

UNIT 3

Advances SQL and PL/SQL

Structured Query Language (SQL): Select Commands, Union, Intersection, Except, Nested Queries, Aggregate Operators, Null values, Relational set operators, SQL join operators

Relational Algebra(RA): Selection, Projection, Set operations, Joins

Relational Calculus (TRC, DRC): Tuple Relational Calculus, Domain Relational Calculus

PL/SQL, Assertions, Triggers

Set Operators

One of the characteristics of RDBMS is that it should support all the transaction on the records in the table by means relational operations.

- That means it should have strong query language which supports relational algebra.
- There are three main relational algebras on sets – UNION, SET DIFFERENCE and SET INTERSECT. The same is implemented in database query language using set operators.
- Relational set operators are used to combine or subtract the records from two tables.
- These operators are used in the SELECT query to combine the records or remove the records.
- In order to set operators to work in database, it should have same number of columns participating in the query and the datatypes of respective columns should be same. This is called Union Compatibility.
- The resulting records will also have same number of columns and same datatypes for the respective column.

• UNION

- It combines the similar columns from two tables into one resultant table. All columns that are participating in the UNION operation should be Union Compatible.
- This operator combines the records from both the tables into one. If there are duplicate values as a result, then it eliminates the duplicate. The resulting records will be from both table and distinct.
- Suppose we have to see the employees in EMP_TEST and EMP_DESIGN tables. If we are using UNION, then it will combine both the results from tables in to one set.

SELECT EMP_ID, EMP_NAME, EMP_ADDRESS, EMP_SSN FROM EMP_TEST

UNION

SELECT EMP_ID, EMP_NAME, EMP_ADDRESS, EMP_SSN FROM EMP_DESIGN;

EMP_TEST			
EMP_ID	EMP_NAME	EMP_ADDRESS	EMP_SSN
100	James	Troy	232434
104	Kathy	Holland	324343

Union



EMP_DESIGN			
EMP_ID	ENAME	EMP_ADDRESS	SSN
103	Rose	Freser Town	6744545
102	Marry	Novi	343613
105	Laurry	Rochester Hills	97676
104	Kathy	Holland	324343

UNION			
EMP_ID	EMP_NAME	EMP_ADDRESS	EMP_SSN
100	James	Troy	232434
102	Marry	Novi	343613
103	Rose	Freser Town	6744545
104	Kathy	Holland	324343
105	Laurry	Rochester Hills	97676

- We can notice that Result will have same column names as first query. Duplicate record – 104 from EMP_TEST and EMP_DESIGN are showed only once in the result set. Records are sorted in the result.

• UNION ALL

- This operation is also similar to UNION, but it does not eliminate the duplicate records. It shows all the records from both the tables.

- All other features are same as UNION. We can have conditions in the SELECT query. It need not be a simple SELECT query.

Look at the same example below with UNION ALL operation.

SELECT EMP_ID, EMP_NAME, EMP_ADDRESS, EMP_SSN FROM EMP_TEST

UNION ALL

SELECT EMP_ID, EMP_NAME, EMP_ADDRESS, EMP_SSN FROM EMP_DESIGN;

EMP_TEST			
EMP_ID	EMP_NAME	EMP_ADDRESS	EMP_SSN
100	James	Troy	232434
104	Kathy	Holland	324343

Union ALL



EMP_DESIGN			
EMP_ID	ENAME	EMP_ADDRESS	SSN
103	Rose	Freser Town	6744545
102	Marry	Novi	343613
105	Laurry	Rochester Hills	97676
104	Kathy	Holland	324343

UNION			
EMP_ID	EMP_NAME	EMP_ADDRESS	EMP_SSN
100	James	Troy	232434
102	Marry	Novi	343613
103	Rose	Freser Town	6744545
104	Kathy	Holland	324343
104	Kathy	Holland	324343
105	Laurry	Rochester Hills	97676

• INTERSECT

- This operator is used to pick the records from both the tables which are common to them. In other words it picks only the duplicate records from the tables.

- Even though it selects duplicate records from the table, each duplicate record will be displayed only once in the result set. It should have UNION Compatible columns to run the query with this operator.

