UNIT III - SQL

PRASHANT TOMAR

Simple Queries in SQL

- SQL stands for **Structured Query Language**, and it is used to communicate with the Database. This is a standard language used to perform tasks such as retrieval, updation, insertion and deletion of data from a database.
- Structured Query Language (SQL) is a domain-specific language used in programming and designed for managing data held in a relational database management system (RDBMS), or for stream processing in a relational data stream management system (RDSMS). It is particularly useful in handling structured data where there are relations between different entities/variables of the data.

- All the SQL statements start with any of the keywords like SELECT, INSERT, UPDATE, DELETE, ALTER, DROP, CREATE, USE, SHOW and all the statements end with a semicolon (;).
- The most important point to be noted here is that SQL is case insensitive, which means SELECT and select have same meaning in SQL statements. Whereas, SQL makes difference in table names. So, if you are working with SQL, then you need to give table names as they exist in the database.
- The CREATE DATABASE statement is used to create a new SQL database.

CREATE DATABASE *databasename*;

- The DROP DATABASE statement is used to drop an existing SQL database.

```
DROP DATABASE databasename;
```

- Creating a basic table involves naming the table and defining its columns and each column's data type. The SQL **CREATE TABLE** statement is used to create a new table.

```
CREATE TABLE table_name
(
        column1 datatype [ NULL | NOT NULL ],
        column2 datatype [ NULL | NOT NULL ],
        ...
        PRIMARY KEY( one columns)
);
```

- For example:

```
CREATE TABLE CUSTOMERS
(
        ID   INT   NOT NULL,
        NAME VARCHAR (20) NOT NULL,
        AGE  INT  NOT NULL,
        ADDRESS  CHAR (25) NULL,
        SALARY   DECIMAL (18, 2) NULL,
        PRIMARY KEY (ID)
 );
```

Column Name	DataType	Allow NULL	KEY	Default Value
ID	INT	NOT NULL	PRIM	
NAME	VARCHAR (20)	NOT NULL		
AGE	INT	NOT NULL		
ADDRESS	CHAR (25)	NULL		
SALARY	DECIMAL (18,2)	NULL		

- The SQL **DROP TABLE** statement is used to remove a table definition and all the data, indexes, triggers, constraints and permission specifications for that table.

DROP TABLE table_name;

- SELECT DISTINCT *column1*, *column2, ...* FROM *table_name*;
- SELECT *column1*, *column2* FROM *table_name* ORDER BY *column1, column2, ...* ASC|DESC;
- SELECT MIN(*column_name*) FROM *table_name* WHERE *condition*;
- SELECT MAX(*column_name*) FROM *table_name* WHERE *condition*;
- SELECT COUNT(*column_name*) FROM *table_name* WHERE *condition*;
- SELECT AVG(*column_name*) FROM *table_name* WHERE *condition*;
- SELECT *column1, column2, ...*FROM *table_name* WHERE *columnN* LIKE *pattern*;

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that start with "a"
WHERE CustomerName LIKE '%a'	Finds any values that end with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%_%'	Finds any values that start with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that start with "a" and ends with "o"

Subquery

- A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause. A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.
- Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.
- There are a few rules that subqueries must follow −
- Subqueries must be enclosed within parentheses.

- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
- The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
- A subquery cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

```
SELECT column_name [, column_name ]
FROM   table1 [, table2 ] WHERE  column_name OPERATOR
(SELECT column_name [, column_name ] FROM table1 [, table2 ] [WHERE])
```

```
SELECT * FROM CUSTOMERS
WHERE ID IN (SELECT ID
FROM CUSTOMERS WHERE SALARY > 4500) ;
```

VIEW

- A VIEW in SQL is a logical subset of data from one or more tables. View is used to restrict data access. Views in SQL are kind of virtual tables. A view also has rows and columns as they are in a real table in the database.
- In a database, a view is the result set of a stored query on the data, which the database users can query just as they would in a persistent database collection object. This pre-established query command is kept in the database dictionary.
- We can create a view by selecting fields from one or more tables present in the database. A View can either have all the rows of a table or specific rows based on certain condition.

- Views, which are a type of virtual tables allow users to do the following −
  - Structure data in a way that users or classes of users find natural or intuitive.
  - Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
  - Summarize data from various tables which can be used to generate reports.
- Views can provide advantages over tables:
  - Views can represent a subset of the data contained in a table. Consequently, a view can limit the degree of exposure of the underlying tables to the outer world: a given user may have permission to query the view, while denied access to the rest of the base table.
  - Views can join and simplify multiple tables into a single virtual table.

