37
Distributed Transactions

This chapter discusses the Oracle Java Database Connectivity (JDBC) implementation of distributed transactions. These are multiphased transactions, often using multiple databases, which must be committed in a coordinated way. There is also related discussion of XA, which is a general standard, and not specific to Java, for distributed transactions.

The following topics are discussed:

- AboutDistributed Transactions
- XA Components
- Error Handling and Optimizations
- About Implementing a Distributed Transaction
- Native-XA in Oracle JDBC Drivers

> **Note:**
>
> This chapter discusses features of the JDBC 2.0 Optional Package, formerly known as the JDBC 2.0 Standard Extension application programming interface (API) that is available through the `javax` packages.

For further introductory and general information about distributed transactions, refer to the specifications for the JDBC 2.0 Optional Package and the Java Transaction API (JTA).

37.1 About Distributed Transactions

The section covers the following topics:

- Overview of Distributed Transaction
- Distributed Transaction Components and Scenarios
- Distributed Transaction Concepts
- About Switching Between Global and Local Transactions
- Oracle XA Packages

37.1.1 Overview of Distributed Transaction

A **distributed transaction**, sometimes referred to as a **global transaction**, is a set of two or more related transactions that must be managed in a coordinated way. The transactions that constitute a distributed transaction might be in the same database, but more typically are in different databases and often in different locations. Each individual transaction of a distributed transaction is referred to as a **transaction branch**.

For example, a distributed transaction might consist of money being transferred from an account in one bank to an account in another bank. You would not want either transaction committed without assurance that both will complete successfully.

In JDBC, distributed transaction functionality is built on top of connection pooling functionality. This distributed transaction functionality is also built upon the open XA standard for distributed transactions. XA is part of the X/Open standard and is not specific to Java.

JDBC is used to connect to database resources. However, to include all changes to multiple databases within a transaction, you must use the JDBC connections within a JTA global transaction. The process of including database SQL updates within a transaction is referred to as enlisting a database resource.

37.1.2 Distributed Transaction Components and Scenarios

In reading the remainder of the distributed transactions section, it will be helpful to keep the following points in mind:

- A distributed transaction system typically relies on an external transaction manager, such as a software component that implements standard JTA functionality, to coordinate the individual transactions.

  Many vendors offer XA-compliant JTA modules, including Oracle, which includes JTA in Oracle9*i* Application Server and Oracle Application Server 10*g*.

- XA functionality is usually isolated from a client application, being implemented instead in a middle-tier environment, such as an application server.

  In many scenarios, the application server and transaction manager will be together on the middle tier, possibly together with some of the application code as well.

- Discussion throughout this section is intended mostly for middle-tier developers.

- The term resource manager is often used in discussing distributed transactions. A resource manager is simply an entity that manages data or some other kind of resource. Wherever the term is used in this chapter, it refers to a database.

> **Note:**
>
> Using JTA functionality requires `jta.jar` to be in the `CLASSPATH` environment variable. This file is located at *`ORACLE_HOME`*`/jlib`. Oracle includes this file with the JDBC product.

37.1.3 Distributed Transaction Concepts

When you use XA functionality, the transaction manager uses XA resource instances to prepare and coordinate each transaction branch and then to commit or roll back all transaction branches appropriately.

XA functionality includes the following key components:

- XA data sources

  These are extensions of connection pool data sources and other data sources, and similar in concept and functionality.

There will be one XA data source instance for each resource manager that will be used in the distributed transaction. You will typically create XA data source instances in your middle-tier software.

XA data sources produce XA connections.

- XA connections

  These are extensions of pooled connections and similar in concept and functionality. An XA connection encapsulates a physical database connection. Individual connection instances are temporary handles to these physical connections.

  An XA connection instance corresponds to a single Oracle session, although the session can be used in sequence by multiple logical connection instances, as with pooled connection instances.

  You will typically get an XA connection instance from an XA data source instance in your middle-tier software. You can get multiple XA connection instances from a single XA data source instance if the distributed transaction will involve multiple sessions in the same database.

  XA connections produce `OracleXAResource` instances and JDBC connection instances.

- XA resources

  These are used by a transaction manager in coordinating the transaction branches of a distributed transaction.

  You will get one `OracleXAResource` instance from each XA connection instance, typically in your middle-tier software. There is a one-to-one correlation between `OracleXAResource` instances and XA connection instances. Equivalently, there is a one-to-one correlation between `OracleXAResource` instances and Oracle sessions.

  In a typical scenario, the middle-tier component will hand off `OracleXAResource` instances to the transaction manager, for use in coordinating distributed transactions.

  Each `OracleXAResource` instance corresponds to a single Oracle session. So, there can be only a single active transaction branch associated with an `OracleXAResource` instance at any given time. However, there can be additional suspended transaction branches.

  Each `OracleXAResource` instance has the functionality to start, end, prepare, commit, or roll back the operations of the transaction branch running in the session with which the `OracleXAResource` instance is associated.