Same example above when used with INTERSECT operator, gives below result.

SELECT EMP_ID, EMP_NAME, EMP_ADDRESS, EMP_SSN FROM EMP_TEST

INTERSECT

SELECT EMP_ID, EMP_NAME, EMP_ADDRESS, EMP_SSN FROM EMP_DESIGN;

- We have INTERSECT ALL operator too. But it is same as INTERSET. There is no difference between them like we have between UNION and UNION ALL.

SQL - EXCEPT Clause

The SQL EXCEPT clause/operator is used to combine two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement. This means EXCEPT returns only rows, which are not available in the second SELECT statement.

Just as with the UNION operator, the same rules apply when using the EXCEPT operator. MySQL does not support the EXCEPT operator.

Syntax

The basic syntax of EXCEPT is as follows.

```
SELECT column1 [, column]
```

```
FROM table1 [, table2]
```

```
[WHERE condition]
```

```
EXCEPT
```

```
SELECT column1 [, column2]
```

```
FROM table1 [, table2]
```

```
[WHERE condition]
```

Here, the given condition could be any given expression based on your requirement.

Example

Consider the following two tables.

Table 1 – CUSTOMERS Table is as follows.

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

Table 2 – ORDERS table is as follows.

OID	DATE	CUSTOMER_ID	AMOUNT
102	2009-10-08 00:00:00	3	3000
100	2009-10-08 00:00:00	3	1500
101	2009-11-20 00:00:00	2	1560
103	2008-05-20 00:00:00	4	2060

Now, let us join these two tables in our SELECT statement as shown below.

```
SQL> SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
EXCEPT
SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
RIGHT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

```
+---+-----+-----+-----+
| ID | NAME | AMOUNT | DATE |
+---+-----+-----+-----+
| 1  | Ramesh | NULL   | NULL |
| 5  | Hardik | NULL   | NULL |
| 6  | Komal  | NULL   | NULL |
| 7  | Muffy  | NULL   | NULL |
+---+-----+-----+-----+
```

```
SQL> select * from emp_design;

      EID ENAME          AGE JOIN_DATE      SALARY JOB
-----
      10 aaa             28 01-SEP-95      50000 manager
      20 bbb             25 01-SEP-01      20000 admin
      30 ccc             22 01-MAR-03      28000 developer

SQL> commit;

Commit complete.
```

```
SQL> select * from emp_test;

      EID ENAME          AGE JOIN_DATE      SALARY JOB
-----
      20 bbb             25 01-SEP-01      20000 admin
      30 ccc             25 01-JUL-03      28000 developer

SQL> commit;
```

```
SQL> select * from emp_design
2 union
3 select * from emp_test;
```

EID	ENAME	AGE	JOIN_DATE	SALARY	JOB
10	aaa	28	01-SEP-95	50000	manager
20	bbb	25	01-SEP-01	20000	admin
30	ccc	22	01-MAR-03	28000	developer
30	ccc	25	01-JUL-03	28000	developer

```
SQL> select * from emp_design
2 intersect
3 select * from emp_test;
```

EID	ENAME	AGE	JOIN_DATE	SALARY	JOB
20	bbb	25	01-SEP-01	20000	admin

```
SQL> select * from emp_design
2 minus
3 select * from emp_test;
```

EID	ENAME	AGE	JOIN_DATE	SALARY	JOB
10	aaa	28	01-SEP-95	50000	manager
30	ccc	22	01-MAR-03	28000	developer

Sub Queries

- Sub query is inner query or nested query where the inner query is executed first and its result is used in the outer query. Sub queries will be executed only once. It has some of the following features:
- Sub queries can be used with SELECT, INSERT, UPDATE and DELETE statements.

1. sub queries in the INSERT statement :

- to insert one-month older record into a backup of student table STD_BKP, by selecting the CLASS_IDs STUDENTS who are inserted into STUDENT one month ago.

