CachedRowSet)rs;
    boolean bool = false;
    try {
      debug("evaluate(): "+crs.getString(col).toUpperCase()+" contains "+
          criteria.toUpperCase()+" = "+
          crs.getString(col).toUpperCase().contains(criteria.toUpperCase()));
      if (crs.getString(col).toUpperCase().contains(criteria.toUpperCase()))
        bool = true;
    } catch(Throwable t) {
      t.printStackTrace();
      throw new RuntimeException(t.getMessage());
    }
    return bool;
  }

  public boolean evaluate(Object o, String s) throws SQLException {
    throw new SQLException("String evaluation is not supported.");
  }

  public boolean evaluate(Object o, int i) throws SQLException {
    throw new SQLException("Int evaluation is not supported.");
  }

}
```

Listing 6-4 Code to Set a Filter for a FilteredRowSet

```
SearchPredicate pred = new SearchPredicate(ROWSET_LASTNAME, lastName);
rs.setFilter(pred);
```

WebLogic SQL-Style Filter

Weblogic Server provides the `weblogic.jdbc.rowset.SQLPredicate` class, which implements the `javax.sql.rowset.Predicate` interface. You can use the `SQLPredicate` class to define a filter using SQL-like WHERE clause syntax to filter rows in a rowset. For example:

```
SQLPredicate filter = new SQLPredicate("ID >= 103");
rs.setFilter(filter);
```

See "SQLPredicate, a SQL-Style RowSet Filter" on page 6-36 for more information.

Working with Data in a FilteredRowSet

Working with data in a FilteredRowSet is much the same as working with data in a CachedRowSet, except that when you insert a row or update a row, the changes that you make must be within the filtering criteria so that the row will remain in the set of rows displayed. For example, if the filter on the rowset allowed only rows with an ID column value of less than 105 to be displayed, if you tried to insert a row with a value of 106 in the ID column or update an ID value to 106, that operation would fail and throw an SQLException.

For more details about working with data, see "Working with Data in a CachedRowSet" on page 6-11.

WebRowSets

A WebRowSet is a cached rowset that can read and write a rowset in XML format. WebRowSets have the following characteristics:

- Uses the `readXml(java.io.InputStream iStream)` method to populate the rowset from an XML source.
- Uses the `writeXml(java.io.OutputStream oStream)` method to write data and metadata in XML for use by other application components or to send to a remote client.
- The XML code used to populate the rowset or written from the rowset conforms to the standard WebRowSet XML Schema definition available at `http://java.sun.com/xml/ns/jdbc/webrowset.xsd`.

For more information, see the Sun Web site at http://java.sun.com/products/jdbc/download.html and the Javadoc for the `javax.sql.rowset.WebRowSet` interface.

Note: WebLogic Server supports two schemas for rowsets: one for the standard WebRowSet and one for the WLCachedRowSet, which was implemented before JSR-114 was finalized.

Special Programming Considerations

- The WebLogic WebRowSets implementation supports two XML schemas (and APIs): one for the standard WebRowSet specification (available at `http://java.sun.com/xml/ns/jdbc/webrowset.xsd`.) and one for the WLCachedRowSet, which was implemented before JSR-114 was finalized.

- If you are using only WebLogic Server rowsets, you can use either schema. The proprietary schema offers the following benefits:
  - Has more element types.
  - Is used by rowsets in BEA Workshop for WebLogic Platform.
- To interact with other rowset implementations, you must use the standard schema.

JoinRowSets

A JoinRowSet is a number of disconnected RowSet objects joined together in a single rowset by a SQL JOIN. JoinRowSets have the following characteristics:

- Each rowset added to the JoinRowSet must have a "match" column specified in the addRowSet method used to add the rowset to the JoinRowSet. For example:

  ```
  addRowSet(javax.sql.RowSet[] rowset,java.lang.String[] columnName);
  ```

- You can set the join type using setJoinType method. The following join types are supported:

  ```
  CROSS_JOIN
  FULL_JOIN
  INNER_JOIN
  LEFT_OUTER_JOIN
  RIGHT_OUTER_JOIN
  ```

- Enables you to join data while disconnected from the database.
- JoinRowSets are for read-only use. JoinRowSets cannot be used to update data in the database.
- Match columns in a JoinRowSet are limited to four data types: Number, Boolean, Date, and String. Table 6-1 provides more details about data types allowed for a match column in a JoinRowSet.

Table 6-1 Data Types Allowed for Match Columns

Left Data Type in the Join	Allowed Right Data Types in the Join
Number	Number String
Boolean	Boolean String
Date	Date String
String	String Number Boolean Date

For more information about JoinRowSets, see the Javadoc for the `javax.sql.rowset.Joinable` and `JoinRowSet` interfaces.

JDBCRowSets

A JDBCRowSet is a wrapper around a ResultSet object that enables you to use the result set as a JavaBeans component. Note that a JDBCRowSet is a connected rowset. All other rowset types are disconnected rowsets.

For more information, see the Javadoc for the `javax.sql.rowset.JdbcRowSet` interface.

Handling SyncProviderExceptions with a SyncResolver

When you call `acceptChanges()` to propagate changes in a rowset to the database, WebLogic Server compares the original data in the rowset (data since the last synchronization) based on an optimistic concurrency policy with the data in the database. If it detects data changes, it throws a `javax.sql.rowset.spi.SyncProviderException`. By default, your application does not have to do anything, but the changes in the rowset will not be synchronized in the database.You can design your application to handle these exceptions and process the data changes as is suitable for your system.

Note: For `javax.sql.rowset.CachedRowSets`, WebLogic Server compares all original values in all rows in the rowset with the corresponding rows in the database. For `weblogic.jdbc.rowset.WLCachedRowSet` or other WebLogic extended rowset types, WebLogic Server makes the data comparison based on the optimistic concurrency setting. See "Optimistic Concurrency Policies" on page 6-37 for more details.

The main steps for handling a `SyncProviderException` are:

1. Catch the `javax.sql.rowset.spi.SyncProviderException`.
2. Get the SyncResolver object from the exception. See "Getting a SyncResolver Object" on page 6-33.
3. Page through conflicts using `nextConflict()` or any other navigation method. "Navigating in a SyncResolver Object" on page 6-33.
4. Determine the correct value, then set it with `setResolvedValue()`, which sets the value in the rowset. See "Setting the Resolved Value for a RowSet Data Synchronization Conflict" on page 6-34.
5. Repeat steps 3 and 4 for each conflicted value.
6. Call `rowset.acceptChanges()` on the rowset (not the SyncResolver) to synchronize changes with the database using the new resolved values. See "Synchronizing Changes" on page 6-34.

For more details about SyncResolvers and the `SyncProviderException`, see the RowSets specification or the Javadoc for the `SyncResolver` interface.

Note: Before you begin to resolve the SyncProviderException, make sure that no other processes will update the data.

RowSet Data Synchronization Conflict Types

Table 6-2 lists the types of conflict scenarios that can occur when synchronizing data changes from a rowset to the database.

Table 6-2 Conflict Types When Synchronizing RowSet Changes in the Database

RowSet Data Change Type	Database Data Change Type	Notes
Update	Update	Values in the same row in the rowset and database have changed. The syncresolver status is SyncResolver.UPDATE_ROW_CONFLICT. Your application may need to supply logic to resolve the conflict or may need to present the new data to the user.
Update	Delete	Values in the row in the rowset have been updated, but the row has been deleted in the database. The syncresolver status is SyncResolver.UPDATE_ROW_CONFLICT. Your application may need to supply logic to decide whether to leave the row as deleted (as it is in the database) or to restore the row and persist changes from the rowset. • To leave the row as deleted, revert the changes to the row in the rowset. • To restore the row with changes, insert a new row with the desired values. Note that if the row is deleted in the database, there is no conflict value. When you call `getConflictValue()`, WebLogic Server throws a `weblogic.jdbc.rowset.RowNotFoundException`.

Table 6-2 Conflict Types When Synchronizing RowSet Changes in the Database

RowSet Data Change Type	Database Data Change Type	Notes
Delete	Update	The row has been deleted in the rowset, but the row has been updated in the database. The syncresolver status is SyncResolver.DELETE_ROW_CONFLICT. Your application may need to supply logic to decide whether to delete the row (as it is in the rowset) or to keep the row and persist changes currently in the database. Note that in this scenario, all values in the row will be conflicted values. To keep the row with the current values in the database, call `setResolvedValue` to set the resolved value for each column in the row to the current value in the database. To proceed with the delete, call `syncprovider.deleteRow()`.
Delete	Delete	The row has been deleted in the rowset and has been deleted in the database by another process.The syncresolver status is SyncResolver.DELETE_ROW_CONFLICT. To resolve the SyncProviderException, you must revert the delete operation on the row in the rowset. Note that there will be no conflict value (not `null`, either) for any column in the row. When you call `getConflictValue()`, WebLogic Server throws a `weblogic.jdbc.rowset.RowNotFoundException`.
Insert	Insert	If a row is inserted in the rowset and a row is inserted in the database, a primary key conflict may occur, in which case an SQL exception will be thrown. You cannot directly handle this conflict type using a SyncResolver because a SyncProviderException is not thrown.

SyncResolver Code Example

Listing 6-5 shows an abbreviated example of how to use a SyncResolver to resolve conflicting values between the rowset and the database. This example checks the value for known column names in each row in the SyncResolver in which there is a conflict. Details about the example are explained in the sections that follow the example.

Listing 6-5 SyncResolver Abbreviated Code Example

