What is the Basic Structure of SQL Queries?

The fundamental structure of SQL queries includes three clauses that are select, from, and where clause. What we want in the final result relation is specified in the **select** clause. Which relations we need to access to get the result is specified in **from** clause. How the relation must be operated to get the result is specified in the **where** clause.

$$\text{select } A_1, A_2, \ldots, A_n$$

$$\text{from } r_1, r_2, \ldots, r_m$$

$$\text{where } P;$$

- In the select clause, you have to specify the attributes that you want to see in the result relation
- In the from clause, you have to specify the list of relations that has to be accessed for evaluating the query.
- In the where clause involves a predicate that includes attributes of the relations that we have listed in the from clause.

Though the SQL query has a sequence select, from, and where. To understand how the query will operate? You must consider the query in the order, from, where and then focus on select.

So with the help of these three clauses, we can retrieve the information we want out of the huge set of data. Let us begin with the structure of queries imposed on the single relation.

Queries on Single Relation

Consider that we have a relation 'instructor' with the attributes instr_id, name, dept_name, and salary. Now we want the names of all the instructors along with their corresponding department names.

Instr_id	Name	Dept_name	Salary
101	Srinivasan	Comp. Sci.	65000
121	Wu	Finance	90000
151	Mozart	Music	40000
222	Einstein	Physics	95000
343	El Said	History	60000
456	Gold	Physics	87000
565	Katz	Comp. Sci.	75000
583	Cali Fieri	History	62000
543	Singh	Finance	80000
766	Crick	Biology	72000
821	Brandt	Comp. Sci.	92000
345	Kim	Elec. Eng.	80000

Figure 1. Instructor Relation

The SQL query we would structure to get a result relation with instructor's names along with their department name.

select name,

from instructor;

Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Cali Fieri
Singh
Crick
Brandt
Kim

Observe that here we have not included **where** clause in the query above as we want the name of all the instructors with their department name. So there is no need of imposing any condition.

Now, if we have asked only for the dept_name in the above query by default it would have listed names of all the departments retaining the duplicates. To eliminate the duplicates you can make use of the **distinct** keyword.

select distinct dept_name

from instructor;

Comp. Sci.
Finance
Music
Physics
History
Biology
Elec. Eng.

Further, you can even include the arithmetic expression in the select clause using operators such as +, -, *, and /. In case, you want the result relation to display instructor name along with their salary which reduced by 10%. Then the SQL query you will impose on the data set is:

select instr_name, salary*0.9

from instructor;

To get specific tuples in the result relation we can use the where clause to specify the particular condition. Like if we want the result relation to display names of instructors that have salaries greater than 50000.

In where clause we can make use of logical connectives and, or & not. Along with these, we can include expressions where we can use comparison operators to compare operands in the expression. This was a query imposed on a single relation now let's move ahead and discuss queries requested on multiple relations.

Queries on Multiple Relation

The SQL queries often need to access multiple relations from the data set in order to get the required result. Let us take an example we have two relations instructor and department.

Now, if you want to retrieve the names of all the instructors along with their department names and the corresponding department building. We will get the instructor's name and department name in the instructor relation but building name in the department relation. So the query would be:

select name, instructor.dept_name, building

from instructor, department

where instructor.dept_name= department.dept_name;

Name	Dept_name	Building
Srinivasan	Comp. Sci.	Taylor
Wu	Finance	Painter
Mozart	Music	Packard
Einstein	Physics	Watson
El Said	History	Painter
Gold	Physics	Watson
Katz	Comp. Sci.	Taylor
Cali Fieri	History	Painter
Singh	Finance	Painter
Crick	Biology	Watson
Brandt	Comp. Sci.	Taylor
Kim	Elec. Eng.	Taylor

Here department name of each tuple of instructor relation will be matched with the department name of each tuple of department relation. Observe

that we have used relation name as a prefix to the attribute name in where clause, as both the attributes to be compared in where clause has the same name. The result of the query above is:

The from the clause in the queries with multiple relations act as a Cartesian product of the relations present the from clause.

