Lab Session 10

(Triggers and Transactions)

By: Ammara Yaseen

Triggers:

A trigger is an event within the DBMS that can cause some code to execute automatically.

Triggers are named PL/SQL blocks which are stored in the database. We can also say that they are specialized stored programs which execute implicitly when a triggering event occurs. This means we cannot call and execute them directly instead they only get triggered by events in the database.

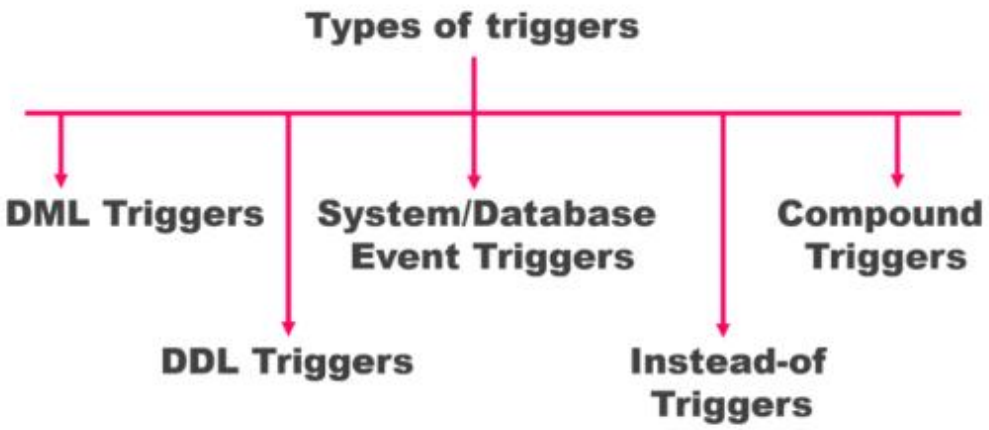
Events Which Fires the Database Triggers

These events can be anything such as

1. **A DML Statement** – An Update, Insert or Delete statement executing on any table of your database. You can program your trigger to execute either BEFORE or AFTER executing your DML statement. For example, you can create a trigger which will get fired *Before* the Update. Similarly, you can create a trigger which will get triggered after the execution of your INSERT DML statement.
2. **A DDL Statement** – Next type of triggering statement can be a DDL Statement such as CREATE or ALTER. These triggers can also be executed either BEFORE or AFTER the execution of your DDL statement. These triggers are generally used by DBAs for auditing purposes. And they really come in handy when you want to keep an eye on the various changes on your schema. For instance, who created the object or which user. Just like some cool spy tricks.
3. **A system event**-You can create a trigger on a system event. And by a system event, shut down or startup of your database.
4. **A User Events** – Another type of triggering event can be User Events such as log off or log on onto your database. You can create a trigger which will either execute before or after the event. Furthermore, it will record the information such as time of event occur, the username who created it.

Types of Database Triggers

There are 5 types of triggers in the Oracle database. 3 of them are based on the triggering event.



Syntax:

```
CREATE [OR REPLACE] TRIGGER Trigger_name
{BEFORE|AFTER} Triggering_event ON table_name
[FOR EACH ROW]
[ENABLE/DISABLE]
[WHEN condition]
DECLARE
  declaration statements
BEGIN
  Executable statements
EXCEPTION
  Exception-handling statements
END;
```

Example:

The price of a product changes constantly. It is important to maintain the history of the prices of the products. We can create a trigger to update the 'product_price_history' table when the price of the product is updated in the 'product' table.

1. **Create the product table and product_price_history.**

```
CREATE TABLE product_price_history
(product_id number(5),
product_name VARCHAR(20),
supplier_name VARCHAR (20),
unit_price number(7,2) );

CREATE TABLE product
(product_id number(5),
product_name VARCHAR(20),
supplier_name VARCHAR (20),
unit_price number(7,2) ) ;
```

2. **Create the product_history_trigger and execute it.**

```
CREATE OR REPLACE TRIGGER price_history_trigger
BEFORE UPDATE OF unit_price
ON product
FOR EACH ROW
BEGIN
INSERT INTO product_price_history
VALUES (:old.product_id,:old.product_name,:old.supplier_name,:old.unit_price);

END;
```

3. **Lets insert & update the price of a product.**

```
Insert into product values (100, 'Laptop', 'Dell', 262.22);
Insert into product values (101, 'Laptop', 'HP', 362.22);
UPDATE PRODUCT SET unit_price=800 WHERE product_id=100;
```

Once the above query is executed, the trigger fires and updates the 'product_price_history' table.

Types of PL/SQL Triggers:

There are two types of triggers based on the which level it is triggered.

