Database Concepts

10 Transactions

This chapter defines a transaction and describes how the database processes transactions.

This chapter contains the following sections:

Introduction to Transactions

A **transaction** is a logical, atomic unit of work that contains one or more SQL statements.

A transaction groups SQL statements so that they are either all committed, which means they are applied to the database, or all rolled back, which means they are undone from the database. Oracle Database assigns every transaction a unique identifier called a **transaction ID** (glossary.htm#GUID-71C31D71-F2E8-4FB9-9010-062C1C407CEF) .

All Oracle transactions obey the basic properties of a database transaction, known as **ACID properties** (glossary.htm#GUID-2684886B-AB0E-4F65-9B24-688511AED791) . ACID is an acronym for the following:

- Atomicity

  All tasks of a transaction are performed or none of them are. There are no partial transactions. For example, if a transaction starts updating 100 rows, but the system fails after 20 updates, then the database rolls back the changes to these 20 rows.

- Consistency

  The transaction takes the database from one consistent state to another consistent state. For example, in a banking transaction that debits a savings account and credits a checking account, a failure must not cause the database to credit only one account, which would lead to inconsistent data.

- Isolation

The effect of a transaction is not visible to other transactions until the transaction is committed. For example, one user updating the `hr.employees` table does not see the uncommitted changes to `employees` made concurrently by another user. Thus, it appears to users as if transactions are executing serially.

- Durability

  Changes made by committed transactions are permanent. After a transaction completes, the database ensures through its recovery mechanisms that changes from the transaction are not lost.

The use of transactions is one of the most important ways that a database management system differs from a file system.

Sample Transaction: Account Debit and Credit

To illustrate the concept of a transaction, consider a banking database.

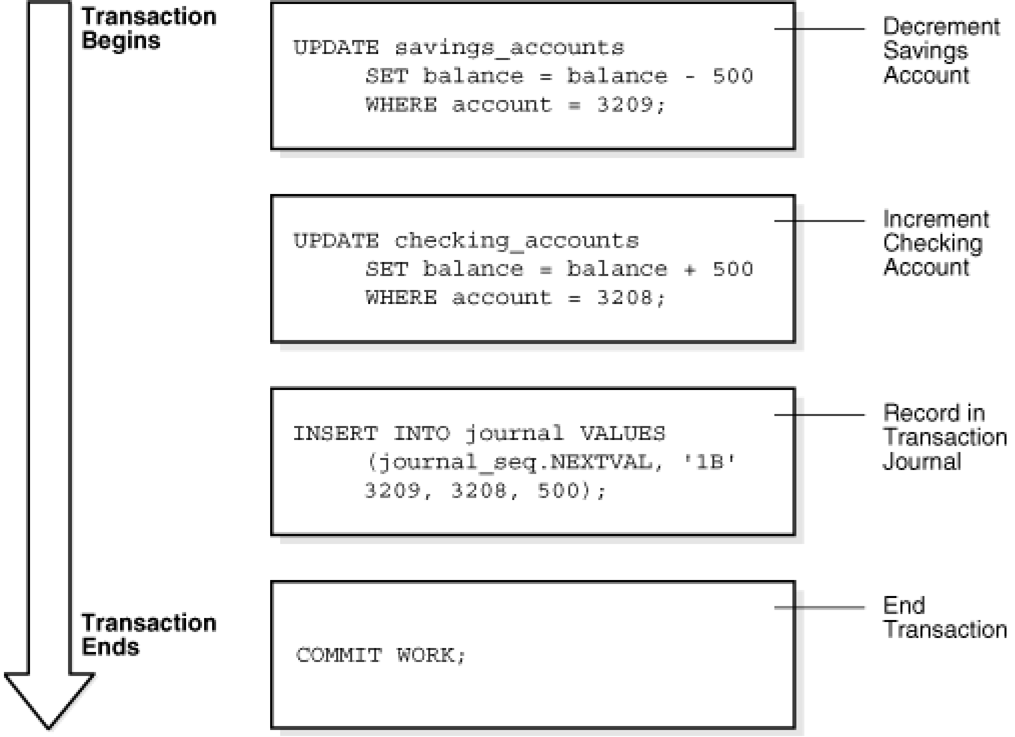
When a customer transfers money from a savings account to a checking account, the transaction must consist of three separate operations:

- Decrement the savings account
- Increment the checking account
- Record the transaction in the transaction journal

Oracle Database must allow for two situations. If all three SQL statements maintain the accounts in proper balance, then the effects of the transaction can be applied to the database. However, if a problem such as insufficient funds, invalid account number, or a hardware failure prevents one or two of the statements in the transaction from completing, then the database must roll back the entire transaction so that the balance of all accounts is correct.

The following graphic illustrates a banking transaction. The first statement subtracts $500 from savings account 3209. The second statement adds $500 to checking account 3208. The third statement inserts a record of the transfer into the journal table. The final statement commits the transaction.

Figure 10-1 A Banking Transaction


Transaction Begins
UPDATE savings_accounts SET balance = balance - 500 WHERE account = 3209;
Decrement Savings Account
UPDATE checking_accounts SET balance = balance + 500 WHERE account = 3208;
Increment Checking Account
INSERT INTO journal VALUES (journal_seq.NEXTVAL, '1B' 3209, 3208, 500);
Record in Transaction Journal
Transaction Ends
COMMIT WORK;
End Transaction

Description of "Figure 10-1 A Banking Transaction" (img_text/GUID-ABBF8589-449F-44BA-946B-858569B2ECB6-print.htm)

Structure of a Transaction

A database transaction consists of one or more statements. Specifically, a transaction consists of one of the following:

- One or more data manipulation language (DML) statements that together constitute an atomic change to the database
- One data definition language (DDL) statement

A transaction has a beginning and an end.

See Also:

"Overview of SQL Statements (sqllangu.htm#GUID-662EE4B0-7D5E-43F5-806D-A2AE404D77BF) "

Beginning of a Transaction

A transaction begins when the first executable SQL statement is encountered.

An **executable SQL statement** (glossary.htm#GUID-2F2F02A4-2405-4A0A-ABAF-91DB48DD4B82) is a SQL statement that generates calls to a **database instance** (glossary.htm#GUID-CFB1A30E-76B6-44EA-839E-9E63C8DA31AC) , including DML and DDL statements and the `SET TRANSACTION` statement.

When a transaction begins, Oracle Database assigns the transaction to an available **undo data** (glossary.htm#GUID-297B963A-989C-4720-B061-A2352FF72892) segment to record the undo entries for the new transaction. A transaction ID is not allocated until an undo segment and **transaction table** (glossary.htm#GUID-BFF71130-2760-4C64-8829-7772C803FBE7) slot are allocated, which occurs during the first DML statement. A transaction ID is unique to a transaction and represents the undo segment number, slot, and sequence number.

The following example execute an `UPDATE` statement to begin a transaction and queries `V$TRANSACTION` for details about the transaction:

```
SQL> UPDATE hr.employees SET salary=salary; 107 rows updated. SQL> SELECT XID AS "txn id",
XIDUSN AS "undo seg", XIDSLOT AS "slot", 2 XIDSQN AS "seq", STATUS AS "txn status" 3 FROM
V$TRANSACTION; txn id undo seg slot seq txn status ---------------- ---------- ---------- ----
------ ---------------- 0600060037000000 6 6 55 ACTIVE
```

See Also:

"Undo Segments (logical.htm#GUID-6E206D3A-E0E7-4B23-9C41-516FB35BC3FE) "

End of a Transaction

A transaction can end under different circumstances.

A transaction ends when any of the following actions occurs:

- A user issues a `COMMIT` or `ROLLBACK` statement *without* a `SAVEPOINT` clause.

  In a **commit** (glossary.htm#GUID-0680EB2C-ADF4-431A-A259-FB2227E5AA93) , a user explicitly or implicitly requested that the changes in the transaction be made permanent. Changes made by the transaction are permanent and visible to other users only after a transaction commits. The transaction shown in "Sample Transaction: Account Debit and Credit (transact.htm#GUID-A049FE81-8B67-4386-B599-9CDD7E6B6C59) " ends with a commit.

