


DBS211

 This page is still under construction !!

Week 6 - Transactions and Security

Table of Contents

- [Welcome](#)
- [What is a Transaction](#)
- [Transaction Scope](#)
- [Commit, Save Point and Rollback](#)
- [DDL and Transactions](#)
- [Concurrency](#)
- [User Level Security](#)
- [Grant and Revoke](#)

Reading Materials

- [Text - Chapter 1 \(Before the Advent of Database Systems\)](#)
- [Text - Chapter 2 \(Fundamental Concepts\)](#)
- [Text - Chapter 3 \(Characteristics and Benefits of a Database\)](#)

Welcome to Week 6

Welcome to Week 6. This week we tackle the subject of transactions and security. Transactions are performed millions of times every day and it is important that the computer programmer understands how these real-world scenarios are programmed and maintained. In today's information technology industry, protection of data and privacy is top of the list of important database considerations. After completing this week, you will be able to:

- Create multi-SQL statement transactions and use error handling to determine the ultimate success or failure of the transaction,
- Use effectively, the concepts of Commit, Save Point and Rollback
- Describe and program against the concepts of concurrency and record locking
- grant and revoke various privileges from both public and local users to a database

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.

Real World Transaction Example

Let us say you have 2 bank accounts at your local bank; one chequing and one savings account. You log into your online banking and go through the simple process of transferring \$200 from your chequings to your savings account. Easy enough right? But now think about it from the bank's perspective and how the database process may go. You are customer ID 24.

Table: ACCOUNTS			
<u>Account Number</u>	Account Type	CustID	Balance
1234	Chq	24	25.27
3456	Sav	24	1589.42

Table: TRANSACTIONS				
<u>TransID</u>	Date	Account Number	Transaction Type	Amount
33	May 21, 2020	5678	Debit	99.99
44	May 21, 2020	7890	Credit	1257.41

<u>Account Number</u>	Account Type	CustID	Balance	<u>TransID</u>	Date	Account Number	Transaction Type	Amount
5678	Chq	32	4892.34	55	May 22, 2020	3456	Debit	200.00
7890	Chq	56	789.46	66	May 22, 2020	1234	Credit	200.00

Note: A credit is a deposit (positive) into your account and a debit is a withdrawal (negative).

Using just the simplified tables above the basic process would be something like:

1. Check to see if user has permissions to withdrawal money from the savings account

```
SELECT accountno WHERE accountno = 3456 AND custID = &UsersCustomerID; -- if there is a result, permission granted
```

2. Check to see if there is enough money in the savings account to withdrawal \$200

```
SELECT accountno WHERE accountno = 3456 AND balance >= 200; -- if there is a result, sufficient funds
```

3. Create a record in the transactions table for the withdrawal from savings (*transactionid 55*)

```
INSERT INTO transactions VALUES(55, sysdate, 3456, 'Debit', 200);
```

4. Update the accounts table record for the savings account to make sure the balance is correct (*update account number 3456*)

```
UPDATE accounts SET balance = balance - 200 WHERE accountNo = 3456;
```

5. Check to make sure the user has permission to deposit money into the chequings account

```
SELECT accountno WHERE accountno = 1234 AND custID = &UsersCustomerID; -- if there is a result, permission granted
```

6. Create a record in the transactions table for the deposit to chequings (*transactionid 66*)

```
INSERT INTO transactions VALUES(66, sysdate, 1234, 'Credit', 200);
```

7. Update the accounts table record for the chequing account to make sure the balance is correct (*update account number 1234*)

```
UPDATE accounts SET balance = balance + 200 WHERE accountNo = 1234;
```

8. Send out a confirmation code stating everything was successful

So maybe not as simple as you first thought? But still relatively simple. But let us now imagine that you successfully got past step 4 and found out that the user now does not have permission to deposit into the chequings account and there for steps 6 and 7 will fail. If you were running one SQL statement at a time, where would the \$200 that was taken out of Savings be now? hmmm - in cyber oblivion and you would now have \$200 less in the savings, and no deposit into the chequing account. You would not be happy starving student. \$200 is a couple weeks groceries.

