34 Using Transaction Guard

Transaction Guard provides a generic tool for applications to use for at-most-once execution in case of planned and unplanned outages. Applications use the logical transaction ID to determine the commit outcome of the last transaction open in a database session following an outage. Without Transaction Guard, applications that attempt to replay operations following outages can cause logical corruption by committing duplicate transactions. Transaction Guard is used by Application Continuity for automatic and transparent transaction replay.

Transaction Guard provides these benefits:

- Preserves the returned outcome - committed or uncommitted so that it can be relied on
- Ensures a known commit outcome for every transaction
- Can be used to provide at-most-once transaction execution for applications that wish to resubmit themselves
- Is used by Application Continuity for automatic and transparent transaction replay

This chapter assumes that you are familiar with the major relevant concepts and techniques of the technology or product environment in which you are using Transaction Guard.

Topics:

See Also:

- *Oracle Database JDBC Developer's Guide* for more information about using Transaction Guard with Oracle Java Database Connectivity (JDBC)
- *Oracle Call Interface Programmer's Guide* for more information about using Transaction Guard with OCI

34.1 Problem That Transaction Guard Solves

In applications without Transaction Guard, a fundamental problem 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 if the submission executed any commit operations, if a procedural call completed and executed all expected commits and session state changes, or if a call failed part way through or, yet worse, is still running disconnected from the client.

Without Transaction Guard, it is impossible or extremely difficult to determine the outcome of the last commit operation, in a guaranteed and scalable manner, after a communication failure to the server. If an application must determine whether the submission to the database was committed, the application must 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 almost impractical because the query must be specific to each submission. After an application is built and is in production, this is completely impractical. Moreover, a query cannot give the correct answer because the transaction could commit immediately after that query executed. Indeed, after a communication failure the server may still be running the submission not yet aware that the client has disconnected. For PL/SQL or Java in the database, for a procedural submission, there is also no record as to whether that submission ran to completion or was canceled part way through. While such a procedure may have committed, subsequent work may not have been done for the procedure.

Failing to recognize that the last submission has committed, or will commit sometime soon or has not run to completion, can lead applications that attempt to replay, thus causing duplicate transaction submissions and other forms of "logical corruption" because the software might try to reissue already persisted changes.

Without 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. The transaction cannot be validly resubmitted if the nontransactional state is incorrect or if it already committed. In the absence of guaranteed commit and completion information, resubmission can lead to transactions applied more than once and in a session with the incorrect state.

34.2 Solution That Transaction Guard Provides

Effective with Oracle Database 12*c* Release 1 (12.1.0.1), Transaction Guard provides new, integrated tools for applications to use to achieve idempotence automatically and transparently, and in a manner that scales. Its key features are the following:

- Durability of `COMMIT` outcome by saving a logical transaction identifier (LTXID) at commit for all supported transaction types against the database (Oracle Database 12*c* Release 1 (12.1.0.1) or later). This includes idempotence for transactions executed using autocommit, from inside PL/SQL, from remote transactions, One-Phase XA transactions, and from callouts that cannot otherwise be identified using generic means.
- Use of the LTXID to support at-most-once execution semantics, such that database transactions protected by logical transaction identifiers cannot be duplicated when there are multiple copies of that transaction in flight identified by the LTXID.
- Blocking of a commit of in-flight work to ensure that regardless of the outage situation, another submission of the same transaction protected by that LTXID cannot commit.
- Identification of whether work committed at an LTXID 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 be guaranteed to have completed until that procedure itself returns to the database engine.
- Identification of whether the database to which the commit resolution is directed is ahead of, in sync with, or behind the original submission, and rejection when there are gaps in the

submission sequence of transactions from a client. It is considered an error to attempt to obtain an outcome if the server or client are not in sync on an LTXID sequence.

- A callback on the JDBC Thin client driver that fires when the LTXID changes. This can be used by higher layer applications such as WebLogic Server and third parties to maintain the current LTXID ready to use if needed.
- Namespace uniqueness across globally disparate databases and across databases that are consolidated into infrastructure. This includes Oracle Real Application Clusters (Oracle RAC) and RAC One, Data Guard.
- Service name uniqueness across global databases and across databases that are consolidated into a Multitenant infrastructure. This ensures that connections are properly directed to the transaction information.