- A user runs a DDL command such as `CREATE`, `DROP`, `RENAME`, or `ALTER`.

  The database issues an implicit `COMMIT` statement before and after every DDL statement. If the current transaction contains DML statements, then Oracle Database first commits the transaction and then runs and commits the DDL statement as a new, single-statement transaction.

- A user exits normally from most Oracle Database utilities and tools, causing the current transaction to be implicitly committed. The commit behavior when a user disconnects is application-dependent and configurable.

  Note:

  Applications should always explicitly commit or undo transactions before program termination.

- A client process terminates abnormally, causing the transaction to be implicitly rolled back using metadata stored in the transaction table and the undo segment.

After one transaction ends, the next executable SQL statement automatically starts the following transaction. The following example executes an `UPDATE` to start a transaction, ends the transaction with a `ROLLBACK` statement, and then executes an `UPDATE` to start a new transaction (note that the transaction IDs are different):

```
SQL> UPDATE hr.employees SET salary=salary; 107 rows updated. SQL> SELECT XID, STATUS FROM
V$TRANSACTION; XID STATUS ---------------- ---------------- 0800090033000000 ACTIVE SQL>
ROLLBACK; Rollback complete. SQL> SELECT XID FROM V$TRANSACTION; no rows selected SQL> UPDATE
hr.employees SET last_name=last_name; 107 rows updated. SQL> SELECT XID, STATUS FROM
V$TRANSACTION; XID STATUS ---------------- ---------------- 0900050033000000 ACTIVE
```

See Also:

- "Tools for Database Administrators (cncptdba.htm#GUID-FA659979-25B7-4611-AA8D-48B5404301FE) " and "Tools for Database Developers (cncptdev.htm#GUID-D1C1BC1D-03C7-4A18-BB88-0D76F311DAF0) "
- *Oracle Database SQL Language Reference* (../SQLRF/statements_4011.htm#SQLRF01110) to learn about `COMMIT`

Statement-Level Atomicity

Oracle Database supports **statement-level atomicity**, which means that a SQL statement is an atomic unit of work and either completely succeeds or completely fails.

A successful statement is different from a committed transaction. A single SQL statement executes successfully if the database parses and runs it without error as an atomic unit, as when all rows are changed in a multirow update.

If a SQL statement causes an error during execution, then it is not successful and so all effects of the statement are rolled back. This operation is a **statement-level rollback** (glossary.htm#GUID-187EACC0-1FDD-498A-BEAD-892AC03B50D0) . This operation has the following characteristics:

- A SQL statement that does not succeed causes the loss only of work it would have performed itself.

  The unsuccessful statement does not cause the loss of any work that preceded it in the current transaction. For example, if the execution of the second `UPDATE` statement in "Sample Transaction: Account Debit and Credit (transact.htm#GUID-A049FE81-8B67-4386-B599-9CDD7E6B6C59) " causes an error and is rolled back, then the work performed by the first `UPDATE` statement is not rolled back. The first `UPDATE` statement can be committed or rolled back explicitly by the user.

- The effect of the rollback is as if the statement had never been run.

  Any side effects of an atomic statement, for example, triggers invoked upon execution of the statement, are considered part of the atomic statement. Either all work generated as part of the atomic statement succeeds or none does.

An example of an error causing a statement-level rollback is an attempt to insert a duplicate **primary key** (glossary.htm#GUID-8640EFA5-276C-4812-A078-1F21F55F4200) . Single SQL statements involved in a **deadlock** (glossary.htm#GUID-E33D1853-7F99-4FDD-9CC7-D6308E943D61) , which is competition for the same data, can also cause a statement-level rollback. However, errors discovered during SQL statement parsing, such as a syntax error, have not yet been run and so do not cause a statement-level rollback.

See Also:

- "SQL Parsing (sqllangu.htm#GUID-B3F2B5B8-B679-4A7C-B1E8-286F36319FCB) "
- "Locks and Deadlocks (consist.htm#GUID-C1971E9B-849A-4634-9575-4F8FAD697750) "
- "Overview of Triggers (srvrside.htm#GUID-40297ADF-0968-42F8-B8B9-84AD6ADCBE63) "

System Change Numbers (SCNs)

A **system change number (SCN)** (glossary.htm#GUID-B421FB8D-EA59-4CC3-B9D7-8B149BDC2D5C) is a logical, internal time stamp used by Oracle Database. SCNs order events that occur within the database, which is necessary to satisfy the ACID properties of a transaction. Oracle Database uses SCNs to mark the SCN before which all changes are known to be on disk so that recovery avoids applying unnecessary redo. The database also uses SCNs to mark the point at which no redo exists for a set of data so that recovery can stop.

SCNs occur in a monotonically increasing sequence. Oracle Database can use an SCN like a clock because an observed SCN indicates a logical point in time, and repeated observations return equal or greater values. If one event has a lower SCN than another event, then it occurred at an earlier time in the database. Several events may share the same SCN, which means that they occurred at the same time in the database.

Every transaction has an SCN. For example, if a transaction updates a row, then the database records the SCN at which this update occurred. Other modifications in this transaction have the same SCN. When a transaction commits, the database records an SCN for this commit.

Oracle Database increments SCNs in the **system global area (SGA)** (glossary.htm#GUID-78C0E867-233A-4857-B9FE-A3852A9B7BDF) . When a transaction modifies data, the database writes a new SCN to the undo data segment assigned to the transaction. The log writer process then writes the commit record of the transaction immediately to the **online redo log** (glossary.htm#GUID-2A8BC112-AB70-4B06-9F85-FE975861CEE0) . The commit record has the unique SCN of the transaction. Oracle Database also uses SCNs as part of its **instance recovery** (glossary.htm#GUID-FFDBC27D-CE99-49C7-8BB9-C8C2D8D52801) and **media recovery** (glossary.htm#GUID-938A3E7D-CC65-496C-9DB3-3CFC45AFA8DD) mechanisms.

See Also:

"Overview of Instance Recovery (startup.htm#GUID-728C6BE1-5687-4DC5-B570-D2042C88F935) " and "Backup and Recovery (cncptdba.htm#GUID-DCE361D9-B486-43B4-B4FD-5722A93203F0) "

Overview of Transaction Control

Transaction control is the management of changes made by DML statements and the grouping of DML statements into transactions.

In general, application designers are concerned with transaction control so that work is accomplished in logical units and data is kept consistent.

Transaction control involves using the following statements, as described in "Transaction Control Statements (sqllangu.htm#GUID-73F24816-881A-4849-B8A7-EA9B446A24A7) ":

- The `COMMIT` statement ends the current transaction and makes all changes performed in the transaction permanent. `COMMIT` also erases all savepoints in the transaction and releases transaction locks.
- The `ROLLBACK` statement reverses the work done in the current transaction; it causes all data changes since the last `COMMIT` or `ROLLBACK` to be discarded. The `ROLLBACK TO SAVEPOINT` statement undoes the changes since the last savepoint but does not end the entire transaction.
- The `SAVEPOINT` statement identifies a point in a transaction to which you can later roll back.

The session in Table 10-1 (transact.htm#GUID-5BB15405-8A03-47DE-8A20-63E1B83E1361__CHDCIJGA) illustrates the basic concepts of transaction control.

Table 10-1 Transaction Control

T	Session	Explanation
t0	`COMMIT;`	This statement ends any existing transaction in the session.

