ORACLE
DATABASE

Transaction Guard with Oracle Database12*c* Release 2

Hiding Outages

ORACLE WHITE PAPER | MARCH 2017

ORACLE

Table of Contents

.

Executive Overview

Transaction Guard with Oracle Database 12^{c} is a reliable protocol and interface that returns the commit outcome of the current in-flight transaction when an error or timeout is returned to the client instead of the real result. Applications or infrastructure embed the Transaction Guard API's in their error handling and use Transaction Guard to return the real result when outages occur.

Transaction Guard avoids the costs of clients receiving ambiguous errors that lead to user frustration, customer support calls and lost opportunities. Without Transaction Guard, applications and users who attempt to retry operations following an error or timeout can cause logical corruption by committing duplicate transactions or committing transactions out of order. Transaction Guard guarantees correctness and scales to cloud and internet levels, with lower overheads and a great deal better performance, than home grown and external solutions can achieve for known commit outcome and at-most-once execution

Customer quotes from Oracle Database 12c Reference and Oracle Open World 2015

"About Transaction Guard, I think that it should be used generally and made a standard for the internet."

"Having Transaction Guard available will allow our application programmers to deal gracefully with most errors conditions."

"Using Transaction Guard we can submit POS requests asynchronously and not keep customers waiting."

"Transaction Guard will tell us that the proceeds transferred with no additional overhead in infrastructure."

"The savings with Transaction Guard are multiplied in satisfaction, reduced support calls, and reduced overheads".

"With Transaction Guard we replace unsafe cancelling of requests, with Transaction Guard's safe cancelling of requests and our own replay is now safe from duplicates."

Introduction

Using Transaction Guard, the end user experience is vastly improved by returning to the application and user, following an outage, whether the last submission committed and completed or did not. After submitting a request, it is far better to know whether your funds transfer, bill payment, form submission and so on has executed or, that it was not done and that it is safe to re-submit. Without using Transaction Guard, users can receive a vague, ambiguous error message following an outage, and are left not knowing what happened to the last in-flight operation. Applications typically display messages that can be frustrating such as the following:

- Please call customer support
- Do not press resubmit or reload the page
- Do not use the backspace key
- The system is currently experiencing problems. Please try again later.
-

Developers embed the Transaction Guard APIs in their application or mid-tier error handling to force a guaranteed request outcome. Transaction Guard's protocol enforces that when the commit outcome is returned to the application, that outcome persists with the value that is returned to the application. Using Transaction Guard, once a committed or uncommitted result is returned to the application, the result stays this way. This is critically important. A committed result stays committed. An uncommitted result stays uncommitted, and is a green light, for example, for a user to resubmit. It can also be a green light for applications to resubmit themselves in case of an outage. Figure one shows the previous experience without Transaction Guard. When an error or timeout is returned to request, users are left not knowing what happened to the work that was submitted.

Figure one. Users can receive errors even when the request committed.

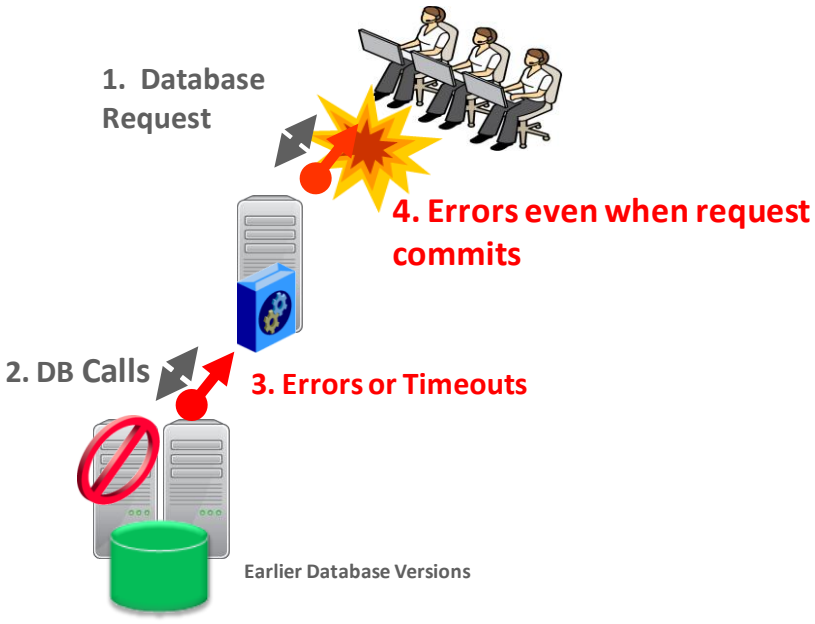


Figure Two. By adopting Transaction Guard end users receive the real response rather than ambiguous errors

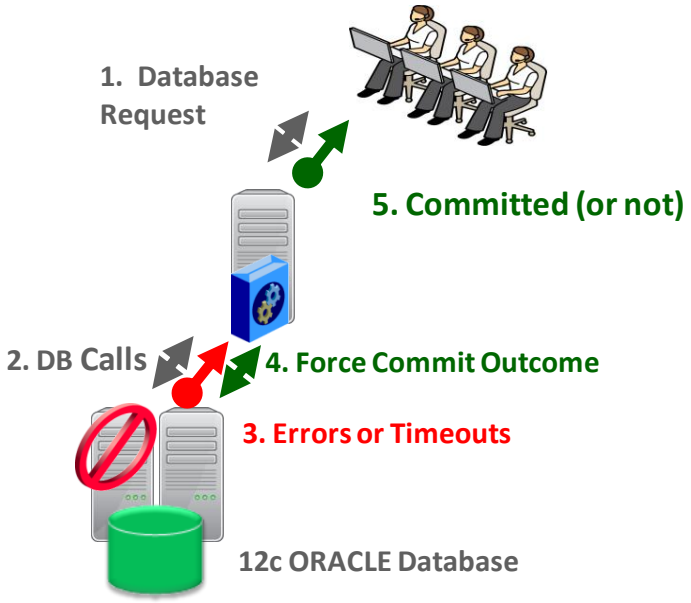


Figure two shows the far better user experience when Transaction Guard is adopted to return the real commit outcome. Following the flow in figure two -

