DEVELOPING JAVA APPLICATIONS WITH ORACLE'S JDBC DRIVERS ¾ NEW FEATURES

Douglas Surber, Oracle Corp.

ABSTRACT

The JDBC 2.0 API provides several new features to develop Java applications that access a database. Oracle JDBC drivers already provide support for structured types, and Oracle8 objects. In Oracle 8i Release 2, the Oracle drivers fully comply with Java2 and JDBC 2.0. Oracle JDBC drivers support the JDBC extensions for XA and various types of scrollable cursors. Applications are able to use these features in exactly the same way in all of Oracle's drivers, including the server side driver inside the database. This paper discusses Oracle's support for the latest JDBC 2.0 features as well as other new features.

OVERVIEW

The purpose of this paper is to provide a survey of some of the new technology in the Oracle 8i Release 2 JDBC drivers. You should gain enough information about the new stuff to recognize opportunities for applying it in your work. This paper is not a tutorial on how to use these features, nor is it a reference manual. You should refer to the appropriate Oracle manual for detailed information on how to use any of the features discussed here. Additionally, this paper previews some forthcoming technology. As always, information about forthcoming technology can change at any time and in any way up until it is released

Here is an outline of the topics covered in this paper:

- JDBC 2.0 features
    - ResultSet enhancements
    - Batch updates
    - Advanced data types
    - RowSets
    - JNDI
    - Connection Pooling
    - Distributed Transactions
    - Other Features
- JNI 1.1 support
- JDK Support
    - Support for JDK 1.2 with all Drivers
    - Support for JDK 1.1 with Thick and Thin Drivers
    - Discontinued support for JDK 1.0.2
- Performance
- Coming Attractions
    - New Interfaces

- Statement Cache
- New Driver

JDBC 2.0

JDBC 2.0 defines lots of neat stuff. We implemented nearly all of it.

ResultSet Enhancements

The Oracle 8i Release 2 JDBC drivers support the JDBC 2.0 scrollable result set capability. JDBC 2.0 defines three types of result set: forward-only, scroll-insensitive, and scroll-sensitive. As their names suggest, a forward-only result set can only move the cursor forward. This is the result set defined in JDBC 1.0 API. A scroll-insensitive result set can move both backward and forward but is insensitive to changes committed by other transactions or other statements within the same transaction. A scroll-sensitive result set can move both backward and forward and is sensitive to changes committed by other transactions or other statements within the same transaction.

For each result set type, an application may choose from two different concurrency types for a result set: read-only or updatable. A read-only result set does not support update of its contents; an updatable result set supports updating its contents through the result set. Given three result set types and two concurrency types, there are six different kinds of result sets:

- Forward_only/Read_only
- Forward_only/Updatable
- Scroll_insensitive/Read_only
- Scroll_insensitive/Updatable
- Scroll_sensitive/Read_only
- Scroll_sensitive/Updatable

The first result set kind is the one defined in JDBC 1.0 API, and supported in all releases of Oracle 8i Release 2 JDBC drivers. The remaining five kinds are additions in JDBC 2.0. All result set kinds are supported in both the JDK 1.1 and JDK 1.2 OracleOracle 8i Release 2 JDBC Drivers. Support for these additional kinds in the JDK 1.1 drivers is an Oracle extension.

Implementation

Since the Oracle database does not support scrollable cursors, the Oracle 8i Release 2 JDBC drivers implement scrolling as part of the driver itself. The driver maintains a client side in-memory cache to store all the query results. When the user application moves the cursor, the logical cursor is moved to point to the requested row in the cache. Since every row of the result is cached, if the result set contains many rows, many columns, or very large columns, the cache may consume all available memory and crash the VM. It's very important to NOT to use a scrollable result set for a "big" result set.

The Oracle 8i Release 2 JDBC drivers allow you to provide your own implementation of ResultSet cache. You do this by implementing the OracleResultSetCache interface and passing an instance of your implementing class to OracleStatement:setResultSetCache() before statement execution. OralceResultSetCache is an interface defined by Oracle which defines the accessors to put/get the data in/from the cache. You might want to provide your own cache implementation is to utilize client side resource that the default implementation does not, such as disk.

To support updatability and change sensitivity, Oracle JDBC drivers cache the ROWID along with each row. The drivers generate SQL to update or read changes using this ROWID. You do not have to include the ROWID in your query, the drivers automatically add it when necessary.