  The prepare step is the first step of a two-phase commit operation. The transaction manager will issue a `PREPARE` to each `OracleXAResource` instance. Once the transaction manager sees that the operations of each transaction branch have prepared successfully, it will issue a `COMMIT` to each `OracleXAResource` instance to commit all the changes.

- Transaction IDs

  These are used to identify transaction branches. Each ID includes a transaction branch ID component and a distributed transaction ID component. This is how a branch is associated with a distributed transaction. All `OracleXAResource` instances associated with a given distributed transaction would have a transaction ID that includes the same distributed transaction ID component.

- `OracleXAResource.ORATRANSLOOSE`

  Start a loosely coupled transaction with transaction ID `xid`.

37.1.4 About Switching Between Global and Local Transactions

Applications can share connections between local and global transactions. Applications can also switch connections between local transactions and global transactions.

A connection is always in one of the following modes:

- `NO_TXN`

  No transaction is actively using this connection.

- `LOCAL_TXN`

  A local transaction with auto-commit turned off or disabled is actively using this connection.

- `GLOBAL_TXN`

  A global transaction is actively using this connection.

Each connection switches automatically between these modes depending on the operations carried out on the connection. A connection is always in `NO_TXN` mode when it is instantiated.

> **Note:**
>
> The modes are maintained internally by the JDBC drivers in association with Oracle Database.

Table 37-1 describes the connection mode transition rules.

Table 37-1 Connection Mode Transitions

Current Mode	Switches to NO_TXN When	Switches to LOCAL_TXN When	Switches to GLOBAL_TXN When
`NO_TXN`	NA	Auto-commit mode is false and an Oracle data manipulation language (DML) statement is run.	The `start` method is called on an `XAResource` obtained from the `XAconnection` that provided the current connection.
`LOCAL_TXN`	Any of the following happens: • An Oracle data definition language (DDL) statement is run. • `commit` is called. • `rollback` is called, but without parameters.	NA	The `start` method is called on an `XAResource` obtained from the `XAconnection` that provided the current connection. This feature is available starting from Oracle Database 12*c* Release 1 (12.1.0.2).
`GLOBAL_TXN`	Within a global transaction open on this connection, `end` is called on an `XAResource` obtained from the `XAconnection` that provided this connection.	NEVER	NA

If none of these rules is applicable, then the mode does not change.

Mode Restrictions on Operations

The current connection mode restricts which operations are valid within a transaction.

- In the `LOCAL_TXN` mode, applications must not call `prepare`, `commit`, `rollback`, `forget`, or `end` on an `XAResource`. Doing so causes an `XAException` to be thrown.
- In the `GLOBAL_TXN` mode, applications must not call `commit`, `rollback`, `rollback(Savepoint)`, `setAutoCommit(true)`, or `setSavepoint` on a `java.sql.Connection`, and must not call `OracleSetSavepoint` or `oracleRollback` on an `oracle.jdbc.OracleConnection`. Doing so causes a `SQLException` to be thrown.

> **Note:**
>
> This mode-restriction error checking is in addition to the standard error checking on the transaction and savepoint APIs.

37.1.5 Oracle XA Packages

Oracle supplies the following three packages that have classes to implement distributed transaction functionality according to the XA standard:

- `oracle.jdbc.xa`
- `oracle.jdbc.xa.client`
- `oracle.jdbc.xa.server`

Classes for XA data sources, XA connections, and XA resources are in both the `client` package and the `server` package. An abstract class for each is in the top-level package. The `OracleXid` and `OracleXAException` classes are in the top-level `oracle.jdbc.xa` package, because their functionality does not depend on where the code is running.

In middle-tier scenarios, you will import `OracleXid`, `OracleXAException`, and the `oracle.jdbc.xa.client` package.

If you intend your XA code to run in the target Oracle Database, however, you will import the `oracle.jdbc.xa.server` package instead of the `client` package.

If code that will run inside a target database must also access remote databases, then do not import either package. Instead, you must fully qualify the names of any classes that you use from the `client` package to access a remote database or from the `server` package to access the local database. Class names are duplicated between these packages.

37.2 XA Components

This section discusses the XA components, that is, the standard XA interfaces specified in the JDBC standard, and the Oracle classes that implement them. The following topics are covered:

- XADatasource Interface and Oracle Implementation
- XAConnection Interface and Oracle Implementation
- XAResource Interface and Oracle Implementation

- OracleXAResource Method Functionality and Input Parameters
- Xid Interface and Oracle Implementation

37.2.1 XADatasource Interface and Oracle Implementation

The `javax.sql.XADataSource` interface outlines standard functionality of XA data sources, which are factories for XA connections. The overloaded `getXAConnection` method returns an XA connection instance and optionally takes a user name and password as input:

```
public interface XADataSource
{
   XAConnection getXAConnection() throws SQLException;
   XAConnection getXAConnection(String user, String password)
      throws SQLException;
   ...
}
```

Oracle JDBC implements the `XADataSource` interface with the `OracleXADataSource` class, located both in the `oracle.jdbc.xa.client package` and the `oracle.jdbc.xa.server` package.