34.3 Transaction Guard Concepts and Scope

This section explains some key concepts for Transaction Guard, and what Transaction Guard covers and does not cover.

Topics:

- Logical Transaction Identifier (LTXID)
- At-Most-Once Execution
- Transaction Guard Coverage
- Transaction Guard Exclusions

See Also:

- *Oracle Database Concepts* for more information about how Transaction Guard works
- *Oracle Database JDBC Developer's Guide* for more information about using Transaction Guard with Oracle Java Database Connectivity (JDBC)

34.3.1 Logical Transaction Identifier (LTXID)

Applications use a concept called the **logical transaction identifier (LTXID)** to determine the outcome of the last transaction open in a database session following an outage. The logical transaction ID is stored in the OCI session handle and in a connection object for the JDBC Thin and ODP.NET drivers. The logical transaction ID is the foundation of the at-most-once semantics.

The Transaction Guard protocol ensures that:

- Execution of each logical transaction is unique.
- Duplication is detected at supported commit time to ensure that for all commit points, the protocol must not be circumvented.
- When the transaction is committed, the logical transaction ID is persisted for the duration of the retention period for retries (default = 24 hours, maximum = 30 days).
- When obtaining the outcome, an LTXID is blocked to ensure that an earlier in-flight version of that LTXID cannot commit, by enforcing the uncommitted status. If the earlier version

with the same LTXID was already committed or forced, then blocking the LTXID returns the same result.

The logical session number is automatically assigned at session establishment. It is an opaque structure that cannot be read by an application. For scalability, each LTXID carries a running number called the commit number, which is increased when a database transaction is committed for each round trip to the database. This running commit number is zero-based.

34.3.2 At-Most-Once Execution

Transaction Guard uses the logical transaction identifier (LTXID) to avoid duplicate transactions. This ability to ensure at most one execution of a transaction is referred to as *transaction idempotence*. The LTXID is persisted on commit and is reused following a rollback. During normal runtime, an LTXID is automatically held in the session at both the client and server for each database transaction. At commit, the LTXID is persisted as part of committing the transaction.

The at-most-once protocol requires that the database maintain the LTXID for the retention period agreed for replay. The default retention period is 24 hours, although you might need a shorter or longer period, conceivably even a week or longer. The longer the retention period, the longer the at-most-once check blocks an old transaction using an old LTXID from replay. The setting is available on each service. When multiple databases are involved, as is the case when using Data Guard and Active Data Guard, the LTXID is replicated to each database involved through the use of redo.

The `getLTXID` API, provided for Oracle JDBC Thin (with similar APIs for OCI, OCCI, and ODP.NET clients), lets an application retrieve the logical transaction identifier that was in use on the terminated session. This is needed to determine the status of this last transaction.

The `DBMS_APP_CONT.GET_LTXID_OUTCOME` PL/SQL subprogram lets an application find the outcome of an action for a specified logical transaction identifier. Calling `DBMS_APP_CONT.GET_LTXID_OUTCOME` may involve the server blocking the LTXID from committing so that the outcome is known. This is a requirement if a transaction using that LTXID is in flight or is about to commit. An application using Transaction Guard obtains the LTXID following a recoverable error, and then calls `DBMS_APP_CONT.GET_LTXID_OUTCOME` before attempting a replay.