- Views can act as aggregated tables, where the database engine aggregates data (sum, average, etc.) and presents the calculated results as part of the data.
- Views can hide the complexity of data. For example, a view could appear as Sales2000 or Sales2001, transparently partitioning the actual underlying table.
- Views take very little space to store; the database contains only the definition of a view, not a copy of all the data that it presents.
- Depending on the SQL engine used, views can provide extra security.

CREATE VIEW ‘view_name’ AS SELECT statement;

create view emp_view as select emp_id, lname, location from emp;

Constraints

- Constraints are the rules enforced on the data columns of a table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.
- Constraints could be either on a column level or a table level. The column level constraints are applied only to one column, whereas the table level constraints are applied to the whole table.
- **Types of constraints**
  - **NOT NULL**
  - **UNIQUE**
  - **DEFAULT**
  - **CHECK**
  - **Key Constraints – PRIMARY KEY, FOREIGN KEY**

- **NOT NULL:**

NOT NULL constraint makes sure that a column does not hold NULL value. When we don't provide value for a particular column while inserting a record into a table, it takes NULL value by default. By specifying NULL constraint, we can be sure that a particular column(s) cannot have NULL values.

```
CREATE TABLE STUDENT(
ROLL_NO INT NOT NULL,
STU_NAME VARCHAR (35) NOT NULL,
STU_AGE INT NOT NULL,
STU_ADDRESS VARCHAR (235),
PRIMARY KEY (ROLL_NO)
);
```

- **UNIQUE:**

UNIQUE Constraint enforces a column or set of columns to have unique values. If a column has a unique constraint, it means that particular column cannot have duplicate values in a table.

```
CREATE TABLE STUDENT(
ROLL_NO INT NOT NULL,
STU_NAME VARCHAR (35) NOT NULL UNIQUE,
STU_AGE INT NOT NULL,
STU_ADDRESS VARCHAR (35) UNIQUE,
PRIMARY KEY (ROLL_NO)
);
```

- **DEFAULT:**

The DEFAULT constraint provides a default value to a column when there is no value provided while inserting a record into a table.

```
CREATE TABLE STUDENT(
ROLL_NO   INT  NOT NULL,
STU_NAME VARCHAR (35) NOT NULL,
STU_AGE INT NOT NULL,
EXAM_FEE INT  DEFAULT 10000,
STU_ADDRESS VARCHAR (35) ,
PRIMARY KEY (ROLL_NO)
);
```

- **CHECK:**

This constraint is used for specifying range of values for a particular column of a table. When this constraint is being set on a column, it ensures that the specified column must have the value falling in the specified range.

```
CREATE TABLE STUDENT(
ROLL_NO   INT  NOT NULL CHECK(ROLL_NO >1000) ,
STU_NAME VARCHAR (35)  NOT NULL,
STU_AGE INT  NOT NULL,
EXAM_FEE INT DEFAULT 10000,
STU_ADDRESS VARCHAR (35) ,
PRIMARY KEY (ROLL_NO)   );
```

- In the above example we have set the check constraint on **ROLL_NO** column of **STUDENT** table. Now, the **ROLL_NO** field must have the value greater than 1000.

- **PRIMARY KEY:**

Primary key uniquely identifies each record in a table. It must have unique values and cannot contain nulls. In the below example the **ROLL_NO** field is marked as primary key, that means the **ROLL_NO** field cannot have duplicate and null values.

```
CREATE TABLE STUDENT(
ROLL_NO   INT  NOT NULL,
STU_NAME VARCHAR (35)  NOT NULL UNIQUE,
STU_AGE INT NOT NULL,
STU_ADDRESS VARCHAR (35) UNIQUE,
CONSTRAINT PK_ STUDENT PRIMARY KEY (ROLL_NO)
);
```

FOREIGN KEY Constraint

- A FOREIGN KEY is a key used to link two tables together. A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table. The table containing the foreign key is called the child table, and the table containing the candidate key is called the referenced or parent table. Look at the following two tables:

PersonID	LastName	FirstName	Age
1	Hansen	Ola	30
2	Svendson	Tove	23
3	Pettersen	Kari	20

OrderID	OrderNumber	PersonID
1	77895	3
2	44678	3
3	22456	2
4	24562	1

- Notice that the "PersonID" column in the "Orders" table points to the "PersonID" column in the "Persons" table. The "PersonID" column in the "Persons" table is the PRIMARY KEY in the "Persons" table.
- The "PersonID" column in the "Orders" table is a FOREIGN KEY in the "Orders" table. The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.
- The FOREIGN KEY constraint also prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

```
CREATE TABLE Orders (
   OrderID int NOT NULL PRIMARY KEY,
   OrderNumber int NOT NULL,
     PersonID int FOREIGN KEY REFERENCES Persons(PersonID)
     );


     CREATE TABLE Orders (
        OrderID int NOT NULL,
        OrderNumber int NOT NULL,
        PersonID int,
        PRIMARY KEY (OrderID),
        CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID)
        REFERENCES Persons(PersonID)
     );
```