1) **Row Level Trigger**- An event is triggered for each row updated, inserted or deleted.
2) **Statement Level Trigger**- An event is triggered for each sql statement executed.

PL/SQL Triggers Execution Hierarchy:

The following hierarchy is followed when a trigger is fired.

1) BEFORE statement trigger fires first.

2) Next BEFORE row level triggers fires, once for each row affected.

3) Then AFTER row level trigger fires once for each affected row. This events will alternates between BEFORE and AFTER row level triggers.

4) Finally the AFTER statement level trigger fires.

Example:

Let's create a table 'product_check' which we can use to store messages when triggers are fired.

```
CREATE TABLE product_check
(Message VARCHAR(50),
Current_Date Date
);
```

Let's create a BEFORE and AFTER statement and row level triggers for the product table.

```
CREATE or REPLACE TRIGGER Before_Update_Stat_product
BEFORE
UPDATE ON product
BEGIN
INSERT INTO product_check
Values ('Before update, statement level', sysdate);
END;
/
```

This above trigger will insert a record into the table 'product_check' before a Sql update statement is executed, at the statement level .

```
CREATE or REPLACE TRIGGER Before_Update_Row_product
BEFORE
UPDATE ON product
FOR EACH ROW
BEGIN
INSERT INTO product_check
Values ('Before update row level', sysdate);
END;
```

This trigger will insert a record into the table 'product_check' before each row is update.

```
CREATE or REPLACE TRIGGER After_Update_Stat_product
AFTER
UPDATE ON product
BEGIN
INSERT INTO product_check
Values ('After update, statement level', sysdate);
END;
/
```

This trigger will insert a record into the table
'product_check' after a Sql update statement is executed, at the statement level.

```
CREATE or REPLACE TRIGGER After_Update_Row_product
AFTER
UPDATE ON product
FOR EACH ROW
BEGIN
INSERT INTO product_check
Values ('After update, row level', sysdate);
END;
```

This trigger will insert a record into the table 'product-check'
after each row is updated.

Now let's execute an update statement on table product.

```
UPDATE PRODUCT SET unit_price = 800 WHERE product_id in (100,101);
```

Let's check the data in 'product_check' table to see the order in which the trigger is fired.

```
SELECT * FROM product_check;
```

Output:

	MESSAGE	CURRENT_DATE
1	Before update, statement level	28-OCT-19
2	Before update row level	28-OCT-19
3	After update, row level	28-OCT-19
4	Before update row level	28-OCT-19
5	After update, row level	28-OCT-19
6	After update, statement level	28-OCT-19

Transactions

When the application logic needs to execute a sequence of SQL commands in an atomic fashion, then the commands need to be grouped as a logical unit of work (LUW) called SQL transaction In everyday life, people conduct different kind of business transactions buying products, ordering travels, changing or canceling orders, buying tickets to concerts, paying rents, electricity bills, insurance invoices, etc. Transactions are recoverable units of data access tasks in terms of database content manipulation.

Commit: Any successful execution of the transaction is ended by a **COMMIT** command.

Rollback: Unsuccessful execution need to be ended by a **ROLLBACK** command which automatically recovers from the database all changes made by the transaction.

Savepoint: savepoint command is used to temporarily save a transaction so that you can roll back to that point whenever required.

Savepoint savepoint_name;

Dirty Read:

The meaning of this term is as bad as it sounds. You're permitted to read uncommitted, or dirty, data. You can achieve this effect by just opening an OS file that someone else is writing and reading whatever data happens to be there.

Nonrepeatable read:

This simply means that if you read a row at time T1 and try to reread that row at time T2, the row may have changed. It may have disappeared; it may have been updated, and so on.

Phantom read:

This means that if you execute a query at time T1 and re-execute it at time T2, additional rows may have been added to the database, which may affect your results.

This differs from a Nonrepeatable read in that with a phantom read, data you already read hasn't been changed, but instead, more data satisfies your query criteria than before.

Anomaly	Example
Dirty Reads A dirty read happens when a transaction reads data that is being modified by another transaction that has not yet committed.	Transaction A begins. UPDATE emp SET salary = 31650 WHERE empno = '7782' Transaction B begins. SELECT * FROM emp (Transaction B sees data updated by transaction A. Those updates have not yet been committed.)
Non-Repeatable Reads Non-repeatable reads happen when a query returns data that would be different if the query were repeated within the same transaction. Non-repeatable reads can occur when other transactions are modifying data that a transaction is reading.	Transaction A begins. SELECT * FROM emp WHERE empno = '7782' Transaction B begins. UPDATE empl SET salary = 30100 WHERE empno = '7782' (Transaction B updates rows viewed by transaction A before transaction A commits.) If Transaction A issues the same SELECT statement, the results will be different.
Phantom Reads Records that appear in a set being read by another transaction. Phantom reads can occur when other transactions insert rows that would satisfy the WHERE clause of another transaction's statement.	Transaction A begins. SELECT * FROM emp WHERE sal > 30000 Transaction B begins. INSERT INTO emp (empno, ename,job , , sal) VALUES ('7356', 'NICK', 'CLERK','35000') Transaction B inserts a row that would satisfy the query in Transaction A if it were issued again.