```
INSERT INTO STD_BKP
```

```
SELECT STD_ID, STD_NAME, ADDRESS, CLASS_ID
```

```
FROM STUDENT
```

```
WHERE CLASS_ID
```

```
IN
```

```
(SELECT CLASS_ID
```

```
FROM STUDENT
```

```
WHERE CREATED_DATE<= SYSDATE-30);
```

2. sub queries in the UPDATE statement :

- to update the salary of DESIGN employees by 10%.

```
UPDATE EMPLOYEE  
SET SALARY = SALARY + (SALARY * 0.1)  
WHERE DEPT_ID = (SELECT DEPT_ID FROM DEPT  
WHERE DEPT_NAME = 'DESIGN');
```

3. sub queries in the DELETE statement :

- to delete DESIGN employees from EMPLOYEE table.

```
DELETE FROM EMPLOYEE  
WHERE DEPT_ID = (SELECT DEPT_ID FROM DEPT  
WHERE DEPT_NAME = 'DESIGN');
```

- We can use sub queries to compare the results using =, <, >, >=, <=, IN, BETWEEN etc operators in the WHERE clause.
- Inner queries are enclosed inside a parenthesis.
- It should return only one row of data. If there is more than one row are returned by sub query then IN operator has to be used.
- In most of the cases only one column is used in inner SELECT statement.

4. sub queries in the SELECT statement :

To display the details of all employees whose salary is equal to minimum salary.

```
SELECT first_name, salary, department_id  
FROM employees  
WHERE salary = (SELECT MIN (salary) FROM employees);
```

‘=’ operator for subquery comparison:

To display employee information for the employees with the highest salary.

```
SELECT *  
FROM employees  
WHERE salary =  
(SELECT max(salary) FROM employees);
```

‘>’ operator for subquery comparison:

To display employee details whose salaries are greater than the highest salary.

```
SELECT * FROM employees
WHERE salary >
(SELECT max(salary) FROM employees);
```

multiple columns sub query:

- We can have multiple columns in the subquery, provided we have multiple columns to compare in the outer query.
- When **multiple columns** are used the order of columns in outer and inner query should be same.
- **For example, if we need to find the students who are of Alex's age and are from his place, then the inner / sub query can be written as below:**

```
SELECT * FROM STUDENT
WHERE (AGE, ADDRESS) =
(SELECT AGE, ADDRESS
FROM STUDENT
WHERE STD_NAME = 'Alex');
```

- **ORDER BY clause cannot be used inside sub query.** Ideally it does not signify anything if used too because it should return only one row. Even if we have to sort the columns, then we can use GROUP BY clause.

- **In SQL, one can have up to 255 sub queries in the WHERE clause.**

Some / Any operator:

- Sub queries can be used to compare the values using SOME clause. SOME is used when we have to compare the list of values using OR clause. If any one of the list value is matching, the result will be displayed.

```
SELECT * FROM STUDENT
WHERE (AGE) > SOME (or ANY)
(SELECT AGE FROM STUDENT
WHERE STD_ID BETWEEN 100 AND 105);
```

- Here if there is any student whose age are above ANY students from 100 to 105, then result will be displayed.
- Here SOME clause can be replaced by ANY clause. Both of these clauses do the same task.

SQL subquery with the ANY operator

The following shows the syntax of a subquery with the [ANY](#) operator:

```
comparison_operator ANY (subquery)
```


Code language: SQL (Structured Query Language) (sql)

For example, the following condition evaluates to true if x is greater than any value returned by the subquery. So the condition `x > SOME (1,2,3)` evaluates to true if x is greater than 1.

`x > ANY (subquery)`

Code language: SQL (Structured Query Language) (sql)

Note that the `SOME` operator is a synonym for the `ANY` operator so you can use them interchangeably.

The following query finds all employees whose salaries are greater than or equal to the highest salary of every department.