If you update a row, the driver generates a SQL statement that updates all of the retrieved columns of the row that has the cached ROWID.

UPDATE (original query) SET col_1 = val_1 col_2 = val_2, … , col_n = val_m
WHERE ROWID = rowid

A result set implements change sensitivity by retrieving some rows again after they have already been retrieved initially by the original query. The result set maintains a window that consists of a range of rows. Those rows are presumed to be up-to-date. If you scroll within that window no fetches are executed. If you scroll outside that window a new window is defined with a new range of rows. The result set then refreshes those rows by generating a SQL query like

```
SELECT col₁,col₂,...,colₙ FROM tablename
WHERE ROWID = rowid₁ OR ROWID = rowid ₂ OR … OR ROWID = rowid ₘ
```

After updating the cache with the newly retrieved values, the result set considers the new window to be up-to-date. Whether on not a row is considered up-to-date depends not on changes in the database, or even on time, but rather on the size of the window and the pattern of access. If you wish to manually refresh the window you can call ResultSet.refreshRow which will refresh the current window. The size of the window is the fetch size (see "Other Features" below). A scroll-sensitive result set is not magic; it just retrieves the rows again and again, making it hugely less efficient than other types of result set.

RESTRICTIONS

Since scrollable result sets are implemented by the driver rather than by the underlying database there are a number of restrictions.

- Not all SQL queries can return all result set types. If you request an unsupported result set type, the execute method will still return a result set, but it will be of the supported type that best approximates your request. Many queries cannot return any type other than forward-only/insensitive. One quick way to check a questionable statement is to execute it using SQL*Plus (or some other tool) and add ROWID to the list of selected columns. If this fails then the query must be forward-only/insensitive.
- The driver does not check for conflicts when doing ResultSet.deleteRow() and ResultSet.updateRow()
- The driver does not enforce write lock for updatable result set.

An update conflict occurs when the value(s) in a row is changed by other committed transaction, but you try to update the same row. For both ResultSet.deleteRow() and ResultSet.updateRow(), an Oracle 8i Release 2 JDBC driver uses the "ROWID" value to uniquely identify a row in a database table. As long as the ROWID remains valid, the driver performs the delete/update operation on the row with the matched ROWID. If the row's column values in the database are changed by other committed transaction, the driver ignores these changes and writes the new values or deletes the row without warning. This has the potential to affect data integrity.

If detecting update conflicts is important in your application, you may impose a write lock the select rows by modifying your SQL query to contain the "FOR UPDATE" keywords. Since only a single write lock may be held at a time on a data item, this can reduce concurrency and substantially reduce performance, but it guarantees no update conflict will occur.

- Any rows inserted or deleted by others are not visible. Once the result set is open, the row's order and membership are fixed. The only exception is that calling deleteRow() on a ResultSet removes the row from that ResultSet.

The driver does not detect that changes have been made by self or others. Therefore, you cannot call the rowUpdated, rowDeleted, or rowInserted in ResultSet to determine whether the current row has been changed or not.

BATCH UPDATES

Prior to 8i Release 2, Oracle JDBC drivers provided a batch update facility. JDBC 2.0 specifies a slightly different batch update facility. The Oracle 8i Release 2 versions of the Oracle drivers support both. Although there are slight differences in behavior, much of the implementing code and data storage is the same. For this reason, you absolutely cannot mix the two types of batching on a single connection. You must use one or the other (or neither) exclusively on any given connection. If you mix the two, the results will be unpredictable.

The Oracle style of batching is automatic; once it is turned on, batches are accumulated and sent to the server without any other change in the code or action by the program. The JDBC 2.0 style of batching is explicit; the program must explicitly add statements to the batch and explicitly execute the batch.

Oracle Style:

```
stmt.setBatch(10);
stmt.setInt(1, 100);
stmt.execute();
  .
  .    repeat eight more times
  .
stmt.execute();
```

The last execute will cause all ten batched values to be flushed to the database. Note that if the argument to setBatch were set to zero, each execute would be immediately sent to the database and executed. If the argument were set to five, then the statements would be executed in two groups of five.

JDBC 2.0 Style

```
stmt.setInt(1, 1);
stmt.addBatch();
  .
  .    repeat eight more times
  .
stmt.addBatch();
stmt.executeBatch();
```

This example also executes the statements in a batch of ten, but this is explicitly determined by the calls to addBatch and executeBatch. The size of the batch is not determined to be a specific number as in the Oracle style, but rather by whatever number of times addBatch is called between intervening calls to executeBatch.