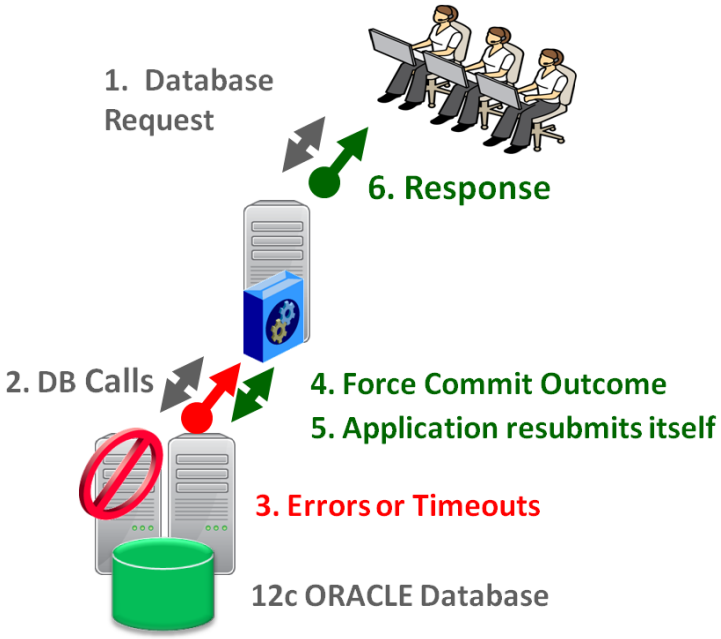
- (step 1) The client submits a request to the application
- (step 2) The application submits database calls to the Oracle 12c database
- (step 3) An error or timeout occurs due a failure in the underlying system or networks when a request is underway. The standard error handling traps this exception.
- (step 4).The error handling invokes Transaction Guard to return the commit outcome of the last in-flight work on that session.
- (step 5) A reliable commit outcome is returned rather than the ambiguous error, so users know whether the work succeeded or did not succeed.

Once Transaction Guard is adopted, it is safe for applications and mid-tiers to return success or resubmit themselves following an uncommitted result. Figure three enhances the workflow in figure two, with steps five and six where the application resubmits safely. With 12c, many applications are replacing their unsafe cancelling of requests, with Transaction Guard's safe cancelling of requests followed by their own replay. Customers can also use Application continuity that does all the work for them beneath the application.

Figure three continues from figure two, and replaces step five as follows:

- (step 5) If committed, return the committed result to the users. If uncommitted, it is safe for the application to resubmit itself.
- (step 6) When successful, a response is returned to the users so in most cases the work is completed as if an error had not occurred

Figure three. By adopting Transaction Guard, Applications can cancel and replay themselves safely



The benefits of using Transaction Guard include:

- For businesses, a much better user experience with fewer support calls and lost opportunities.
- For users, a reliable commit outcome for the last work submitted following outages.
- For developers, increased productivity by correctly handling and processing outages and errors.
- Overall, cloud scalability, increased performance, and higher safety than home built and external solutions for handling request cancellation and resubmission.

New Concepts for Application Failover with Oracle 12c

Logical Transaction Identifier (LTXID)

Applications use a concept called the logical transaction identifier (LTXID) to determine the commit outcome of the last transaction open in a database session following an outage. The LTXID is owned by the database and a copy is held in the OCI session handle and in a connection object for the JDBC-Thin driver, and both managed and unmanaged Oracle Data Provider for .NET (ODP.NET) drivers. The logical transaction ID is used to obtain the commit outcome and is used to enforce the at-most once semantics for local, one-phase transactions, and two-phase transactions over database links.

Reliable Commit Outcome

From the client perspective, the transaction is committed when an Oracle message termed the Commit Outcome, generated after the transaction redo is written, is received by the client. However, the COMMIT message is not durable. When this message is lost the application may receive errors or timeouts that do not have a relationship as to whether the transaction committed or not. Transaction Guard obtains the Commit Outcome reliably when it has been lost following a recoverable error. The reliability is a critical element. Transaction Guard returns, committed or uncommitted, no matter where and when the error. Once a transaction is forced by Transaction Guard, the returned result never changes no matter how often it is asked for.

Recoverable Error Classification

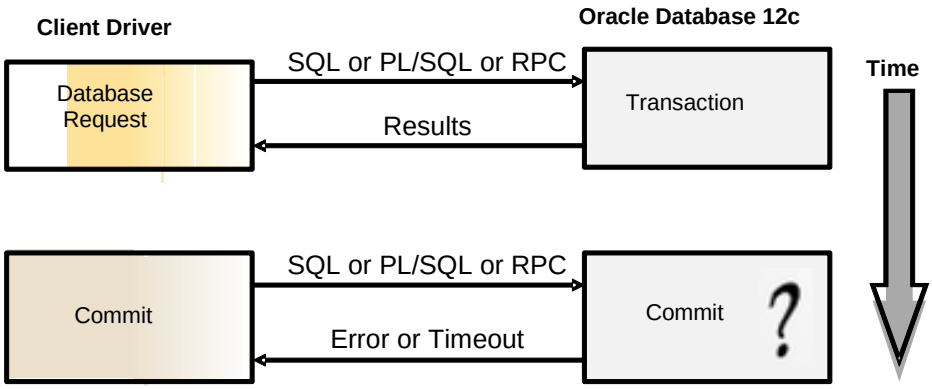
A recoverable error is an error that arises due to an external system failure, independent of the application session logic that is executing. Recoverable errors occur during planned maintenance, and following unplanned outages of sessions, networks, nodes, storage, and databases. Recoverable errors also occur when systems are unresponsive and causing requests to time-out. The application receives an error code that can leave the application not knowing the status of the last user call that was submitted. Recoverable errors are enhanced in Oracle Database 12c, to include more errors and to include a public API for OCI. Applications should no longer list error numbers in their code and instead use the new API.

- As it pertains to Transaction Guard whether something committed or not should not be hinged on whether the error code was recoverable. For example, just because an OSD error was received does not mean that it did not first commit. However, if the application plans to resubmit automatically, the resubmission may have a better chance of succeeding if the original error code received was recoverable. The recoverable error APIs are per driver:
- SQLRecoverableException for JDBC thin
- OracleException.IsRecoverable property for ODP.NET
- OCI_ATTR_ERROR_IS_RECOVERABLE error handle attribute for OCI

Problem that Transaction Guard Solves

One of the fundamental problems for recovering applications after an outage is that the commit message that is sent back to the client is not durable. If there is a break between the client and the server, the client sees an error message indicating that the communication failed. This error does not inform the application whether the submission executed any commit operations or if a procedural call, ran to completion executing all expected commits and session state changes or failed part way through or yet worse, is still running disconnected from the client, as illustrated in figure four.

Figure Four. Transaction Guard solves determining a lost commit outcome



Determining the outcome of the last commit operation in a reliable and scalable manner, following a communication error or timeout, in a performing and reliable way, has been an unsolved problem. If an application needs to know whether the submission to the database was committed, the application needs to add custom exception code to query the outcome for every possible commit point in the application. Given that a system can fail anywhere, this is impractical in general as the check must be specific to each submission. After an application is built and is in production, this is completely impractical. Moreover, a check cannot give the correct answer because the transaction could commit immediately after that check executed. Indeed, following a timeout received at the client, the server may still be running the submission not yet aware that the client has abandoned the request. In addition to the commit dilemma, when using PL/SQL or Java in the database, there is also no record for a procedural submission as to whether that submission ran to completion or was aborted part way through. While such a procedure may have committed, subsequent work that is needed may not have been done for the procedure.