```
SELECT
    employee_id, first_name, last_name, salary
FROM
    employees
WHERE
    salary >= SOME (SELECT
        MAX(salary)
        FROM
            employees
        GROUP BY department_id);
```

Code language: SQL (Structured Query Language) (sql)

	employee_id	first_name	last_name	salary
▶	121	Adam	Fripp	8200.00
	103	Alexander	Hunold	9000.00
	104	Bruce	Ernst	6000.00
	179	Charles	Johnson	6200.00
	109	Daniel	Faviet	9000.00
	105	David	Austin	4800.00
	114	Den	Raphaely	11000.00
	204	Hermann	Baer	10000.00
	111	Ismael	Sciarra	7700.00
	177	Jack	Livingston	8400.00
	200	Jennifer	Whalen	4400.00

In this example, the subquery finds the highest salary of employees in each department. The outer query looks at these values and determines which employee's salaries are greater than or equal to any highest salary by department.

All operator: • Sub queries can be used to compare the values using `ALL` clauses. `ALL` is used when we have to compare the list of values using `AND` clause. If all of the list values are matching, then result will be displayed.

```
SELECT * FROM STUDENT
```

```
WHERE (AGE) > ALL
```

```
(SELECT AGE FROM STUDENT
```

```
WHERE STD_ID BETWEEN 100 AND 105);
```

• Here if there is any student whose age is above students from 100 to 105, then result will be displayed.

SQL subquery with the ALL operator

The syntax of the subquery when it is used with the [ALL](#) operator is as follows:

```
comparison_operator ALL (subquery)
```

Code language: SQL (Structured Query Language) (sql)

The following condition evaluates to true if x is greater than every value returned by the subquery.

```
 $x > \text{ALL (subquery)}$ 
```

Code language: SQL (Structured Query Language) (sql)

For example, suppose the subquery returns three value one, two, and three. The following condition evaluates to true if x is greater than 3.

```
 $x > \text{ALL (1,2,3)}$ 
```

Code language: SQL (Structured Query Language) (sql)

The following query uses the [GROUP BY](#) clause and [MIN\(\)](#) function to find the lowest salary by department:

```
SELECT  
  MIN(salary)  
FROM  
  employees  
GROUP BY department_id  
ORDER BY MIN(salary) DESC;
```

Code language: SQL (Structured Query Language) (sql)

	MIN(salary)
▶	17000.00
	10000.00
	8300.00
	6900.00
	6500.00
	6200.00
	6000.00
	4400.00
	4200.00
	2700.00
	2500.00

The following example finds all employees whose salaries are greater than the lowest salary of every department:

```
SELECT  
  employee_id, first_name, last_name, salary  
FROM  
  employees  
WHERE  
  salary >= ALL (SELECT  
    MIN(salary)  
  FROM  
    employees)
```

GROUP BY department_id)
ORDER BY first_name , last_name;

Code language: SQL (Structured Query Language) (sql)

	employee_id	first_name	last_name	salary
▶	102	Lex	De Haan	17000.00
	101	Neena	Kochhar	17000.00
	100	Steven	King	24000.00

UNIQUE clause with subqueries:

- Sub queries are used along with UNIQUE clause to check if the sub query returns duplicate values. When we apply UNIQUE to a subquery, the resulting condition returns true if no row appears twice in the answer to the subquery, that is, there are no duplicates;

```
SELECT * FROM STUDENT
```

```
WHERE UNIQUE
```

```
(SELECT AGE FROM STUDENT
```

```
WHERE STD_ID BETWEEN 100 AND 105);
```

IN operator with subqueries

find all employees working in “HYD” location

```
SELECT  
    employee_id, first_name, last_name  
FROM  
    employees  
WHERE  
    department_id IN (SELECT  
        department_id  
  
        FROM  
            departments  
        WHERE  
            location = 'HYD');
```

NOT IN operator with subqueries

find all employees not working in “HYD” location

```
SELECT  
    employee_id, first_name, last_name  
FROM  
    employees  
WHERE  
    department_id NOT IN (SELECT  
        department_id  
        FROM  
            departments  
        WHERE  
            location = 'HYD');
```

5. SQL subquery in the FROM clause

You can use a subquery in the `FROM` clause of the [SELECT](#) statement as follows:

```
SELECT * FROM (subquery) AS table_name
```

- to calculate the average of average salary of departments

```
SELECT  
  ROUND(AVG(average_salary), 0)  
FROM  
  (SELECT  
    AVG(salary) average_salary  
  FROM  
    employees  
  GROUP BY department_id) department_salary
```

6. SQL Subquery in the SELECT clause

- To find the salaries of all employees, their average salary, and the difference between the salary of each employee and the average salary.