Trigger

- A SQL trigger is a set of SQL statements stored in the database catalog. A SQL trigger is executed or fired whenever an event associated with a table occurs e.g., insert, update or delete.
- A trigger is a special kind of Stored Procedure or stored program that is automatically fired or executed when some event (insert, delete and update) occurs.
- It is special because it is not called directly like a stored procedure. The main difference between a trigger and a stored procedure is that a trigger is called automatically when a data modification event is made against a table whereas a stored procedure must be called explicitly.

- DML triggers is a special type of stored procedure that automatically takes effect when a data manipulation language (DML) event takes place that affects the table or view defined in the trigger. DML events include INSERT, UPDATE, or DELETE statements.
- DML triggers can be used to enforce business rules and data integrity, query other tables, and include complex Transact-SQL statements. The trigger and the statement that fires it are treated as a single transaction, which can be rolled back from within the trigger. If a severe error is detected (for example, insufficient disk space), the entire transaction automatically rolls back.

- DML triggers are similar to constraints in that they can enforce entity integrity or domain integrity. In general, entity integrity should always be enforced at the lowest level by indexes that are part of PRIMARY KEY and UNIQUE constraints or are created independently of constraints. Domain integrity should be enforced through CHECK constraints, and referential integrity (RI) should be enforced through FOREIGN KEY constraints. DML triggers are most useful when the features supported by constraints cannot meet the functional needs of the application.
- The following list compares DML triggers with constraints and identifies when DML triggers have benefits over .
- DML triggers can cascade changes through related tables in the database; however, these changes can be executed more efficiently using cascading referential integrity constraints. FOREIGN KEY constraints can validate a column value only with an exact match to a value in another column, unless the REFERENCES clause defines a cascading referential action.

- They can guard against malicious or incorrect INSERT, UPDATE, and DELETE operations and enforce other restrictions that are more complex than those defined with CHECK constraints.
- Unlike CHECK constraints, DML triggers can reference columns in other tables. For example, a trigger can use a SELECT from another table to compare to the inserted or updated data and to perform additional actions, such as modify the data or display a user-defined error message.
- They can evaluate the state of a table before and after a data modification and take actions based on that difference.
- Multiple DML triggers of the same type (INSERT, UPDATE, or DELETE) on a table allow multiple, different actions to take place in response to the same modification statement.

- Constraints can communicate about errors only through standardized system error messages. If your application requires, or can benefit from, customized messages and more complex error handling, you must use a trigger.
- DML triggers can disallow or roll back changes that violate referential integrity, thereby canceling the attempted data modification.
- Such a trigger might go into effect when you change a foreign key and the new value does not match its primary key. However, FOREIGN KEY constraints are usually used for this purpose.
- If constraints exist on the trigger table, they are checked after the INSTEAD OF trigger execution but prior to the AFTER trigger execution. If the constraints are violated, the INSTEAD OF trigger actions are rolled back and the AFTER trigger is not executed.

AFTER TRIGGERS

- These triggers run after an insert, update or delete on a table. They are **not supported for views.**

AFTER TRIGGERS can be classified further into three types as:

- AFTER INSERT Trigger
- AFTER UPDATE Trigger
- AFTER DELETE Trigger

Let’s create After triggers. First of all, let’s create a table and insert some sample data. Then, on this table, I will be attaching several triggers.

```
CREATE TRIGGER <Schema_Name>.<Trigger_Name, sysname, Trigger_Name>

ON  <Schema_Name, sysname, Schema_Name>.<Table_Name, sysname, Table_Name>

AFTER <Data_Modification_Statements, , INSERT,DELETE,UPDATE>

AS

BEGIN

    -- Insert statements for trigger here

END
GO
```