See Also:

Oracle Database PL/SQL Packages and Types Reference for more information about `DBMS_APP_CONT.GET_LTXID_OUTCOME` PL/SQL subprogram

34.3.3 Transaction Guard Coverage

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 with the following Oracle Database configurations:

- Single Instance Oracle RDBMS
- Real Application Clusters
- Data Guard

- Active Data Guard
- Multitenant including unplug/plug and relocates across the PDB/CDB, but excludes "with clone" option
- Global Data Services for the above database configurations

Transaction Guard supports the following transaction types against Oracle Database:

- Local transactions
- Data definition language (DDL) transactions
- Data control language (DCL) transactions
- Distributed transactions
- Remote transactions
- Parallel transactions
- Commit on success (auto-commit)
- PL/SQL with embedded commit-supported client drivers
- 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 :

- 12*c* JDBC type 4 driver
- 12*c* OCI and OCCI client drivers
- 12*c* Oracle Data Provider for .NET (ODP.NET), Unmanaged Driver
- 12*c* ODP.NET, Managed Driver in ODAC 12*c* Release 4 or higher

34.3.4 Transaction Guard with XA Transactions

Starting with Oracle Database 12.2 Release, Transaction Guard supports XA transactions to determine the outcome of one phase transactions. 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 the following XA transactions:

1. Local transactions that use `autocommit`
2. Local transactions that use an explicit `commit`
3. 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 monitors own the commit outcome for `TMTWOPHASE` transactions. This functionality allows TP monitors to return an unambiguous outcome for `TMONEPHASE` operations.

34.3.5 Transaction Guard Exclusions

Transaction Guard intentionally excludes recursive transactions and autonomous transactions so that they can be re-executed.

As of Oracle Database 12*c* Release 2, Transaction Guard also excludes:

- Two Phase XA transactions are managed externally. When using XA transactions, Transaction Guard maintains the commit outcome for one-phase XA transactions, and silently disables itself for externally-managed two-phase transactions because this outcome is owned by the TP monitor.
- Active Data Guard with read/write database links for forwarding transactions
- Golden Gate and Logical Standby for determining the outcome when failing across logical databases. Golden Gate and Logical Standby endpoints can use Transaction Guard
- 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.
- TAF and Application Continuity handle Transaction Guard internally. Do not code Transaction Guard in your application in the following places:
  - A failed return from TAF
  - TAF Callback for TAF or for Application Continuity for OCI and ODP.NET
  - JDBC initialization callback for Application Continuity for Java

Transaction Guard excludes failover across databases maintained by replication technology:

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

If you are using a database replica using any replication technology such as Golden Gate, or Logical Standby, or 3rd party replication, you may not use Transaction Guard between the primary and the secondary databases in this configuration.

You may use Transaction Guard on each database that participates in the replication. In this case, each database must use a different database unique identifier. Use V$DATABASE to obtain the DBID for each database.

34.4 Database Configuration for Transaction Guard

This section contains information relevant to configuring the database for using Transaction Guard.

Topics:

- Configuration Checklist
- Transaction History Table
- Service Parameters

34.4.1 Configuration Checklist

To use Transaction Guard with an application, you must do the following:

- Use Oracle Database 12*c* Release 1 (12.1.0.1) or later.
- Use an application service for all database work. Create the service using the `srvctl` command if you are using Oracle RAC, or using the `DBMS_SERVICE.CREATE_SERVICE` PL/SQL subprogram if you are not using Oracle RAC.

  Do ***not*** use the default database services, because these services are for administration purposes and cannot be manipulated. That is, do not use a service name that is set to *db_name* or *db_unique_name*.
- Grant permission on the `DBMS_APP_CONT` package to the database users who will call `GET_LTXID_OUTCOME`:

  ```
  GRANT EXECUTE ON DBMS_APP_CONT TO <user-name>;
  ```
- Increase `DDL_LOCK_TIMEOUT` if using Transaction Guard with DDL statements..

To use Transaction Guard with an application, Oracle recommends that you do the following:

- Locate and define the transaction history table for optimal performance.
- If you are using Oracle RAC or Oracle Data Guard, ensure that FAN is configured to communicate to interrupt clients fast on error.
- Set the following parameter: `AQ_HA_NOTIFICATIONS = TRUE` (if using OCI FAN).