Both approaches have their advantages and disadvantages. The Oracle 8i Release 2 versions of both the JDK 1.2 and JDK 1.2 Oracle JDBC drivers support both.

ADVANCED DATA TYPES

Previous releases of Oracle JDBC drivers have support for object data. This support was nearly identical with the JDBC 2.0 specification. The JDK 1.2 drivers introduced in Oracle 8i Release 2 have fully compliant support for object data. The JDK 1.1 drivers support for object data is unchanged.

In JDK 1.1 the package java.sql does not define the types that support object data, Array, Blob, Clob, Ref, SQLData, and Struct. As a result it is impossible for a JDK 1.1 compatible driver to exactly support the JDBC 2.0 object data features. The JDBC 2.0 method ResultSet.getBlob returns a java.sql.Blob. Since that type does not exist in JDK 1.1 it is not possible to implement that method.

Oracle addressed this problem by defining a package, oracle.jdbc2, that contains the JDBC 2.0 types that are not defined in JDK 1.1. So, in Oracle's JDK 1.1 compatible drivers, the method OracleResultSet.getBlob returns an oracle.jdbc2.Blob. Other than the name of the package containing these types, the JDK 1.1 drivers exactly matched the JDBC 2.0 specification with respect to advanced data types.

Oracle 8i Release 2 introduced JDK 1.2 compatible drivers. In JDK 1.2 the type java.sql.Blob is defined as are the other advanced data types. So, in Oracle's JDK 1.2 compatible drivers, the method OracleResultSet.getBlob returns a java.sql.Blob. Because the two getBlob methods differ only in their result type, it is not possible to implement both. Thus the JDK 1.2 drivers do not contain getBlob methods that return oracle.jdbc2.Blob, in fact, the package oracle.jdbc2 does not exist at all.

The impact of this is that porting code from the JDK 1.1 drivers to the JDK 1.2 drivers can require you to edit the source and recompile. If your program mentions an oracle.jdcb2 type such as by using getArray, getBlob, getClob, getObject, or getRef, you will have to edit the source to replace oracle.jdbc2 with java.sql. If you do not mention oracle.jdbc2 then you do not have to do anything.

ROWSETS

Oracle does not currently provide any implementations of the RowSet interface.

JNDI for Naming Databases

JNDI is an API that allows an application to find network resources (among other things) in a relatively vendor independent fashion. Prior to JDBC 2.0 an application got a database connection from a java.sql.Driver. The application had to create an instance of the Driver object in a vendor specific way, register the Driver, and then ask the DriverManager to return an appropriate driver given a vendor specific URL. There is a lot of vendor specific stuff in the application. JNDI attempts to separate the vendor specific bits from the vendor neutral bits and enable them to cooperate via a naming and directory service.

JDBC 2.0 standard extensions specifies an interface, javax.sql.DataSource, for a Connection factory. An application would use the JNDI API to locate an appropriate DataSource and then get connections from that DataSource. Ideally, a separate application would register the DataSource with the naming service. That clearly separates the vendor specific bit, the registering application, from the vendor neutral bit, the business application.

All Oracle DataSources are JNDI referencable

- oracle.jdbc.pool.OracleConnectionPoolDataSource
- oracle.jdbc.pool.OracleDataSource
- oracle.jdbc.pool.OracleConnectionCacheImpl

All DataSource properties except roleName and timeout are provided. A few additional ones like driverType, url, and tnsEntry are also provided.

All Oracle DataSources are supported in both the JDK 1.1 and JDK 1.2 drivers.

Connection Pooling

The JDBC 2.0 standard extensions define an infrastructure for supporting connection pooling. Connection pooling is a scheme where multiple consumers share a limited set of connections rather than having each consumer create new connections. The connection pooling specification does not actually provide any additional runtime functionality. What it does provide is a set of tools that can be used to implement a connection cache. The Oracle 8i Release 2 JDBC drivers also include support for connection caching (see below) implemented using the connection pooling infrastructure.

Connection pooling is supported in both the JDK 1.1 and JDK 1.2 OCI and Thin drivers. Connection pooling is not supported for the Sever driver since the Server driver can have only one connection which is to the logged-in session in which it is running.