The `OracleXADataSource` classes also extend the `OracleConnectionPoolDataSource` class, which extends the `OracleDataSource` class, and therefore, include all the connection properties.

The `getXAConnection` methods of the `OracleXADataSource` class returns the Oracle implementation of XA connection instances, which are `OracleXAConnection` instances.

> **Note:**
>
> You can register XA data sources in Java Naming Directory and Interface (JNDI) using the same naming conventions as discussed previously for nonpooling data sources.

> **See Also:**
>
> For information about Fast Connection Failover, refer to *Oracle Universal Connection Pool for JDBC Developer's Guide*.

37.2.2 XAConnection Interface and Oracle Implementation

An XA connection instance, as with a pooled connection instance, encapsulates a physical connection to a database. This would be the database specified in the connection properties of the XA data source instance that produced the XA connection instance.

Each XA connection instance also has the facility to produce the `OracleXAResource` instance that will correspond to it for use in coordinating the distributed transaction.

An XA connection instance is an instance of a class that implements the standard `javax.sql.XAConnection` interface:

```
public interface XAConnection extends PooledConnection
{
   javax.jta.xa.XAResource getXAResource() throws SQLException;
}
```

As you see, the `XAConnection` interface extends the `javax.sql.PooledConnection` interface, so it also includes the `getConnection`, `close`, `addConnectionEventListener`, and `removeConnectionEventListener` methods.

Oracle JDBC implements the `XAConnection` interface with the `OracleXAConnection` class, located both in the `oracle.jdbc.xa.client package` and the `oracle.jdbc.xa.server` package.

The `OracleXAConnection` classes also extend the `OraclePooledConnection` class.

The `OracleXAConnection` class `getXAResource` method returns the Oracle implementation of an `OracleXAResource` instance, which is an `OracleXAResource` instance. The `getConnection` method returns an `OracleConnection` instance.

A JDBC connection instance returned by an XA connection instance acts as a temporary handle to the physical connection, as opposed to encapsulating the physical connection. The physical connection is encapsulated by the XA connection instance. The connection obtained from an `XAConnection` object behaves exactly like a regular connection, until it participates in a global transaction. At that time, auto-commit status is set to `false`. After the global transaction ends, auto-commit status is returned to the value it had before the global transaction. The default auto-commit status on a connection obtained from `XAConnection` is `false` in all releases prior to Oracle Database 10*g*. Starting from Oracle Database 10*g*, the default status is `true`.

Each time an XA connection instance `getConnection` method is called, it returns a new connection instance that exhibits the default behavior, and closes any previous connection instance that still exists and had been returned by the same XA connection instance. However, it is advisable to explicitly close any previous connection instance before opening a new one.

Calling the `close` method of an XA connection instance closes the physical connection to the database. This is typically performed in the middle tier.

37.2.3 XAResource Interface and Oracle Implementation

The transaction manager uses `OracleXAResource` instances to coordinate all the transaction branches that constitute a distributed transaction.

Each `OracleXAResource` instance provides the following key functionality, typically invoked by the transaction manager:

- It associates and disassociates distributed transactions with the transaction branch operating in the XA connection instance that produced this `OracleXAResource` instance. Essentially, it associates distributed transactions with the physical connection or session encapsulated by the XA connection instance. This is done through use of transaction IDs.
- It performs the two-phase commit functionality of a distributed transaction to ensure that changes are not committed in one transaction branch before there is assurance that the changes will succeed in all transaction branches.

> **Note:**
>
> – Because there must always be a one-to-one correlation between XA connection instances and `OracleXAResource` instances, an `OracleXAResource` instance is implicitly closed when the associated XA connection instance is closed.
>
> – If a transaction is opened by a given `OracleXAResource` instance, then it must also be closed by the same `OracleXAResource` instance.

An `OracleXAResource` instance is an instance of a class that implements the standard `javax.transaction.xa.XAResource` interface. Oracle JDBC implements the `XAResource` interface with the `OracleXAResource` class, located both in the `oracle.jdbc.xa.client` package and the `oracle.jdbc.xa.server` package.

Oracle JDBC driver creates and returns an `OracleXAResource` instance whenever the `getXAResource` method of the `OracleXAConnection` class is called, and it is Oracle JDBC driver that associates an `OracleXAResource` instance with a connection instance and the transaction branch being run through that connection.

This method is how an `OracleXAResource` instance is associated with a particular connection and with the transaction branch being run in that connection.

37.2.4 OracleXAResource Method Functionality and Input Parameters

The `OracleXAResource` class has several methods to coordinate a transaction branch with the distributed transaction with which it is associated. This functionality usually involves two-phase commit operations.

A transaction manager, receiving `OracleXAResource` instances from a middle-tier component, such as an application server, typically invokes this functionality.