Consider the relation instructor and teaches and the Cartesian product between the two can be expressed as:

(instructor.ID, instructor.name, instructor.dept_name, instructor.salary, teaches.ID, teaches.course id, teaches.sec id, teaches.semester, teaches.year)

The prefix should not be added to the attributes that are only in one of the relations. So this would be like:

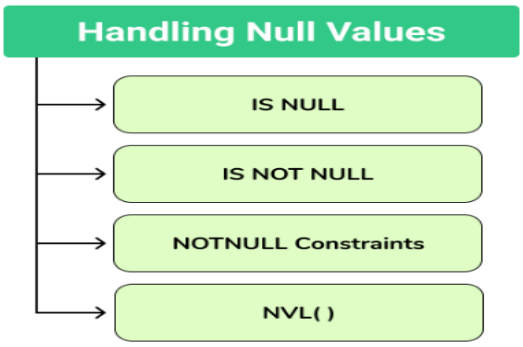
(instructor.ID, name, dept_name, salary

teaches.ID, course id, sec id, semester, year)

The Cartesian product combines each tuple of instructor relation to every tuple of teaches relation. This Cartesian product results in extremely large relations which are hardly of any use. So, it is the where clause that restricts the unnecessary combinations, and let's retain the meaningful combination that will help in retrieving the desired result.

Null Values in DBMS

- Special value that is supported by SQL is called as `null` which is used to represent values of attributes that are unknown or do not apply for that particular row
- For example age of a particular student is not available in the age column of student table then it is represented as null but not as zero
- It is important to know that ***null values is always different from zero value***
- A null value is used to represent ***the following different interpretations***
    - ***Value unknown*** (value exists but is not known)
    - ***Value not available*** (exists but is purposely hidden)
    - ***Attribute not applicable*** (undefined for that row)
- SQL provides special operators and functions to deal with data involving null values



Consider the sample table '*emp'*

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT	–	17-NOV-81	5000	–	10
7698	BLAKE	MANAGER	7839	01-MAY-81	2850	500	30
7782	CLARK	MANAGER	7839	09-JUN-81	2450	500	10
7566	JONES	MANAGER	7839	02-APR-81	2975	–	20
7788	SCOTT	ANALYST	7566	19-APR-87	3000	–	20
7902	FORD	ANALYST	7566	03-DEC-81	3000	–	20
7369	SMITH	CLERK	7902	17-DEC-80	800	500	20

IS NULL operator

All ***operations upon null values present in the table must be done using this 'is null' operator*** .we cannot compare null value using the assignment operator

Example

```
select * from emp
where comm is null
```

O/P

```
4 rows selected.
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT	–	17-NOV-81	5000	–	10
7566	JONES	MANAGER	7839	02-APR-81	2975	–	20

7788	SCOTT	ANALYST	7566	19-APR-87	3000	–	20
7902	FORD	ANALYST	7566	03-DEC-81	3000	–	20

The details of those employees whose commission value is Null are displayed.

IS NOT NULL

```
select * from emp
where comm is not null;
```

O/P

```
3 rows selected.
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7698	BLAKE	MANAGER	7839	01-MAY-81	2850	500	30
7782	CLARK	MANAGER	7839	09-JUN-81	2450	500	10
7369	SMITH	CLERK	7902	17-DEC-80	800	500	20

Details of all those employees whose Commission value is not null value are displayed.

NOT NULL Constraint

- Not all constraints prevents a column to contain null values
- Once ***not null is applied to a particular column, you cannot enter null values to that column*** and restricted to maintain only some proper value other than null
- A ***not-null constraint cannot be applied at table level***

Example

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

- In the above example, we have applied not null on three columns ID, name and age which means ***whenever a record is entered using insert statement all three columns should contain a value other than null***
- We have two other columns address and salary, ***where not null is not applied*** which means that ***you can leave the row as empty or use null value while inserting the record into the table.***

NVL() NULL Function

- Using NVL function you can ***substitute a value in the place of NULL values.***
- The substituted value then ***temporarily replaces the NULL values in your calculations or expression***. Remember that the substituted value ***only replaces the NULL value temporarily*** for the session and ***does not affect the value stored in the table.***
- Here is the syntax of NVL function.

```
NVL (exp, replacement-exp)
```

- As you can see NVL function ***takes two parameters exp and replacement exp***. ***First parameter exp can be a column name of a table or an arithmetic expression*** and the ***second parameter replacement expression will be the value which you want to substitute*** when a NULL value is encountered.
- Always remember the ***data type of both the parameters must match otherwise*** the compiler will raise an error.