Distributed Transactions

The JDBC 2.0 standard extensions define an API to support distributed transactions. This API builds upon the connection pooling infrastructure (see above). The Oracle 8i Release 2 Oracle JDBC drivers fully support this API, with one exception. The method javax.sql.XAResource.recover(int) is not supported. It will be supported in a future release, probably 8.1.7.

Support for distributed transactions is provided by two packages, oracle.jdbc.xa.client and oracle.jdbc.xa.server. Each package contains three classes, OracleXAConnection, OracleXADataSource, and OracleXAResource. It is important that your application use the proper package. Applications using the OCI or Thin drivers, including the Thin driver in the server, should use oracle.jdbc.xa.client. Applications using the Server side driver inside the JServer should use oracle.jdbc.xa.server.

If your application is running in the JServer and is using both the Server side driver and the Thin driver, then you must take care to use the right class in the right place. Generally, you will want to use fully qualified class names, oracle.jdbc.xa.client.OracleDataSource and oracle.jdbc.xa.server.OracleDataSource, rather than using import statements. If you import both packages into the same class, you will have problems since the class names are the same. Similarly you will have problems if you import one package but intend to use classes from both. In this case an unqualified name would refer to the imported package and a fully qualified name would be required to refer to the other. Mistakenly omitting the qualification would lead to runtime errors. If you do not import either and use only fully qualified names, omission of the qualification will cause a compile error rather than a runtime error.

Distributed transactions are supported in both the JDK 1.1 and JDK 1.2 drivers. Distributed transaction support is NOT backward compatible with older versions of the Oracle database server. Distributed transactions are supported only on Oracle 8i Release 2 and later Oracle database servers with JServer, the Oracle Java VM, installed on the server.

Other Features

To minimize the database round trips to fetch the query results, JDBC 2.0 API provides the methods to define the number of rows retrieved for each database fetch, Statement.setFetchSize. The fetch size also determines the size of the window in a scroll sensitive result set. The number of rows specified affects only result sets created using this statement object. You can change the fetch size value in an existing result set by calling ResultSet.setFetchSize.

Previous releases of the Oracle JDBC drivers provided a similar mechanism, OracleStatment.setRowPrefetch. setRowPrefetch changes the fetch size of the result set object that is currently activated with the statement object. Since a statement object can only have one activated result set.setRowPrefetch() has the same effect as ResultSet.setFetchSize() even though one call is against OracleStatement object and the other is against the result set object. You can use the Oracle specific setRowPrefetch, or you can the JDBC 2.0 standard setFetchSize. The result of either is the same. However, you can not mix the two against one Statement, PreparedStatement, CallableStatement or ResultSet object. If you do so, the result will be unpredictable.

Note that the fetch size will be set to 1 automatically if any of the select-column types is streaming (long data or long raw data). This overrides any fetch size value the user might set. Also, this will be done regardless of whether the streaming columns are read or not.

Support for character streams has been added. This means that character data can be retrieved and sent to the database as a stream of internationalized Unicode characters.

We have also added methods to allow `java.math.BigDecimal` values to be returned with full precision.

The APIs which take Calendar arguments are supported, but they perform the same operations those without Calendar parameters. The reason this is so is that the DATE SQL type in an Oracle database does not have locale or timezone information. The database just accepts whatever value inserted. If we try to enforce the Calendar conversion in the client side, there will be no way to keep track of the timezone and locale in the Calendar. The Calendar parameter will be used whenever the database provides support for timezone and locale, certainly no sooner that 8.2. This support will possibly require a new SQL type.

Connection Cache

Connection pooling, a JDBC 2.0 feature described above, is an infrastructure for supporting connection caching. It provides no real additional functionality alone, without a connection caching mechanism. The Oracle connection cache provides three commonly used caching schemes as well as making it easy for middle-tier developers to implement their own scheme.

- Dynamic Scheme: This is the default scheme. New connections can be created beyond the maximum limit upon request but are closed and freed when the logical connections are closed. When all the connections are active and busy, requests for new connections will create new physical connections, but these physical connections are closed when the corresponding logical connections are closed. A typical grow and shrink scheme.

- Fixed with NoWait: At no time will there be more active connections than the maximum limit. Requests for new connections beyond the maximum limit will return null.

- Fixed with Wait: Same as the above except that requests for new connections beyond the maximum will wait until a connection becomes available.