Transaction Isolation Levels:

There are four levels of transaction isolation defined in ANSI/ISO SQL standard, with different possible outcomes for the same transaction scenario. That is, the same work performed in the same fashion with the same inputs may result in different answers, depending on your isolation level.

Serializable

With a lock-based concurrency control DBMS implementation, serializability requires read and write locks (acquired on selected data) to be released at the end of the transaction.

Repeatable reads

In this isolation level, a lock-based concurrency control DBMS implementation keeps read and write locks (acquired on selected data) until the end of the transaction.

Read committed

Read committed is an isolation level that guarantees that any data read is committed at the moment it is read.

Read uncommitted

This is the lowest isolation level. In this level, dirty reads are allowed, so one transaction may see not-yet-committed changes made by other transactions.

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
READ UNCOMMITTED	Permitted	Permitted	Permitted
READ COMMITTED	--	Permitted	Permitted
REPEATABLE READ	--	--	Permitted
SERIALIZABLE	--	--	--

COMMAND TO SET ISOLATION LEVEL:

```
SET TRANSACTION ISOLATION LEVEL {SERIALIZABLE | READ COMMITTED};
```

User User1

```
SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

Transaction set.

SQL> CREATE TABLE Primes (p INT);

Table created.

SQL> GRANT SELECT, UPDATE, INSERT ON Primes to pagh2;

Grant succeeded.

SQL> SELECT * FROM Primes;

no rows selected

SQL> INSERT INTO Primes VALUES (41);

1 row created.

SQL> SELECT * FROM Primes;

         P
----------
        41

SQL> COMMIT;

Commit complete.
```

User User2

```
SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

Transaction set.

SQL> SELECT * FROM user1.Primes;

no rows selected

SQL> INSERT INTO user1.Primes VALUES (43);

1 row created.

SQL> SELECT * FROM user1.Primes;

         P
----------
        43
```

SQL> COMMIT;

Commit complete.

SQL> SELECT * FROM user1.Primes;

```
         P
----------
        41
        43
```

SQL> SELECT * FROM user1.Primes;

```
         P
----------
        41
        43
```

SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

Transaction set.

SQL> INSERT INTO Primes VALUES (2);

1 row created.

SQL> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

Transaction set.

SQL> SELECT * FROM user1.Primes;

```
         P
----------
        41
        43
```

SQL> INSERT INTO user1.Primes VALUES (2003);

1 row created.

SQL> SELECT * FROM user1.Primes;

```
                                                    P
                                              ----------
                                                   41
                                                   43
                                                 2003

SQL> COMMIT;

Commit complete.

                                              SQL> SELECT * FROM user1.Primes;

                                                    P
                                              ----------
                                                   41
                                                   43
                                                    2
SQL> SELECT * FROM Primes;                       2003

      P
----------
      41
      43
       2

                                              SQL> COMMIT;

                                              Commit complete.


SQL> SELECT * FROM Primes;

      P
----------
      41
      43
       2
    2003

SQL> INSERT INTO Primes VALUES (3);

1 row created.

SQL> ROLLBACK;

Rollback complete.

SQL> SELECT * FROM Primes;
```

Exercise

ROLLBACK AND COMMIT

Create a table named "T", having two columns: id (of type integer, the primary key), s (of type character string with a length varying from 1 to 40 characters)

- Insert 5 rows to the newly created table
- Select * from T;
- Observe the data in Table.
- ROLLBACK
- Select * from T;
- Observe the data in Table.
- SET AUTOCOMMIT ON;
- Insert 5 rows again in table.

Transaction Isolation:

STEP 1:
Create a table TAB with an attribute tno.
STEP 2:
Create a table t_log with attribute t_count.
STEP 3:
Implement a PL/SQL procedure p1 that writes into a table t the numbers 1 to 100 (each in a separate tuple).
STEP 4:
Write a procedure p2 that counts 5 times the number of tuples in it.
STEP 5:
Inserts each of the counting results into a tuple in a log table.
STEP 6:
Set isolation transaction mode to read committed with the following command.
Set transaction isolation level read committed

STEP 7:

Execute p1 and p2 concurrently on two pc's separately, where both transactions are
in read committed isolation.

STEP 8:

Check the content of the log table. How do you justify the obtained result??