Transaction Support in SQL

We will now discuss the support provided by SQL to specify transaction-level behavior. We will refer Oracle database in our example.

So you may not mark the beginning explicitly by SET TRANSACTION

- **Beginning of a Transaction**

A transaction begins when the **first** executable SQL statement (DML/DDL and the SET TRANSACTION statement is encountered). When a transaction begins,

  - Oracle Database assigns the transaction to an available **undo data** segment to record the undo entries for the new transaction.
  - A transaction ID is allocated after an undo segment and **transaction table** slot are allocated, which occurs during the first DML statement.

- **Setting Transaction Properties with SET TRANSACTION**

The SET TRANSACTION command can be used to initiate a database transaction. This command is used to specify characteristics for the transaction that follows. For example, you can specify a transaction to be read only, or read write. The syntax for SET TRANSACTION is as follows:

```
SET TRANSACTION [ READ WRITE | READ ONLY ];
```

In following example a store manager uses a read-only transaction to gather order totals for the day, the past week, and the past month. The totals are unaffected by other users updating the database during the transaction.

1. The SET TRANSACTION statement must be the first SQL statement in a read-only transaction
2. Can only appear once in a transaction.
3. If you set a transaction to READ ONLY, subsequent queries see only changes committed before the transaction began. The use of READ ONLY does not affect other users or transactions.

Example - Using SET TRANSACTION to Begin a Read-only Transaction

```
DECLARE
  daily_order_total   NUMBER(12,2);        weekly_order_total  NUMBER(12,2);
  monthly_order_total NUMBER(12,2);
BEGIN
  COMMIT; -- ends previous transaction
  SET TRANSACTION READ ONLY NAME 'Calculate Order Totals';
  SELECT SUM (order_total) INTO daily_order_total FROM orders      WHERE order_date = SYSDATE;
  SELECT SUM (order_total) INTO weekly_order_total FROM orders   WHERE order_date = SYSDATE - 7;
  SELECT SUM (order_total) INTO monthly_order_total FROM orders
                          WHERE order_date = SYSDATE - 30;
  COMMIT; -- ends read-only transaction
```

END;

- **Terminating Transactions**

A transaction is automatically started when we execute any SQL statement (SELECT, UPDATE , CREATE etc. Once a transaction is started, other statements can be executed as part of this transaction until the transaction is terminated by either a COMMIT command or a ROLLBACK command.

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database since the last COMMIT or ROLLBACK command was issued. The syntax for ROLLBACK command is as follows: ROLLBACK;

For involve long-running transactions, or that must run several transactions one after the other two features are available. One such feature called a **savepoint,** allows us to identify a point in a transaction and selectively roll back operations carried out after this point. This is useful if the transaction carries out what-if-kinds of operations, and wishes to undo or keep the changes based on the result.

In a long-running transaction, we may want to define a series of savepoints. The syntax is **: SAVEPOINT** < savepoint name >

A subsequent rollback command can specify the savepoint to roll back to

ROLLBACK TO SAVEPOINT < savepoint name >

A simple rollback or commit **erases all savepoints**. When you roll back to a savepoint, any savepoints marked after that savepoint are erased. The savepoint to which you roll back remains.

You can reuse savepoint names within a transaction. The savepoint moves from its old position to the current point in the transaction.

An implicit savepoint is marked before executing an INSERT, UPDATE, or DELETE statement. If the statement fails, a rollback to the implicit savepoint is done. Normally, just the failed SQL statement is rolled back, not the whole transaction; if the statement raises an unhandled exception, the host environment (such as SQL*Plus) determines what is rolled back.

Example of Savepoint and Rollback

Following is the **class** table

ID	NAME
1	abhi
2	adam
4	alex

Let's use some SQL queries on this table and see the results.

```
INSERT into class values(5,'Rahul');
commit;
UPDATE class set name='abhijit' where id='5';
savepoint A;
```

The resultant table will look like,

ID	NAME
1	abhi
2	adam

INSERT into class values(6,'Chris');
savepoint B;
INSERT into class values(7,'Bravo');
savepoint C;
SELECT * from class;

4	alex
5	abhijit
6	chris
7	bravo

Now **rollback** to **savepoint B** table will
rollback to B;
SELECT * from class;