This is where transactions come in. The concept that all steps must complete successfully, or none are completed. If the deposit into chequing fails, the withdrawal from savings must be reversed and undone like it never happened.

Transaction Scope

So the scope of a transaction is regarding when it starts and finishes. Each related set of commands must start and end independantly from other statements. In most DBMS systems, in order to perform transactions you would use a statement like this, in its' simplest form.

It is possible to have more than one transaction at a time and have them interact with each other, however this is an advanced topic not covered until much later. For now we will only consider autonomous multiple statement transactions.

```
BEGIN transaction;  -- or  START transaction;  or  just  BEGIN;
.... do stuff...
COMMIT;
END transaction;    -- or  just  END;
```

This clearly states when the transaction starts and ends and the statements between are the statements of the transaction that must all complete successfully.

However, in Oracle, transactions are constantly automatically created for you any time you are communicating with the database, regardless if you mean to or not. They are automated, but somewhat invisible. So let's explain how they work.

Everything you do in the worksheet of SQL Developer will be in a transaction. They start and stop, new ones are created all the time. It is critical you know when they start and end such that you can use the transaction statements appropriately.

There are many ways that a transaction is started, the 4 most common are:

1. The user has established a new connection to the server and has a blank sheet ready to go, starts a new transaction
2. The user uses the BEGIN statement in Oracle SQL, this will start a new transaction
3. The user executes a COMMIT statement, the current transaction is committed, and a new transaction starts.
4. The user executes ANY DDL statement. This automatically triggers an auto-commit of the current transaction and starts a new transaction.

Transaction statements, when executed, only apply against the currently running transaction. Once a transaction is ended, all objects, references and statements with respect to the current transaction are terminated and no longer exist (they are completed). DML statements that INSERT, UPDATE or DELETE data are not actually completed in the server database until they are committed, **even though it appears like they are if you run a SELECT statement to view the data**. Some examples below will explain this further.

Commit, Save Points, Rollback

In Oracle, the three main statements used in transactions are COMMIT, SAVEPOINT and ROLLBACK.

Commit

The commit statement simply ends the current transaction and any pending changes to the data are permanently made on the database server. Immediately upon the commit statements successfully executing, the current transaction is terminated and a new transaction is

started.

So what happens to the DML statements you execute if they are not committed?

To demonstrate this, it is easiest to try it for yourself. The following is a series of statements and tasks to perform in sequence. It does require you to close SQL Developer, so make sure you save your .sql file.

1. Open a new worksheet connected to your database.
2. Create an INSERT statement on any table and execute that statement
3. Create a SELECT statement that will recall that new record and see that it is in the table.
4. SAVE YOUR open .sql file(s)
5. Now crash SQL Developer (On Windows use Ctrl-Alt-Delete and go to task manager, right click on SQL Developer and click End Task)
Alternatively you can just close SQL Developer and when prompted, select to abort the current session. (But it is a more effective demonstration if you crash it)
6. Now reopen SQL Developer, connect to the database and open the .sql file you were working on.
7. Run the SELECT statement again.

What happened??? The record you inserted, and saw was there, is no longer there.

Because the transaction was not committed, the INSERT statement you executed was simply a trial to see if it would work. Because transactions are made up of multiple statements that often depend on each other it is important that the data appears to be there. If the next statement required that record to be there for it to work, it has to appear to be there. The Primary key also gets "reserved" so noone else can use it if they insert another record. But this is all undone, or rolled back, if the transaction never gets committed.

Example 2:

1. Rerun the INSERT statement you ran above,
2. run the SELECT statement again to see it is there
3. create and execute a COMMIT statement
4. SAVE YOUR .sql file

5. now crash or close and abort SQL Developer again.
6. Reload SQL Developer, reconnect and open your file again
7. Execute the SELECT statement once again.