```
SELECT  
  employee_id, first_name, last_name, salary,  
  (SELECT ROUND(AVG(salary), 0) FROM employees) average_salary,  
  salary - (SELECT ROUND(AVG(salary), 0) FROM employees) difference  
FROM  
  employees;
```

Correlated Sub Queries

- Correlated Sub Queries are also similar to sub queries but here the outer query is executed first and inner query is executed for each records of outer query. That is inner query is executed as many times as the outer query results.
- A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a `SELECT`, `UPDATE`, or `DELETE` statement.

EXAMPLE of Correlated Subqueries :

1. Find all the employees who earn more than the average salary in their department.

```
SELECT last_name, salary, department_id  
FROM employees outer  
WHERE salary >  
      (SELECT AVG(salary)
```

```
FROM employees
WHERE department_id =
       outer.department_id);
```

2. Use a correlated subquery to delete rows in one table based on the rows from another table.

```
delete from emp e where deptno = (select deptno from dept where
deptno = e.deptno);
```

3. update employee commission as average salary of their department

```
update emp
set comm = (select avg(sal)
from emp
where a.deptno = b.deptno);
```

Difference between Correlated and NonCorrelated Subquery

1. In case of correlated subquery inner query depends on outer query while in case of noncorrelated query inner query or subquery doesn't depend on outer query and runs by its own.
2. In case of correlated subquery, outer query is executed before inner query or subquery while in case of NonCorrelated subquery inner query executes before outer query.
3. Correlated Sub-queries are slower than non correlated subquery and should be avoided in favor of sql joins.
4. Common example of correlated subquery is using exists and not exists keyword while non correlated query mostly uses IN or NOT IN keywords.

EXISTS Operator :

The EXISTS operator tests for existence of rows in the results set of the subquery. If a subquery row value is found the condition is flagged **TRUE** and the search does not continue in the inner query, and if it is not found then the condition is flagged **FALSE** and the search continues in the inner query.

EXIST	operator	:
--------------	-----------------	----------

To find employees who have at least one person reporting to them.

```
SELECT empno, ename, job, deptno
FROM emp outer
```

```
WHERE EXISTS (SELECT *
FROM emp
WHERE mgrid = outer.empno);
```

NOT	EXIST	operator	:
To find all departments that do not have any employees.			

```
SELECT department_id, department_name
FROM departments d
WHERE NOT EXISTS (SELECT * FROM employees
WHERE department_id = d.department_id);
```

SQL NULL Values

A field with a NULL value is a field with no value.

If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

The IS NULL Operator

The IS NULL operator is used to test for empty values (NULL values).

The following SQL lists all customers with a NULL value in the "Address" field:

Example

```
SELECT CustomerName, ContactName, Address
FROM Customers
WHERE Address IS NULL;
```

The IS NOT NULL Operator

The IS NOT NULL operator is used to test for non-empty values (NOT NULL values).

The following SQL lists all customers with a value in the "Address" field:

Example:

```
SELECT CustomerName, ContactName, Address  
FROM Customers  
WHERE Address IS NOT NULL;
```

Joins

SQL Join is used to fetch data from two or more tables, which is joined to appear as single set of data.

- SQL Join is used for combining column from two or more tables by using values common to both tables.
- Join Keyword is used in SQL queries for joining two or more tables. Minimum required condition for joining table, is $(n-1)$ where n , is number of tables.
- A table can also join to itself known as, Self Join.

Types of Join

The following are the types of JOIN that we can use in SQL.

- Inner Join
 - Equi Join
 - Natural Join
 - Non-equi join
 - self join
- Outer Join
 - Left outer join
 - Right outer join
 - Full outer join
- Cross JOIN or Cartesian Product

Consider the following tables

Emp, dept, salgrade tables:

```
SQL> insert all
  2 into emp values(1,'suresh', 10000, 'clerk', 5, 11)
  3 into emp values(2,'dinesh', 15000, 'admin', 5, 11)
  4 into emp values(3,'mahesh', 5000, 'telecaller', 5, 11)
  5 into emp values(5,'rama', 50000, 'manager', 10, 22)
  6 into emp values(4,'krish', 45000, 'manager', 10, 22)
  7 into emp values(10,'krishna',80000, 'gm', 8, 22)
  8 select * from dual;
```

6 rows created.