The resultant look like

ID	NAME
1	abhi
2	adam
4	alex
5	abhijit
6	chris

Now **rollback** to **savepoint A**
rollback to A;
SELECT * from class;

The resultant table will look like

ID	NAME
1	abhi
2	adam
4	alex
5	abhijit

Example of COMMIT Command:

Consider the CUSTOMERS table having the following records:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Now delete records from the table having age = 25 and then COMMIT the changes in the database.

```
DELETE FROM CUSTOMERS WHERE AGE = 25;
COMMIT;
```

As a result, two rows from the table would be deleted and SELECT statement would produce t following result:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
3	kaushik	23	Kota	2000.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

The RELEASE SAVEPOINT Command:

The RELEASE SAVEPOINT command is used to remove a SAVEPOINT that you have created. The syntax for RELEASE SAVEPOINT is as follows:

```
RELEASE SAVEPOINT SAVEPOINT_NAME;
```

Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the SAVEPOINT.

Overview of Transaction Processing in PL/SQL

We will now discuss transaction processing with PL/SQL using SQL COMMIT, SAVEPOINT, and ROLLBACK statements that ensure the consistency of a database. You can include these SQL statements directly in your PL/SQL programs. Transaction processing available through all programming languages lets multiple users work on the database concurrently, and ensures that each user sees a consistent version of data and that all changes are applied in the right order.

You usually do not need to write extra code to prevent problems with multiple users accessing data concurrently. Oracle uses locks to control concurrent access to data, and locks only the minimum amount of data necessary, for as little time as possible. You can request locks on **tables** or **rows** if you really do need this level of control. You can choose from several modes of locking such as **row share** and **exclusive**.

Using COMMIT in PL/SQL

- The COMMIT statement ends the current transaction, making any changes made during that transaction permanent, and visible to other users.
- Transactions are not tied to PL/SQL BEGIN-END blocks.
- A block can contain multiple transactions, and a transaction can span multiple blocks.

Following Example illustrates a transaction that transfers money from one bank account to another. It is important that the money come out of one account, and into the other, at exactly the same moment. Otherwise, a problem partway through might make the money be lost from both accounts or be duplicated in both accounts.

```
DECLARE
  transfer NUMBER(8,2) := 250;
BEGIN
  UPDATE accounts SET balance = balance - transfer WHERE account_id = 7715;
  UPDATE accounts SET balance = balance + transfer WHERE account_id = 7720;
  COMMIT COMMENT 'Transfer From 7715 to 7720' WRITE IMMEDIATE NOWAIT;
END;
/
```

- All clauses after the **COMMIT** keyword are optional. If you specify only **COMMIT**, then the default is **COMMIT WORK WRITE IMMEDIATE WAIT**.

  The **WORK** keyword is supported for compliance with standard SQL. The statements **COMMIT** and **COMMIT WORK** are equivalent.

- **Comment clause** : Specify a comment to be associated with the current transaction. This comment can help you diagnose the failure of a distributed transaction.

- **WRITE Clause :** Use this clause to specify the priority with which the redo information generated by the commit operation is written to the redo log.

  If you omit this clause, then the behavior of the commit operation is controlled by the **COMMIT_WRITE** initialization parameter, if it is been set.

- The **IMMEDIATE** parameter initiates I/O, causing the redo for the commit of the transaction to be written out immediately by sending a message to the LGWR process. If you specify neither **IMMEDIATE** nor **BATCH**, then **IMMEDIATE** is the default.

  The **BATCH** parameter causes the redo to be buffered to the redo log. No I/O is initiated.

- The **WAIT** parameter ensures that the commit will not return until the corresponding redo is persistent in the online redo log. If you specify neither **WAIT** nor **NOWAIT**, then **WAIT** is the default.
  The **NOWAIT** parameter allows the commit to return before the redo is persistent in the redo log.

How Oracle Does Implicit Rollbacks

Before executing an INSERT, UPDATE, or DELETE statement, Oracle marks an **implicit savepoint** (unavailable to you). If the statement fails, Oracle rolls back to the savepoint. Usually, just the failed SQL statement is rolled back, not the whole transaction. If the statement raises an unhandled exception, the host environment determines what is rolled back.