Connection caching is supported in both the JDK 1.1 and JDK 1.2 JDBC drivers.

JNI

Up through Oracle 8i Release 1, the OCI driver has used NMI to call the OCI C code libraries. NMI was the first mechanism specified by Sun for calling C code from Java and the only mechanism supported by JDK 1.0.2. In Oracle 8i Release 2 we dropped support for JDK 1.0.2 (see below) so we are free to transition to JNI, the current mechanism for calling C code. This means that the Oracle 8i Release 2 OCI driver can be used with VMs other than Sun's, such as IBM and Microsoft.

Sun's VMs supports both external call mechanisms, the current JNI and the legacy NMI. Other VMs only supports JNI. Sun's earliest VMs, JDK 1.0.2 and earlier, only support NMI. Up through Oracle 8i Release 1, we supported JDK 1.0.2. This required that we support NMI. Due to limited developer resources we were unable to support both NMI and JNI. That meant that the OCI driver, which was implemented using NMI, would work under JDK 1.0.2, but it also meant that it would not work under VMs that only supported JNI.

Both the JDK 1.1 and the JDK 1.2 versions of the Oracle 8i Release 2 Oracle OCI driver uses JNI and so both are compatible with the JDK 1.1, JDK 1.2, IBM, and Microsoft VMs.

JDK Support

The big news for the Oracle 8i Release 2 JDBC drivers is the introduction of full support for JDK 1.2. Of course we are continuing support for JDK 1.1, but we dropped support for JDK 1.0.2.

Support for JDK 1.2

In response to overwhelming popular demand, we support JDK 1.2 and JDBC 2.0. The JDK 1.2 versions of the Oracle 8i Release 2 JDBC drivers are fully compliant with JDBC 2.0.

The Oracle 8i Release 2 release of the JDBC drivers contains a file classes12.zip You should use this file when developing with JDK 1.2. You should continue to use classes11.zip when developing with JDK 1.1. The file classes10.zip is gone (see below).

Support for JDK 1.1

There are no changes in the support for JDK 1.1. The package oracle.jdbc2 is still included in classes11.zip and the appropriate OracleResultSet methods still use values of those types. Some JDBC 2.0 features are supported in JDK 1.1: DataSources, ConnectionPooling, ConnectionCaches, Distributed Transactions, scrollable ResultSets, and the JDBC 2.0 batching interface.

Discontinued Support for JDK 1.0.2

Beginning with Oracle 8i Release 2 we no longer provide drivers that support JDK 1.0.2. This means that the Oracle 8i Release 2 release of the Oracle JDBC drivers does not contain classes10.zip. We will continue maintenance of the JDK 1.0.2 drivers provided as part of prior releases, Oracle 8i Release 1 and earlier, for so long as those releases are supported. If you still have need for JDK 1.0.2 compatible drivers, you can continue to use the classes10.zip from those releases.

If you use the Thin driver then you can use classes10.zip to access any supported Oracle database including Oracle 8i Release 2 and beyond. If you use the OCI driver, you can use classes10.zip to access any supported Oracle database including Oracle 8i Release 2 and beyond, but you cannot use an Oracle 8i Release 2 or later Oracle client. The JDBC OCI drivers are specific to the particular version of the Oracle client and there will not be an Oracle 8i Release 2 or later Oracle client version of the JDK 1.0.2 JDBC OCI driver.

Performance

A broad set of improvements have been made to the Oracle drivers, resulting in enhanced performance and efficient memory utilization. Data structures, binds, defines, and data type conversions have all been streamlined and optimized. LOB and BFILE performance has been extensively improved and fine-tuned. Context switches and data transfer between Java and C have been minimized (for the OCI driver). Memory resource usage has also become substantially more efficient through effective use of caching, scratch and call memory, recyclable resources, and lazy memory allocation.

Coming Attractions

We are planning some new features for the near future, ideally in Oracle 8i Release 3. Three of these new features are:

- Statement Cache
- New Public Interfaces

- New Driver

Statement Cache

There is substantial cost in creating a new Statement object. In many cases applications use the same SQL statement repeatedly. In such cases either the application pays the performance cost of recreating the statement or the programmer develops code to keep track of the various statements. We are well aware that many of our customers have implemented statement caches and we have done so ourselves for our own internal use. Multiple times. In Oracle 8i Release 3 the JDBC drivers will have two kinds of built in statement caching. One is totally transparent, to use it all you have to do is turn it on. The other is more explicit, your program has to explicitly add thing to and get things from the cache.