T	Session	Explanation
t1	`SET TRANSACTION NAME 'sal_update';`	This statement begins a transaction and names it `sal_update`.
t2	`UPDATE employees SET salary = 7000 WHERE last_name = 'Banda';`	This statement updates the salary for Banda to 7000.
t3	`SAVEPOINT after_banda_sal;`	This statement creates a savepoint named `after_banda_sal`, enabling changes in this transaction to be rolled back to this point.
t4	`UPDATE employees SET salary = 12000 WHERE last_name = 'Greene';`	This statement updates the salary for Greene to 12000.
t5	`SAVEPOINT after_greene_sal;`	This statement creates a savepoint named `after_greene_sal`, enabling changes in this transaction to be rolled back to this point.
t6	`ROLLBACK TO SAVEPOINT after_banda_sal;`	This statement rolls back the transaction to t3, undoing the update to Greene's salary at t4. The `sal_update` transaction has *not* ended.
t7	`UPDATE employees SET salary = 11000 WHERE last_name = 'Greene';`	This statement updates the salary for Greene to 11000 in transaction `sal_update`.
t8	`ROLLBACK;`	This statement rolls back all changes in transaction `sal_update`, ending the transaction.
t9	`SET TRANSACTION NAME 'sal_update2';`	This statement begins a new transaction in the session and names it `sal_update2`.
t10	`UPDATE employees SET salary = 7050 WHERE last_name = 'Banda';`	This statement updates the salary for Banda to 7050.
t11	`UPDATE employees SET salary = 10950 WHERE last_name = 'Greene';`	This statement updates the salary for Greene to 10950.

T	Session	Explanation
t1 2	`COMMIT;`	This statement commits all changes made in transaction `sal_update2`, ending the transaction. The commit guarantees that the changes are saved in the online redo log files.

See Also:

Oracle Database SQL Language Reference (../SQLRF/statements_1001.htm#SQLRF30043) to learn about transaction control statements

Transaction Names

A transaction name is an optional, user-specified tag that serves as a reminder of the work that the transaction is performing. You name a transaction with the `SET TRANSACTION ... NAME` statement, which if used must be first statement of the transaction.

In Table 10-1 (transact.htm#GUID-5BB15405-8A03-47DE-8A20-63E1B83E1361__CHDCIJGA) , the first transaction was named `sal_update` and the second was named `sal_update2`.

Transaction names provide the following advantages:

- It is easier to monitor long-running transactions and to resolve in-doubt distributed transactions.
- You can view transaction names along with transaction IDs in applications. For example, a database administrator can view transaction names in Oracle Enterprise Manager (Enterprise Manager) when monitoring system activity.
- The database writes transaction names to the transaction auditing redo record, so you can use LogMiner to search for a specific transaction in the redo log.
- You can use transaction names to find a specific transaction in data dictionary views such as `V$TRANSACTION`.

See Also:

- "Oracle Enterprise Manager (cncptdba.htm#GUID-BE08A8FA-501A-418E-AD32-3E42AE762026) "
- *Oracle Database Reference* (../REFRN/GUID-FAE908F8-24B1-4B90-8FC5-86FCB532431C.htm#REFRN30291) to learn about `V$TRANSACTION`
- *Oracle Database SQL Language Reference* (../SQLRF/statements_10005.htm#SQLRF01705) to learn about `SET TRANSACTION`

Active Transactions

An **active transaction** is one that has started but not yet committed or rolled back.

In Table 10-1 (transact.htm#GUID-5BB15405-8A03-47DE-8A20-63E1B83E1361__CHDCIJGA) , the first statement to modify data in the `sal_update` transaction is the update to Banda's salary. From the successful execution of this update until the `ROLLBACK` statement ends the transaction, the `sal_update` transaction is active.

Data changes made by a transaction are temporary until the transaction is committed or rolled back. Before the transaction ends, the state of the data is as follows:

- Oracle Database has generated undo information in the SGA.

  The undo data contains the old data values changed by the SQL statements of the transaction. See "Read Consistency in the Read Committed Isolation Level (consist.htm#GUID-432C4EDC-F7F2-467C-B85F-4E82D3D58A2F) ".

- Oracle Database has generated redo in the online redo log buffer of the SGA.

  The redo log record contains the change to the data block and the change to the undo block. See "Redo Log Buffer (memory.htm#GUID-C2AD1BF6-A5AE-42E9-9677-0AA08126864B) ".

- Changes have been made to the database buffers of the SGA.

  The data changes for a committed transaction, stored in the database buffers of the SGA, are not necessarily written immediately to the data files by the **database writer (DBW)** (glossary.htm#GUID-C27AAA54-E60B-49BC-AB04-7B3848EBAFD6) . The disk write can happen before or after the commit. See "Database Buffer Cache (memory.htm#GUID-4FF66585-E469-4631-9225-29D75594CD14) ".

- The rows affected by the data change are locked.

  Other users cannot change the data in the affected rows, nor can they see the uncommitted changes. See "Summary of Locking Behavior (consist.htm#GUID-1D60EFCC-03F4-4A04-B099-1B4DE5D02C47) ".

Savepoints

A **savepoint** is a user-declared intermediate marker within the context of a transaction.

Internally, the savepoint marker resolves to an SCN. Savepoints divide a long transaction into smaller parts.

If you use savepoints in a long transaction, then you have the option later of rolling back work performed before the current point in the transaction but after a declared savepoint within the transaction. Thus, if you make an error, you do not need to resubmit every statement. Table 10-1 (transact.htm#GUID-5BB15405-8A03-47DE-8A20-63E1B83E1361__CHDCIJGA) creates savepoint `after_banda_sal` so that the update to the Greene salary can be rolled back to this savepoint.

Rollback to Savepoint

A rollback to a savepoint in an uncommitted transaction means undoing any changes made after the specified savepoint, but it does not mean a rollback of the transaction itself.

When a transaction is rolled back to a savepoint, as when the `ROLLBACK TO SAVEPOINT after_banda_sal` is run in Table 10-1 (transact.htm#GUID-5BB15405-8A03-47DE-8A20-63E1B83E1361__CHDCIJGA) , the following occurs:

1. Oracle Database rolls back only the statements run after the savepoint.

   In Table 10-1 (transact.htm#GUID-5BB15405-8A03-47DE-8A20-63E1B83E1361__CHDCIJGA) , the `ROLLBACK TO SAVEPOINT` causes the `UPDATE` for Greene to be rolled back, but not the `UPDATE` for Banda.

2. Oracle Database preserves the savepoint specified in the `ROLLBACK TO SAVEPOINT` statement, but all subsequent savepoints are lost.

   In Table 10-1 (transact.htm#GUID-5BB15405-8A03-47DE-8A20-63E1B83E1361__CHDCIJGA) , the `ROLLBACK TO SAVEPOINT` causes the `after_greene_sal` savepoint to be lost.

3. Oracle Database releases all table and row locks acquired after the specified savepoint but retains all data locks acquired before the savepoint.

The transaction remains active and can be continued.

See Also:

- *Oracle Database SQL Language Reference* (../SQLRF/statements_9023.htm#SQLRF01610) to learn about the `ROLLBACK` and `SAVEPOINT` statements
- *Oracle Database PL/SQL Language Reference* (../LNPLS/static.htm#LNPLS00608) to learn about transaction processing and control

Enqueued Transactions

Depending on the scenario, transactions waiting for previously locked resources may still be blocked after a rollback to savepoint.

When a transaction is blocked by another transaction it enqueues on the blocking transaction itself, so that the entire blocking transaction must commit or roll back for the blocked transaction to continue.

In the scenario shown in the following table, session 1 rolls back to a savepoint created before it executed a DML statement. However, session 2 is still blocked because it is waiting for the session 1 transaction to complete.

Table 10-2 Rollback to Savepoint Example

T	Session 1	Session 2	Session 3	Explanation
t0	`UPDATE employees SET salary=7000 WHERE last_name= 'Banda';`			Session 1 begins a transaction. The session places an exclusive lock on the `Banda` row (TX) and a subexclusive table lock (SX) on the table.
t1	`SAVEPOINT after_banda_sal;`			Session 1 creates a savepoint named `after_banda_sal`.