Failing to recognize that the last submission has committed, or shall commit sometime soon, or has not run to completion, can lead users and applications that attempt to resubmit to cause duplicate transactions and other forms of “logical corruption” as they might try to re-issue already persisted changes. Users may also progress to the next operation losing a required step in the process.

Transaction Guard Coverage with Oracle Database 12c

You may use Transaction Guard on each database in your system including restarting on and failing over between single instance database, Real Application Clusters, Data Guard and Active Data Guard. Transaction Guard is supported on Enterprise Edition and higher with the following Oracle Database 12c configurations:

- Single Instance Oracle RDBMS
- Real Application Clusters
- RAC One
- Data Guard
- Active Data Guard
- Multitenant including unplug/plug and online PDB relocate
- Global Data Services for the above database configurations

Transaction Guard supports the following transaction types against Oracle Database 12c:

- Local transactions
- DDL and DCL transactions
- Distributed and Remote transactions
- Parallel transactions
- Commit on Success (auto-commit)
- PL/SQL with embedded COMMIT
- Starting with Oracle Database 12c Release 2 (12.2.0.1), XA transactions using One Phase Optimizations including XA commit flag TMONEPHASE and read optimizations
- ALTER SESSION SET Container with Service clause, where the service uses Transaction Guard

Transaction Guard supports the following client drivers:

- 12c JDBC-Thin Driver
- 12c OCI and OCCI client drivers
- 12c ODP.NET, Managed Driver
- 12c ODP.NET, Unmanaged Driver

Transaction Guard Exclusions

- Transaction Guard intentionally excludes recursive transactions and autonomous transactions so that these can be re-executed. Transaction Guard for Oracle 12c Release 2 excludes
- Active Data Guard with read/write DB Links for forwarding transactions
- JDBC OCI is not supported, use JDBC thin driver
- Two Phase XA transactions managed externally. When using XA, Transaction Guard returns the commit outcome for one phase XA transactions, and silently disables for externally-managed two-phase as this outcome is owned by the TP monitor (see Transaction Guard with XA Transactions)

- Transaction Guard cannot be used in the TAF Callback for TAF or Application Continuity for OCI and ODP.NET, or in the JDBC initialization callback for Application Continuity for Java. TAF and Application Continuity are handling Transaction Guard internally.
- Full database import cannot be executed with Transaction Guard enabled. Use an admin service without Transaction Guard for full database imports. User and object imports are not excluded.

Transaction Guard excludes failover across databases maintained by replication technology:

- Replication to Golden Gate
- Replication to Logical Standby
- Third Party replication solutions
- PDB clones clause (excluding PDB online relocation 12c Release 2 and later)

If you are using a database replica using any replication technology such as Golden Gate or Logical Standby or 3rd party replication, you cannot use Transaction Guard between the primary and the secondary databases in this configuration. You may use Transaction Guard within each database that participates in the replication. In this use case, each database must use a different database identifier (DBID). Use V$DATABASE to obtain the DBID for each database.

Transaction Guard with XA Transactions

Starting with Oracle Database 12c Release 2, Transaction Guard supports XA transactions to determine the outcome of one phase optimizations. Transaction Guard supports local transactions and XA transactions that use TMONEPHASE during the commit operation. When the application issues an XA transaction that uses TMTWOPHASE, the Transaction Guard disables itself for that transaction and automatically re-enables to prepare itself for the next transaction. This allows Transaction Guard to support transactions on XA data sources:

- Local transactions that use autocommit
- Local transactions that use an explicit commit
- XA transactions that commit with TMONEPHASE flag

TP Monitors and Applications can use Transaction Guard to obtain the outcome of commit operation for these transaction types. Transaction Guard disables itself for externally-managed TMTWOPHASE commit operations and automatically re-enables for the next transaction. If the Transaction Guard APIs are used with a TMTWOPHASE transaction, a warning message is returned as Transaction Guard is disabled. The TP monitor itself owns the commit outcome for TMTWOPHASE. This functionality allows TP monitor to now return an unambiguous outcome for TMONEPAHSE operations.

Oracle Database Configuration for Transaction Guard

Required Steps

Use Oracle Database Release 12.1 or later.

Use an application service for all database work. Create the service using srvctl if using RAC or DBMS_SERVICE if not using RAC. You may also use GDSCTL.

Set the following properties on the service – COMMIT_OUTCOME = TRUE for Transaction Guard

Grant execute permission on DBMS_APP_CONT package to the application user

Increase DDL_LOCK_TIMEOUT if using Transaction Guard with DDL statements (for example, 10 seconds).

Recommended Steps

Check the service parameter RETENTION_TIMEOUT - how long in seconds to maintain history. Keep this value high, 24 hours or longer for example. The default value should be sufficient.

Check performance. If necessary, locate the transaction history table (LTXID_TRANS) for optimal performance. When using Multitenant, there is one transaction history table per PDB.

If using Oracle Real Application Clusters (RAC) or Oracle Data Guard, ensure that FAN is configured with ONS to talk with 12c database clients

On the service, set AQ_HA_NOTIFICATIONS = TRUE (for OCI and ODP.NET FAN)

Use connection pools and do not ever use a low expire time on sessions. This is important for basic performance. It also ensures Transaction Guard is exceptionally low cost.

Service Parameters

Do NOT use the database service as this service is for administration purposes, not for application usage. That is, it is do not to use the default service that is set to db_name or db_unique_name or pdb_name

COMMIT_OUTCOME

Description – COMMIT_OUTCOME determines whether the transaction COMMIT outcome is accessible after the COMMIT has executed. While the database has always made COMMIT durable, Transaction Guard makes the outcome of the COMMIT durable and is used by applications to enforce the status of the last transaction executed before an outage.

Default – FALSE

Values – TRUE and FALSE

Restrictions

Using the GET_LTXID_OUTCOME Pl/SQL call requires that COMMIT_OUTCOME attribute is set.

COMMIT_OUTCOME has no effect on Active Data Guard in read mode or read only databases. Using Transaction Guard with Active Data Guard combined with DML forwarding is not supported.

COMMIT_OUTCOME is allowed on user defined database services. It is not supported to be set on the default database service or the default pluggable database service.

RETENTION_TIMEOUT

Description – retention timeout is used in conjunction with COMMIT_OUTCOME. It determines the amount of time that the COMMIT_OUTCOME is retained. The transaction history table is small. Keep the retention_timeout high so that sessions returning much later can determine their outcome. The default value should be sufficient for most installations.