```
SQL> select * from emp;
```

ENO	ENAME	SALARY	JOB	MGRID	DEPTNO
1	suresh	10000	clerk	5	11
2	dinesh	15000	admin	5	11
3	mahesh	5000	telecaller	5	11
5	rama	50000	manager	10	22
4	krish	45000	manager	10	22
10	krishna	80000	gm	8	22

6 rows selected.

```
SQL> insert into emp(eno,ename,salary,job,deptno) values(8,'sam',85000,'director',22);
```

1 row created.

```
SQL> insert into emp(eno,ename,salary,job,deptno) values(9,'shyam',85000,'director',22);
```

1 row created.

```
SQL> select * from emp;
```

ENO	ENAME	SALARY	JOB	MGRID	DEPTNO
1	suresh	10000	clerk	5	11
2	dinesh	15000	admin	5	11
3	mahesh	5000	telecaller	5	11
5	rama	50000	manager	10	22
4	krish	45000	manager	10	22
10	krishna	80000	gm	8	22
8	sam	85000	director		22
9	shyam	85000	director		22

8 rows selected.

```
SQL> select * from dept;
```

DEPTNO	DNAME	LOC
11	cse	hyd
22	aiml	blr
33	DS	hyd
44	ML	hyd


```
SQL> select * from salgrade;
```

LOSAL	HISAL	G
5000	10000	A
40000	50000	B
80000	90000	C

EQUI Join

This is a simple JOIN in which the result is based on matched data as per the equality condition specified in the query. Syntax is,

SELECT column-name-list

from table-name1 JOIN table-name2

WHERE table-name1.column-name = table-name2.column-name;

```
SQL> select e.eno, e.ename, e.deptno, d.dname from emp e, dept d where e.deptno = d.deptno;
```

ENO	ENAME	DEPTNO	DNAME
1	suresh	11	cse
2	dinesh	11	cse
3	mahesh	11	cse
5	rama	22	aiml
4	krish	22	aiml
10	krishna	22	aiml
8	sam	22	aiml
9	shyam	22	aiml

8 rows selected.

```
SQL> commit;
```

Commit complete.

```
SQL> select e.eno, e.ename, e.deptno, d.dname from emp e join dept d on ( e.deptno = d.deptno );
```

ENO	ENAME	DEPTNO	DNAME
1	suresh	11	cse
2	dinesh	11	cse
3	mahesh	11	cse
5	rama	22	aiml
4	krish	22	aiml
10	krishna	22	aiml
8	sam	22	aiml
9	shyam	22	aiml

8 rows selected.

Cross JOIN or Cartesian Product

This type of JOIN returns the cartesian product of rows from the tables in Join. It will return a table which consists of records which combines each row from the first table with each row of the second table.

Cross JOIN Syntax is,

SELECT column-name-list

from table-name1 CROSS JOIN table-name2;

```
SQL> set linesize 100
SQL> select * from emp e cross join dept d;
```

ENO	ENAME	SALARY	JOB	MGRID	DEPTNO	DEPTNO	DNAME	LOC
1	suresh	10000	clerk	5	11	11	cse	hyd
2	dinesh	15000	admin	5	11	11	cse	hyd
3	mahesh	5000	telecaller	5	11	11	cse	hyd
5	rama	50000	manager	10	22	11	cse	hyd
4	krish	45000	manager	10	22	11	cse	hyd
10	krishna	80000	gm	8	22	11	cse	hyd
8	sam	85000	director		22	11	cse	hyd
9	shyam	85000	director		22	11	cse	hyd
1	suresh	10000	clerk	5	11	22	aiml	blr
2	dinesh	15000	admin	5	11	22	aiml	blr
3	mahesh	5000	telecaller	5	11	22	aiml	blr
5	rama	50000	manager	10	22	22	aiml	blr
4	krish	45000	manager	10	22	22	aiml	blr
10	krishna	80000	gm	8	22	22	aiml	blr
8	sam	85000	director		22	22	aiml	blr
9	shyam	85000	director		22	22	aiml	blr