-- Trigger on an INSERT, UPDATE, or DELETE statement to a table or view (DML Trigger) SQL Server Syntax

```
CREATE [ OR ALTER ] TRIGGER [ schema_name . ]trigger_name

ON { table | view }

{ FOR | AFTER | INSTEAD OF } { [ INSERT ] , [ UPDATE ] , [ DELETE ] }

AS
{ sql_statement  [ ; ] [ ,...n ] | EXTERNAL NAME <method specifier [ ; ] > }

<dml_trigger_option> ::=
    [ ENCRYPTION ]
    [ EXECUTE AS Clause ]

<method_specifier> ::=
    assembly_name.class_name.method_name
```

```
CREATE TRIGGER trg_AfterInsert ON [dbo].[Employee_Test]
FOR INSERT
AS
    declare @empid int;
    declare @empname varchar(100);
    declare @empsal decimal(10,2);
    declare @audit_action varchar(100);
    select @empid = i.Emp_ID from inserted i;
    select @empname = i.Emp_Name from inserted i;
    select @empsal = i.Emp_Sal from inserted i;
    set @audit_action = 'Inserted Record -- After Insert Trigger.';

insert into Employee_Test_Audit
(Emp_ID, Emp_Name, Emp_Sal, Audit_Action, Audit_Timestamp)
Values
(@empid, @empname, @empsal, @audit_action, getdate());

PRINT 'AFTER INSERT trigger fired.'
GO
```