Set using DBMS_SERVICE, srvctl or gdsctl

Units – seconds

Default – 24 hours (86400)

Maximum value – 30 days (2592000)

Example Service Settings

Example for creating and modifying services on RAC:

If using Oracle RAC or RAC One, use srvctl to create and modify services:

Example for policy- managed services:

```
srvctl add service -d orcl -s GOLD -g ora.Srvpool  -commit_outcome TRUE  -retention 86400

srvctl modify service -d orcl -s GOLD -commit_outcome TRUE
```

Example for administration managed services:

```
srvctl add service -d codedb -s GOLD -r serv1 -a serv2 -commit_outcome TRUE  60 –stopoption immediate

srvctl modify service -d orcl -s GOLD -commit_outcome TRUE
```

Example for modifying services on single instance database:

If using a single instance database (not RAC and not Global Database Services), use DBMS_SERVICE to modify services. To create or modify a service on a PDB using DBMS_SERVICE, you must be connected to that PDB.

```
declare

params dbms_service.svc_parameter_array;

begin

params('COMMIT_OUTCOME'):='true';

params('RETENTION_TIMEOUT'):=86400;

dbms_service.modify_service('[your service]',params);

end;
```

Grant Permission to use Commit_Outcome

Ensure that permission on the DBMS_APP_CONT package has been granted to the database users that will call GET_LTXID_OUTCOME:

```
GRANT EXECUTE ON DBMS_APP_CONT TO <user-name>;
```

Performance

Resource Usage

Transaction Guard is designed as a scalable protocol. It is necessarily light weight and scales using partitioning across Real Application Clusters and on single instance database. Performance measures using instruction counts through to high OLTP workloads show less than 0.05% increase in CPU for all workloads measured. Adding Transaction Guard should not be discernible, and should reduce costs where it is used to replace home grown solutions.

With FAST_START_MTTR_TARGET set measures show that in high OLTP performance tests, client average and total elapsed is similar when using Transaction Guard due to the optimized code paths. The difference is below 0.04% Client Transaction Average Elapsed running high OLTP workloads. The benchmark used was AROLTP (eBusiness Suite) at 500 and 1000 concurrent sessions comparing Commit_OUTCOME False to True.

Transaction History Table

The transaction history table (LTXID_TRANS) is created by default in the SYSAUX tablespace at database creation and upgrade. New partitions are added when instances are added, using the storage of the last partition. SYSAUX typically uses automatic storage management (ASSM)

If the location of this tablespace is not optimal for performance, the DBA can move partitions to another tablespace. To move the history table, the alter table is executed for each partition.

For example:

```
alter TABLE LTXID_TRANS move partition LTXID_TRANS_4

tablespace FastPace

storage ( initial 100M  next 100M minextents 20 maxextents 121 );
```

Application Development using Transaction Guard

If you are using TAF, skip to Transaction Guard with Transparent Application Failover

To use Transaction Guard, several steps are followed –

- Trap the error that has made the user session unavailable. The most important step here is to ensure that the DBA has FAN configured. Nil application code is needed for FAN. FAN needs to be enabled so the application receives an error in real time, rather than hangs on TCP/IP timeouts. (12c Release 2 has FAN at the JDBC and OCI driver level enabled automatically using the recommended TNS for high availability, AQ_HA_NOTIFICATION set to TRUE and EVENTS in oraccess.xml for OCI.)
- If the plan is to resubmit following an uncommitted outcome, resubmission is more likely to succeed if the error is recoverable. There is no need to check the error class for commit outcome, as a commit can complete regardless of the error class that was returned to the client. For example, just because an osd error was received does not mean that the work did not commit.
- Obtain the LTXID from the previous failed session using the client driver provided APIs – getLogicalTransactionId for JDBC, LogicalTransactionId for ODP.NET, and OCI_ATTR_GET with LTXID for OCI and OCCI.
- Obtain a new session. This new session will have its own logical transaction ID.
- Invoke the GET_LTXID_OUTCOME PL/SQL procedure with the LTXID received from the API. The return state tells the driver if the last transaction COMMITTED (TRUE/FALSE) and USER_CALL_COMPLETED (TRUE/FALSE). This PL/SQL function returns an error if the client and database are out of sync (for example, not the same database or restored database).
- The application can return the result to the user to decide. Some applications may choose to replay themselves when the outcome is uncommitted. If the replay itself incurs an outage then the LTXID for the replaying session is used for the GET_LTXID_OUTCOME function.

Typical Transaction Guard Usage

12c client driver receives a FAN down event or error (and the transaction is not TP-managed with TMTWOPHASE commit FLAG).

FAN aborts the dead session

In the exception handling

Close the old session

Get last LTXID from dead session (see APIs below)

If the LTXID is null // Transaction Guard is disabled throw the original error

Else

// Transaction Guard is enabled

Obtain a new database session

Call DBMS_APP_CONT.GET_LTXID_OUTCOME with last LTXID to obtain COMMITTED and USER_CALL_COMPLETED status

If an error is returned obtaining the commit outcome throw the original error

Else

If COMMITTED AND USER_CALL_COMPLETED

Then the application can safely return committed

ELSEIF COMMITTED and NOT USER_CALL_COMPLETED

Then the application can return the committed result. However, if the application relies on details such as out binds or row count or DML with the returning clause that were not returned in commit, the application may not be able to continue. Most applications do not rely on result sets at commit. The completed status is provided for those applications that do.

ELSEIF NOT COMMITTED

If the original error is recoverable (OCI_ATTRIBUTE, isRecoverable for JDBC clients, the application can decide to resubmit or to return the uncommitted status to the user

Developer Step-by-Step using the LTXID

For replaying and returning results, the application or third party container needs access to the LTXID that is the next that will be committed at the server for each session. The LTXID can be obtained using APIs – getLogicalTransactionId for JDBC, LogicalTransactionId for ODP.NET, and OCI_ATTR_GET with LTXID for OCI from a failed session following a recoverable outage.

The JDBC-thin driver also provides a commit-outcome callback that executes on each LTXID change received from the server. The event gives you the new LTXID but it doesn't tell you if the previous committed or not. A third party container can use this callback to obtain the LTXID for use if the commit outcome is lost, for example to resubmit. http://adc2180604.us.oracle.com/JDBC_Javadoc/MAIN/latest/oracle/jdbc/OracleConnection.html#addLogicalTransactionIdEventListener-oracle.jdbc.LogicalTransactionIdEventListener

IMPORTANT Rules for Developers

If further outages occur in the process of resubmitting, the application MUST use the LAST LTXID in effect at the time of the failure with GET_LTXID_OUTCOME, not an earlier one, as seen in table 1. In the table, LTXID-A, LTXID-B, LTXID-C represent different LTXID on different sessions. Conversely, if further outages occur in the process of obtaining a commit outcome for a particular LTXID, you can keep asking for that outcome using the SAME LTXID.