```
try {
    rs.acceptChanges();
} catch (SyncProviderException spex) {
    SyncResolver syncresolver = spex.getSyncResolver();
    while (syncresolver.nextConflict())  {
        int status = syncresolver.getStatus();
        int rownum = syncresolver.getRow();

        rs.absolute(rownum);
        //check for null in each column
        //write out the conflict
        //set resolved value to value in the db for this example
        //handle exception for deleted row in the database
        try {
           Object idConflictValue = syncresolver.getConflictValue("ID");
           if (idConflictValue != null)  {
              System.out.println("ID value in db: " + idConflictValue);
             System.out.println("ID value in rowset: " + rs.getInt("ID"));
              syncresolver.setResolvedValue("ID", idConflictValue);
            System.out.println("Set resolved value to " + idConflictValue);
            }
           else {
              System.out.println("ID: NULL - no conflict");
           }
        } catch (RowNotFoundException e)  {
          System.out.println("An exception was thrown when requesting a ");
          System.out.println("value for ID. This row was ");
          System.out.println("deleted in the database.");
        }
. . .
    }
    try {
        rs.acceptChanges();
    } catch (Exception ignore2) {
        }
```

```
}
```

Getting a SyncResolver Object

To handle a `SyncProviderException`, you can catch the exception and get a `SyncResolver` object from it. For example:

```
try {
   rowset.acceptChanges();
} catch (SyncProviderException spex) {
   SyncResolver syncresolver = spex.getSyncResolver();
. . .
}
```

A SyncResolver is a rowset that implements the `SyncResolver` interface. A SyncResolver object contains a row for every row in the original rowset. For values without a conflict, the value in the SyncResolver is null. For values with a conflict, the value is the current value in the database.

Navigating in a SyncResolver Object

With a SyncResolver object, you can page through all conflicts and set the appropriate value for each conflict value. The SyncResolver interface includes the `nextConflict()` and `previousConflict()` methods that you can use to navigate directly to the next row in the SyncResolver that has a conflict value other than `null`. Because a SyncResolver object is a rowset, you can also use all of the rowset navigation methods to move the cursor to any row in the SyncResolver. However, the `nextConflict()` and `previousConflict()` methods enable you to easily skip rows that do not contain conflict values.

After you move the cursor to a conflict row, you must check the value in each column with the `getConflictValue()` method to find the values in the database that conflict with the values in the rowset, and then compare values to determine how to handle the conflict. For rows with values that do not conflict, the return value is `null`. If the row was deleted in the database, there is no value to return, so an exception is thrown.

Note: In the WebLogic rowsets implementation, a value conflict occurs if any value in a row in the database differs from the values read into the rowset when the rowset was created or when it was last synchronized.

An example of code to compare values in the rowset and database:

```
syncresolver.nextConflict()
```

```
for (int i = 1; i <= colCount; i++) {
   if (syncresolver.getConflictValue(i) != null)  {
      rsValue = rs.getObject(i);
      resolverValue = syncresolver.getConflictValue(i);
      . . .
     // compare values in the rowset and SyncResolver to determine
     // which should be the resolved value (the value to persist)
   }
}
```

Setting the Resolved Value for a RowSet Data Synchronization Conflict

To set the appropriate value to persist in the database, you call `setResolvedValue()`. For example:

```
syncresolver.setResolvedValue(i, resolvedValue);
```

The `setResolvedValue()` method makes the following changes:

- Sets the value to persist in the database. That is, it sets the current value in the rowset. When changes are synchronized, the new value will be persisted to the database.
- Changes the original value for the rowset data to the current value in the database. The original value was the value since the last synchronization. After calling `setResolvedValue()`, the original value becomes the current value in the database.
- Changes the WHERE clause in the synchronization call so that updates are made to appropriate rows in the database.

Synchronizing Changes

After resolving conflicting values in the SyncResolver, you must synchronize your changes with the database. To do that, you call `rowset.acceptChanges()`. again. The `acceptChanges()` call closes the SyncResolver object and releases locks on the database after the synchronization completes.

WLCachedRowSets

A WLCachedRowSet is an extension of CachedRowSets, FilteredRowSets, WebRowSets, and SortedRowSets. JoinRowSets have the following characteristics:

- In the WebLogic Server RowSets implementation, all rowsets originate as a WLCachedRowset. WLCachedRowSets can be interchangeably used as any of the standard rowset types that it extends.
- WLCachedRowSets include convenience methods that help make using rowsets easier and also include methods for setting optimistic concurrency options and data synchronization options.

For more information, see the Javadoc for the `weblogic.jdbc.rowset.WLCachedRowSet` interface.

SharedRowSets

Rowsets can be used by a single thread. They cannot be shared by multiple threads. A SharedRowSet extends CachedRowSets so that additional CachedRowSets can be created for use in other threads based on the data in an original CachedRowSet. SharedRowSets have the following characteristics:

- Each SharedRowSet is a shallow copy of the original rowset (with references to data in the original rowset instead of a copy of the data) with its own context (cursor, filter, sorter, pending changes, and sync provider).
- When data changes from any of the SharedRowSets are synchronized with the database, the base CachedRowSet is updated as well.
- Using SharedRowSets can increase performance by reducing the number of database round-trips required by an application.

To create a SharedRowSet, you use the `createShared()` method in the WLCachedRowSet interface and cast the result as a WLCachedRowSet. For example:

```
WLCachedRowSet sharedrowset = (WLCachedRowSet)rowset.createShared();
```

SortedRowSets

A SortedRowSet extends CachedRowSets so that rows in a CachedRowSet can be sorted based on the Comparator object provided by the application. SortedRowSets have the following characteristics:

- Sorting is set in a way similar to way filtering is set for a FilteredRowSet, except that sorting is based on a `java.util.Comparator` object instead of a `javax.sql.rowset.Predicate` object:

a. The application creates a `Comparator` object with the desired sorting behavior.

b. The application then sets the sorting criteria with the `setSorter(java.util.Comparator)` method.

- Sorting is done in memory rather than depending on the database management system for sort processing. Using SortedRowSets can increase application performance by reducing the number of database round-trips.
- WebLogic Server provides the SQLComparator object, which implements `java.util.Comparator`. You can use it to sort rows in a SortedRowSet by passing the list of columns that you want use as sorting criteria. For example:

```
rs.setSorter(new
weblogic.jdbc.rowset.SQLComparator("columnA,columnB,columnC"));
```

For more information, see the Javadocs for the following:

- `weblogic.jdbc.rowset.SortedRowSet` interface
- `weblogic.jdbc.rowset.SQLComparator` class

SQLPredicate, a SQL-Style RowSet Filter

What is SQLPredicate?

WebLogic Server provides the `weblogic.jdbc.rowset.SQLPredicate` class, which is an implementation of the `javax.sql.rowset.Predicate` interface. You can use the `SQLPredicate` class to define a filter for a FilteredRowSet using SQL-like WHERE clause syntax.

SQLPredicate Grammar

The SQLPredicate class borrows its grammar from the JMS selector grammar, which is very similar to the grammar for an SQL select WHERE clause.

Some important notes:

- When referencing a column, you must use the column name; you cannot use column index number.
- The grammar supports the use of operators and mathematical operations, for example:

```
(colA + ColB) >=100.
```

- In constructing the WHERE clause, you can use simple datatypes only, including:
  - String
  - Int
  - Boolean
  - Float
- Complex data types are *not* supported:
  - Array
  - BLOB
  - CLOB
  - Date

Code Example

```
//S E T   F I L T E R
//use SQLPredicate class to create a SQLPredicate object,
//then pass the object in the setFilter method to filter the RowSet.

SQLPredicate filter = new SQLPredicate("ID >= 103");
rs.setFilter(filter);
```

For more information, see the Javadoc for the `weblogic.jdbc.rowset.SQLPredicate` class.

Optimistic Concurrency Policies

In most cases, populating a rowset with data and updating the database occur in separate transactions. The underlying data in the database can change in the time between the two transactions. The WebLogic Server rowset implementation (WLCachedRowSet) uses optimistic concurrency control to ensure data consistency.

With optimistic concurrency, RowSets work on the assumption that multiple users are unlikely to change the same data at the same time. Therefore, as part of the disconnected rowset model, the rowset does not lock database resources. However, before writing changes to the database, the rowset must check to make sure that the data to be changed in the database has not already changed since the data was read into the rowset.

The UPDATE and DELETE statements issued by the rowset include WHERE clauses that are used to verify the data in the database against what was read when the rowset was populated. If the rowset detects that the underlying data in the database has changed, it issues an `OptimisticConflictException`. The application can catch this exception and determine how to proceed. Typically, applications will refresh the updated data and present it to the user again.

The WLCachedRowSet implementation offers several optimistic concurrency policies that determine what SQL the rowset issues to verify the underlying database data:

- VERIFY_READ_COLUMNS
- VERIFY_MODIFIED_COLUMNS
- VERIFY_SELECTED_COLUMNS
- VERIFY_NONE
- VERIFY_AUTO_VERSION_COLUMNS
- VERIFY_VERSION_COLUMNS

To illustrate the differences between these policies, we will use an example that uses the following:

- A very simple employees table with 3 columns:

```
CREATE TABLE employees (
  e_id integer primary key,
  e_salary integer,
  e_name varchar(25)
);
```

- A single row in the table:

```
e_id = 1, e_salary = 10000, and e_name = 'John Smith'
```

In the example for each of the optimistic concurrency policies listed below, the rowset will read this row from the employees table and set John Smith's salary to 20000. The example will then show how the optimistic concurrency policy affects the SQL code issued by the rowset.

VERIFY_READ_COLUMNS

The default rowset optimistic concurrency control policy is VERIFY_READ_COLUMNS. When the rowset issues an UPDATE or DELETE, it includes all columns that were read from the database in the WHERE clause. This verifies that the value in all columns that were initially read into the rowset have not changed.

In our example update, the rowset issues:

```
UPDATE employees SET e_salary = 20000
   WHERE e_id = 1 AND e_salary=10000 AND e_name = 'John Smith';
```

VERIFY_MODIFIED_COLUMNS

The VERIFY_MODIFIED_COLUMNS policy only includes the primary key columns and the updated columns in the WHERE clause. It is useful if your application only cares if its updated columns are consistent. It does allow your update to commit if columns that have not been updated have changed since the data has been read.

In our example update, the rowset issues:

```
UPDATE employees SET e_salary = 20000
   WHERE e_id = 1 AND e_salary=10000
```

The `e_id` column is included since it is a primary key column. The `e_salary` column is a modified column so it is included as well. The `e_name` column was only read so it is not verified.

VERIFY_SELECTED_COLUMNS

The VERIFY_SELECTED_COLUMNS includes the primary key columns and columns you specify in the WHERE clause.