```
16 rows selected.
```

Natural JOIN

Natural Join is a type of Inner join which is based on column having same name and same datatype present in both the tables to be joined. Natural Join Syntax is,

```
SELECT * from
```

```
table-name1
```

```
NATURAL JOIN
```

```
table-name2;
```

Example:

```
SQL> select * from emp e natural join dept d;
```

DEPTNO	ENO	ENAME	SALARY	JOB	MGRID	DNAME	LOC
11	1	suresh	10000	clerk	5	cse	hyd
11	2	dinesh	15000	admin	5	cse	hyd
11	3	mahesh	5000	telecaller	5	cse	hyd
22	5	rama	50000	manager	10	aiml	blr
22	4	krish	45000	manager	10	aiml	blr
22	10	krishna	80000	gm	8	aiml	blr
22	8	sam	85000	director		aiml	blr
22	9	shyam	85000	director		aiml	blr

```
8 rows selected.
```

Non-equi join:

This is a simple JOIN in which the result is based on matched data as per the condition that includes the operator other than equality operator specified in the query. Syntax is,

To display the employee details, their salaries, and their salary grades

```
SQL> select e.eno, e.ename, e.deptno, s.grade from emp e, salgrade s where e.salary between losal and hisal;
```

ENO	ENAME	DEPTNO	G
1	suresh	11	A
3	mahesh	11	A
5	rama	22	B
4	krish	22	B
10	krishna	22	C
8	sam	22	C
9	shyam	22	C

7 rows selected.

Self join:

A table can also join to itself known as, Self Join.

To display the employee details and their manager details from emp table

```
SQL> select e.eno, e.ename, e.mgrid, m.ename from emp e, emp m where e.mgrid = m.eno;
```

ENO	ENAME	MGRID	ENAME
3	mahesh	5	rama
2	dinesh	5	rama
1	suresh	5	rama
4	krish	10	krishna
5	rama	10	krishna
10	krishna	8	sam

6 rows selected.

Outer joins:

Outer Join is based on both matched and unmatched data

Consider the emp and dept tables as given below

```
SQL> select * from emp;
```

ENO	ENAME	SALARY	JOB	MGRID	DEPTNO
1	suresh	10000	clerk	5	11
2	dinesh	15000	admin	5	11
3	mahesh	5000	telecaller	5	11
5	rama	50000	manager	10	22
4	krish	45000	manager	10	22
10	krishna	80000	gm	8	22
8	sam	85000	director		22
9	shyam	85000	director		22
6	sriram	30000	analyst	8	
7	laxman	35000	developer	8	

10 rows selected.

```
SQL> select * from dept;
```

DEPTNO	DNAME	LOC
11	cse	hyd
22	aiml	blr
33	DS	hyd
44	ML	hyd

Left Outer join:

The left outer join returns a result table with the matched data of two tables then remaining rows of the left table and null for the right table's column. Left Outer Join syntax is,

```
SELECT column-name-list  
from table-name1  
LEFT OUTER JOIN  
table-name2  
on table-name1.column-name = table-name2.column-name;  
OR
```

```
select column-name-list  
from table-name1  
LEFT OUTER JOIN  
table-name2  
using column-name;
```

Example:

```
SQL> select eno, ename, deptno, dname from emp e left outer join dept d using(deptno);
```

ENO	ENAME	DEPTNO	DNAME
3	mahesh	11	cse
2	dinesh	11	cse
1	suresh	11	cse
9	shyam	22	aiml
8	sam	22	aiml
10	krishna	22	aiml
4	krish	22	aiml
5	rama	22	aiml
7	laxman		
6	sriram		

