32 Transaction Guard for Java

The Transaction Guard feature of Oracle Database provides a generic infrastructure for at-most-once execution during planned and unplanned outages and duplicate submissions. This chapter discusses Transaction Guard for Java in the following sections:

- Overview of Transaction Guard for Java
- Transaction Guard Support for XA Transactions
- Transaction Guard for Java APIs
- Complete Example:Using Transaction Guard APIs
- About Using Server-Side Transaction Guard APIs

32.1 Overview of Transaction Guard for Java

For most of the current applications, determining the outcome of the last commit operation in a guaranteed and scalable manner, following a communication failure to the server, is an unsolved problem. In many cases, the applications prompt the end users to follow certain steps to avoid resubmitting duplicate requests. For example, some applications warn users not to click the Submit button twice because if it is not followed, then users may unintentionally purchase the same items twice and submit multiple payments for the same invoice.

To solve this problem, Transaction Guard for Java provides transaction idempotence, that is, every transaction has at-most-once execution that prevents applications from submitting duplicate transactions. You can tag every transaction with a Logical Transaction Identifier (`LTXID`), which the application can use after the occurrence of a failure to verify whether the transaction had committed before the failure or not. For example, if the commit calls do not return, then using the `LTXID`, the application can find out whether the calls succeeded or not.

The Application Continuity for Java feature uses Transaction Guard for Java internally, which enables transparent session recovery and replay of SQL statements (queries and DMLs) since the beginning of the in-flight transaction. Application Continuity enables recovery of work after the occurrence of a planned or unplanned outage and Transaction Guard for Java ensures transaction idempotence. When an outage occurs, the recovery restores the state exactly as it was before the failure occurred.

Related Topics

- Application Continuity for Java
  The outages of the underlying software, hardware, communications, and storage layers can cause application execution to fail. In the worst cases, the middle-tier servers may need to be restarted to deal with a logon storm, which is a sudden increase in the number of client connection requests.

32.2 Transaction Guard Support for XA Transactions

Starting from Oracle Database 12*c* Release 2 (12.2.0.1), Transaction Guard provides support for XA transactions for one-phase commit optimization, read-only optimization, and promotable XA. Transaction Guard with XA provides safe replay following recoverable outages for XA transactions. With the addition of XA support, Transaction Managers can now provide replay with idempotence enforced more easily using Transaction Guard.

> **Note:**
>
> For using Transaction Guard with XA, during session check-out from the connection pool, you must verify that the database version is Oracle 12*c* Release 2 (12.2.0.1) or later, and Transaction Guard is enabled.

A new server protocol provides a guaranteed commit outcome when the commit is one-phase managed by the database, and switches to a disabled mode while the Transaction Manager coordinates a transaction for that session. The new protocol sets a status flag in the LTXID that validates and invalidates access to the LTXID, based on the current transaction owner.

The protocol is intelligent in its handling that XA can encompass many sessions and many branches for the one XA transaction. As a further challenge, once a branch is suspended, a session is available for different transactions, while the original transaction remains active. There is no requirement to prepare or commit XA transactions on the same session or RAC instance that created the original branches. Transaction Guard for XA uses the following two new methods for handling commit outcome at the database for one-phase XA transactions, while the Transaction Manager continues to handle commit-outcome for two-phase transactions:

- Using the first method, the driver marks the LTXID provisional until a recoverable error, or any other named condition, occurs on that session. When a recoverable error (or any other condition) occurs, the LTXID at the client is marked final. The guaranteed commit outcome is provided only when the LTXID is final at the client, and at the server that LTXID has a VALID status, indicating that the database owns that transaction. Any other access attempt returns an error.
- Using the second method, the client driver does not provide the LTXID to the application until a recoverable error, or other named condition, occurs on that session.

32.3 How to Use Transaction Guard with XA

This section contains the following sections:

Obtaining the Commit Outcome with Promotable XA

For local transactions, the request obtains an LTXID as the transaction key, when there is a recoverable exception. When a second branch starts, then the request is promoted to XA, or converted to XA, and a Global Transaction ID (GTRID) is allocated to it. If a recoverable outage occurs during commit processing, where the application does not receive a reply from the Transaction Manager, then the application can ask a Transaction Manager for the outcome. Most requests to the database use either local transactions or single branch optimization. When you use either local transactions or promotable XA, then there is no overhead in round trips and management for XA, because the majority of the transactions are local. The workflow of these transactions follows:

1. Prior to converting to XA, transaction processing is local. Authentication, `SELECT` statements, and local transactions carry and use the LTXID of the local transaction.
2. The Transaction Manager allocates a GTRID to the transaction only when it starts to use XA due to opening a second branch of the transaction.

3. Following a recoverable error, when the application does not receive a commit outcome, if the transaction is local, then the Transaction Manager can use the LTXID with the `GET_LTXID_OUTCOME` procedure to retrieve the commit outcome and return `COMMITTED` or `UNCOMMITTED` outcome to the application.

Replaying if Promotable XA Is Added

Before being promoted, promotable XA supports RDBMS commits through calls and settings that are not supported by static XA. These calls include auto-commit mode, DDL, DCL, COMMIT embedded in PL/SQL, and COMMIT through remote procedure calls. The COMMIT outcome for these user calls and modes is controlled by the RDBMS, and following an error, the commit outcome can be found using Transaction Guard.

Until promoted, the Transaction Manager is unaware whether the request has issued any COMMIT or not. If the Transaction Manager wishes to replay a request following a recoverable error, then the Transaction Manager must determine if any RDBMS COMMIT has occurred. If any RDBMS COMMIT occurs, or can occur, then replay does not happen. The `GET_LTXID_OUTCOME` procedure is insufficient in determining this because the procedure only reports the current transaction outcome. If the LTXID is changed, then the transaction is committed. So, the invocation of the LTXID callback indicates that the transaction is committed.