TABLE 1. CONDITIONS AND ACTIONS FOR DEVELOPERS USING LTXID

Condition	**Application Action**	**Next LTXID to use** **Callback on LTXID Change – JDBC-Thin Driver only**
Application receives a recoverable error and calls GET_LTXID_OUTCOME to determine the transaction status	Application takes a new connection (with its own LTXID-B 0) and calls GET_LTXID_OUTCOME with the LTXID of the last failed session (LTXID-A)	New LTXID-B 0 Also sent via the JDBC callback when registered
Application finds that the last session transaction status is COMMITTED and USER_CALL_COMPLETED	Returns committed status to client, the application may be able to continue.	
Application finds that the last session transaction status is COMMITTED and NOT USER_CALL_COMPLETED	Returns committed status to client and exits – some applications cannot progress as the work in the call is not complete. (e.g. an outbind or row count was not returned). Whether the application can continue is application dependent.	
Application finds that the last session transaction status is NOT COMMITTED	Application returns the result to the user or cleans up if needed, and resubmits with the LTXID on the new session in effect, LTXID-B 0 If the new request executes any commits server returns commit messages with LTXID-B 2 and increasing…	New LTXID-B 2 .. N Also sent via the JDBC callback when registered
Application receives another recoverable error	Application takes a new connection (with LTXID-C 0) and calls GET_LTXID_OUTCOME with the LTXID of LAST session (LTXID-B N).	LTXID-C 0 on the new session. Also set via the JDBC callback when registered
Application receives another recoverable error during replay	Application takes a new connection (with LTXID-D 0) and calls GET_LTXID_OUTCOME again with the LTXID of LAST session (LTXID-C N).	LTXID-D 0 on the new session. Also set via the JDBC callback when registered

Transaction Guard with ODP.NET

1) The LTXID is not available once promoted to XA when using ODP.NET - both providers

2) ODP.NET in 12.2 handles Transaction Guard for the application whenever it is able to. When using ODP.NET the LTXID is only exposed to the application when ODP.NET is unable to obtain the commit outcome on behalf of the application. This might occur for example during an extended failover to Data Guard.

3) TAF and Application Continuity handle Transaction Guard for the application. There is no need to code for Transaction Guard.

Connection-Pool LTXID Usage

Connection pools create a different use case for managing LTXIDs because connections and sessions are pre-established and shared. The simplest model for connection pools and mid-tiers has an LTXID on each pooled session. An LTXID is associated with an application request at check-out from the connection pool, and is disassociated from the application request at check-in back to the pool. Between check-out and check-in, the LTXID on the session is exclusively held by that application request. After check-in, the LTXID belongs to an idle, pooled session. It is associated with the next application request that checks-out that connection.

This model allows

- Duplicate detection and failover for the present http request
- Basic replay by a third party container – replaying last request following a non-recoverable outages

Integration with Transparent Application Failover

Do not use Transaction Guard directly when TAF is enabled. TAF handles Transaction Guard on the developers' behalf. This behavior applies to both TAF BASIC and TAF SELECT modes. If Transparent Application Failover (TAF) is enabled or could be enabled, code similar to the example provided for "ODP.NET with TAF" should be used when developing for Transaction Guard with ODP.NET or OCI.

At failover, Transparent Application Failover (TAF) in 12c obtains a new connection and when Transaction Guard is enabled invokes Transaction Guard to force the commit outcome. If Transaction Guard returns committed and completed, TAF continues and the application sees no errors. If Transaction Guard returns uncommitted or committed but not completed, TAF returns a TAF error to the application. TAF maintains the new connection.

When TAF and Transaction Guard are both used, developers can use the TAF error codes (ORA-25402, ORA-25408, ORA-25405) to decide to safely resubmit transactions or to return a message indicating uncommitted to the user. It is ONLY safe to resubmit or to return uncommitted on these errors codes when BOTH TAF and Transaction Guard are enabled. It is not safe to resubmit or to return uncommitted if only TAF is enabled. The code sample for ODP.NET provides an example.

Resubmitting requires that the correct environment is established in the TAF callback, and that the entire transaction is rolled back and resubmitted. A boolean variable, as illustrated in the example below, that tracks if Transaction Guard is enabled. This variable MUST be evaluated in the TAF callback. Note that Application Continuity does all of this for the application plus more with no additional code requirement. Application Continuity does not require TAF, nor explicit coding for Transaction Guard or resubmitting lost work.

Note: TAF is not invoked on session failure (this includes 'kill -9' at operating system level, alter system kill session, and timeouts). TAF is invoked on an INSTANCE failure and on a FAN NODE DOWN event, and on shutdown transactional and disconnect POST_TRANSACTION.

Improved Commit Outcome for XA One Phase Optimizations

Transaction Processing Monitors (TPM) cannot determine the outcome of a commit operation when using the one phase optimization (TMONEPHASE flag), if the connection to the resource manager is lost or returns an ambiguous error. An improved outcome is available by using Transaction Guard 12c Release 2 and later with TPMs.

If the result from the one-phase commit operation is ambiguous or the connection is lost after issuing commit, TPMs can get a new connection to the Oracle resource and use GET_LTXID_OUTCOME to provide a definitive result.

If the commit has not been issued, the transaction has rolled back – so the TPM returns rollback.

If the commit has been issued and returned an ambiguous result, the TPM can use Transaction Guard to determine the commit outcome when the error is recoverable.

- If COMMITTED return COMMITTED.
- If UNCOMMITTED, borrow a new connection and reissue the COMMIT. The original LTXID is blocked by calling GET_LTXID_OUTCOME.

Example Transaction Managers using Transaction Guard

An application is using a Transaction Manager with an XA data source. For the XA data source, XA does not support local commits, DDL or DCL, or embedded COMMITS in PL/SQL, or remote COMMITs over remote procedure calls.

The application submits a servlet or EJB to the transaction manager to execute with one XA transaction. The transaction manager captures this servlet or EJB plus the initial state at the mid-tier in case a replay is required. The transaction manager starts executing this application code. No partial replays are allowed as state must be correct – any successful commit disables replay. If a recoverable error occurs, the transaction manager knows whether the error has occurred inside or outside commit processing and whether any commits have occurred:

1) No commit processing by the transaction manager has occurred and an XA data source.

Transaction manager restores the initial state, calls a callback to reset any application states, and resubmits the request. Any side-effects at either the mid-tier or database may be repeated

2) Commit processing by the transaction manager is in progress and the commit is two-phase

The transaction manager manages the two-phase commit processing. The transaction manager can repeat xa_prepare requests, if an ambiguous reply is received.

If the transaction manager has determined that all participants have prepared to commit, the transaction manager can repeat the xa_commit to a participant that fails to reply or replies with an ambiguous error. Likewise the transaction manager can repeat xa_rollback if rollback is the decided outcome and any participant returns a recoverable error from xa_rollback.

3) Commit processing by the transaction manager is in progress and the commit operation is one-phase.

One-phase commit processing is determined by the database. The TM requests that the commit is one-phase because the single branch optimization or read-only optimization apply. The TM passes the flag TMONEPHASE. If an ambiguous error is returned, the transaction manager obtains a new database session, and uses the LTXID obtained from the exact same session that was used for committing to obtain the reliable commit outcome based on that LTXID.

If Transaction Guard returns UNCOMMITTED, the transaction manager can resubmit.

If the Transaction Guard returns COMMITTED and COMPLETED (both), and commit was the last call for the servlet, the transaction manager can return this result.