T	Session 1	Session 2	Session 3	Explanation
t2	`UPDATE employees SET salary=12000 WHERE last_name= 'Greene';`			Session 1 locks the `Greene` row.
t3		`UPDATE employees SET salary=14000 WHERE last_name= 'Greene';`		Session 2 attempts to update the `Greene` row, but fails to acquire a lock because session 1 has a lock on this row. No transaction has begun in session 2.
t4	`ROLLBACK TO SAVEPOINT after_banda_sal;`			Session 1 rolls back the update to the salary for `Greene`, which releases the row lock for `Greene`. The table lock acquired at t0 is not released. At this point, session 2 is *still* blocked by session 1 because session 2 enqueues on the session 1 *transaction*, which has not yet completed.
t5			`UPDATE employees SET salary=11000 WHERE last_name= 'Greene';`	The `Greene` row is currently unlocked, so session 3 acquires a lock for an update to the `Greene` row. This statement begins a transaction in session 3.

T	Session 1	Session 2	Session 3	Explanation
t6	`COMMIT;`			Session 1 commits, ending its transaction. Session 2 is now enqueued for its update to the `Greene` row behind the transaction in session 3.

See Also:

"Lock Duration (consist.htm#GUID-DB33AD91-8A4E-44CE-A6B2-654909143E5F) " to learn more about when Oracle Database releases locks

Rollback of Transactions

A rollback of an uncommitted transaction undoes any changes to data that have been performed by SQL statements within the transaction. After a transaction has been rolled back, the effects of the work done in the transaction no longer exist.

In rolling back an entire transaction, without referencing any savepoints, Oracle Database performs the following actions:

- Undoes all changes made by all the SQL statements in the transaction by using the corresponding undo segments

  The transaction table entry for every active transaction contains a pointer to all the undo data (in reverse order of application) for the transaction. The database reads the data from the undo segment, reverses the operation, and then marks the undo entry as applied. Thus, if a transaction inserts a row, then a rollback deletes it. If a transaction updates a row, then a rollback reverses the update. If a transaction deletes a row, then a rollback reinserts it. In Table 10-1 (transact.htm#GUID-5BB15405-8A03-47DE-8A20-63E1B83E1361__CHDCIJGA) , the `ROLLBACK` reverses the updates to the salaries of Greene and Banda.

- Releases all the locks of data held by the transaction

- Erases all savepoints in the transaction

  In Table 10-1 (transact.htm#GUID-5BB15405-8A03-47DE-8A20-63E1B83E1361__CHDCIJGA) , the `ROLLBACK` deletes the savepoint `after_banda_sal`. The `after_greene_sal` savepoint was removed by the `ROLLBACK TO SAVEPOINT` statement.

- Ends the transaction

  In Table 10-1 (transact.htm#GUID-5BB15405-8A03-47DE-8A20-63E1B83E1361__CHDCIJGA) , the `ROLLBACK` leaves the database in the same state as it was after the initial `COMMIT` was executed.

The duration of a rollback is a function of the amount of data modified.

See Also:

"Undo Segments (logical.htm#GUID-6E206D3A-E0E7-4B23-9C41-516FB35BC3FE) "

Commits of Transactions

A commit ends the current transaction and makes permanent all changes performed in the transaction. In Table 10-1 (transact.htm#GUID-5BB15405-8A03-47DE-8A20-63E1B83E1361__CHDCIJGA) , a second transaction begins with `sal_update2` and ends with an explicit `COMMIT` statement. The changes that resulted from the two `UPDATE` statements are now made permanent.

When a transaction commits, the following actions occur:

- The database generates an SCN for the `COMMIT`.

  The internal transaction table for the associated **undo tablespace** (glossary.htm#GUID-78A7FBF2-2EB5-4BD6-AECC-D61A5AEF1158) records that the transaction has committed. The corresponding unique SCN of the transaction is assigned and recorded in the transaction table. See "Serializable Isolation Level (consist.htm#GUID-8DA9A191-4CA3-4B1A-995F-4B17471C2738) ".

- The **log writer process (LGWR)** (glossary.htm#GUID-E96BC851-0B78-4250-8EAB-26EBDF4FE5A6) process writes remaining redo log entries in the redo log buffers to the online redo log and writes the transaction SCN to the online redo log. *This atomic event constitutes the commit of the transaction.*

- Oracle Database releases locks held on rows and tables.

  Users who were enqueued waiting on locks held by the uncommitted transaction are allowed to proceed with their work.

- Oracle Database deletes savepoints.

  In Table 10-1 (transact.htm#GUID-5BB15405-8A03-47DE-8A20-63E1B83E1361__CHDCIJGA) , no savepoints existed in the `sal_update` transaction so no savepoints were erased.

- Oracle Database performs a **commit cleanout** (glossary.htm#GUID-5E20E19B-FC72-40E3-A129-306B52DC9607) .

  If modified blocks containing data from the committed transaction are still in the SGA, and if no other session is modifying them, then the database removes lock-related transaction information (the ITL entry) from the blocks.

  Ideally, the `COMMIT` cleans the blocks so that a subsequent `SELECT` does not have to perform this task. If no ITL entry exists for a specific row, then it is not locked. If an ITL entry exists for a specific row, then it is possibly locked, so a session must check the undo segment header to determine whether this interested transaction has committed. If the interested transaction has committed, then the session cleans out the block, which generates redo. However, if the `COMMIT` cleaned out the ITL previously, then the check and cleanout are unnecessary.

  > **Note:**
  >
  > Because a block cleanout generates redo, a query may generate redo and thus cause blocks to be written during the next **checkpoint** (glossary.htm#GUID-95DBDA37-4C57-444F-B660-D52B4A99D919) .

- Oracle Database marks the transaction complete.

After a transaction commits, users can view the changes.

Typically, a commit is a fast operation, regardless of the transaction size. The speed of a commit does not increase with the size of the data modified in the transaction. The lengthiest part of the commit is the physical disk I/O performed by LGWR. However, the amount of time spent by LGWR is reduced because it has been incrementally writing the contents of the redo log buffer in the background.

The default behavior is for LGWR to write redo to the online redo log synchronously and for transactions to wait for the buffered redo to be on disk before returning a commit to the user. However, for lower transaction commit latency, application developers can specify that redo be written asynchronously so that transactions need not wait for the redo to be on disk and can return from the `COMMIT` call immediately.

See Also:

- "Locking Mechanisms (consist.htm#GUID-AD960556-7F7B-4242-8B91-6DA22AABA27D) "
- "Overview of Background Processes (process.htm#GUID-D8AE1B78-69D5-4F0F-8BE3-C91AA2514F2D) " for more information about LGWR
- *Oracle Database PL/SQL Language Reference* (../LNPLS/static.htm#LNPLS592) for more information on asynchronous commit

Overview of Transaction Guard

Transaction Guard (glossary.htm#GUID-FA547925-576A-4FD8-BA44-AED3AE438856) is an API that applications can use to provide **transaction idempotence** (glossary.htm#GUID-09AF097B-326A-454D-8125-43BA717DAC0E) , which is the ability of the database to preserve a guaranteed commit outcome that indicates whether a transaction committed and completed. Oracle Database provides the API for JDBC thin, OCI, OCCI, and ODP.Net.

A **recoverable error** (glossary.htm#GUID-2F7DE61F-BD7C-4A89-91E6-CE0D193E8294) is caused by an external system failure, independent of the application session logic that is executing. Recoverable errors occur following planned and unplanned outages of foreground processes, networks, nodes, storage, and databases. If an outage breaks the connection between a client application and the database, then the application receives a disconnection error message. The transaction that was running when the connection broke is called an **in-flight transaction** (glossary.htm#GUID-4A49C543-CD3B-4F01-BF01-21F15CEABEE1) .

To decide whether to resubmit the transaction or to return the result (committed or uncommited) to the client, the application must determine the outcome of the in-flight transaction. Before Oracle Database 12*c*, commit messages returned to the client were not persistent. Checking a transaction was no guarantee that it would not commit after being checked, permitting duplicate transactions and other forms of logical corruption. For example, a user might refresh a web browser when purchasing a book online and be charged twice for the same book.