See Also:

- *Oracle Database Reference* for more information about `DDL_LOCK_TIMEOUT`
- Transaction History Table

34.4.2 Transaction History Table

The transaction history table maintains the mapping of logical transaction identifiers (LTXIDs) to database transaction. This table can be accessed only by databases users with DBA privileges. It is maintained automatically by Oracle Database, and users must not issue DDL or DML statements directly against the 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. However, if the location of this tablespace is not optimal for performance, the DBA can move partitions to another tablespace. For example, the following statement alters the transaction history table to move it to a tablespace named `FastPace`:

```
ALTER TABLE LTXID_TRANS move partition LTXID_TRANS_4
 tablespace FastPace
 storage ( initial 10G next 10G
 minextents 1 maxextents 121 );
```

See Also:

- *Oracle Database SQL Language Reference* for information about the `ALTER TABLE` statement

34.4.3 Service Parameters

Configure the services for commit outcome and retention.

For example:

```
COMMIT_OUTCOME = TRUE
RETENTION_TIMEOUT = <retention-value>
```

`COMMIT_OUTCOME` determines whether transaction commit outcome is accessible after the commit has executed. This feature makes the outcome of the commit durable, and it is used by applications to enforce the status of the last transaction executed before an outage. The feature is used internally by the Oracle replay driver and by WebLogic Server, and it is available for use by other applications to determine an outcome. The `COMMIT_OUTCOME` possible values are `FALSE` (the default) and `TRUE`, and the value must be `TRUE` for Transaction Guard to be in effect.

The following considerations apply to `COMMIT_OUTCOME`:

- Using the `DBMS_APP_CONT.GET_LTXID_OUTCOME` procedure requires that `COMMIT_OUTCOME` be `TRUE`.
- `COMMIT_OUTCOME` has no effect on Active Data Guard and read-only databases. Using Transaction Guard with read/write Active Data Guard combined with database links that forward DMLs is not supported.
- `COMMIT_OUTCOME` is allowed on user-defined database services. Use on the database service is excluded because this service does not switch across Data Guard and cannot be started, stopped, or disabled for planned outages at the primary database.

`RETENTION_TIMEOUT` is used in conjunction with `COMMIT_OUTCOME` to set the amount of time that the commit outcome is retained. The retention timeout value is specified in seconds; the default is 86400 (24 hours), and the maximum is 2592000 (30 days). You can use the srvctl command or the `DBMS_SERVICE` PL/SQL package to specify the retention timeout value.

See Also:

- *Oracle Database Administrator's Guide* for information about the `srvctl add service` and `srvctl modify service` commands
- *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_SERVICE` package.
- *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_APP_CONT.GET_LTXID_OUTCOME` procedure

34.4.3.1 Example: Adding and Modifying a Service for a Server Pool

If you are using Oracle RAC or Oracle RAC One, then use the `srvctl` command to create and modify services.

Example 34-1 shows the use of `srvctl`. You can also use Global Data Services (GDSCTL).

Example 34-1 Adding and Modifying a Service for a Server Pool

```
srvctl add service -database orcl -service GOLD -poolname ora.Srvpool  -commit_outcome
TRUE  -retention 604800
srvctl modify service -database orcl -service GOLD -commit_outcome TRUE  -retention
604800
```

34.4.3.2 Example: Adding an Administrator-Managed Service

If you are using Oracle RAC or Oracle RAC One, then use the `srvctl` command to create and modify services.

Example 34-2 shows the use of `srvctl`. You can also use Global Data Services (GDSCTL)

Example 34-2 Adding an Administrator-Managed Service

```
srvctl add service -database codedb -service GOLD -preferred serv1 -available serv2 -
commit_outcome TRUE  -retention 604800
```

34.4.3.3 Example: Modifying a Service (PL/SQL)

If you are using a single-instance database, use the `DBMS_SERVICE.MODIFY_SERVICE` PL/SQL procedure to modify services and use FAN.

Example 34-3 modifies a service (but substitute the actual service name for `<service-name>`).