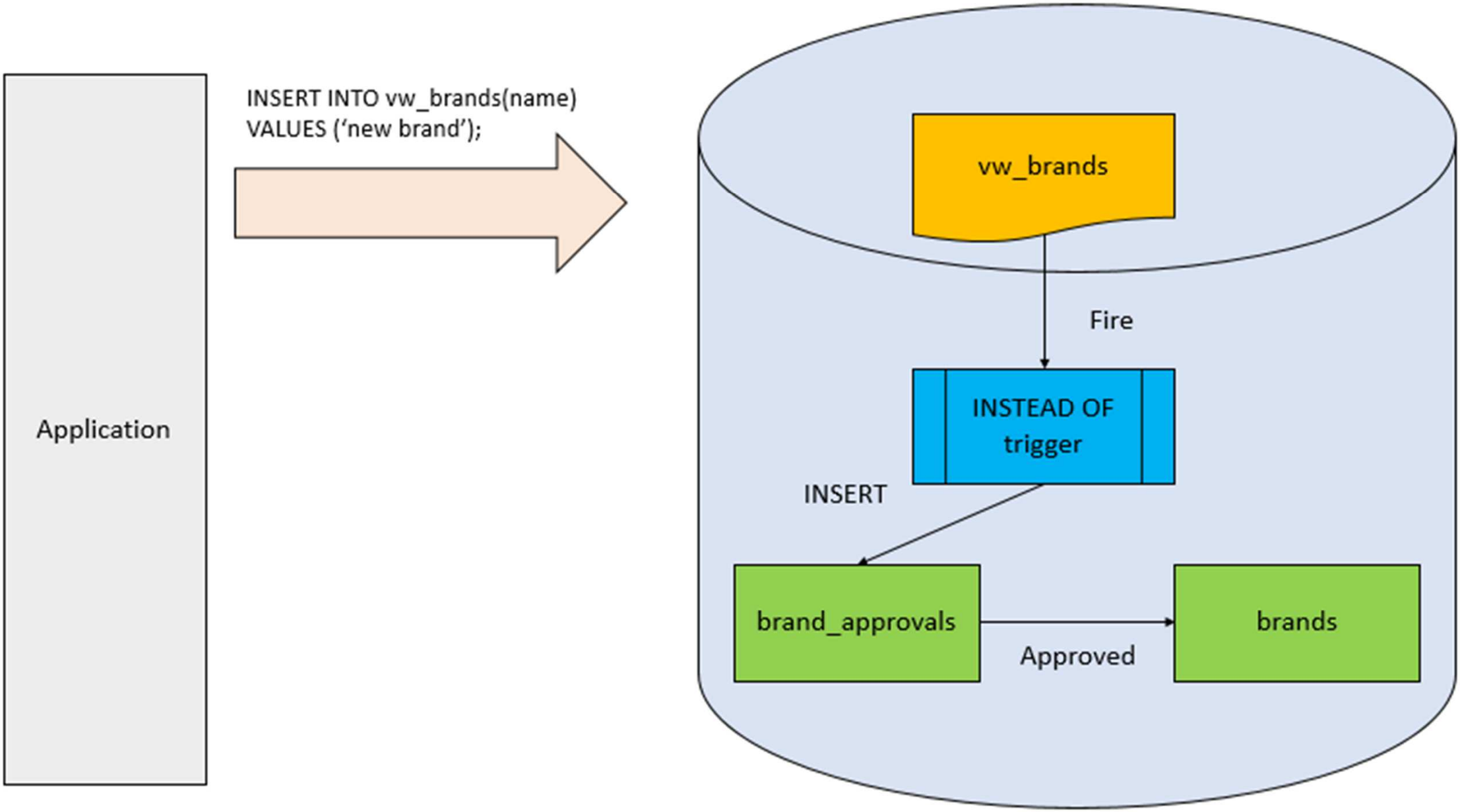
Instead of Trigger

- An INSTEAD OF trigger is a trigger that allows you to skip an INSERT, DELETE, or UPDATE statement to a table or a view and execute other statements defined in the trigger instead. The actual insert, delete, or update operation does not occur at all. In other words, an INSTEAD OF trigger skips a DML statement and execute other statements.
- These kinds of triggers fire before the execution of an action query that can only be DML statements like Insert, Update and Delete but after the execution of that query. The table data will not be affected, in other words if you want to insert or update the data of the table then you need to write it in the trigger using "inserted" or "deleted" virtual tables.

```
CREATE TRIGGER [schema_name.] trigger_name
ON {table_name | view_name }
INSTEAD OF {[INSERT] [,] [UPDATE] [,] [DELETE] }
AS
        {sql_statements}
```

- First, specify the name of the trigger and optionally the name of the schema to which the trigger belongs in the CREATE TRIGGER clause.
- Second, specify the name of the table or view which the trigger associated with.
- Third, specify an event such as INSERT, DELETE, or UPDATE which the trigger will fire in the INSTEAD OF clause. The trigger may be called to respond to one or multiple events.
- Fourth, place the trigger body after the AS keyword. A trigger's body may consist of one or more Transact-SQL statements.

- A typical example of using an INSTEAD OF trigger is to override an insert, update, or delete operation on a view.
- Suppose, an application needs to insert new brands into the **production.brands** table. However, the new brands should be stored in another table called **production.brand_approvals** for approval before inserting into the **production.brands** table.
- To accomplish this, you create a view called **production.vw_brands** for the application to insert new brands. If brands are inserted into the view, an INSTEAD OF trigger will be fired to insert brands into the **production.brand_approvals** table.


Application
INSERT INTO vw_brands(name)
VALUES ('new brand');
vw_brands
Fire
INSTEAD OF
trigger
INSERT
brand_approvals
Approved
brands

```
CREATE TABLE production.brand_approvals(
brand_id INT IDENTITY PRIMARY KEY,
brand_name VARCHAR(255) NOT NULL );
```

- Create View:

```
CREATE VIEW production.vw_brands
AS
SELECT brand_name, 'Approved' approval_status FROM production.brands
UNION
SELECT brand_name, 'Pending Approval' approval_status FROM
production.brand_approvals;
```

- Once a row is inserted into the **production.vw_brands** view, we need to route it to the **production.brand_approvals** table via the following **INSTEAD OF trigger**:

```
CREATE TRIGGER production.trg_vw_brands  ON production.vw_brands
INSTEAD OF INSERT
AS
BEGIN
   SET NOCOUNT ON;
   INSERT INTO production.brand_approvals (brand_name)
   SELECT i.brand_name FROM inserted i
   WHERE i.brand_name NOT IN (SELECT brand_name FROM production.brands);
END
```

Java Database Connectivity

- There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:
- Register the Driver class
- Create connection
- Create statement
- Execute queries
- Close connection

Register the driver class

```
import java.sql.* ;  // for standard JDBC programs
import java.math.* ; // for BigDecimal and BigInteger support
```

- The most common approach to register a driver is to use Java's Class.forName() method, to dynamically load the driver's class file into memory, which automatically registers it. This method is preferable because it allows you to make the driver registration configurable and portable.

```
try {
  Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch(ClassNotFoundException ex) {
  System.out.println("Error: unable to load driver class!");
  System.exit(1);
}
```

Create the connection object

- You should use the *registerDriver()* method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

```
try {
        Driver myDriver = new oracle.jdbc.driver.OracleDriver();
        DriverManager.registerDriver( myDriver );
    }
catch(ClassNotFoundException ex) {
        System.out.println("Error: unable to load driver class!");
        System.exit(1);
}
```

```
import java.sql.*;
class OracleCon{
public static void main(String args[]){
try{
        //step1 load the driver class
        Class.forName("oracle.jdbc.driver.OracleDriver");
        //step2 create  the connection object
        Connection con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
        //step3 create the statement object
        Statement stmt = con.createStatement();
        //step4 execute query
        ResultSet rs = stmt.executeQuery("select * from emp");
        while(rs.next())
                System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));

        //step5 close the connection object
        con.close();
  }
catch(Exception e){ System.out.println(e);}
  }
}
```