See Also:

- "Introduction to Transactions (transact.htm#GUID-31319EA7-994C-4D25-8814-0214ABD3CBDA) "
- *Oracle Database Development Guide* (../ADFNS/adfns_trans_idemp_guard.htm#ADFNS1062) to learn about Transaction Guard
- *Oracle Real Application Clusters Administration and Deployment Guide* (../RACAD/GUID-097A5067-8CCF-4FF0-B97B-BF109DFA1841.htm#RACAD8423) to learn how to configure services for Transaction Guard

Benefits of Transaction Guard

Starting in Oracle Database 12*c*, Transaction Guard provides applications with a tool for determining the status of an in-flight transaction following a recoverable outage. Using Transaction Guard, an application can ensure that a transaction executes no more than once. For example, if an online bookstore application determines that the previously submitted commit failed, then the application can safely resubmit.

Transaction Guard provides a tool for at-most-once execution to avoid the application executing duplicate submissions. Transaction Guard provides a known outcome for every transaction.

Transaction Guard is a core Oracle Database capability. Application Continuity uses Transaction Guard when masking outages from end users. Without Transaction Guard, an application retrying after an error may cause duplicate transactions to be committed.

See Also:

- *Oracle Database Development Guide* (../ADFNS/adfns_avail.htm#ADFNS924) to learn about Transaction Guard, including the types of supported and included transactions
- "Overview of Application Continuity (transact.htm#GUID-0B463F72-73C9-4EB6-B98D-5EC828CDB1E7) " to learn about Application Continuity, which works with Transaction Guard to help developers achieve high application availability

How Transaction Guard Works

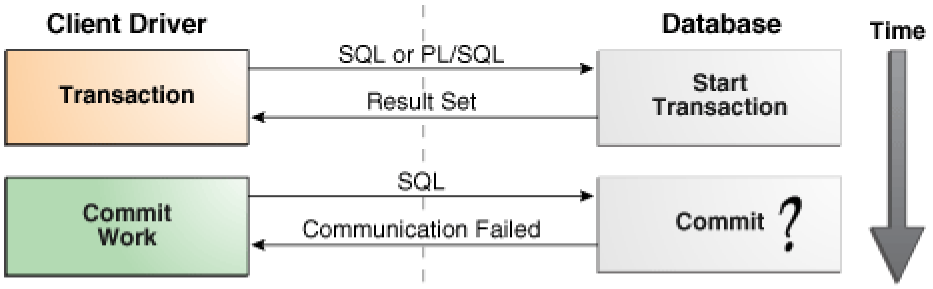
This section explains the problem of lost commit messages and how Transaction Guard uses logical transaction IDs to solve the problem.

Lost Commit Messages

When designing for idempotence, developers must address the problem of communication failures after submission of commit statements. Commit messages do not persist in the database and so cannot be retrieved after a failure.

The following graphic is a high-level representation of an interaction between a client application and a database.

Figure 10-2 Lost Commit Message



Description of "Figure 10-2 Lost Commit Message" (img_text/GUID-A13D6AC3-1310-4FFF-B2BB-3878EB6A9341-print.htm)

In the standard commit case, the database commits a transaction and returns a success message to the client. In Figure 10-2 (transact.htm#GUID-E567DC50-32EC-440E-9352-18E8AD79340B__CIHEHJGB) , the client submits a commit statement and receives a message stating that communication failed. This type of failure can occur for several reasons, including a database instance failure or network outage. In this scenario, the client does not know the state of the transaction.

Following a communication failure, the database may still be running the submission and be unaware that the client disconnected. Checking the transaction state does not guarantee that an active transaction will not commit after being checked. If the client resends the commit because of this out-of-date information, then the database may repeat the transaction, resulting in logical corruption.

Logical Transaction ID

Oracle Database solves the communication failure by using a globally unique identifier called a **logical transaction ID** (glossary.htm#GUID-B87A5F1E-E193-4A3D-AEAE-5EFE8DFFB3DA) . This ID contains the logical session number allocated when a session first connects, and a running commit number that is updated each time the session commits or rolls back.Foot 1 (#fn_1) From the application perspective, the logical transaction ID uniquely identifies the last database transaction submitted on the session that failed.

For each round trip from the client in which one or more transactions are committed, the database persists a logical transaction ID. This ID can provide transaction idempotence for interactions between the application and the database for each round trip that commits data.

The at-most-once protocol enables access to the commit outcome by requiring the database to do the following:

- Maintain the logical transaction ID for the retention period agreed for retry
- Persist the logical transaction ID on commit

While a transaction is running, both the database and client hold the logical transaction ID. The database gives the client a logical transaction ID at authentication, when borrowing from a connection pool, and at each round trip from the client driver that executes one or more commit operations.

Before the application can determine the outcome of the last transaction following a recoverable error, the application obtains the logical transaction ID held at the client using Java, OCI, OCCI, or ODP.Net APIs. The application then invokes the PL/SQL procedure `DBMS_APP_CONT.GET_LTXID_OUTCOME` with the logical transaction ID to determine the outcome of the last submission: committed (`true` or `false`) and user call completed (`true` or `false`).

When using Transaction Guard, the application can replay transactions when the error is recoverable and the last transaction on the session has not committed. The application can continue when the last transaction has committed and the user call has completed. The application can use Transaction Guard to return the known outcome to the client so that the client can decide the next action to take.

See Also:

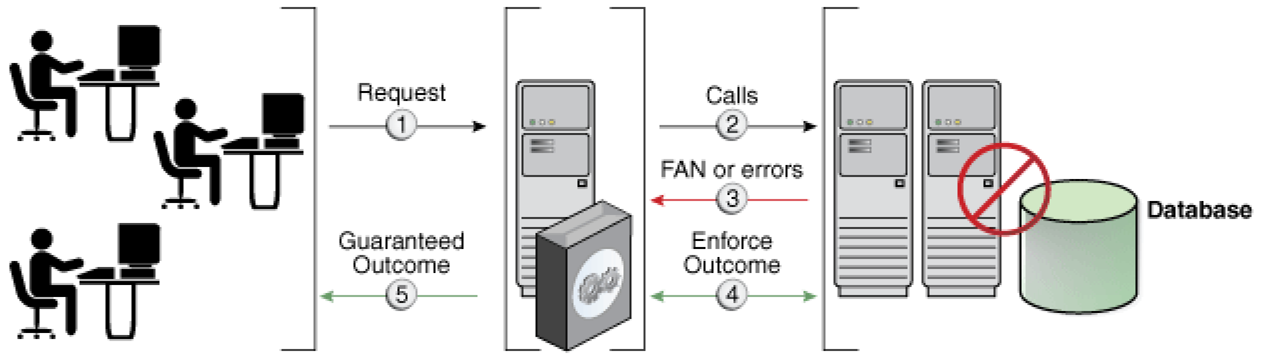
- *Oracle Database Development Guide* (../ADFNS/adfns_trans_idemp_guard.htm#ADFNS324) to learn about logical transaction IDs
- *Oracle Database PL/SQL Packages and Types Reference* (../ARPLS/d_app_cont.htm#ARPLS73456) to learn more about the `DBMS_APP_CONT.GET_LTXID_OUTCOME` procedure

Transaction Guard: Example

In this scenario, the commit message is lost because of a recoverable error.

Transaction Guard uses the logical transaction ID to preserve the outcome of the `COMMIT` statement, ensuring that there is a known outcome for the transaction.

Figure 10-3 Check of Logical Transaction Status



Description of "Figure 10-3 Check of Logical Transaction Status" (img_text/GUID-0164150C-892E-4FF7-8524-E817DD6BD7C9-print.htm)

In Figure 10-3 (transact.htm#GUID-6AEE7250-D39B-4DB7-BF71-E2C1841BF598__CIHGGHBA) , the database informs the application whether the transaction committed and whether the last user call completed. The application can then return the result to the end user. The possibilities are:

- If the transaction committed and the user call completed, then the application can return the result to the end user and continue.
- If the transaction committed but the user call did not complete, then the application can return the result to the end user with warnings. Examples include a lost out bind or lost number of rows processed. Some applications depend on the extra information, whereas others do not.
- If the user call was not committed, then the application can return this information to the end user, or safely resubmit. The protocol is guaranteed. When the commit status returns false, the last submission is blocked from committing.