Example 34-3 Modifying a Service (PL/SQL)

```
DECLARE
  params dbms_service.svc_parameter_array;
BEGIN
  params('COMMIT_OUTCOME'):='true';
  params('RETENTION_TIMEOUT'):=604800;
  params('aq_ha_notifications'):='true';
  dbms_service.modify_service('<service-name>',params);
END;
/
```

34.5 Developing Applications That Use Transaction Guard

To use Transaction Guard, review the requirements and recommendations in Configuration Checklist, and follow these steps in the error handling when a recoverable error occurs:

Note:

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

1. Check that the error is a recoverable error that has made the database session unavailable.
2. Acquire the LTXID from the previous failed session using the client driver provided APIs (`getLTXID` for JDBC, `OCI_ATTR_GET` with LTXID for OCI, and `LogicalTransactionId` for ODP.NET).
3. Acquire a new session with that sessions' own LTXID.
4. Invoke the `DBMS_APP_CONT.GET_LTXID_OUTCOME` PL/SQL procedure with the LTXID obtained from the API. The return state tells the driver if the last transaction was `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).
5. The application can return the result to the user to decide. An application can replay itself. If the replay itself incurs an outage, then the LTXID for the replaying session is used for the `DBMS_APP_CONT.GET_LTXID_OUTCOME` procedure.

34.5.1 Typical Transaction Guard Usage

The following pseudocode shows a typical usage of Transaction Guard:

```
Receive a FAN down event (or recoverable error)

FAN cancels the terminated session

If recoverable error  (new OCI_ATTRIBUTE for OCI, isRecoverable for JDBC)
     Get last LTXID from terminated session using getLTXID or from your callback
     Obtain a new session
     Call GET_LTXID_OUTCOME with last LTXID to obtain COMMITTED and USER_CALL_COMPLETED
status

If COMMITTED and USER_CALL_COMPLETED
     Then return result

ELSEIF COMMITTED and NOT USER_CALL_COMPLETED
     Then return result with a warning (that details such as out binds or row count were
not returned)

ELSEIF NOT COMMITTED
     Cleanup and resubmit request, or return uncommitted result to the client
```

34.5.2 Details for Using the LTXID

For replay and returning results, the application or third party container needs access to the next LTXID to be committed at the server for each session. The LTXID can be obtained using APIs (`getLTXID` for JDBC and `OCI_ATTR_GET` with LTXID for OCI) from a failed session after a recoverable outage.

The JDBC Thin driver also provides a callback that executes on each commit number change received from the database. A third party container can use this callback to save the current LTXID in preparation to use if failover is needed. Within each session, the current LTXID is in use, so the callback can override earlier ones.

If failovers cascade without completing (that is, if during recovery from one failure, another failure occurs), the application ***must*** obtain and then pass the LTXID in effect on the current session into `GET_LTXID_OUTCOME`.

Table 34-1 shows several conditions or situations that require some LTXID-related action, and for each the application action and next LTXID to use.

Table 34-1 LTXID Condition or Situation, Application Actions, and Next LTXID to Use

Condition or Situation	Application Action	Next LTXID to Use (Callback on LTXID Change for Containers - JDBC Thin 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 set using 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.	(Not applicable)
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. (for example, an out bind or row count was not returned). Whether the application can continue is application dependent.	(Not applicable)
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 set using the JDBC callback when registered
Application receives a recoverable error if it has decided to replay.	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 using the JDBC callback when registered
Application receives another recoverable error if it has decided to 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 using the JDBC callback when registered

34.5.3 Transaction Guard and Transparent Application Failover

When Transparent Application Failover (TAF) is enabled with Transaction Guard, TAF handles the errors for developers. Do not code Transaction Guard when you are using TAF because it has embedded the Transaction Guard code starting with Oracle Database 12*c* Release 1

(12.1.0.1). When both TAF and Transaction Guard are used, developers can use the following TAF errors to rollback and safely resubmit, or return uncommitted.

- `ORA-25402`
- `ORA-25408`
- `ORA-25405`

Developers must not use `GET_LTXID_OUTCOME` procedure directly when TAF is enabled because TAF is already processing Transaction Guard.