Each of these methods takes a transaction ID as input, in the form of an `Xid` instance, which includes a transaction branch ID component and a distributed transaction ID component. Every transaction branch has a unique transaction ID, but transaction branches belonging to the same global transaction have the same global transaction component as part of their transaction IDs.

start

Starts work on behalf of a transaction branch, associating the transaction branch with a distributed transaction.

```
void start(Xid xid, int flags)
```

The `flags` parameter must be one or more of the following values:

- `XAResource.TMNOFLAGS`

  Flags the start of a new transaction branch for subsequent operations in the session associated with this XA resource instance. This branch will have the transaction ID `xid`, which is an `OracleXid` instance created by the transaction manager. This will map the transaction branch to the appropriate distributed transaction.

- `XAResource.TMJOIN`

Joins subsequent operations in the session associated with this XA resource instance to the existing transaction branch specified by `xid`.

- `XAResource.TMRESUME`

  Resumes the transaction branch specified by `xid`.

  > **Note:**
  >
  > A transaction branch can be resumed only if it had been suspended earlier.

- `OracleXAResource.TMPROMOTE`

  Promotes a local transaction to a global transaction

- `OracleXAResource.ORATMSERIALIZABLE`

  Starts a serializable transaction with transaction ID `xid`.

- `OracleXAResource.ORATMREADONLY`

  Starts a read-only transaction with transaction ID `xid`.

- `OracleXAResource.ORATMREADWRITE`

  Starts a read/write transaction with transaction ID `xid`.

- `OracleXAResource.ORATRANSLOOSE`

  Starts a loosely coupled transaction with transaction ID `xid`.

`TMNOFLAGS`, `TMJOIN`, `TMRESUME`, `TMPROMOTE`, `ORATMSERIALIZABLE`, `ORATMREADONLY`, and `ORATMREADWRITE` are defined as `static` members of the `XAResource` interface and `OracleXAResource` class. `ORATMSERIALIZABLE`, `ORATMREADONLY`, and `ORATMREADWRITE` are the isolation-mode flags. The default isolation behavior is `READ COMMITTED`.

> **Note:**
>
> - Instead of using the `start` method with `TMRESUME`, the transaction manager can cast to `OracleXAResource` and use the `resume(Xid xid)` method, an Oracle extension.
> - If you use `TMRESUME`, then you must also use `TMNOMIGRATE`, as in `start(xid, XAResource.TMRESUME | OracleXAResource.TMNOMIGRATE)`. This prevents the application from receiving the error `ORA 1002: fetch out of sequence`.
> - If you use the isolation-mode flags incorrectly, then an exception with code `XAER_INVAL` is raised. Furthermore, you cannot use isolation-mode flags when resuming a global transaction, because you cannot set the isolation level of an existing transaction. If you try to use the isolation-mode flags when resuming a transaction, then an external Oracle exception with code ORA-24790 is raised.
> - In order to avoid `Error ORA 1002: fetch out of sequence`, include the `TMNOMIGRATE` flag as part of the `start` method. For example:
>
>   ```
>   start(xid, XAResource.TMSUSPEND | OracleXAResource.TMNOMIGRATE);
>   ```
>
> - All the flags defined in `OracleXAResource` are Oracle extensions. When writing a transaction manager that uses these flags, you should be mindful of this.

Note that to create an appropriate transaction ID in starting a transaction branch, the transaction manager must know to which distributed transaction the transaction branch belongs. The mechanics of this are handled between the middle tier and transaction manager.

end

Ends work on behalf of the transaction branch specified by `xid`, disassociating the transaction branch from its distributed transaction.

```
void end(Xid xid, int flags)
```

The `flags` parameter can have one of the following values:

- `XAResource.TMSUCCESS`

  This is to indicate that this transaction branch is known to have succeeded.

- `XAResource.TMFAIL`

  This is to indicate that this transaction branch is known to have failed.

- `XAResource.TMSUSPEND`

  This is to suspend the transaction branch specified by `xid`. By suspending transaction branches, you can have multiple transaction branches in a single session. Only one can be active at any given time, however. Also, this tends to be more expensive in terms of resources than having two sessions.

`TMSUCCESS`, `TMFAIL`, and `TMSUSPEND` are defined as static members of the `XAResource` interface and `OracleXAResource` class.

Note:

- Instead of using the `end` method with `TMSUSPEND`, the transaction manager can cast to `OracleXAResource` and use the `suspend(Xid xid)` method, an Oracle extension.
- This XA functionality to suspend a transaction provides a way to switch between various transactions within a single JDBC connection. You can use the XA classes to accomplish this, even if you are not in a distributed transaction environment and would otherwise have no need for the XA classes.
- If you use `TMSUSPEND`, then you must also use `TMNOMIGRATE`, as in `end(xid, XAResource.TMSUSPEND | OracleXAResource.TMNOMIGRATE)`. This prevents the application from receiving the error `ORA 1002: fetch out of sequence`.
- In order to avoid `Error ORA 1002: fetch out of sequence`, include the `TMNOMIGRATE` flag as part of the end method. For example:

  ```
  end(xid, XAResource.TMSUSPEND | OracleXAResource.TMNOMIGRATE);
  ```

- All the flags defined in `OracleXAResource` are Oracle extensions. Any transaction manager that uses these flags should take heed of this.

prepare

Prepares the changes performed in the transaction branch specified by `xid`. This is the first phase of a two-phase commit operation, to ensure that the database is accessible and that the changes can be committed successfully.

```
int prepare(Xid xid)
```

This method returns an integer value as follows:

- `XAResource.XA_RDONLY`

  This is returned if the transaction branch runs only read-only operations such as `SELECT` statements.