If the Transaction Guard returns COMMITTED but not COMPLETED, and commit was the last call for the servlet, this can also be indicated to the application.

4) Commit processing by the transaction manager has completed successfully and there is more servlet code beyond.

Once a commit has executed, replay beyond this commit is not supported. The transaction manager does not replay committed transactions. The SERVLET or EJB has built states during the first part of execution that cannot be resurrected to continue beyond the first commit. Ordinarily commit is the last operation.

Additional Requirements for Transaction Guard Development

Transaction Guard is a tool to for developers to use that provides a reliable commit outcome following errors and timeouts. It is supported only when an error or timeout is returned indicating that the last session is dead. Transaction Guard API's should NOT be used in the following cases. Doing so will throw errors.

- DO NOT get the LTXID and hold it outside exception handling. That is, DO NOT obtain the LTXID and use it later.
- DO NOT use GET_LTXID_OUTCOME on the current session with the LTXID of the current session. It will return an error. The purpose of the LTXID is to find the outcome for dead sessions, not your own.
- DO NOT use GET_LTXID_OUTCOME against a session that did not receive a recoverable error. It will block that session from committing.
- DO NOT use GET_LTXID_OUTCOME with an LTXID from a different user or at a different database. It will return an error.
- DO NOT save LTXID from exception handling. GET_LTXID_OUTCOME is valid only for the last open or completed submission. If used with earlier transactions on the same session, it will return an error.
- DO NOT code Transaction Guard if the application is using TAF. Use the new TAF error codes to return the results instead.

Conclusion

Without using Transaction Guard, if a transaction has been started and commit has been issued, the commit message that is sent back to the client is not durable. The client is left not knowing whether the transaction committed or not. The transaction cannot be validly resubmitted if the non-transactional state is incorrect or if it already committed. In the absence of reliable commit and completion information, resubmission can lead to transactions applied more than once, out of order, or in the incorrect state.

Transaction Guard avoids the costs of ambiguous errors that lead to user frustration, customer support calls, and lost opportunities. Transaction Guard is safer and performs better with lower overheads than home-grown solutions for a known outcome.

JDBC Transaction Guard Code Sample

This use case is created by Jean de Lavarene.
https://blogs.oracle.com/dev2dev/entry/write_recovery_code_with_transaction
This simple request can return an ambiguous outcome when a recoverable error occurs. The commit embedded in this request can itself succeed while the return message is lost.

```
void giveRaiseToAllEmployees(Connection conn, int percentage) throws SQLException {
  Statement stmt = null;
  try {
    stmt = conn.createStatement();
    stmt.executeUpdate("UPDATE emp SET sal=sal+(sal*"+percentage+"/100)");
  } catch (SQLException sqle ) {
    throw sqle;
  }
  finally {
    if(stmt != null)
      stmt.close();
  }
  // At the end of the request we commit our changes:
  conn.commit();   // commit can succeed but the commit outcome is lost
}
```

Next add an exception block that retries when the error is unrecoverable and Transaction Guard reports that the commit did not succeed. In the example, if getTransactionOutcome returns true for the previous attempt then the Oracle Database guarantees that the previous transaction successfully committed and the application doesn't need to retry. Conversely, when getTransactionOutcome returns false, then the Oracle Database guarantees that the previous attempt didn't commit and will not commit. Hence it is safe to retry.

```
Connection jdbcConnection = getConnection();
boolean isJobDone = false;
while(!isJobDone) {
  try {
    // apply the raise (DML + commit):
    giveRaiseToAllEmployees(jdbcConnection,5);
    // no exception, the procedure completed:
    isJobDone = true;
  } catch (SQLRecoverableException recoverableException) {
    // Retry only if the error was recoverable.
    try {
      jdbcConnection.close();   // close old  connection:
    } catch (Exception ex) {} // pass through other exception s
    Connection newJDBCConnection = getConnection();   // reconnect to allow retry
    // Use Transacton Guard to force last request: committed or uncommitted
    LogicalTransactionId ltxid
      = ((OracleConnection)jdbcConnection).getLogicalTransactionId();
```

```
    isJobDone = getTransactionOutcome(newJDBCConnection, ltxid);
    jdbcConnection = newJDBCConnection;
  }
}

/**
 * GET_LTXID_OUTCOME_WRAPPER wraps DBMS_APP_CONT.GET_LTXID_OUTCOME
 */
private static final String GET_LTXID_OUTCOME_WRAPPER =
  "DECLARE PROCEDURE GET_LTXID_OUTCOME_WRAPPER("+
  "  ltxid IN RAW,"+
  "  is_committed OUT NUMBER ) "+
  "IS " +
  "  call_completed BOOLEAN; "+
  "  committed BOOLEAN; "+
  "BEGIN "+
  "  DBMS_APP_CONT.GET_LTXID_OUTCOME(ltxid, committed, call_completed); "+
  "  if committed then is_committed := 1;  else is_committed := 0; end if; "+
  "END; "+
  "BEGIN GET_LTXID_OUTCOME_WRAPPER(?,?); END;";


/**
 * getTransactionOutcome returns true if the LTXID committed or false otherwise.
 * note that this particular version is not considering user call completion
 */
boolean getTransactionOutcome(Connection conn, LogicalTransactionId ltxid)
    throws SQLException {
  boolean committed = false;
  CallableStatement cstmt = null;
  try {
    cstmt = conn.prepareCall(GET_LTXID_OUTCOME_WRAPPER);
    cstmt.setObject(1, ltxid); // use this starting in 12.1.0.2
    cstmt.registerOutParameter(2, OracleTypes.BIT);
    cstmt.execute();
    committed = cstmt.getBoolean(2);
  }
  catch (SQLException sqlexc) {
    throw sqlexc;
  }
  finally {
    if(cstmt != null)
      cstmt.close();
  }
  return committed;
}
```

ODP.NET Transaction Guard Code Sample when TAF is not used

```
======================== start ========================
using System;
using Oracle.DataAccess.Client; // For ODP.NET, Unmanaged Driver
// Or use “Oracle.ManagedDataAccess.Client;” for ODP.NET, Managed Driver

class TransactionGuardSample
{
   static void Main()
   {
      bool bReadyToCommit = false;
      string constr = "user id=hr;password=hr;data source=oracle";
      OracleConnection con = new OracleConnection(constr);
      OracleTransaction txn = null;
      OracleCommand cmd = null;

      try
      {
         string sql = " update employees set salary=10000 where employee_id=103";
         con.Open();
         txn = con.BeginTransaction();
         cmd = new OracleCommand(con, sql);
         cmd.ExecuteNonQuery();
         bReadyToCommit = true;
      }
      catch (Exception ex)
      {
         // rollback here as the SQL execution is unsuccessful
         txn.Rollback();
         Console.WriteLine(ex.ToString());
      }

      try  // progress to here as TX is successful
      {
         if (bReadyToCommit)
            txn.Commit();
      }
      catch (Exception ex)
      {

         if (ex is OracleException)
         {
//  It's safe to re-submit the work if the error is recoverable and the transaction has not been committed and
Transaction Guard is enabled
```

```
            if (ex.IsRecoverable && ex.OracleLogicalTransaction != null && !ex.OracleLogicalTransaction.Committed)
            {
                // safe to re-submit work
            }
            else
            {
                // do not re-submit work
            }
        }       finally
        {
            // dispose all objects
            txn.Dispose();
            cmd.Dispose();
            con.Dispose(); // place the connection back to the connection pool
        }
    }
}
```