Note:

TAF is not invoked on session failure (this includes “kill -9” at operating system level, or `ALTER SYSTEM KILL` session). TAF is invoked on the following conditions:

- `INSTANCE` failure
- `FAN NODE DOWN` event
- `SHUTDOWN` transactional
- Disconnect `POST_TRANSACTION`

34.5.4 Using Transaction Guard with ODP.NET

The following rules apply to using Transaction Guard with ODP.NET:

- The LTXID is not available after promoting to XA in both the ODP.NET providers.
- Starting with Oracle Database 12c Release 2 (12.2.0.1), ODP.NET handles Transaction Guard for application based on its availability and handling abilities. When using ODP.NET, the LTXID is exposed to the application only when ODP.NET is unable to obtain the commit outcome on behalf of the application. For example, during an extended failover to Data Guard.
- Developers must not code Transaction Guard in the TAF callback or JDBC initialization callback. Transaction Guard is handled for you.

34.5.5 Connection-Pool LTXID Usage

Connection pools create a different use case for managing LTXIDs because connections and sessions are preestablished and shared. In the simplest model for connection pools and middle tiers, an LTXID exists on each session handle (client-side session). It 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.

Using Transaction Guard in this way:

- Can support duplicate detection and failover for the present request
- Allows to cancel (real `Cancel` operation and not **Ctrl-C**) timed out requests, and optional re-submission by the application

34.5.6 Improved Commit Outcome for XA One Phase Optimizations

Starting with Oracle Database 12*c* Release 2 (12.2.0.1), Transaction Guard is used with Transaction Processing Monitors (TPM) to determine the outcome of a commit operation when using one-phase optimizations (TMONEPLHASE flag). The Transaction Guard uses the `GET_LTXID_OUTCOME` package to help the TPM to determine if the connection to the resource manager is lost or if an ambiguous error is returned.

Table 34-2 Transaction Manager Conditions/ Situations and Actions

Condition or Situation	Transaction Manager Action
`Commit` has not been issued, and if the transaction has rolled back.	Transaction Manager returns a `rollback`.
`Commit` has been issued and if an ambiguous result is returned.	Transaction Manager can use Transaction Guard (XA) to determine the outcome when the error is recoverable.
If the transaction is `COMMITTED`.	Transaction Manager returns `COMMITTED`.
If the transaction is `UNCOMMITTED`.	The Transaction Manager borrows a new connection and reissues the `COMMIT`. The original `LTXID` is blocked by calling `GET_LTXID_OUTCOME`.

34.5.7 Additional Requirements for Transaction Guard Development

Transaction Guard is a tool for developers to use after recoverable errors to provide a known outcome. It must be used when an error is returned indicating that the last session is terminated.

The Transaction Guard APIs must *not* be used in the following cases:

- Do not use `GET_LTXID_OUTCOME` on the current session. It will return an error.
- Do not use `GET_LTXID_OUTCOME` against a session that did not receive a recoverable error —that is, a live session. It will block that session from committing.
- Do not use `GET_LTXID_OUTCOME` from a different user or to a different database. It will return an error.
- Do not obtain the LTXID and save it for use later, as opposed to using it immediately. The result of `GET_LTXID_OUTCOME` is valid only for the last open or completed transaction. If it is used with an earlier transaction 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.

Note:

This rule does not apply to Application Continuity.

See Also:

Transaction Guard and Transparent Application Failover for more information about TAF

34.6 Transaction Guard and Its Relationship to Application Continuity

Transaction Guard provides a unique identifier (LTXID) for each database transaction. This identifier can be used to query the commit outcome of the transaction, and can also be used to ensure that the transaction is applied only once. Transaction Guard is used by Application Continuity and automatically enabled by it, but it can also be enabled independently. Transaction Guard prevents the transaction being replayed by Application Continuity from being applied more than once. If the application has implemented an application-level replay, then it requires the application to be integrated with transaction guard to provide idempotence.

For a solution that does not require coding, configure your application to use Application Continuity. For developing your own replay, the application developer codes using Transaction Guard. You can have an application coded for both Transaction Guard and Application Continuity. The Application Continuity takes effect first and the custom Transaction Guard code takes effect only when the Application Continuity is unable to replay. It is not required to use both, but, they are compatible if an application uses both Transaction Guard and Application Continuity. If an application wishes to add Transaction Guard API's in addition to Application Continuity, Transaction Guard can return the commit outcome when replay is disabled or unsuccessful.