- `XAResource.XA_OK`

  This is returned if the transaction branch runs updates that are all prepared without error.

`XA_RDONLY` and `XA_OK` are defined as `static` members of the `XAResource` interface and `OracleXAResource` class.

Note:

- The `prepare` method sometimes does not return any value if the transaction branch runs updates and any of them encounters errors during preparation. In this case, an XA exception is thrown.
- Always call the `end` method on a branch before calling the `prepare` method.
- If there is only one transaction branch in a distributed transaction, then there is no need to call the `prepare` method. You can call the `OracleXAResource commit` method without preparing first.

commit

Commits prepared changes in the transaction branch specified by `xid`. This is the second phase of a two-phase commit and is performed only after all transaction branches have been successfully prepared.

```
void commit(Xid xid, boolean onePhase)
```

Set the `onePhase` parameter as follows:

- `true`

  This is to use one-phase instead of two-phase protocol in committing the transaction branch. This is appropriate if there is only one transaction branch in the distributed transaction; the `prepare` step would be skipped.

- `false`

  This is to use two-phase protocol in committing the transaction branch.

rollback

Rolls back prepared changes in the transaction branch specified by `xid`.

```
void rollback(Xid xid)
```

forget

Tells the resource manager to forget about a heuristically completed transaction branch.

```
public void forget(Xid xid)
```

recover

The transaction manager calls this method during recovery to obtain the list of transaction branches that are currently in prepared or heuristically completed states.

```
public Xid[] recover(int flag)
```

> **Note:**
>
> Values for `flag` other than `TMSTARTRSCAN`, `TMENDRSCAN`, or `TMNOFLAGS`, cause an exception to be thrown, otherwise `flag` is ignored.

The resource manager returns zero or more `Xid`s for the transaction branches that are currently in a prepared or heuristically completed state. If an error occurs during the operation, then the resource manager throws the appropriate `XAException`.

> **Note:**
>
> The `recover` method requires SELECT privilege on `DBA_PENDING_TRANSACTIONS` and EXECUTE privilege on `SYS.DBMS_XA` in Oracle database server. For database versions prior to Oracle Database 11*g* Release 1, where an Oracle patch including a fix for bug 5945463 is not available, or it is infeasible to apply the patch for the particular application scenario, the `recover` method requires `SYSBDBA` privilege. Regular use of `SYSDBA` privilege is a security risk. So, Oracle strongly recommends that you upgrade your Database or apply the fix for bug 5945463, if you need to use the `recover` method.

isSameRM

To determine if two `OracleXAResource` instances correspond to the same resource manager, call the `isSameRM` method from one `OracleXAResource` instance, specifying the other `OracleXAResource` instance as input. In the following example, presume `xares1` and `xares2` are `OracleXAResource` instances:

```
boolean sameRM = xares1.isSameRM(xares2);
```

37.2.5 Xid Interface and Oracle Implementation

The transaction manager creates transaction ID instances and uses them in coordinating the branches of a distributed transaction. Each transaction branch is assigned a unique transaction ID, which includes the following information:

- Format identifier

  A format identifier specifies a Java transaction manager. For example, there could be a format identifier `orcl`. This field *cannot* be null. The size of a format identifier is 4 bytes.

- Global transaction identifier

  It is also known as a distributed transaction ID component. The size of a global transaction identifier is 64 bytes.

- Branch qualifier

  It is also known as transaction branch ID component. The size of a branch qualifier is 64 bytes.

The 64-byte global transaction identifier value will be identical in the transaction IDs of all transaction branches belonging to the same distributed transaction. However, the overall transaction ID is unique for every transaction branch.

An XA transaction ID instance is an instance of a class that implements the standard `javax.transaction.xa.Xid` interface, which is a Java mapping of the X/Open transaction identifier XID structure.

Oracle implements this interface with the `OracleXid` class in the `oracle.jdbc.xa` package. `OracleXid` instances are employed only in a transaction manager, transparent to application programs or an application server.

> **Note:**
>
> Oracle does not require the use of `OracleXid` for `OracleXAResource` calls. Instead, use any class that implements the `javax.transaction.xa.Xid` interface.

A transaction manager may use the following in creating an `OracleXid` instance:

```
public OracleXid(int fId, byte gId[], byte bId[]) throws XAException
```

`fId` is an integer value for the format identifier, `gId[]` is a byte array for the global transaction identifier, and `bId[]` is a byte array for the branch qualifier.