The statement cache is a fixed size, least recently used cache. By default the cache size is zero, or no caching. This provides exactly the same behavior as the current drivers. To turn on statement caching your program calls Connection.setStatementCacheSize with a positive value. No other code changes are required to use transparent caching.

Statements are cached on a connection basis, that is, each connection has its own cache. The size of the cache is fixed, unless the program explicitly changes it. Different connections may have different size caches. The cache uses a least recently used (LRU) algorithm to determine which statement to delete when adding a new statement to the cache. Statements are added to the cache when they are closed. A statement is reused when the program attempts to create a new statement of the same type with the same SQL string. Note: only closed statements are eligible for reuse.

For example, if you are developing an ecommerce server, you might have a bit of code like this:

```
PreparedStatement login = connection.prepareStatement(
             “SELECT ID FROM USER WHERE NAME = ? AND PASSWORD_HASH = ?”);
login.setString(1, userName);
login.setString(2, hashedPassword);
ResultSet r = login.executeQuery();
. . .
login.close();
```

Each time your application executed prepareStatement a new PreparedStatement would be created. In an ecommerce server handling tens of thousands of customers, this can be a substantial performance hit.

Let us suppose that your application uses a total of 14 distinct SQL statements. If you set the statement cache size to 14, then the second time your application attempts to create the login PreparedStatement, the previously used login PreparedStatement will be found in the statement cache and returned instead of a new one. The statement will have been cleaned of any state from its previous use, yet it will still contain information that the JDBC driver needs to execute the statement. This plus the savings of not having to allocate the object in memory can result in substantial performance improvement.

There are cases where the statement cache can remember too much. If one of your statements changed the type of the ID column in the USER table, then a later use of the login statement would fail. In the event that statement caching assumes too much about your application and causes problems, you turn it off and you are no worse off than before. For a substantial number of applications, an appropriately sized statement cache can result in appreciable performance increase.

Explicit statement caching uses the same cache as transparent caching and the two can be arbitrarily intermingled. To explicitly get a statement from the cache your program would call OracleConnection.getStatementWithKey. If a statement has been cached with the same key value as the argument and is still in the cache, that statement is returned. Otherwise, null is returned. You add a statement to the cache with a key by calling OracleStatement.closeWithKey. Multiple statements can be cached with the same key. It is probably a good idea to cache only identical statements with a given key.

Aside from the code differences the major difference between the transparent caching and explicit caching is the state of the returned statement. When you retrieve a statement using transparent caching the statement is indistinguishable from a new statement. When you retrieve a statement using explicit caching the statement has exactly the same state as when it was closed including defined column types and bind parameter values. The explicit statement cache is optimized for things such a middle-tier applications that use the same few statements repeatedly in exactly the same way. Transparent statement caching is useful for existing applications with minimal code changes and in developing less structured applications.

NEW INTERFACES

The Oracle JDBC drivers provide a number of extensions to the JDBC standard. You access these extensions by casting the result of a standard method to an Oracle specific type, which makes the extension methods visible. The names of these types are documented as part of the public interface for the Oracle JDBC drivers. Examples are oracle.jdbc.driver.OracleDriver and oracle.jdbc.driver.OracleStatement. In all versions of the Oracle JDBC drivers up to and including Oracle 8i Release 2 these are the names of the concrete classes that implement this functionality. In Oracle 8i Release 3 this will change, although in a 100% backward compatible way.

For a number of reasons, including support for the new driver discussed below, we can no longer make visible the concrete classes that implement the Oracle JDBC drivers. For example, the new driver does not return an oracle.jdbc.driver.OracleConnection when you call getConnection. It returns an instance of a different class which I will call JConnection. JConnection provides all the same extensions as oracle.jdbc.driver.OracleConnection and should be type compatible with it, but it is not. For technical reasons which you can probably guess, JConnection is not a subclass of oracle.jdbc.driver.OracleConnection. This means that if you are using the new driver and do

```
OracleConnection conn = (OracleConnection) driver.getConnection(newDriverURL);
```

you will get a IllegalCastException at runtime. Unfortunately this is exactly what we have told you to do.