32.4 Transaction Guard for Java APIs

This section discusses the APIs associated with Transaction Guard for Java for the following activities:

- Retrieving the Logical Transaction Identifiers
- Retrieving the Updated Logical Transaction Identifiers

32.4.1 Retrieving the Logical Transaction Identifiers

Use the `getLogicalTransactionId` method of the `oracle.jdbc.OracleConnection` interface to retrieve the current Logical Transaction Identifiers that are sent by the server. This method call does not make a database round-trip.

Example

```
OracleConnection oconn = (OracleConnection) ods.getConnection();
...
// Getting the 1st LTXID after connecting
LogicalTransactionId firstLtxid = oconn.getLogicalTransactionId();
```

32.4.2 Retrieving the Updated Logical Transaction Identifiers

Use the `oracle.jdbc.LogicalTransactionIdEventListener` interface for receiving updates to Logical Transaction Identifiers. You must implement this interface in your application to process the Logical Transaction Identifier events.

32.4.2.1 Registering Event Listeners

Use the `addLogicalTransactionIdEventListener` method to register a listener to the Logical Transaction Identifier events.

Example

```
OracleConnection oconn = (OracleConnection) ods.getConnection();
...
// The subsequent LTXID updates can be obtained through the listener
oconn.addLogicalTransactionIdEventListener(this);
```

You can also use the `addLogicalTransactionIdEventListener(LogicalTransactionIdEventListener listener, java.util.concurrent.Executor executor)` method to register a listener with an executor.

32.4.2.2 Unregistering Event Listeners

Use the `removeLogicalTransactionIdEventListener` method to unregister a listener from the Logical Transaction Identifier events.

Example

```
OracleConnection oconn = (OracleConnection) ods.getConnection();
...
// The subsequent LTXID updates can be obtained through the listener
oconn.removeLogicalTransactionIdEventListener(this);
```

32.5 Complete Example:Using Transaction Guard APIs

The following is a complete example using the Transaction Guard APIs.

```
import oracle.jdbc.pool.OracleDataSource;
import oracle.jdbc.OracleConnection;
import oracle.jdbc.LogicalTransactionId;
import oracle.jdbc.LogicalTransactionIdEvent;
import oracle.jdbc.LogicalTransactionIdEventListener;

public class transactionGuardExample
{
      ...
      ...
      OracleDataSource ods = new OracleDataSource();
      ods.setURL(url);
      ods.setUser("user");
      ods.setPassword("password");

      OracleConnection oconn = (OracleConnection) ods.getConnection();

      // Getting the 1st LTXID after connecting
      LogicalTransactionId firstLtxid = oconn.getLogicalTransactionId();

      // The subsequent LTXID updates can be obtained via the listener
      oconn.addLogicalTransactionIdEventListener(this);
  }

public class LtxidListenerImpl
  implements LogicalTransactionIdEventListener
{
  ...

  public void onLogicalTransactionIdEvent(LogicalTransactionIdEvent ltxidEvent)
```

```
            {
              LogicalTransactionId newLtxid = ltxidEvent.getLogicalTransactionId();
              // process newLtxid ......
            }
          }
```

32.6 About Using Server-Side Transaction Guard APIs

The `DBMS_APP_CONT` package contains the `GET_LTXID_OUTCOME` procedure that contains the server-side Transaction Guard APIs. This procedure forces the outcome of a transaction. If the transaction is not committed, then a fake transaction is committed. Otherwise, the state of the transaction is returned. By default, the `EXECUTE` privilege for this package is granted to Database Administrators.

Syntax

```
  PROCEDURE GET_LTXID_OUTCOME(CLIENT_LTXID        IN  RAW,
                              committed           OUT BOOLEAN,
                              USER_CALL_COMPLETED OUT BOOLEAN);
```

Input Parameter

`CLIENT_LTXID` specifies the `LTXID` from the client driver.

Output Parameter

`COMMITTED` specifies that the transaction is committed.

`USER_CALL_COMPLETED` specifies that the user call, which committed the transaction, is complete.

Exceptions

`SERVER_AHEAD` is thrown when the server is ahead of the client. So, the transaction is an old transaction and must have already been committed.

`CLIENT_AHEAD` is thrown when the client is ahead of the server. This can only happen if the server is flashed back or the `LTXID` is corrupted. In either of these situations, the outcome cannot be determined.

`ERROR` is thrown when an error occurs during processing and the outcome cannot be determined. It specifies the error code raised during the execution of the current procedure.

Example

Example 32-1 shows how you can call the `GET_LTXID_OUTCOME` procedure and find out the outcome of an `LTXID`:

Example 32-1 Finding Out the Outcome of an LTXID

```
...
OracleConnection oconn = (OracleConnection) ods.getConnection();
LogicalTransactionId ltxid = oconn.getLogicalTransactionId();
boolean committed = false;
boolean call_completed = false;

try
{
  CallableStatement cstmt = oconn.prepareCall(GET_LTXID_OUTCOME);
  cstmt.setObject(1, ltxid);
```

```
    cstmt.registerOutParameter(2, OracleTypes.BIT);
    cstmt.registerOutParameter(3, OracleTypes.BIT);

    cstmt.execute();

    committed = cstmt.getBoolean(2);
    call_completed = cstmt.getBoolean(3);

    System.out.println("LTXID committed ? " + committed);
    System.out.println("User call completed ? " + call_completed);
  }
  catch (SQLException sqlexc)
  {
    System.out.println("Calling GET_LTXID_OUTCOME failed");
    sqlexc.printStackTrace();
  }
```