Now you see that the record you inserted is still there and therefore has been permanently saved to the server.

Auto-Commit

```
SET AUTOCOMMIT ON
```

```
SET AUTOCOMMIT OFF
```

are two statements that can be run to change how DML statements are handled. By default auto-commit is off. If autocommit is on, then DML statements are automatically committed immediately upon successful execution without the need for commit. Why don't we always do this you say? Remember the banking example above where your money went into cyber space, sometimes we need a way to make sure other statements are also successful before committing to previous ones.

Rollback

The rollback statement will undo all the executed statements in the current transaction as if nothing had happened. REMEMBER, the current transaction. So knowing where the current transaction started is important.

Example 1:

```
COMMIT;  -- force starts a new transaction
```

```
INSERT INTO players (playerID, firstname, lastname, DOB, email)
VALUES (1667, 'George', 'Dunkirk', to_date('19860428','yyyymmdd'), 'gdunkirk@email.com');
```

```
SELECT * FROM players WHERE playerID = 1667;    -- there will be one record shown

ROLLBACK;

SELECT * FROM players WHERE playerID = 1667;    -- no records shown
```

This ROLLBACK undid all statements in the transaction, so the current transaction is now empty.

From my experience, learners often get confused here. It is important to understand the difference between the statements you see in the .sql file (or worksheet) you are using and statements that have actually been executed. Just because it is in the file in front of you does not mean it has been executed. Transactions will only contain statements that have been executed, not the ones written in the file. So to make this easier and clearer, we write the statements from top to bottom in the order we are going to execute them, regardless if we are repeating the statements. This indicates we are running it again.

Example 2:

```
COMMIT;    -- force starts a new transaction

INSERT INTO players (playerID, firstname, lastname, DOB, email)
VALUES (1667, 'George', 'Dunkirk', to_date('19860428','yyyymmdd'), 'gdunkirk@email.com');

SELECT * FROM players WHERE playerID = 1667;    -- there will be one record shown

COMMIT;

ROLLBACK;

SELECT * FROM players WHERE playerID = 1667;    -- the record is still there
```

In this example, the COMMIT statement permanently makes the change on the server and starts a new transaction. So the ROLLBACK statement is actually executed in a new transaction, that at the time is empty, and therefore does absolutely nothing.

Savepoints

SAVEPOINT is a statement that can be used in larger transactions to place markers at specific locations throughout the transaction. This now allows ROLLBACK to only undo part of the transaction, and not the whole thing. SQL code can be written in a way that multiple tries can take place, so a partial rollback may allow another try before giving up and rolling back the entire transaction.

Example 1:

```
COMMIT;    -- force starts a new transaction

INSERT INTO players (playerID, firstname, lastname, DOB, email)
VALUES (1667, 'George', 'Dunkirk', to_date('19860428','yyyymmdd'), 'gdunkirk@email.com');

SAVEPOINT A;

SELECT * FROM players WHERE playerID = 1667 OR playerID = 1668    -- there will be one record shown

INSERT INTO players (playerID, firstname, lastname, DOB, email)
VALUES (1668, 'Henry', 'Buggles', to_date('19851203','yyyymmdd'), 'gbuggles@email.com');

SELECT * FROM players WHERE playerID = 1667 OR playerID = 1668;    -- there will be two records shown

ROLLBACK TO A;

SELECT * FROM players WHERE playerID = 1667 OR playerID = 1668    -- only one record will be shown as 1668 was rolled back
```

Note, that in the above example, there is still an active statement in the current transaction: the INSERT of player 1667. A COMMIT would make that permanent, end the transaction and start a new transaction where a ROLLBACK would rollback that player as well and make the current transaction now empty as if nothing happened at all.