Overriding Default Locking

By default, Oracle locks data structures for you automatically, which is a major strength of the Oracle database: different applications can read and write to the same data without harming each other's data or coordinating with each other.

You can request data locks on specific rows or entire tables if you need to override default locking. Explicit locking lets you deny access to data for the duration of a transaction.

- With the **LOCK TABLE** statement, you can explicitly lock entire tables.
- With the **SELECT FOR UPDATE** statement, you can explicitly lock specific rows of a table to make sure they do not change after you have read them. That way, you can check which or how many rows will be affected by an UPDATE or DELETE statement before issuing the statement, and no other application can change the rows in the meantime.

Using FOR UPDATE

When you declare a cursor that will be referenced in the **CURRENT OF** clause of an UPDATE or DELETE statement, you must use the **FOR UPDATE** clause to acquire exclusive row locks. An example follows:

```
DECLARE
  CURSOR c1  IS SELECT employee_id, salary FROM employees
                WHERE job_id = 'SA_REP' AND commission_pct > .10
                FOR UPDATE NOWAIT;
```

The **SELECT ... FOR UPDATE** statement identifies the rows that will be updated or deleted, then locks each row in the result set. This is useful when you want to base an update on the existing values in a row. In that case, you must make sure the row is not changed by another user before the update.

The optional keyword **NOWAIT** tells Oracle not to wait if requested rows have been locked by another user. Control is immediately returned to your program so that it can do other work before trying again to acquire the lock. If you omit the keyword NOWAIT, Oracle waits until the rows are available.

All rows are locked when you open the cursor, not as they are fetched. The rows are unlocked when you commit or roll back the transaction. Since the rows are no longer locked, you cannot fetch from a FOR UPDATE cursor after a commit.

When querying multiple tables, you can use the FOR UPDATE clause to confine row locking to particular tables. Rows in a table are locked only if the **FOR UPDATE OF** clause refers to a column in that table. For example, the following query locks rows in the employees table but not in the departments table:

```
DECLARE
  CURSOR c1 IS SELECT last_name, department_name FROM employees,  departments
          WHERE employees.department_id = departments.department_id AND job_id = 'SA_MAN'
          FOR UPDATE OF salary;
```

Autonomous Transactions

Autonomous transactions allow you to leave the context of the calling transaction, perform an independent transaction, and return to the calling transaction without affecting it's state. The autonomous transaction has no link to the calling transaction, so only committed data can be shared by both transactions.

- In Oracle, an autonomous transaction can commit or rollback the data in the same session without committing or rolling back in the main transaction. **PRAGMA** (compiler directive) statement is used to define autonomous transaction in Oracle.

Example : We create following test table and populate it with two rows.

```
CREATE TABLE at_test (
    id              NUMBER      NOT NULL,
    description  VARCHAR2(50) NOT NULL
);
```

```
INSERT INTO at_test (id, description) VALUES (1, 'Description for 1');
INSERT INTO at_test (id, description) VALUES (2, 'Description for 2');

SELECT * FROM at_test;

SQL>
```

Output

```
        ID DESCRIPTION
---------- --------------------
         1 Description for 1
         2 Description for 2
2 rows selected.
```

Notice that the data is not committed.

Next, we insert another 8 rows using an anonymous block declared as an autonomous transaction, which contains a commit statement.

```
DECLARE
   PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
   FOR i IN 3 .. 10 LOOP
         INSERT INTO at_test (id, description)   VALUES (i, 'Description for ' || i);
   END LOOP;
   COMMIT;
END;
/
```

PL/SQL procedure successfully completed.

```
SELECT * FROM at_test;
```

```
        ID DESCRIPTION
---------- --------------------
         1 Description for 1
         2 Description for 2
         3 Description for 3
         4 Description for 4
         5 Description for 5
         6 Description for 6
         7 Description for 7
         8 Description for 8
         9 Description for 9
        10 Description for 10

10 rows selected.
```

As expected, we now have 10 rows in the table.

If we now issue a rollback statement we get the following result.

```
ROLLBACK;
SELECT * FROM at_test;
```

```
        ID DESCRIPTION
---------- --------------------
         3 Description for 3
         4 Description for 4
         5 Description for 5
         6 Description for 6
         7 Description for 7
         8 Description for 8
         9 Description for 9
        10 Description for 10

8 rows selected.
```

The 2 rows inserted by our current session (transaction) have been rolled back, while the rows inserted by the autonomous transactions remain.