The `Xid` interface specifies the following getter methods:

- `public int getFormatId()`
- `public byte[] getGlobalTransactionId()`
- `public type[] getBranchQualifier()`

37.3 Error Handling and Optimizations

This section focuses on the functionality of XA exceptions and error handling and the Oracle optimizations in its XA implementation. It covers the following topics:

- XAException Classes and Methods
- Mapping Between Oracle Errors and XA Errors
- XA Error Handling
- Oracle XA Optimizations

The exception and error-handling discussion includes the standard XA exception class and the Oracle-specific XA exception class, as well as particular XA error codes and error-handling techniques.

37.3.1 XAException Classes and Methods

XA methods throw XA exceptions, as opposed to general exceptions or `SQLExceptions`. An XA exception is an instance of the standard class `javax.transaction.xa.XAException` or a subclass.

An Oracle XAException is an instance that consists of an Oracle error portion and an XA error portion. Oracle provides the `oracle.jdbc.xa.OracleXAException` subclasses of the standard `javax.transaction.xa.XAException` class. An `OracleXAException` instance is constructed using one of the following constructors:

```
public OracleXAException()

public OracleXAException(int error)
```

The error value is an error code that combines an Oracle SQL error value and an XA error value. The JDBC driver determines exactly how to combine the Oracle and XA error values.

The `OracleXAException` class has the following methods:

- `public int getOracleError()`

This method returns the Oracle SQL error code pertaining to the exception, a standard ORA error number or 0 if there is no Oracle SQL error.

- `public int getXAError()`

  This method returns the XA error code pertaining to the exception. XA error values are defined in the `javax.transaction.xa.XAException` class.

37.3.2 Mapping Between Oracle Errors and XA Errors

Oracle errors correspond to XA errors in `OracleXAException` instances as documented in Table 37-2.

Table 37-2 Oracle-XA Error Mapping

Oracle Error Code	XA Error Code
ORA 24756	XAException.XAER_NOTA
ORA 24764	XAException.XA_HEURCOM
ORA 24765	XAException.XA_HEURRB
ORA 24766	XAException.XA_HEURMIX
ORA 24767	XAException.XA_RDONLY
ORA 25351	XAException.XA_RETRY
ORA 30006	XAException.XA_RETRY
ORA 24763	XAException.XAER_PROTO
ORA 24769	XAException.XAER_PROTO
ORA 24770	XAException.XAER_PROTO
ORA 24776	XAException.XAER_PROTO
ORA 2056	XAException.XAER_PROTO
ORA 17448	XAException.XAER_PROTO
ORA 24768	XAException.XAER_PROTO
ORA 24775	XAException.XAER_PROTO
ORA 24761	XAException.XA_RBROLLBACK
ORA 2091	XAException.XA_RBROLLBACK
ORA 2092	XAException.XA_RBROLLBACK
ORA 24780	XAException.XAER_RMERR
All other ORA errors	XAException.XAER_RMFAIL

37.3.3 XA Error Handling

The following code snippet uses the `OracleXAException` class to process an XA exception:

```
try {
   ...
   ...Perform XA operations...
   ...
} catch(OracleXAException oxae) {
  int oraerr = oxae.getOracleError();
  System.out.println("Error " + oraerr);
```

```
}
  catch(XAException xae)
{...Process generic XA exception...}
```

In case the XA operations did not throw an Oracle-specific XA exception, the code drops through to process a generic XA exception.

37.3.4 Oracle XA Optimizations

Oracle JDBC has functionality to improve performance if two or more branches of a distributed transaction use the same database instance, meaning that the `OracleXAResource` instances associated with these branches are associated with the same resource manager.

In such a circumstance, the `prepare` method of only one of these `OracleXAResource` instances will return `XA_OK` or will fail. The rest will return `XA_RDONLY`, even if updates are made. This allows the transaction manager to implicitly join all the transaction branches and commit or roll back, in case of failure, the joined transaction through the `OracleXAResource` instance that returned `XA_OK` or failed.

The transaction manager can use the `OracleXAResource` class `isSameRM` method to determine if two `OracleXAResource` instances are using the same resource manager. This way it can interpret the meaning of `XA_RDONLY` return values.

37.4 About Implementing a Distributed Transaction

This section provides an example of how to implement a distributed transaction using Oracle XA functionality. This section covers the following topics:

- Summary of Imports for Oracle XA
- Oracle XA Code Sample

37.4.1 Summary of Imports for Oracle XA

You must import the following for Oracle XA functionality:

```
import oracle.jdbc.xa.OracleXid;
import oracle.jdbc.xa.OracleXAException;
import oracle.jdbc.pool.*;
import oracle.jdbc.xa.client.*;
import javax.transaction.xa.*;
```

The `oracle.jdbc.pool` package has classes for connection pooling functionality, some of which have XA-related classes as subclasses.

Alternatively, if the code will run inside Oracle Database and access that database for SQL operations, you must import `oracle.jdbc.xa.server` instead of `oracle.jdbc.xa.client`.

```
import oracle.jdbc.xa.server.*;
```

If your application must access another Oracle Database as part of an XA transaction using the server-side Thin driver, then your code can use the fully qualified names of the `oracle.xa.client` classes.