See Also:

- *Oracle Real Application Clusters Administration and Deployment Guide* for information about Transaction Guard and Application Continuity with Oracle RAC
- *Oracle Database JDBC Developer's Guide* for information about connecting to the database with JDBC
- *Oracle Call Interface Programmer's Guide* for information about connecting to the database with Oracle Call Interface (OCI)
- *Oracle Data Provider for .NET Developer's Guide for Microsoft Windows* for more information about ODP.Net Driver

34.7 Transaction Guard Support during DBMS_ROLLING Operations

Transaction Guard ensures continuous application operation during the rolling upgrade operations performed on Oracle Database.

The rolling upgrade (`DBMS_ROLLING`) procedure is used when Oracle Database requires a major release upgrade. During the `DBMS_ROLLING` procedure, the upgrades are done on a logical standby while the primary database remains open for production.

34.7.1 Rolling Upgrade Using Transient Logical Standby

The transient logical process used in the rolling upgrade begins and ends with a physical standby database while temporarily being converted to a logical standby database for the upgrade. A Transient Logical Standby is used for major upgrades to ensure that user applications do not fail during the major upgrades.

A rolling upgrade using Transient Logical Standby:

- Temporarily converts an existing physical standby to a logical standby database for the duration of the upgrade
- Executes a rolling upgrade on the logical standby database to release 'n+1' while the production runs on the primary database at release 'n'
- Returns the logical standby back to its original status as a physical standby database after the upgrade is successful
- Resynchronizes the physical standby with the primary database using SQL apply
- Performs a switchover to transition the physical standby to the production role running on the new release. The physical standby becomes the new primary database
- Converts original primary database back to physical standby database and resynchronizes with the new primary, automatically completing the upgrade process
- Upgrades the physical standby using the redo stream

34.7.2 Transaction Guard Support During Major Database Version Upgrades

Starting with Oracle Database 23ai, Transaction Guard works during `DBMS_ROLLING` operations to ensure continuous application functions during switchover, issued by `DBMS_ROLLING` to Transient Logical Standby.

Transaction Guard returns the commit outcome of the current in-flight transaction when an error or outage occurs. Applications embed the Transaction Guard APIs in their error handling procedures to ensure that work continues without any in-flight work lost or duplicate submissions after an outage. Transaction Guard provides idempotence support to ensure that a commit occurs not more than once when a transaction is re-processed (replay) after an outage.

Transaction Guard ensures continuous application operation during the `DBMS_ROLLING` switchover operation to Transient Logical Standby. Transaction Guard ensures that the last commit outcome of transactions in the in-flight sessions during a switchover outage is used to protect the applications from duplicate submissions of the transactions on replay.

Transaction Guard maintains a transaction history table called `LTXID_TRANS` that has the mapping of logical transaction identifiers (`LTXID`s) to database transactions. For a failover to succeed after an outage, the changes to `LTXID_TRANS` from the primary database must first replicate and apply to Transient Logical Standby. With supplemental logging enabled for the `DBMS_ROLLING` procedure, Transaction Guard uses SQL to allow supplemental capture of `LTXID_TRANS` at CDB and PDB levels. The capture process replicates the `LTXID_TRANS` table and the apply process reads and recreates the `LTXID_TRANS` tables for the logical standby, along with the committed user transactions.

As a part of its support for the `DBMS_ROLLING` procedure, Transaction Guard performs the following functions:

- Tracks when the primary database is in `DBMS_ROLLING` mode (when the database upgrade is initiated)
- Checks that supplemental logging is in use
- Records the redo vector for the primary key (PK) at runtime while in supplemental logging mode
- Waits for all current updates to finish and replicate to the logical standby before performing the LTXID replication
- Replicates the `LTXID_TRANS` tables and applies the redo to Transient Logical Standby for each PDB
- Provides a mechanism for failover to know about successful LTXID replication
- Enforces last commit outcome for inflight sessions on replay after an outage
- Handles new users during supplemental capture and apply process to ensure that any apply does not create mismatched logged-in UIDs (user IDs) at the target database