The presence of the PRAGMA AUTONOMOUS_TRANSACTION compiler directive made the anonymous block run in its own transaction, so the internal commit statement did not affect the calling session. As a result rollback was still able to affect the DML issued by the current statement.

Autonomous transactions are commonly used by **error logging routines**, where the error messages must be preserved, regardless of the commit/rollback status of the transaction. For example, the following table holds basic error messages.

```
CREATE TABLE error_logs (
  id                NUMBER(10)    NOT NULL,
  log_timestamp     TIMESTAMP     NOT NULL,
  error_message     VARCHAR2(4000),
  CONSTRAINT error_logs_pk PRIMARY KEY (id)
);
```

CREATE SEQUENCE error_logs_seq;
It is used to generate auto incremented id.

We define a procedure to log error messages as an autonomous transaction.

```
CREATE OR REPLACE PROCEDURE log_errors (p_error_message  IN  VARCHAR2) AS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO error_logs (id, log_timestamp, error_message)
       VALUES (error_logs_seq.NEXTVAL, SYSTIMESTAMP, p_error_message);
  COMMIT;
END;
/
```

The following code forces an error, which is trapped and logged.

```
BEGIN
  INSERT INTO at_test (id, description)  VALUES (998, 'Description for 998');
   -- Force invalid insert.
  INSERT INTO at_test (id, description)  VALUES (999, NULL);
EXCEPTION
  WHEN OTHERS THEN
     log_errors (p_error_message => SQLERRM);
     ROLLBACK;
END;
/

PL/SQL procedure successfully completed.

SELECT * FROM at_test WHERE id >= 998;

no rows selected

SELECT * FROM error_logs;

      ID LOG_TIMESTAMP
```

From this we can see that the LOG_ERRORS transaction was separate to the anonymous block.
If it weren't, we would expect the first insert in the anonymous block to be preserved by the commit statement in the LOG_ERRORS procedure.

---------- ----------

ERROR_MESSAGE

---

1 28-FEB-2006 11:10:10.107625
ORA-01400: cannot insert NULL into ("TIM_HALL"."AT_TEST"."DESCRIPTION")

1 row selected.

Be careful how you use autonomous transactions. If they are used indiscriminately they can lead to deadlocks, and cause confusion when analyzing session trace. To hammer this point home, here's a quote

"... in 999 times out of 1000, if you find yourself "forced" to use an autonomous transaction - it likely means you have a serious data integrity issue you haven't thought about.

Another Example : Create Function fnc_test

The following function will insert some data into the **test_data** table, and after that, it will generate the error because it is dividing by 0 in the next line. On error, in the exception section, it is calling the procedure **prc_log_errors** to log the error. If the function executes without error, then it will return TRUE else it will return FALSE. In the below case, it will return the FALSE after logging the error.

```
CREATE OR REPLACE FUNCTION fnc_test
  RETURN BOOLEAN
IS
  n   NUMBER;
BEGIN
  INSERT INTO test_data   VALUES ('abc');
  n := 2 / 0;     /* generate error */
  RETURN TRUE;
EXCEPTION
  WHEN OTHERS
  THEN
     log_errors (p_error_message => SQLERRM);
     RETURN FALSE;
END fnc_test;
/
```

Check the test_data table, should have no records.

```
SELECT * FROM test_data;
```

Output

no rows selected.

Check data in the error_log table

```
SELECT * FROM error_log;
```

Output

```
ERROR_CODE ERROR_MSG                    DATE_OCCURRED        PLSQL_PROGRAM_REF
-1476      ORA-01476: divisor is equal to zero 27/03/2019 15:43:12 FNC_TEST
```

Test

Call the function **fnc_test**.

```
BEGIN
  IF fnc_test    THEN    COMMIT;
     ELSE                ROLLBACK;
  END IF;
EXCEPTION
  WHEN OTHERS   THEN   ROLLBACK;
END;
/
```

Even it is rolling back on fail, still, the data will be saved in the error_log table, because the **procedure log_errors** is using **PRAGMA AUTONOMOUS_TRANSACTION**.