See Also:

Oracle Database Development Guide (../ADFNS/adfns_trans_idemp_guard.htm#ADFNS8000) to learn how to use Transaction Guard

Overview of Application Continuity

Application Continuity attempts to mask outages from applications by replaying incomplete application requests after unplanned and planned outages. In this context, a request is a unit of work from the application. Typically, a request corresponds to the DML statements and other database calls of a single web request on a single database connection. In general, a request is demarcated by the calls made between check-out and check-in of a database connection from a connection pool.

This section contains the following topics:

- Benefits of Application Continuity (transact.htm#GUID-0B2F6AD0-92ED-431D-BEB6-10DFC3715FCC)
- Application Continuity Architecture (transact.htm#GUID-2E973242-2B39-42FC-AA16-CD97B460B6D2)
- How Application Continuity Works (transact.htm#GUID-4B6C5116-D396-45C6-8FA0-F7B607BDBBF6)

Benefits of Application Continuity

Application Continuity attempts to solve the lost session problem by restoring the database session when any component disrupts the conversation between database and client.

In a typical case, a client has submitted a request to the database, which has built up both transactional and nontransactional states. The state at the client remains current, potentially with entered data, returned data, and cached data and variables. However, the database session state, which the application needs to operate within, is lost.

If the client request has initiated one or more transactions, then the application is faced with the following possibilities:

- If a commit *has* been issued, then the commit message returned to the client is not durable. The client does not know whether the request committed, and where in the nontransactional processing state it reached.
- If a commit has *not* been issued, or if it was issued but did not execute, then the in-flight transaction is rolled back and must be resubmitted using a session in the correct state.

If the replay is successful, then database user service for planned and unplanned outages is not interrupted. The restored database session includes all states, cursors, variables, and the most recent transaction when one exists.

If the database detects changes in the data seen and potentially acted on by the application, then the replay is rejected. Replay is not attempted when the time allowed for starting replay is exceeded, the application uses a restricted call, or the application has explicitly disabled replay using the `disableReplay` method.

See Also:

Oracle Database Development Guide (../ADFNS/adfns_app_continuity.htm#ADFNS1058) to learn more about the benefits of application continuity

Application Continuity Architecture

The Application Continuity architecture includes the JDBC replay driver and the continuity director.

The components work together to execute capture and replay as follows:

- JDBC replay driver

  The JDBC replay driver intercepts execution errors and, when these errors are recoverable, automatically replays the user calls from the beginning of the request. When successful, the replay appears to the application as a delayed database interaction.

  In collaboration with the database, the JDBC replay driver maintains a history of calls during a conversation between a client and the database. For each call made at run time, the driver retains the context required for a subsequent replay.

- Continuity director

  The continuity director directs the run time and replay, working in collaboration with the JDBC client-side replay driver to determine what to do. The director knows what is repeatable and how to repeat it, and applies this knowledge as follows:

  - During run time, saving protocol and validation information (and mutables sufficient for replaying), or directing the replay drivers to keep or purge calls
  - During replay, rejecting a replay that:
    - Runs on a different database or a database that has lost transactions and potentially diverged

    - Violates protocols
    - Fails to reproduce the same client-visible data (rows, `OUT` binds, and messages) that the application or client saw and potentially made decisions on during the original execution

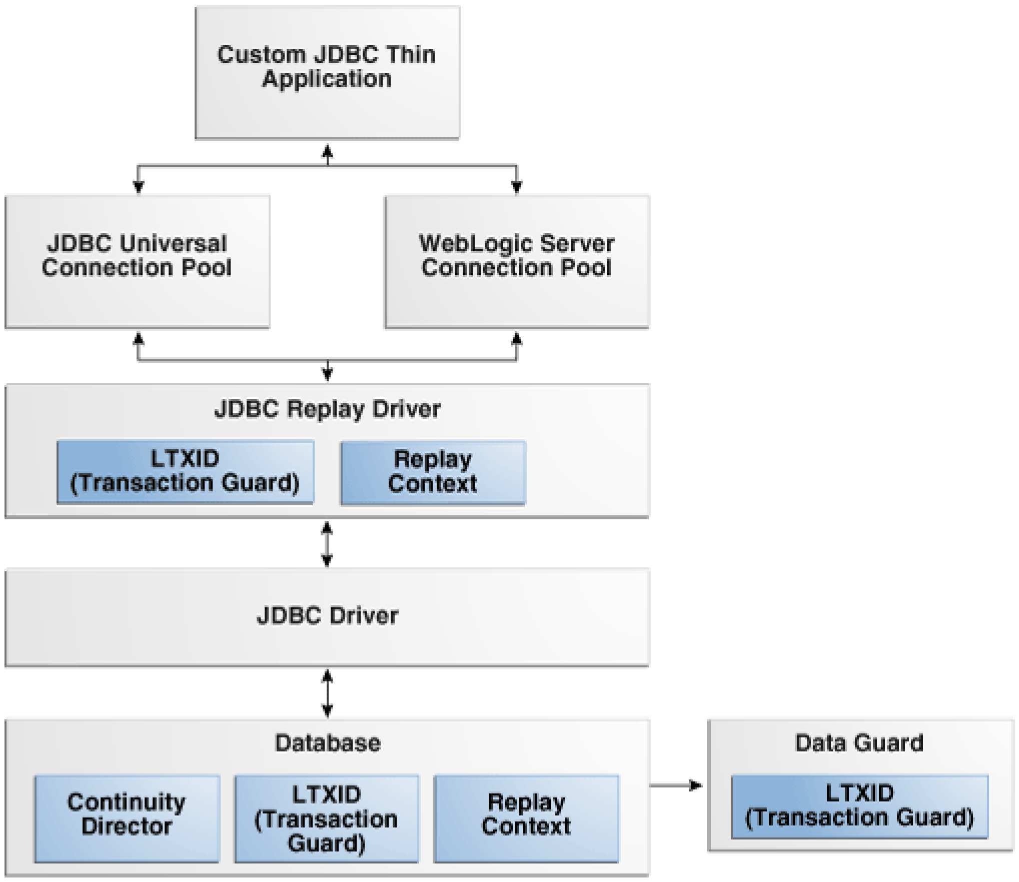
- Replay context

  The **replay context** (glossary.htm#GUID-5D479146-7FD2-486E-8DAA-9C2C13DC379E) is opaque information that the database returns to the client driver during normal application run time. The JDBC replay driver saves the context with each SQL or PL/SQL call that the database has instructed the driver to hold. The replay context contains sufficient knowledge to protect and validate[Foot 2] (#fn_2) replay of each call, and to apply mutables when mutable values exist. When a call is no longer needed for replaying the session, the driver discards the replay context along with the call itself.

- Transaction Guard (see "Overview of Transaction Guard (transact.htm#GUID-47BCD081-8FFF-4D13-A0B1-F531521BC6C3) ")

Figure 10-4 (transact.htm#GUID-2E973242-2B39-42FC-AA16-CD97B460B6D2__CIHHBCIC) depicts the basic architecture of Application Continuity. In the diagram, `LTXID` refers to the logical transaction ID in Transaction Guard (see "Logical Transaction ID (transact.htm#GUID-10F207C2-C59A-4707-A9A0-BDA3B19DE8D3) ").