The `client` and `server` packages each have versions of the `OracleXADataSource`, `OracleXAConnection`, and `OracleXAResource` classes. Abstract versions of these three classes are in the top-level `oracle.jdbc.xa` package.

37.4.2 Oracle XA Code Sample

This example uses a two-phase distributed transaction with two transaction branches, each to a separate database.

Note that for simplicity, this example combines code that would typically be in a middle tier with code that would typically be in a transaction manager, such as the `OracleXAResource` method invocations and the creation of transaction IDs.

For brevity, the specifics of creating transaction IDs and performing SQL operations are not shown here. The complete example is shipped with the product.

This example performs the following sequence:

1. Start transaction branch #1.
2. Start transaction branch #2.
3. Execute DML operations on branch #1.
4. Execute DML operations on branch #2.
5. End transaction branch #1.
6. End transaction branch #2.
7. Prepare branch #1.
8. Prepare branch #2.
9. Commit branch #1.
10. Commit branch #2.

```
// You need to import the java.sql package to use JDBC
import java.sql.*;
import javax.sql.*;
import oracle.jdbc.*;
import oracle.jdbc.pool.*;
import oracle.jdbc.xa.OracleXid;
import oracle.jdbc.xa.OracleXAException;
import oracle.jdbc.xa.client.*;
import javax.transaction.xa.*;

class XA4
{
  public static void main (String args [])
       throws SQLException
  {

    try
    {
        String URL1 = "jdbc:oracle:oci:@";
        // You can put a database name after the @ sign in the connection URL.
        String URL2 ="jdbc:oracle:thin:@(description=(address=(host=localhost)
                     (protocol=tcp)(port=5521))(connect_data=(service_name=orcl)))";
        // Create first DataSource and get connection
        OracleDataSource ods1 = new OracleDataSource();
        ods1.setURL(URL1);
        ods1.setUser("HR");
        ods1.setPassword("hr");
        Connection conna = ods1.getConnection();

        // Create second DataSource and get connection
```

```
OracleDataSource ods2 = new OracleDataSource();
ods2.setURL(URL2);
ods2.setUser("HR");
ods2.setPassword("hr");
Connection connb = ods2.getConnection();

// Prepare a statement to create the table
Statement stmta = conna.createStatement ();

// Prepare a statement to create the table
Statement stmtb = connb.createStatement ();

try
{
  // Drop the test table
  stmta.execute ("drop table my_table");
}
catch (SQLException e)
{
  // Ignore an error here
}

try
{
  // Create a test table
  stmta.execute ("create table my_table (col1 int)");
}
catch (SQLException e)
{
  // Ignore an error here too
}

try
{
  // Drop the test table
  stmtb.execute ("drop table my_tab");
}
catch (SQLException e)
{
  // Ignore an error here
}

try
{
  // Create a test table
  stmtb.execute ("create table my_tab (col1 char(30))");
}
catch (SQLException e)
{
  // Ignore an error here too
}

// Create XADataSource instances and set properties.
OracleXADataSource oxds1 = new OracleXADataSource();
oxds1.setURL("jdbc:oracle:oci:@");
oxds1.setUser("HR");
oxds1.setPassword("hr");

OracleXADataSource oxds2 = new OracleXADataSource();

oxds2.setURL("jdbc:oracle:thin:@(description=(address=(host=localhost)
           (protocol=tcp)(port=5521))(connect_data=(service_name=orcl)))");
```

```
oxds2.setUser("HR");
oxds2.setPassword("hr");

// Get XA connections to the underlying data sources
XAConnection pc1  = oxds1.getXAConnection();
XAConnection pc2  = oxds2.getXAConnection();

// Get the physical connections
Connection conn1 = pc1.getConnection();
Connection conn2 = pc2.getConnection();

// Get the XA resources
XAResource oxar1 = pc1.getXAResource();
XAResource oxar2 = pc2.getXAResource();

// Create the Xids With the Same Global Ids
Xid xid1 = createXid(1);
Xid xid2 = createXid(2);

// Start the Resources
oxar1.start (xid1, XAResource.TMNOFLAGS);
oxar2.start (xid2, XAResource.TMNOFLAGS);

// Execute SQL operations with conn1 and conn2
doSomeWork1 (conn1);
doSomeWork2 (conn2);

// END both the branches -- IMPORTANT
oxar1.end(xid1, XAResource.TMSUCCESS);
oxar2.end(xid2, XAResource.TMSUCCESS);

// Prepare the RMs
int prp1 =  oxar1.prepare (xid1);
int prp2 =  oxar2.prepare (xid2);

System.out.println("Return value of prepare 1 is " + prp1);
System.out.println("Return value of prepare 2 is " + prp2);

boolean do_commit = true;

if (!((prp1 == XAResource.XA_OK) || (prp1 == XAResource.XA_RDONLY)))
   do_commit = false;

if (!((prp2 == XAResource.XA_OK) || (prp2 == XAResource.XA_RDONLY)))
   do_commit = false;

System.out.println("do_commit is " + do_commit);
System.out.println("Is oxar1 same as oxar2 ? " + oxar1.isSameRM(oxar2));

if (prp1 == XAResource.XA_OK)
  if (do_commit)
     oxar1.commit (xid1, false);
  else
     oxar1.rollback (xid1);

if (prp2 == XAResource.XA_OK)
  if (do_commit)
     oxar2.commit (xid2, false);
  else
     oxar2.rollback (xid2);

 // Close connections
```

```
        conn1.close();
        conn1 = null;
        conn2.close();
        conn2 = null;

        pc1.close();
        pc1 = null;
        pc2.close();
        pc2 = null;

        ResultSet rset = stmta.executeQuery ("select col1 from my_table");
        while (rset.next())
          System.out.println("Col1 is " + rset.getInt(1));

        rset.close();
        rset = null;

        rset = stmtb.executeQuery ("select col1 from my_tab");
        while (rset.next())
          System.out.println("Col1 is " + rset.getString(1));

        rset.close();
        rset = null;

        stmta.close();
        stmta = null;
        stmtb.close();
        stmtb = null;

        conna.close();
        conna = null;
        connb.close();
        connb = null;

      } catch (SQLException sqe)
      {
        sqe.printStackTrace();
      } catch (XAException xae)
      {
        if (xae instanceof OracleXAException) {
          System.out.println("XA Error is " +
                       ((OracleXAException)xae).getXAError());
          System.out.println("SQL Error is " +
                       ((OracleXAException)xae).getOracleError());
        }
      }
    }

    static Xid createXid(int bids)
      throws XAException
    {...Create transaction IDs...}

    private static void doSomeWork1 (Connection conn)
     throws SQLException
    {...Execute SQL operations...}

    private static void doSomeWork2 (Connection conn)
     throws SQLException
    {...Execute SQL operations...}
}
```