ODP.NET Transaction Guard Code Sample when TAF is used

```
// NOTE: User requires execution access on SYS.DBMS_APP_CONT
// NOTE: The database service must be set with commit_outcome = true and FAILOVER_TYPE= BASIC or
SELECT
======================== start ========================
//  For the code sample, ODP.NET 12.1.0.2 or higher needs to be used.
using System;
using Oracle.DataAccess.Client;
// Or use “Oracle.ManagedDataAccess.Client;” for ODP.NET, Managed Driver available with ODAC 12c Release 4
or higher

class TransactionGuardSample
{
   static void Main()
   {
      bool bReadyToCommit = false;

      // This code is needed to test whether Transaction Guard is enabled.

      bool bTGEnabled = false;

      string constr = "user id=hr;password=hr;data source=oracle";
      OracleConnection con = new OracleConnection(constr);
      OracleTransaction txn = null;
      OracleCommand cmd = null;

      try
      {
         string sql = " update employees set salary=10000 where employee_id=103";
         con.Open();

            // This bTGEnabled boolean variable is needed to check whether TG is enabled.
            // Set this variable immediately after connecting and again in the TAF callback

            bTGEnabled = con.OracleLogicalTransaction != null;

         txn = con.BeginTransaction();
         cmd = new OracleCommand(con, sql);
         cmd.ExecuteNonQuery();
         bReadyToCommit = true;
      }
      catch (Exception ex)
      {
         // rollback here as the SQL execution is unsuccessful
         txn.Rollback();
```

```
        Console.WriteLine(ex.ToString());
      }

      try  // progress to here as TX is successful
      {
        if (bReadyToCommit)
          txn.Commit();
      }
      catch (Exception ex)
      {
        //ONLY handle the listed TAF errors and ONLY when Transaction Guard is enabled

          if (bTGEnabled &&  (ex.Number == 25402 || ex.Number == 25408 || ex.Number == 25405 ))
         {
            // application may cleanup, then rollback and re-submit the current transaction
          }
          else
          {
            // handle the error as before; do not resubmit
          }
      }
      finally
      {
        // dispose all objects
        txn.Dispose();
        cmd.Dispose();
        con.Dispose(); // place the connection back to the connection pool
      }
    }
}
```

OCCI/OCI Transaction Guard Code Sample when TAF is not enabled

The complete OCI demonstration is located on OTN. Look for tgdemo.c. The following code example is called in the error handling:

```
static void checkTransOutcome(
                        OCIEnv *envhp,
                        OCISvcCtx *svchp,
                        OCIError  *tmpErrhp,
                        OraText   *poolName,
                        ub4       poolNameLen,
                        boolean   *committed,
                        boolean   *callComplete)
{
  OCISession *embUsrhp;       /* session handle embedded in service context */
  ub1       *ltxidPtr;               /* LTXID from the embedded session */
  ub4        ltxidLen;                       /* length of the LTXID */

  OCISvcCtx  *newSvchp = (OCISvcCtx *)0;   /* used to call get_ltxid_outcome */
  OCIStmt    *getLtxidStm = (OCIStmt *)0;  /* used to call get_ltxid_outcome */
  OCIBind    *bnd1p, *bnd2p, *bnd3p;
  boolean     cmtd = FALSE, compl = FALSE;

  /* Get the LTXID from the session so we can call get_ltxid_outcome
   * to determine the transaction status. The LTXID parameter passed
   * to get_ltxid_outcome must correspond to the session that encountered
   * the error.
   */

  /* First get the session handle embedded in the caller's service context */
  (void) OCIAttrGet(svchp, OCI_HTYPE_SVCCTX,
                (dvoid *)&embUsrhp, (ub4 *)0,
                (ub4)OCI_ATTR_SESSION, tmpErrhp);

  /* Next get a pointer to that session's LTXID */
  (void) OCIAttrGet(embUsrhp, OCI_HTYPE_SESSION,
                (dvoid *)&ltxidPtr,(ub4 *)&ltxidLen,
                (ub4)OCI_ATTR_LTXID, tmpErrhp);

  /* The original session received a recoverable error, so it cannot be
   * used to call get_ltxid_out.  Instead, get a new session from the pool.
  if (checkerr(tmpErrhp, OCISessionGet(envhp, tmpErrhp, &newSvchp,
                              (OCIAuthInfo *)0,
                              (OraText *)poolName, (ub4)poolNameLen,
                              NULL, 0, NULL, NULL, NULL,
                              OCI_SESSGET_SPOOL)))
```

```
{
  printf("OCISessionGet failed to get a session from the pool.\n");
  goto done;
}

/* Parse the get_ltxid_id call */
if (checkerr(tmpErrhp,OCIStmtPrepare2(newSvchp,&getLtxidStm, tmpErrhp,
                                (CONST OraText *)getLtxid,
                                (ub4)sizeof(getLtxid),
                                (const oratext *)0, (ub4)0,
                                OCI_NTV_SYNTAX, OCI_DEFAULT)))
{
  printf("OCIStmtPrepare2 failed.\n");
  goto done;
}

/* get_ltxid_outcome takes 3 binds:
 *   1. LTXID from session that encountered the error
 *   2. committed - did the last call commit?
 *   3. completed - did the last call complete?
 */
if (checkerr(tmpErrhp, OCIBindByPos(getLtxidStm, &bnd1p, tmpErrhp, 1,
                                (dvoid *)ltxidPtr,
                                (sword)ltxidLen,
                                SQLT_BIN, (dvoid *)0,
                                (ub2 *) 0, (ub2 *) 0, (ub4) 0, (ub4 *) 0,
                                OCI_DEFAULT)) ||

    checkerr(tmpErrhp, OCIBindByPos(getLtxidStm, &bnd2p, tmpErrhp, 2,
                                (dvoid *) &cmtd,
                                sizeof(cmtd),
                                SQLT_BOL, (dvoid *)0,
                                (ub2 *) 0, (ub2 *) 0, (ub4) 0, (ub4 *) 0,

                                OCI_DEFAULT)) ||

    checkerr(tmpErrhp, OCIBindByPos(getLtxidStm, &bnd3p, tmpErrhp, 3,
                                (dvoid *) &compl,
                                (sword)sizeof(compl),
                                SQLT_BOL, (dvoid *)0,
                                (ub2 *) 0, (ub2 *) 0, (ub4) 0, (ub4 *) 0,
                                OCI_DEFAULT)))

{
  printf("Failed to bind variables for get_ltxid_outcome\n");
  goto done;
```

```
 }
 if (checkerr(tmpErrhp,OCIStmtExecute(newSvchp, getLtxidStm, tmpErrhp,
                        (ub4)1, (ub4)0,
                        (OCISnapshot *)0, (OCISnapshot *)0,
                        OCI_DEFAULT )))
 {
   printf("Failed to execute get_ltxid_outcome\n");
   goto done;
 }

done:

 /* Now determine what to do based on the status of the last call.
  *  1. Not Committed:
  *       The in-flight transaction did not COMMIT and will not COMMIT
  *       in the future.  It is safe to re-submit the last transaction.
  *  2. Committed and Not Complete:
  *       This can happen if, for example, the failure occurred while
  *       executing a PL/SQL procedure that contained a COMMIT.
  *       The caller will need to determine whether it is safe to
  *       continue.
  *  3. Committed and Complete:
  *       Do not report error; application can continue.
  */
 if (!cmtd)
 {                                          /* Case 1 */
   printf("Recoverable error occurred; transaction re-execution is safe\n");
 }
 else if (cmtd && !compl)
 {                                          /* Case 2 */
   printf("Warning: Transaction committed but call not complete\n");
 }
 /* Case 3 (committed and complete): Do not report any errors/warnings */
 /* free up the resources */
 if (newSvchp)
   (void) OCISessionRelease(newSvchp, tmpErrhp, NULL, 0, OCI_DEFAULT);
 if (getLtxidStm)
   (void) OCIStmtRelease((dvoid *) getLtxidStm, tmpErrhp,
                         (void *)0, 0, OCI_DEFAULT);
 /* Finally, update our OUT  values */
 *committed = cmtd;
 *callComplete = compl;
}
```