Figure 10-4 Application Continuity Architecture


Custom JDBC Thin Application
JDBC Universal Connection Pool
WebLogic Server Connection Pool
JDBC Replay Driver
LTXID (Transaction Guard)
Replay Context
JDBC Driver
Database
Continuity Director
LTXID (Transaction Guard)
Replay Context
Data Guard
LTXID (Transaction Guard)

Description of "Figure 10-4 Application Continuity Architecture" (img_text/GUID-E621106A-4066-41B6-A09B-D8AA037940DC-print.htm)

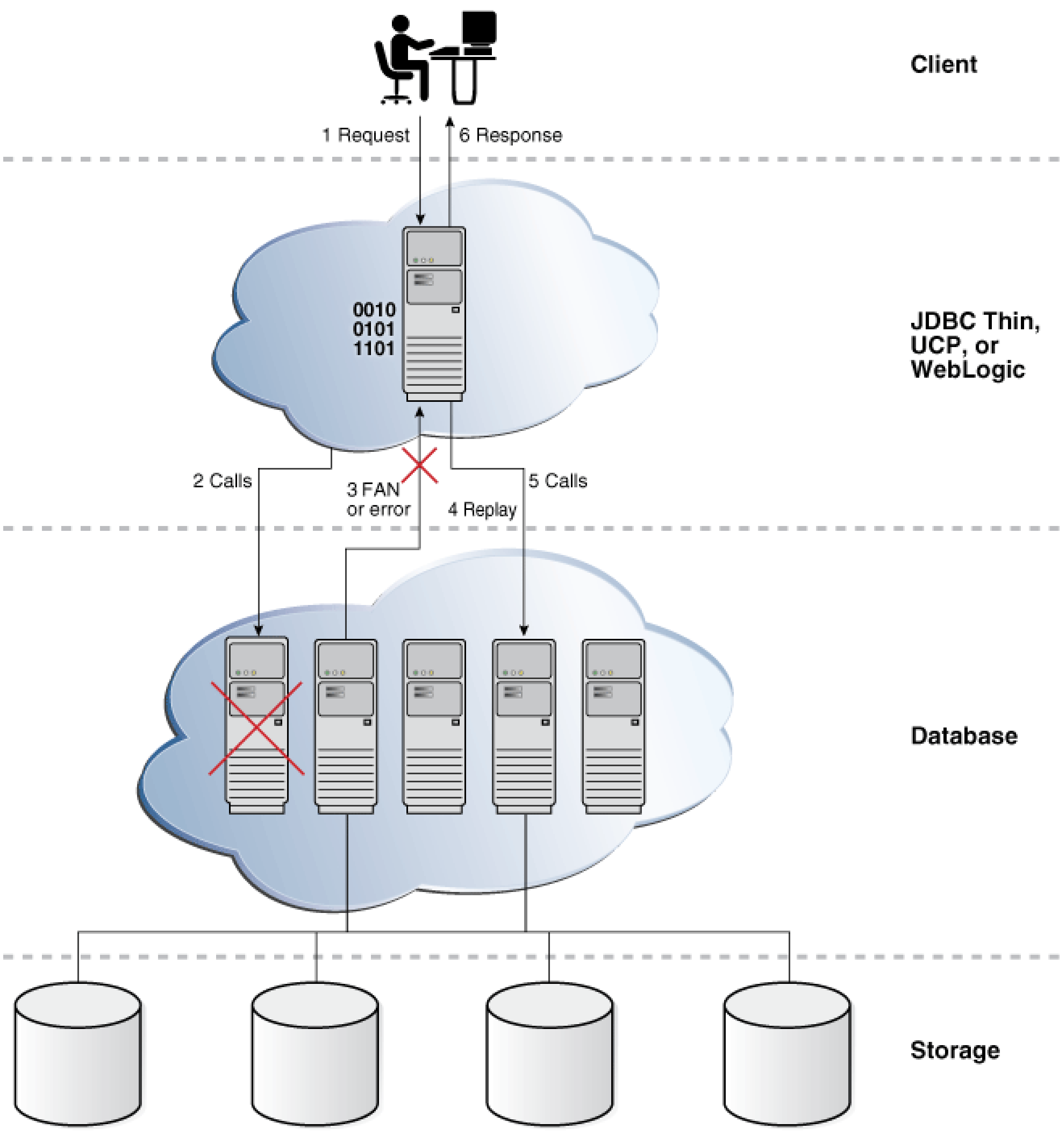
See Also:

- *Oracle Database Development Guide* (../ADFNS/adfns_app_continuity.htm#ADFNS1201) to learn more about Application Continuity
- *Oracle Database JDBC Developer's Guide* (../JJDBC/appcontnew.htm#JJDBC29059) and *Oracle Database JDBC Java API Reference* (../JAJDB/toc.htm) to learn more about JDBC and application continuity

How Application Continuity Works

Figure 10-5 (transact.htm#GUID-4B6C5116-D396-45C6-8FA0-F7B607BDBBF6__CIHIGBHC) shows the basic model for a conversation between a client application and a database using Application Continuity when an outage occurs. In the figure, the "X" indicates the location of the outage.

Figure 10-5 Application Continuity in a Planned or Unplanned Outage



Description of "Figure 10-5 Application Continuity in a Planned or Unplanned Outage" (img_text/GUID-D82923F1-A1D9-44F4-A391-B20CAF927EC7-print.htm)

The steps in the conversation are as follows:

1. The client application submits a request that the JDBC replay driver receives.
2. The replay driver sends the calls that form the request to the database, receiving directions for each call from the database.

3. The replay driver receives a Fast Application Notification (FAN) or recoverable error.

4. The replay driver performs the following actions:

    a. Obtains a new database session, and if a registered callback exists, runs this callback to initialize the session

    b. Checks with the database to determine whether replay can progress (for example, because the database rolled back the transaction)

5. If the continuity directory requires replay, then the replay driver resubmits calls under instruction from the database.

    Each call must establish the same client-visible state as the original.Foot 3 (#fn_3) Before replaying the last call, the replay driver ends the replay, and then returns to run-time mode under instruction from the database.

6. The JDBC driver relays the response to the client application:

    - If the replay is successful, then the database returns the response to the application. The replay appears as a delayed execution.

    - If the replay is unsuccessful, then the application receives the original error.

See Also:

- *Oracle Database Development Guide* (../ADFNS/adfns_app_continuity.htm#ADFNS1201) to learn more about Application Continuity

- *Oracle Real Application Clusters Administration and Deployment Guide* (../RACAD/GUID-921494A6-330F-44D1-A24E-774AFB3152E1.htm#RACAD8424) to learn more about configuring Application Continuity in an Oracle RAC environment

- *Oracle Database JDBC Developer's Guide* (../JJDBC/appcontnew.htm#JJDBC29059) and *Oracle Database JDBC Java API Reference* (../JAJDB/toc.htm) to learn more about JDBC and application continuity

- *Oracle Database Development Guide* (../ADFNS/adfns_avail.htm#ADFNS538) to learn more about FAN

Overview of Autonomous Transactions

An **autonomous transaction** is an independent transaction that can be called from another transaction, which is the main transaction. You can suspend the calling transaction, perform SQL operations and commit or undo them in the autonomous transaction, and then resume the calling transaction.

Autonomous transactions are useful for actions that must be performed independently, regardless of whether the calling transaction commits or rolls back. For example, in a stock purchase transaction, you want to commit customer data regardless of whether the overall stock purchase goes through. Additionally, you want to log error messages to a debug table even if the overall transaction rolls back.