There were at least four ways out of this dilemma:

- Make JConnection a subclass of oracle.jdbc.driver.OracleConnection. This is not possible as JConnection does not share any code with oracle.jdbc.driver.Oracle Connection.
- Make oracle.jdbc.driver.OracleConnection an abstract class and then make JConnection a subclass of oracle.jdbc.driver.OracleConnection. This works for oracle.jdbc.driver.OracleConnection but does not work for oracle.jdbc.driver.OracleStatement, oracle.jdbc.driver.OraclePreparedStatement, and oracle.jdbc.driver.OracleCallableStatement. oracle.jdbc.driver.OraclePreparedStatement is a subclass of oracle.jdbc.driver.OracleStatement. JPreparedStatement would have to be a subclass of both oracle.jdbc.driver.OraclePreparedStatement and JStatement. This is not possible.
- Make oracle.jdbc.driver.OracleConnection an interface and create a new class, OraConnection, which implements the oracle.jdbc.driver.OracleConnection interface. Then JConnection can also implement oracle.jdbc.driver.OracleConnection. This works for oracle.jdbc.driver.OracleStatement, oracle.jdbc.driver.OraclePreparedStatement, and oracle.jdbc.driver.OracleCallableStatement as well. In fact it solves all of the Oracle developers' technical problems. But it would require you to recompile your code since changing a Java class to an interface is not a binary compatible change. This is not acceptable.
- Create a new interface, which I will call oracle.sql.OracleConnection although the name is not yet finalized. Modify oracle.jdbc.driver.OracleConnection to implement oracle.sql.OracleConnection and make JConnection also implement oracle.sql.OracleConnection. Finally, deprecate direct use of oracle.jdbc.driver.OracleConnection and change all of the documentation to refer to oracle.sql.OracleConnection (and oracle.sql.OracleStatement, oracle.sql.OraclePreparedStatement, etc. Again, the names are not yet finalized.)

We chose this last option. It is totally backward compatible. All existing code will run without recompilation and can be recompiled without change.

All new code should be written to use the new interfaces. All this really entails is replacing the import statement. For example replace

```
import oracle.jdbc.driver.OracleConnection;
```

with

```
import oracle.sql.OracleConnection;
```

The body of the code would be unchanged since the interfaces have the same name as the classes. Only the package name is different.

```
OracleConnection conn = (OracleConnection) drvr.getConnection(newDriverURL);
```

This will work with all Oracle extensions and all Oracle JDBC drivers, including the new driver. Casting to oracle.jdbc.driver.OracleConnection (or oracle.jdbc.driver.OracleStatement, or whatever) may not work with some new features provided in Oracle 8i Release 3 and beyond and definitely will not work with the new driver.

To summarize, beginning in Oracle 8i Release 3 we will document a new public API. The old API will continue to be supported and all existing code will continue to work without change or the necessity for recompilation. Beginning in Oracle 8i Release 3 all new code should use the new API as there may be some new features that are not available through the old API. In particular, all code that uses Oracle extensions and the new driver must use the new API. Although the details, including the names, are not yet finalized, the new interfaces will be something like this:

- package oracle.sql
    - class OracleDriver
    - class OracleSqlTypes
    - class OracleLog
    - interface OracleConnection
    - interface OracleStatement
    - interface OraclePreparedStatement
    - interface OracleCallableStatement
    - interface OracleResultSet

If you do not use the Oracle extensions and use only java.sql.* then none of this matters to you at all. Everything still works the same.

NEW DRIVER

The Oracle JDBC Thin driver is not so thin, approaching a megabyte or so. This is the problem.

Since Oracle 8i Release 1 the Oracle database engine has contained a Java VM and a complete implementation of JDBC, the Oracle JDBC Server driver. This is the opportunity.

The *HyperDriver* seizes the opportunity to solve the problem. And provides a powerful new feature in the bargain.

WHAT IS THE HYPERDRIVER?

The HyperDriver is a 100% Java driver, just like the Thin driver. But the HyperDriver is implemented in a totally different way from the Thin driver and will be much thinner.

The OCI driver uses JNI to call the Oracle OCI C libraries. This C code communicates with the database using TTC over SQL*Net. The Thin driver contains Java code that duplicates the OCI C code and also communicates with the database using TTC over SQL*Net. The HyperDriver contains Java code that implements a new protocol, *Jolt*, and communicates with the Oracle JDBC Server driver running in the JServer Java VM inside the database. The Server driver talks to the database.