Appendix

Transaction Guard – Key Features

Transaction Guard is an integrated tool for applications to use to achieve idempotence automatically and transparently, with little effort, and in a manner that scales.

The key features of Transaction Guard are the following:

- Durability of COMMIT outcome for all supported transaction types against Oracle Database 12c. This includes transactions executed using auto-commit, transactions committed inside PL/SQL, distributed or remote transactions, one-phase XA transactions, and transactions issued across remote callouts that cannot otherwise be identified using generic means.
- Guaranteeing the commit outcome by blocking COMMIT of the earlier in-flight work to ensure that another submission of the same transaction protected by that LTXID cannot commit. This approach scales and eliminates hangs that occur with external approaches that rely on keys.
- Acceptance of repeated attempts to obtain the commit outcome for the last in-flight transaction on a session. All attempts receive the same result.
- Support for at-most-once execution semantics such that database transactions cannot be duplicated when there are multiple copies of that transaction in flight identified by the same LTXID.
- Identifying whether work committed was committed as part of a top-level call (client to server), or was embedded in a procedure such as PL/SQL at the server. An embedded commit state indicates that while a commit completed, the entire procedure in which the commit executed has not yet run to completion. Any work beyond the commit cannot guarantee to have been completed until that procedure itself returns to the database engine. For example, suppose that there are multiple commit operations in one round-trip (which by the way is very bad practice for OLTP applications), while a commit may have occurred, all the commits may not have occurred. In this example, COMMITTED is true and USER_CALL_COMPLETED is false.
- Identifying if the database to which the COMMIT resolution is directed is ahead of, in-sync, or behind the original user submission, and rejecting when there are gaps in the submission sequence of transactions from a client. It is deemed an error to attempt to obtain an outcome if the server or client is not in sync with the server as dependent transactions can be missing.
- A callback on the JDBC-thin client driver that fires when the LTXID is incremented by a call from the client to the server. This can be used by higher layer applications such as WebLogic Server and third party Java clients in order to maintain the current LTXID ready to use if needed.
- The service name is unique across databases that are consolidated into an Oracle Multitenant 12c infrastructure. For Multitenant, Transaction Guard operates at the pluggable database level. There is set of LTXID structures per tenant so they can be unplugged and relocated transparently.

Transaction Guard – Protocol

Understanding the Logical Transaction Identifier (LTXID)

The Logical Transaction ID (LTXID) is automatically assigned at session establishment. For scalability, the LTXID itself does not change when a database transaction is committed in that database session. The Transaction Guard protocol ensures that:

Execution of each logical transaction ID is unique.

Duplication is detected for all supported commit points during the retention period.

When obtaining the outcome, the LTXID is blocked to ensure that an earlier in-flight version of the transaction using that LTXID cannot commit.

LTXID is null when the database is not 12c or Transaction Guard is disabled

At-most-once execution

When using Transaction Guard, the LTXID is used to avoid duplicate transactions. The LTXID is modified on commit and is reused following a rollback. During normal runtime, a logical transaction id (LTXID) is automatically held in the session at both the client and server for each database transaction. At commit, the logical transaction ID is modified as part of committing the transaction.

The at-most-once protocol requires that the RDBMS maintains the LTXID for the retention period agreed for retry. The default retention period is 24 hours. It is imaginable that a customer could choose to extend this to several days or longer as needed. The longer the retention period, the longer the at-most-once check lasts that blocks an old transaction using the same LTXID from committing. The setting is available on each service and can be changed. When multiple physical copies of the database are involved, as is the case when using Data Guard or Active Data Guard or PDB unplug/plug (without clones), and PDB online relocate, the logical transaction ID is replicated to each database. Currently, it is not supported to use Transaction Guard across Golden Gate or Logical Standby sites or with other third party replication technology. It is supported to use Transaction Guard within each site.

The getLogicalTransactionId API provided for 12c Oracle JDBC-thin, and similar for OCI, OCCI and ODP.NET clients, allows an application the ability to retrieve the next logical transaction ID that will be used for the next database transaction on that session.

The DBMS_APP_CONT.GET_LTXID_OUTCOME PL/SQL procedure allows an application to determine the outcome of the last in-flight transaction using the current logical transaction ID. Calling GET_LTXID_OUTCOME may involve the database blocking the LTXID from committing so that the outcome is known. This is sometimes referred to as forcing the outcome. An application using Transaction Guard obtains the LTXID following an error or timeout. The application then calls GET_LTXID_OUTCOME before attempting to replay or to return the result to the user.

Oracle Corporation, World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065, USA

Worldwide Inquiries
Phone: +1.650.506.7000
Fax: +1.650.506.7200

ORACLE

CONNECT WITH US

blogs.oracle.com/oracle
facebook.com/oracle
twitter.com/oracle
oracle.com

Integrated Cloud Applications & Platform Services

Transaction Guard with Oracle Database 12cR2
August 2016
Author: Carol Colrain
Contributing Authors: Stefan Roesch, Jean de Lavarene, Kiminari Akiyama, Nancy Ikeda