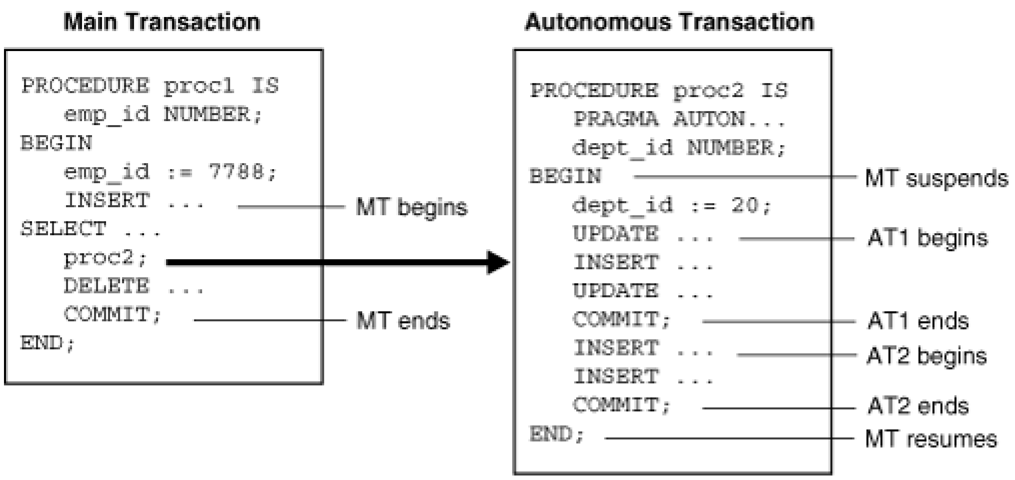
Autonomous transactions have the following characteristics:

- The autonomous transaction does not see uncommitted changes made by the main transaction and does not share locks or resources with the main transaction.
- Changes in an autonomous transaction are visible to other transactions upon commit of the autonomous transactions. Thus, users can access the updated information without having to wait for the main transaction to commit.
- Autonomous transactions can start other autonomous transactions. There are no limits, other than resource limits, on how many levels of autonomous transactions can be called.

In PL/SQL, an autonomous transaction executes within an *autonomous scope*, which is a routine marked with the pragma `AUTONOMOUS_TRANSACTION`. In this context, routines include top-level anonymous PL/SQL blocks and PL/SQL subprograms and triggers. A **pragma** (glossary.htm#GUID-0ED5CC20-407D-42A1-AF92-8816691861DE) is a directive that instructs the compiler to perform a compilation option. The pragma `AUTONOMOUS_TRANSACTION` instructs the database that this procedure, when executed, is to be executed as a new autonomous transaction that is independent of its parent transaction.

The following graphic shows how control flows from the main routine (MT) to an autonomous routine and back again. The main routine is `proc1` and the autonomous routine is `proc2`. The autonomous routine can commit multiple transactions (AT1 and AT2) before control returns to the main routine.

Figure 10-6 Transaction Control Flow



```
PROCEDURE proc1 IS
   emp_id NUMBER;
BEGIN
   emp_id := 7788;
   INSERT ...
SELECT ...
   proc2;
   DELETE ...
   COMMIT;
END;
```

```
PROCEDURE proc2 IS
   PRAGMA AUTON...
   dept_id NUMBER;
BEGIN
   dept_id := 20;
   UPDATE ...
   INSERT ...
   UPDATE ...
   COMMIT;
   INSERT ...
   INSERT ...
   COMMIT;
END;
```

Description of "Figure 10-6 Transaction Control Flow" (img_text/GUID-7DFAB7C0-B6BC-4F65-8F17-BB57F1A241AE-print.htm)

When you enter the executable section of an autonomous routine, the main routine suspends. When you exit the autonomous routine, the main routine resumes.

In Figure 10-6 (transact.htm#GUID-C0C61571-5175-400D-AEFC-FDBFE4F87188__CHDDGDCJ) , the `COMMIT` inside `proc1` makes permanent not only its own work but any outstanding work performed in its session. However, a `COMMIT` in `proc2` makes permanent only the work performed in the `proc2` transaction. Thus, the `COMMIT` statements in transactions AT1 and AT2 have no effect on the MT transaction.

See Also:

Oracle Database Development Guide (../ADFNS/adfns_sqlproc.htm#ADFNS00205) and *Oracle Database PL/SQL Language Reference* (../LNPLS/static.htm#LNPLS00609) to learn how to use autonomous transactions

Overview of Distributed Transactions

A **distributed database** (glossary.htm#GUID-DB7296DF-74E0-45E1-9BC2-5052DD543214) is a set of databases in a distributed system that can appear to applications as a single data source. A **distributed transaction** (glossary.htm#GUID-C4B9A58D-077E-4846-9625-65F5B37F5649) is a transaction that includes one or more statements that update data on two or more distinct nodes of a distributed database, using a schema object called a **database link** (glossary.htm#GUID-1C8C6E07-BE1F-4BC9-93C3-CF6D1D15DC42) . A database link describes how one database instance can log in to another database instance.

Unlike a transaction on a local database, a distributed transaction alters data on multiple databases. Consequently, distributed transaction processing is more complicated because the database must coordinate the committing or rolling back of the changes in a transaction as an atomic unit. The entire transaction must commit or roll back. Oracle Database must coordinate transaction control over a network and maintain data consistency, even if a network or system failure occurs.

> **See Also:**
>
> *Oracle Database Administrator's Guide* (../ADMIN/ds_txns.htm#ADMIN031)

Two-Phase Commit

The **two-phase commit mechanism** (glossary.htm#GUID-567B55F6-9A89-4F86-9711-B157C309CDF6) guarantees that *all* databases participating in a distributed transaction either all commit or all undo the statements in the transaction. The mechanism also protects implicit DML performed by integrity constraints, remote procedure calls, and triggers.

In a two-phase commit among multiple databases, one database coordinates the distributed transaction. The initiating node is called the *global coordinator*. The coordinator asks the other databases if they are prepared to commit. If any database responds with a no, then the entire transaction is rolled back. If all databases vote yes, then the coordinator broadcasts a message to make the commit permanent on each of the databases.

The two-phase commit mechanism is transparent to users who issue distributed transactions. In fact, users need not even know the transaction is distributed. A `COMMIT` statement denoting the end of a transaction automatically triggers the two-phase commit mechanism. No coding or complex statement syntax is required to include distributed transactions within the body of a database application.

> **See Also:**
>
> *Oracle Database Administrator's Guide* (../ADMIN/ds_txns.htm#ADMIN12222) to learn about the two-phase commit mechanism

In-Doubt Transactions

An **in-doubt distributed transaction** occurs when a two-phase commit was interrupted by any type of system or network failure.

For example, two databases report to the coordinating database that they were prepared to commit, but the coordinating database instance fails immediately after receiving the messages. The two databases who are prepared to commit are now left hanging while they await notification of the outcome.

The recoverer (`RECO`) background process automatically resolves the outcome of in-doubt distributed transactions. After the failure is repaired and communication is reestablished, the `RECO` process of each local Oracle database automatically commits or rolls back any in-doubt distributed transactions consistently on all involved nodes.

In the event of a long-term failure, Oracle Database enables each local administrator to manually commit or undo any distributed transactions that are in doubt because of the failure. This option enables the local database administrator to free any locked resources that are held indefinitely because of the long-term failure.

If a database must be recovered to a past time, then database recovery facilities enable database administrators at other sites to return their databases to the earlier point in time. This operation ensures that the global database remains consistent.

See Also:

- "Recoverer Process (RECO) (process.htm#GUID-9FF900D1-7DB8-4D41-8D34-8E99AF650CEC) "
- *Oracle Database Administrator's Guide* (../ADMIN/ds_txnman.htm#ADMIN12252) to learn how to manage in-doubt transactions

Footnote Legend

Footnote 1:
For Oracle Real Application Clusters (Oracle RAC), the logical transaction ID includes the database instance number as a prefix.

Footnote 2:
This validation is hardware-assisted in the database for platforms using current Intel and Sparc chips.

Footnote 3:
Validation is hardware-assisted when using current Intel and Sparc chips.

(https://docs.oracle.com/database/121/CNCPT/database.htm) (https://docs.oracle.com/database/121/CNCPT/part_dbarch.htm)

(#)

About Oracle (http://www.oracle.com/corporate/index.html) | Contact Us (http://www.oracle.com/us/corporate/contact/index.html) | Legal Notices (http://www.oracle.com/us/legal/index.html) |

Terms of Use (http://www.oracle.com/us/legal/terms/index.html) | Your Privacy Rights (http://www.oracle.com/us/legal/privacy/index.html)
Copyright © 1993, 2017, Oracle and/or its affiliates. All rights reserved. (http://www.oracle.com/pls/topic/lookup?ctx=cpyr&id=en-US)