Example

```
SELECT NVL (comm, 500) FROM employees
WHERE  salary>1000;
```

- On execution ***all the null values in the result set will get replaced by 500.***
- Similarly we can use NVL null function while performing arithmetic expression.
- Again let's take the same arithmetic expression which we used in the previous query where we added 100 to the values of commission column.

```
SELECT NVL(comm,100), NVL(comm,100)+100 FROM employees WHERE salary>1000;
```

Aggregate Operators

To **calculate aggregate values**, one requires some **aggregate operators** to perform this task. These operators run over the columns of a relation. The total number of five **aggregate operators** is supported by SQL and these are:

1. **COUNT –** To calculate the number of values present in a particular column.
2. **SUM –** To calculate the sum of all the values present in a particular column.
3. **AVG –** To calculate the average of all the values present in a particular column.
4. **MAX –** To find the maximum of all the values present in a particular column.
5. **MIN –** To find the minimum of all the values present in a particular column.

Mostly, with **COUNT**, **SUM**, and **AVG**, **DISTINCT** is specified in conjunction in order to eliminate any duplicate data. But for **MAX** and **MIN**, **DISTINCT** is not required to be specified as that will anyway not going to affect the output.

For example: Find the average salary of all the employees.

```
SELECT AVG (E.salary)
FROM Employee E
```

GROUP BY Clause

GROUP BY Clause is used to group the attributes with similar features under the given condition.

Let us consider a table of student.

ID	Name	Marks	Section
1	Shiv	89	A
2	Parth	78	B
3	Ankush	95	A
4	Nimish	83	B

Question: Find the highest marks of the student for each section.

To find the highest marks section wise, we need to write two queries as:

```
SELECT MAX (S.Marks)
FROM Student S
```

```
WHERE S.Section = 'A'
```

```
SELECT MAX (S.Marks)
FROM Student S
WHERE S.Section = 'B'
```

As such, if we have many sections, we have to write the query that number of time. This looks quite lengthy.

GROUP BY clause made the solution easier as we don't require writing the queries the number of times of section, Instead, we write:

```
SELECT MAX (S.Marks)
FROM Student S
GROUP BY S.Section
```

In this way, writing just one query, we will get the expected output.

ID	Name	Marks	Section
3	Ankush	95	A
4	Nimish	83	B

HAVING Clause

HAVING Clause determines whether the answer needs to be generated for a given condition or not.

Let us consider the previous example.

ID	Name	Marks	Section
1	Shiv	89	A
2	Parth	78	B
3	Ankush	95	A
4	Nimish	83	B

Question: Find the highest marks of the student for each section having marks greater than 90.

```
SELECT MAX (S.Marks)
FROM Student S
GROUP BY S.Section
HAVING S.Marks > 90
```

Thus, our output will be:

ID	Name	Marks	Section
3	Ankush	95	A

SQL Set Operation

The SQL Set operation is used to combine the two or more SQL SELECT statements.

Types of Set Operation

1. Union
2. UnionAll
3. Intersect
4. Minus

1. Union

- The SQL Union operation is used to combine the result of two or more SQL SELECT queries.
- In the union operation, all the number of datatype and columns must be same in both the tables on which UNION operation is being applied.
- The union operation eliminates the duplicate rows from its resultset.

Syntax

1. SELECT column_name FROM table1
2. UNION
3. SELECT column_name FROM table2;

Example:

The First table

ID	NAME
1	Jack
2	Harry
3	Jackson

The Second table

ID	NAME
3	Jackson
4	Stephan
5	David

Union SQL query will be:

```
1. SELECT * FROM First
2. UNION
3. SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
1	Jack
2	Harry
3	Jackson

4	Stephan
5	David

2. Union All

Union All operation is equal to the Union operation. It returns the set without removing duplication and sorting the data.

Syntax:

```
1. SELECT column_name FROM table1
2. UNION ALL
3. SELECT column_name FROM table2;
```

Example: Using the above First and Second table.

Union All query will be like:

```
1. SELECT * FROM First
2. UNION ALL
3. SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
1	Jack
2	Harry
3	Jackson
3	Jackson
4	Stephan
5	David

3. Intersect

- It is used to combine two SELECT statements. The Intersect operation returns the common rows from both the SELECT statements.
- In the Intersect operation, the number of datatype and columns must be the same.
- It has no duplicates and it arranges the data in ascending order by default.