37.5 Native-XA in Oracle JDBC Drivers

In general, XA commands can be sent to the server in the following ways:

- Through non-native APIs
- Through native APIs

There is a huge performance difference between the two methods of sending XA commands to the server. The use of native APIs provide high performance gains as compared to the use of non-native APIs.

Prior to Oracle Database 10*g*, the Thin driver used non-native APIs to send XA commands to the server because Thin native APIs were not available. The non-native APIs use PL/SQL procedures, so they have the following disadvantages:

- They require different messages on the wire.
- They cause more round-trips to the database.
- They cause more cursors to remain open.
- They damage statement caching by occupying space in the Statement Cache.

Moreover, the implementation of non-native APIs is in the server. So, in order to solve any problem in sending XA commands, it requires a server patch. This creates a major issue because sometimes the patch requires restarting the server.

Starting from Oracle Database 10*g*, the Thin native APIs are available and are used to send XA commands, by default. Native APIs are more than 10 times faster than the non-native ones.

This section covers the following topics:

- OCI Native XA
- Thin Native XA

37.5.1 OCI Native XA

Native XA is enabled through the use of the `tnsEntry` and `nativeXA` properties of the `OracleXADataSource` class.

> **Note:**
>
> Currently, OCI Native XA does not work in a multithreaded environment. The OCI driver uses the C/XA library of Oracle to support distributed transactions, which requires that an XAConnection be obtained for each thread before resuming a global transaction.

Configuration and Installation

On a Solaris or Linux system, you need the `libheteroxa11.so` shared library to enable the Native XA feature. This library must be installed and available in the search path for the Native XA feature to work properly.

On a Microsoft Windows system, you need the `heteroxa11.dll` file to enable the Native XA feature. This file must be installed and available in the Windows DLL path for the Native XA feature to work properly.

Exception Handling

When using the Native XA feature in distributed transactions, it is recommended that the application simply check for `XAException` or `SQLException`, rather than `OracleXAException` or `OracleSQLException`.

> **Note:**
>
> The mapping from SQL error codes to standard XA error codes does not apply to the Native XA feature.

Native XA Code Example

The following portion of code shows how to enable the Native XA feature:

```
...
// Create a XADataSource instance
OracleXADataSource oxds = new OracleXADataSource();
oxds.setURL(url);

// Set the nativeXA property to use Native XA feature
oxds.setNativeXA(true);

// Set the tnsEntry property to an older DB as required
oxds.setTNSEntryName("ora805");
...
```

Related Topics

- Features and Properties of Data Sources

37.5.2 Thin Native XA

Like the JDBC OCI driver, the JDBC Thin driver also provides support for Native XA. However, the JDBC Thin driver provides support for Native XA by default. This is unlike the case of the JDBC OCI driver in which the support for Native XA is not enabled by default.

You can disable Native XA by calling `setNativeXA(false)` on the XA data source as follows:

```
...
// Create a XADataSource instance
OracleXADataSource oxds = new OracleXADataSource();
...
// Disabling Native XA
oxds.setNativeXA(false);
...
```

For example, you may need to disable Native XA as a workaround for a bug in the Native XA code.