Example 2:

```
COMMIT;    -- force starts a new transaction

INSERT INTO players (playerID, firstname, lastname, DOB, email)
  VALUES (1667, 'George', 'Dunkirk', to_date('19860428','yyyymmdd'), 'gdunkirk@email.com');

SAVEPOINT A;

SELECT * FROM players WHERE playerID = 1667 OR playerID = 1668    -- there will be one record shown

INSERT INTO players (playerID, firstname, lastname, DOB, email)
  VALUES (1668, 'Henry', 'Buggles', to_date('19851203','yyyymmdd'), 'gbuggles@email.com');

SELECT * FROM players WHERE playerID = 1667 OR playerID = 1668;    -- there will be two records shown

COMMIT;

ROLLBACK TO A;    -- error

SELECT * FROM players WHERE playerID = 1667 OR playerID = 1668    -- Both records will be shown as they were both committed.
```

In the above example: the ROLLBACK TO A statement will actually fail and cause an error. This is because the COMMIT ends the transaction and starts a new one. The SAVEPOINT statement was in the previous transaction and therefore does not exist in the new transaction, which is where the ROLLBACK TO A is executed.

The Form of a Transaction with Error Checking

So now how do we create a series of SQL statements that make up a transaction and automate the rollback process if an error occurs. For this we use the build in error handling features in Oracle SQL.

```
BEGIN
  INSERT INTO players (playerID, firstname, lastname, DOB, email)
    VALUES (1667, 'George', 'Dunkirk', to_date('19860428','yyyymmdd'), 'gdunkirk@email.com');
  INSERT INTO players (playerID, firstname, lastname, DOB, email)
```

```
VALUES (1668, 'Henry', 'Buggles', to_date('19851203','yyyymmdd'), 'gbuggles@email.com');
COMMIT;
DBMS_OUTPUT.PUT_LINE('Transaction Successful!');
EXCEPTION      -- when any error occurs after BEGIN and therefore COMMIT is skipped
ROLLBACK;      -- rolls back everything in the transaction
DBMS_OUTPUT.PUT_LINE('Transaction Failed, rolled back');
END;
```

The above example cleanly performs an autonomous multiple statement transaction and if an error occurs on any statement after BEGIN, it will jump to the EXCEPTION statements and perform the ROLLBACK. If no error occurs, then the COMMIT statement gets executed as the last statement in the transaction and indicates a successful execution.

DDL and Transactions

As mentioned above, a transaction is automatically committed when any DDL statement is executed. This is a pretty big gotcha in daily operations and can do things in the background that you did not intend and also gives you no feedback stating that this happened. So you just simply have to know this is an issue and not mix your DML and DDL code if you are utilizing transaction features.

Example

```
COMMIT;      -- force starts a new transaction

INSERT INTO players (playerID, firstname, lastname, DOB, email)
VALUES (1667, 'George', 'Dunkirk', to_date('19860428','yyyymmdd'), 'gdunkirk@email.com');

INSERT INTO players (playerID, firstname, lastname, DOB, email)
VALUES (1668, 'Henry', 'Buggles', to_date('19851203','yyyymmdd'), 'gbuggles@email.com');

SELECT * FROM players WHERE playerID = 1667 OR playerID = 1668;      -- there will be two records shown

CREATE VIEW vsShowPlayers AS
SELECT * FROM players WHERE playerID = 1667 OR playerID = 1668;
SELECT * FROM vwShowPlayers;
```

```
ROLLBACK;
```

```
SELECT * FROM players WHERE playerID = 1667 OR playerID = 1668    -- Both records will be shown as they were both committed.
```

In the above example, the CREATE statement is a DDL statement, that when executed, will automatically commit the current transactions. Therefore the 2 INSERT statements are committed; meaning the ROLLBACK statement actually occurs in a new transaction and there is nothing to roll back.

WARNING: This is a huge gotcha when mixing DDL and DML statements together. It is best practice to never mix them together or to create two separate sessions or connections with the database such that transactions can be independent from one another.

Concurrency

here...

User Level Security

here....

Grant and Revoke

here....

Practice Exercises

- [Transactions](#)
- [Concurrency](#)

- [Security](#)