Syntax

```
1. SELECT column_name FROM table1
2. INTERSECT
3. SELECT column_name FROM table2;
```

Example:

Using the above First and Second table.

Intersect query will be:

```
1. SELECT * FROM First
2. INTERSECT
3. SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
3	Jackson

4. Minus

- It combines the result of two SELECT statements. Minus operator is used to display the rows which are present in the first query but absent in the second query.
- It has no duplicates and data arranged in ascending order by default.

Syntax:

```
1. SELECT column_name FROM table1
2. MINUS
3. SELECT column_name FROM table2;
```

Example

Using the above First and Second table.

Minus query will be:

```
1. SELECT * FROM First
2. MINUS
3. SELECT * FROM Second;
```

The resultset table will look like:

ID	NAME
1	Jack
2	Harry

Nested queries in DBMS

A nested query is a query that has another query embedded within it. The embedded query is called a subquery.

A subquery typically appears within the WHERE clause of a query. It can sometimes appear in the FROM clause or HAVING clause.

Example

Let's learn about nested queries with the help of an example.

Find the names of employee who have regno=103

The query is as follows −

```
select E.ename from employee E where E.eid IN (select S.eid from salary S where
S.regno=103);
```

Student table

The student table is created as follows −

```
create table student(id number(10), name varchar2(20),classID number(10), marks
varchar2(20));
Insert into student values(1,'pinky',3,2.4);
Insert into student values(2,'bob',3,1.44);
Insert into student values(3,'Jam',1,3.24);
Insert into student values(4,'lucky',2,2.67);
Insert into student values(5,'ram',2,4.56);
select * from student;
```

Output

You will get the following output −

Id	Name	classID	Marks
1	Pinky	3	2.4
2	Bob	3	1.44
3	Jam	1	3.24
4	Lucky	2	2.67
5	Ram	2	4.56

Teacher table

The teacher table is created as follows −

Example

```
Create table teacher(id number(10), name varchar(20), subject varchar2(10),
classID number(10), salary number(30));
Insert into teacher values(1,'bhanu','computer',3,5000);
Insert into teacher values(2,'rekha','science',1,5000);
Insert into teacher values(3,'siri','social',NULL,4500);
Insert into teacher values(4,'kittu','mathsr',2,5500);
select * from teacher;
```

Output

You will get the following output −

Id	Name	Subject	classID	Salary
1	Bhanu	Computer	3	5000
2	Rekha	Science	1	5000
3	Siri	Social	NULL	4500
4	Kittu	Maths	2	5500

Class table

The class table is created as follows −

Example

```
Create table class(id number(10), grade number(10), teacherID number(10),
noofstudents number(10));
insert into class values(1,8,2,20);
insert into class values(2,9,3,40);
insert into class values(3,10,1,38);
select * from class;
```

Output

You will get the following output −

Id	Grade	teacherID	No.ofstudents
1	8	2	20
2	9	3	40
3	10	1	38

Now let's work on nested queries

Example 1

```
Select AVG(noofstudents) from class where teacherID IN(
Select id from teacher
Where subject='science' OR subject='maths');
```

Output

You will get the following output −

```
20.0
```

Example 2

```
SELECT * FROM student
WHERE classID = (
   SELECT id
   FROM class
   WHERE noofstudents = (
      SELECT MAX(noofstudents)
      FROM class));
```

Output

You will get the following output −