```
WLRowSetMetaData metaData = (WLRowSetMetaData) rowSet.getMetaData();
metaData.setOptimisticPolicy(WLRowSetMetaData.VERIFY_SELECTED_COLUMNS);
// Only verify the e_salary column
metaData.setVerifySelectedColumn("e_salary", true);

metaData.acceptChanges();
```

In our example update, the rowset issues:

```
UPDATE employees SET e_salary = 20000
   WHERE e_id = 1 AND e_salary=10000
```

The `e_id` column is included since it is a primary key column. The `e_salary` column is a selected column so it is included as well.

VERIFY_NONE

The VERIFY_NONE policy only includes the primary key columns in the WHERE clause. It does not provide any additional verification on the database data.

In our example update, the rowset issues:

```
UPDATE employees SET e_salary = 20000 WHERE e_id = 1
```

VERIFY_AUTO_VERSION_COLUMNS

The VERIFY_AUTO_VERSION_COLUMNS includes the primary key columns as well as a separate version column that you specify in the WHERE clause. The rowset will also automatically increment the version column as part of the update. This version column must be an integer type. The database schema must be updated to include a separate version column (`e_version`). Assume for our example this column currently has a value of 1.

```
metaData.setOptimisticPolicy(WLRowSetMetaData.
   VERIFY_AUTO_VERSION_COLUMNS);

metaData.setAutoVersionColumn("e_version", true);

metaData.acceptChanges();
```

In our example update, the rowset issues:

```
UPDATE employees SET e_salary = 20000, e_version = 2
WHERE e_id = 1 AND e_version = 1
```

The `e_version` column is automatically incremented in the SET clause. The WHERE clause verified the primary key column and the version column.

VERIFY_VERSION_COLUMNS

The VERIFY_VERSION_COLUMNS has the rowset check the primary key columns as well as a separate version column. The rowset does not increment the version column as part of the update. The database schema must be updated to include a separate version column (`e_version`). Assume for our example this column currently has a value of 1.

```
metaData.setOptimisticPolicy(WLRowSetMetaData.VERIFY_VERSION_COLUMNS);

metaData.setVersionColumn("e_version", true);

metaData.acceptChanges();
```

In our example update, the rowset issues:

```
UPDATE employees SET e_salary = 20000
WHERE e_id = 1 AND e_version = 1
```

The WHERE clause verifies the primary key column and the version column. The rowset does not increment the version column so this must be handled by the database. Some databases provide automatic version columns that increment when the row is updated. It is also possible to use a database trigger to handle this type of update.

Optimistic Concurrency Control Limitations

The Optimistic policies only verify UPDATE and DELETE statements against the row they are changing. Read-only rows are not verified against the database.

Most databases do not allow BLOB or CLOB columns in the WHERE clause so the rowset never verifies BLOB or CLOB columns.

When multiple tables are included in the rowset, the rowset only verifies tables that have been updated.

Choosing an Optimistic Policy

The default VERIFY_READ_COLUMNS provides a strong-level of consistency at the expense of some performance. Since all columns that were initially read must be sent to the database and compared in the database, there is some additional overhead to this policy. VERIFY_READ_COLUMNS is appropriate when strong levels of consistency are needed, and the database tables cannot be modified to include a version column.

The VERIFY_SELECTED_COLUMNS is useful when the developer needs complete control over the verification and wants to use application-specific knowledge to fine-tune the SQL.

The VERIFY_AUTO_VERSION_COLUMNS provides the same level of consistency as VERIFY_READ_COLUMNS but only has to compare a single integer column. This policy also handles incrementing the version column so it requires a minimal amount of database setup.

The VERIFY_VERSION_COLUMNS is recommended for production systems that want the highest level of performance and consistency. Like VERIFY_AUTO_VERSION_COLUMNS, it provides a high level of consistency while only incurring a single column comparison in the database. VERIFY_VERSION_COLUMNS requires that the database handle incrementing the version column. Some databases provide a column type that automatically increments itself on updates, but this behavior can also be implemented with a database trigger.

The VERIFY_MODIFIED_COLUMNS and VERIFY_NONE decrease the consistency guarantees, but they also decrease the likelihood of an optimistic conflict. You should consider these policies when performance and avoiding conflicts outweigh the need for higher level of data consistency.

Performance Options

Consider the following performance options when using RowSets.

JDBC Batching

The rowset implementation includes support for JDBC batch operations. Instead of sending each SQL statement individually to the JDBC driver, a batch sends a collection of statements in one bulk operation to the JDBC driver. Batching is disabled by default, but it generally improves performance when large numbers of updates occur in a single transaction. It is worthwhile to benchmark with this option enabled and disabled for your application and database.

The WLCachedRowSet interface contains the methods `setBatchInserts(boolean)`, `setBatchDeletes(boolean)`, and `setBatchUpdates(boolean)` to control batching of INSERT, DELETE, and UPDATE statements.

Note: The `setBatchInserts`, `setBatchDeletes`, or `setBatchUpdates` methods must be called before the `acceptChanges` method is called.

Oracle Batching Limitations

Since the WLCachedRowSet relies on optimistic concurrency control, it needs to determine whether an update or delete command has succeeded or an optimistic conflict occurred. The WLCachedRowSet implementation relies on the JDBC driver to report the number of rows updated by a statement to determine whether a conflict occurred or not. In the case where 0 rows were updated, the WLCachedRowSet knows that a conflict did occur.

Oracle JDBC drivers return `java.sql.Statement.SUCCESS_NO_INFO` when batch updates are executed, so the rowset implementation cannot use the return value to determine whether a conflict occurred.

When the rowset detects that batching is used with an Oracle database, it automatically changes its batching behavior:

Batched inserts perform as usual since they are not verified.

Batched updates run as normal, but the rowset issues an extra SELECT query to check whether the batched update encountered an optimistic conflict.

Batched deletes use group deletes since this is more efficient than executing a batched delete followed by a SELECT verification query.

Group Deletes

When multiple rows are deleted, the rowset would normally issue a DELETE statement for each deleted row. When group deletes are enabled, the rowset issues a single DELETE statement with a WHERE clause that includes the deleted rows.

For instance, if we were deleting 3 employees from our table, the rowset would normally issue:

```
DELETE FROM employees WHERE e_id = 3 AND e_version = 1;
DELETE FROM employees WHERE e_id = 4 AND e_version = 3;
DELETE FROM employees WHERE e_id = 5 AND e_version = 10;
```

When group deletes are enabled, the rowset issues:

```
DELETE FROM employees
WHERE e_id = 3 AND e_version = 1 OR
      e_id = 4 AND e_version = 3 OR
      e_id = 5 AND e_version = 10;
```

You can use the `WLRowSetMetaData.setGroupDeleteSize` to determine the number of rows included in a single DELETE statement. The default value is 50.

CHAPTER 7

Troubleshooting JDBC

The following sections describe some common issues when developing JDBC applications:

Problems with Oracle on UNIX

Check the threading model you are using. *Green* threads can conflict with the kernel threads used by OCI. When using Oracle drivers, WebLogic recommends that you use *native* threads. You can specify this by adding the `-native` flag when you start Java.

Thread-related Problems on UNIX

On UNIX, two threading models are available: green threads and native threads. For more information, read about the JDK for the Solaris operating environment on the Sun Web site at `http://www.java.sun.com`.

You can determine what type of threads you are using by checking the environment variable called `THREADS_TYPE`. If this variable is not set, you can check the shell script in your Java installation bin directory.

Some of the problems are related to the implementation of threads in the JVM for each operating system. Not all JVMs handle operating-system specific threading issues equally well. Here are some hints to avoid thread-related problems:

- If you are using Oracle drivers, use *native* threads.
- If you are using HP UNIX, upgrade to version 11.x, because there are compatibility issues with the JVM in earlier versions, such as HP UX 10.20.
- On HP UNIX, the new JDK does not append the green-threads library to the `SHLIB_PATH`. The current JDK can not find the shared library (`.sl`) unless the library is in the path defined by `SHLIB_PATH`. To check the current value of `SHLIB_PATH`, at the command line type:

  ```
  $ echo $SHLIB_PATH
  ```

  Use the `set` or `setenv` command (depending on your shell) to append the WebLogic shared library to the path defined by the symbol `SHLIB_PATH`. For the shared library to be recognized in a location that is not part of your `SHLIB_PATH`, you will need to contact your system administrator.

Closing JDBC Objects

BEA recommends—and good programming practice dictates—that you always close JDBC objects, such as `Connections`, `Statements`, and `ResultSets`, in a `finally` block to make sure that your program executes efficiently. Here is a general example:

Listing 7-1 Closing a JDBC Object

```
try {
Driver d =
(Driver)Class.forName("weblogic.jdbc.oci.Driver").newInstance();
Connection conn = d.connect("jdbc:weblogic:oracle:myserver",
                                "scott", "tiger");
    Statement stmt = conn.createStatement();
    stmt.execute("select * from emp");
    ResultSet rs = stmt.getResultSet();
    // do work
    }
```

```
catch (Exception e) {
  // handle any exceptions as appropriate
}
finally {
  try {rs.close();}
  catch (Exception rse) {}
  try {stmt.close();}
  catch (Exception sse) {}
  try {conn.close();
  catch (Exception cse) {}
}
```

Abandoning JDBC Objects

You should also avoid the following practice, which creates abandoned JDBC objects:

```
//Do not do this.
stmt.executeQuery();
rs = stmt.getResultSet();

//Do this instead
rs = stmt.executeQuery();
```

The first line in this example creates a result set that is lost and can be garbage collected immediately.

Using Microsoft SQL with Nested Triggers

The following section provides troubleshooting information when using nested triggers on some Microsoft SQL databases:

- "Exceeding the Nesting Level" on page 7-4
- "Using Triggers and EJBs" on page 7-5

For information on supported data bases and data base drivers, see Supported Configurations.

Exceeding the Nesting Level

You may encounter a SQL Server error indicating that the nesting level has been exceeded on some SQL Server databases.

For example:

```
CREATE TABLE EmployeeEJBTable (name varchar(50) not null,salary int, card
varchar(50), primary key (name))


CREATE TABLE CardEJBTable (cardno varchar(50) not null, employee
varchar(50), primary key (cardno), foreign key (employee) references
EmployeeEJB Table(name) on delete cascade)

CREATE TRIGGER  card on EmployeeEJBTable for delete as delete
CardEJBTable  where employee in (select name from deleted)


CREATE TRIGGER  emp on CardEJBTable for delete as delete EmployeeEJBTable
where card in (select cardno from deleted)


insert into EmployeeEJBTable values ('1',1000,'1')

insert into CardEJBTable values ('1','1')

DELETE FROM CardEJBTable WHERE cardno = 1
```

Results in the following error message:

```
Maximum stored procedure, function, trigger, or view nesting level
exceeded  (limit 32).
```

To work around this issue, do the following:

1. Run the following script to reset the nested trigger level to 0:

   ```
   -- Start batch
   exec sp_configure 'nested triggers', 0  -- This set's the new value.
   reconfigure with override -- This makes the change permanent
   -- End batch
   ```

2. Verify the current value the SQL server by running the following script:

   ```
   exec sp_configure 'nested triggers'
   ```

Using Triggers and EJBs

Applications using EJBs with a Microsoft driver may encounter situations when the return code from the `execute()` method is 0, when the expected value is 1 (1 record deleted).

For example:

```
CREATE TABLE EmployeeEJBTable (name varchar(50) not null,salary int, card
varchar(50), primary key (name))


CREATE TABLE CardEJBTable (cardno varchar(50) not null, employee
varchar(50), primary key (cardno), foreign key (employee) references
EmployeeEJB Table(name) on delete cascade)

CREATE TRIGGER  emp on CardEJBTable for delete as delete EmployeeEJBTable
where card in (select cardno from deleted)


insert into EmployeeEJBTable values ('1',1000,'1')

insert into CardEJBTable values ('1','1')

DELETE FROM CardEJBTable WHERE cardno = 1
```

The EJB code assumes that the record is not found and throws an appropriate error message.

To work around this issue, run the following script:

```
exec sp_configure 'show advanced options', 1
reconfigure with override
exec sp_configure 'disallow results from triggers',1
reconfigure with override
```