The HyperDriver is optimized for Java-centric applications. The Jolt protocol is based on Java data types and was designed for fast and compact implementation in Java. Anything that is hard to do or takes a lot of code, like converting from Oracle NLS character sets to the Java character set, is done by the server. Thus the HyperDriver does not have a lot of code to support seldom used or highly database-centric operations. The Thin driver does have code to do all of these things, greatly adding to its size. The HyperDriver relies on the server for these operations and so is much smaller.

HOW DOES THE HYPERDRIVER WORK?

The JServer, the Java VM inside the Oracle database engine, supports a feature that we call a *Presentation*. A Presentation is an association between a network port on the Listener, a Java Socket inside the VM, and a Java class. Whenever a client tries to connect to that network port, the Listener tells the VM to create an instance of the class and pass it a socket. That socket is the network connection to the client. At this point the Java class can do whatever it wants by reading from/writing to the socket:

implement a Web server, implement a CORBA ORB, or implement a JDBC server. The HyperDriver is an Oracle JDBC driver that connects to a JDBC server Presentation running in the database. Writing a JDBC server inside the JServer is fairly simple since the JServer already contains a JDBC driver, the Oracle Server driver. It is simply a matter of implementing a protocol to connect the JDBC calls in the client to the equivalent calls in the server. Well, it is not exactly that simple. There are some performance issues to consider, but it is not nearly as complex as implementing a JDBC driver using TTC over SQL*Net.

The HyperDriver implements the Jolt protocol and uses this protocol to communicate with the Server driver running in the JServer. Many of the JDBC methods in the HyperDriver do nothing more than call the equivalent method in the Server driver. Many of the objects that implement the HyperDriver are nothing more than proxies for the equivalent objects in the JServer. For example a HyperDriver JConnection is just a proxy for a Server driver OracleConnection. A call to Connection.getStatement calls getStatement in the server and creates a wrapper for the result. As you can see, the HyperDriver does not work very hard.

For some methods though, the HyperDriver has to be a bit smarter. For example if every getXXX method was implemented as a call to the server, the performance would be unacceptable. (We proved this.) So, the HyperDriver uses the same prefetch model as the other Oracle drivers to fetch groups of rows. And so a call to getXXX is executed entirely on the client and does not require communication with the server.

WHAT IS THE POWERFUL NEW FEATURE?

The Jolt protocol is a simple RPC mechanism that has been highly optimized for Java. In some ways it is similar to RMI and CORBA, but much simpler. Jolt was designed to support a long lasting connection between two Java programs. RMI and CORBA are mostly stateless protocols that support communication between a large number of Java (RMI) or heterogeneous (CORBA) programs. The HyperDriver uses Jolt to call JDBC (and other) methods in the JServer. Customer applications can use this facility as well.

The Jolt driver depends upon instances of a subclass of a Java class ApiDescription. An ApiDescription subclass defines exactly which objects and which methods are accessible via a Jolt connection. There is a JDBC server ApiDescription subclass which defines the objects and methods the JDBC server makes available to the HyperDriver client. This is how the HyperDriver client is implemented. Customers will be able to define their own ApiDescription subclasses which will define the sets of objects and methods that the JServer should make available to their clients. Having defined those objects and methods, programmers can write client applications that call those server side Java methods directly rather than through the current round about path through PL/SQL. You can mix JDBC and direct Java method calls, and we believe that there will be some client applications that make no direct calls to JDBC at all, but instead consist entirely of calls to customer defined Java methods in the JServer.

WHEN WILL THE HYPERDRIVER BE AVAILABLE?

We intend to deliver a beta version of the HyperDriver in Oracle 8i Release 3. It is not clear how much support will be available for customer defined APIs in the initial release. While coding an ApiDescription subclass by hand is not difficult, it is tedious and time consuming. We are developing tools to assist in coding ApiDescription subclasses, but it is not clear when they will be suitable for public consumption. We will also make available an ApiDescription subclass that allows the client to specify at runtime via the Jolt protocol which objects and methods it will use. While not as fast as a purpose built ApiDescription subclass, it is great for prototyping and for applications which are not performance critical.

CONCLUSION

Lots of cool stuff in Oracle 8i Release 2 and some more cool stuff coming in Oracle 8i Release 3. Almost all of the new features are supported in both the JDK 1.1 and JDK 1.2 versions of all three Oracle drivers.

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
+1.650.506.7000
Fax +1.650.506.7200
http://www.oracle.com/