```
4|lucky |2|2.67
```

```
5|ram   |2|4.56
```

Triggers in PL/SQL

Triggers in oracle are blocks of PL/SQL code which oracle engine can execute automatically based on some action or event.

These events can be:

- DDL statements (CREATE, ALTER, DROP, TRUNCATE)
- DML statements (INSERT, SELECT, UPDATE, DELETE)
- Database operation like connecting or disconnecting to oracle (LOGON, LOGOFF, SHUTDOWN)

Triggers are automatically and repeatedly called upon by oracle engine on satisfying certain condition.

Triggers can be activated or deactivated depending on the requirements.

If triggers are activated then they are executed implicitly by oracle engine and if triggers are deactivated then they are executed explicitly by oracle engine.

PL/SQL: Uses of Triggers

Here we have mentioned a few use cases where using triggers proves very helpful:

- Maintaining complex constraints which is either impossible or very difficult via normal constraint(like primary, foreign, unique etc) applying technique.
- Recording the changes made on the table.
- Automatically generating primary key values.
- Prevent invalid transactions to occur.
- Granting authorization and providing security to database.
- Enforcing referential integrity.

PL/SQL: Parts of a Trigger

Whenever a trigger is created, it contains the following three sequential parts:

- **Triggering Event or Statement:** The statements due to which a trigger occurs is called triggering event or statement. Such statements can be DDL statements, DML statements or any database operation, executing which gives rise to a trigger.
- **Trigger Restriction:** The condition or any limitation applied on the trigger is called trigger restriction. Thus, if such a condition is **TRUE** then trigger occurs otherwise it does not occur.
- **Trigger Action:** The body containing the executable statements that is to be executed when trigger occurs that is with the execution of Triggering statement and upon evaluation of Trigger restriction as **True** is called Trigger Action.

PL/SQL: Types of Triggers

The above diagram clearly indicated that Triggers can be classified into three categories:

1. Level Triggers
2. Event Triggers
3. Timing Triggers

which are further divided into different parts.

Level Triggers

There are 2 different types of level triggers, they are:

1. **ROW LEVEL TRIGGERS**
    - It fires for every record that got affected with the execution of DML statements like INSERT, UPDATE, DELETE etc.
    - It always use a FOR EACH ROW clause in a triggering statement.

2. **STATEMENT LEVEL TRIGGERS**
    - It fires once for each statement that is executed.

Event Triggers

There are 3 different types of event triggers, they are:

1. **DDL EVENT TRIGGER**
    - It fires with the execution of every DDL statement(CREATE, ALTER, DROP, TRUNCATE).
2. **DML EVENT TRIGGER**
    - It fires with the execution of every DML statement(INSERT, UPDATE, DELETE).
3. **DATABASE EVENT TRIGGER**
    - It fires with the execution of every database operation which can be LOGON, LOGOFF, SHUTDOWN, SERVERERROR etc.

Timing Triggers

There are 2 different types of timing triggers, they are:

- **BEFORE TRIGGER**
    - It fires before executing DML statement.
    - Triggering statement may or may not executed depending upon the before condition block.
- **AFTER TRIGGER**
    - It fires after executing DML statement.

Syntax for creating Triggers

Following is the syntax for creating a trigger:

```
CREATE OR REPLACE TRIGGER <trigger_name>
```

```
BEFORE/AFTER/INSTEAD OF

      INSERT/DELETE/UPDATE ON <table_name>

REFERENCING (OLD AS O, NEW AS N)

      FOR EACH ROW WHEN (test_condition)

DECLARE

      -- Variable declaration;

BEGIN

      -- Executable statements;

EXCEPTION

      -- Error handling statements;

END <trigger_name>;

END;
```

Copy

where,

CREATE OR REPLACE TRIGGER is a keyword used to create a trigger and **<trigger_name>** is user-defined where a trigger can be given a name.

BEFORE/AFTER/INSTEAD OF specify the timing of the trigger's occurance. INSTEAD OF is used when a view is created.

INSERT/UPDATE/DELETE specify the DML statement.

<table_name> specify the name of the table on which DML statement is to be applied.

REFERENCING is a keyword used to provide reference to old and new values for DML statements.

FOR EACH ROW is the clause used to specify row level tigger.

WHEN is a clause used to specify condition to be applied and is only applicable for row-level trigger.

DECLARE, BEGIN, EXCEPTION, END are the different sections of PL/SQL code block containing variable declaration, executable statements, error handling statements and marking end of PL/SQL block respectively where DECLARE and EXCEPTION part are optional.

Time for an Example!

Below we have a simple program to demonstrate the use of Triggers in PL/SQL code block.

```
CREATE OR REPLACE TRIGGER CheckAge

BEFORE

INSERT OR UPDATE ON student

FOR EACH ROW

BEGIN

      IF :new.Age>30 THEN

            raise_application_error(-20001, 'Age should not be
greater than 30');

      END IF;

END;
```

Copy

```
Trigger created.
```

Following is the STUDENT table,

ROLLNO	SNAME	AGE	COURSE
11	Anu	20	BSC
12	Asha	21	BCOM
13	Arpit	18	BCA
14	Chetan	20	BCA
15	Nihal	19	BBA

After initializing the trigger CheckAge, whenever we will insert any new values or update the existing values in the above table STUDENT our trigger will check the **age** before executing INSERT or UPDATE statements and according to the result of triggering restriction or condition it will execute the statement.

Let's take a few examples and try to understand this,

Example 1:

```
INSERT into STUDENT values(16, 'Saina', 32, 'BCOM');
```

Copy

```
Age should not be greater than 30
```

Example 2:

```
INSERT into STUDENT values(17, 'Anna', 22, 'BCOM');
```

Copy

```
1 row created
```

Example 3:

```
UPDATE STUDENT set age=31 where ROLLNO=12;
```

Copy

```
Age should not be greater than 30
```

Example 4:

```
 UPDATE STUDENT set age=23 where ROLLNO=12;
```

Copy

```
1 row updated.
```

Example

```
1. CREATE OR REPLACE TRIGGER display_salary_changes
2. BEFORE DELETE OR INSERT OR UPDATE ON customers
3. FOR EACH ROW
4. WHEN (NEW.ID > 0)
5. DECLARE
6.   sal_diff number;
7. BEGIN
8.   sal_diff := :NEW.salary  - :OLD.salary;
9.   dbms_output.put_line('Old salary: ' || :OLD.salary);
10.  dbms_output.put_line('New salary: ' || :NEW.salary);
11.  dbms_output.put_line('Salary difference: ' || sal_diff);
12. END;
13. /
```

After the execution of the above code at SQL Prompt, it produces the following result.

```
Trigger created.
```

Check the salary difference by procedure:

Use the following code to get the old salary, new salary and salary difference after the trigger created.

```
1.  DECLARE
2.    total_rows number(2);
3.  BEGIN
4.    UPDATE  customers
5.    SET salary = salary + 5000;
6.    IF sql%notfound THEN
7.      dbms_output.put_line('no customers updated');
8.    ELSIF sql%found THEN
9.      total_rows := sql%rowcount;
10.     dbms_output.put_line( total_rows || ' customers updated ');
11.   END IF;
12. END;
13. /
```

Output:

```
Old salary: 20000
New salary: 25000
Salary difference: 5000
Old salary: 22000
New salary: 27000
Salary difference: 5000
Old salary: 24000
New salary: 29000
Salary difference: 5000
Old salary: 26000
New salary: 31000
Salary difference: 5000
Old salary: 28000
New salary: 33000
Salary difference: 5000
Old salary: 30000
New salary: 35000
Salary difference: 5000
6 customers updated
```

Note: As many times you executed this code, the old and new both salary is incremented by 5000 and hence the salary difference is always 5000.

After the execution of above code again, you will get the following result.

```
Old salary: 25000
New salary: 30000
Salary difference: 5000
Old salary: 27000
New salary: 32000
Salary difference: 5000
Old salary: 29000
New salary: 34000
Salary difference: 5000
```

```
Old salary: 31000
New salary: 36000
Salary difference: 5000
Old salary: 33000
New salary: 38000
Salary difference: 5000
Old salary: 35000
New salary: 40000
Salary difference: 5000
6 customers updated
```

Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters −

```
Select * from customers;
```

```
+----+----------+-----+-----------+----------+
| 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 |
+----+----------+-----+-----------+----------+
```

The following program creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values −

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
   sal_diff number;
BEGIN
   sal_diff := :NEW.salary  - :OLD.salary;
   dbms_output.put_line('Old salary: ' || :OLD.salary);
   dbms_output.put_line('New salary: ' || :NEW.salary);
   dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary:
New salary: 7500
Salary difference:
```

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table −

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

When a record is updated in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```