```
10 rows selected.
```

Right Outer Join

The right outer join returns a result table with the matched data of two tables then remaining rows of the right table and null for the left table's columns.

Right Outer Join Syntax is,

```
select column-name-list  
from table-name1  
RIGHT OUTER JOIN  
table-name2
```

on table-name1.column-name = table-name2.column-name;

or

```
select column-name-list  
from table-name1  
RIGHT OUTER JOIN  
table-name2  
using column-name;
```

Example:

```
SQL> select eno, ename, deptno, dname from emp e right outer join dept d using(deptno);
```

ENO	ENAME	DEPTNO	DNAME
1	suresh	11	cse
2	dinesh	11	cse
3	mahesh	11	cse
5	rama	22	aiml
4	krish	22	aiml
10	krishna	22	aiml
8	sam	22	aiml
9	shyam	22	aiml
		44	ML
		33	DS

```
10 rows selected.
```

Full Join

The full outer join returns a result table with the matched data of two table then remaining rows of both left table and then the right table. Full Outer Join Syntax is,

```
select column-name-list  
from table-name1  
FULL OUTER JOIN  
table-name2  
on table-name1.column-name = table-name2.column-name;
```

or

```
select column-name-list  
from table-name1  
FULL OUTER JOIN  
table-name2  
using column-name;
```

Example:

```
SQL> set pagesize 24
SQL> select eno, ename, deptno, dname from emp e full join dept d using(deptno);
```

ENO	ENAME	DEPTNO	DNAME
1	suresh	11	cse
2	dinesh	11	cse
3	mahesh	11	cse
5	rama	22	aiml
4	krish	22	aiml
10	krishna	22	aiml
8	sam	22	aiml
9	shyam	22	aiml
6	sriram		
7	laxman		
		44	ML
		33	DS

```
12 rows selected.
```

SQL Aggregate Functions

An aggregate function in SQL returns one value after calculating multiple values of a column. We often use aggregate functions with the GROUP BY and HAVING clauses of the SELECT statement.

Useful aggregate functions:

- AVG() - Returns the average value
- COUNT() - Returns the number of rows
- FIRST() - Returns the first value
- LAST() - Returns the last value
- MAX() - Returns the largest value
- MIN() - Returns the smallest value
- SUM() - Returns the sum

Consider the table given below

Students-Table

ID	NAME	MARKS	AGE
1	Harsh	90	19
2	Suresh	50	20
3	Pratik	80	19
4	Dhanraj	95	21
5	Ram	85	18

AVG() Function

The AVG function is used to calculate the average value of the numeric type. AVG function returns the average of all non-Null values.

Syntax

AVG()

or

AVG([ALL|DISTINCT] expression)

Example:

Computing average marks of students.

```
SELECT AVG(MARKS) AS AvgMarks FROM Students;
```

SUM() Function

Sum function is used to calculate the sum of all selected columns. It works on numeric fields only.

Syntax

SUM()

or

SUM([ALL|DISTINCT] expression)

Example: SUM()

```
SELECT SUM(MARKS) AS TotalMarks FROM Students;
```

COUNT() function

COUNT function is used to Count the number of rows in a database table. It can work on both numeric and non-numeric data types.

COUNT function uses the COUNT(*) that returns the count of all the rows in a specified table. COUNT(*) considers duplicate and Null.

Syntax

COUNT(*)

or

COUNT([ALL|DISTINCT] expression)

Example:

Computing total number of students.

```
SELECT COUNT(*) AS NumStudents FROM Students;
```

MAX Function

MAX function is used to find the maximum value of a certain column. This function determines the largest value of all selected values of a column.

Syntax

MAX()

or

MAX([ALL|DISTINCT] expression)

Example:

Fetching maximum marks among students from the Students table. `SELECT MAX(MARKS) AS MaxMarks FROM Students;`

MIN Function

MIN function is used to find the minimum value of a certain column. This function determines the smallest value of all selected values of a column.

Syntax

MIN()

or

MIN([ALL|DISTINCT] expression)

Example

Fetching minimum marks among students from the Students table. `SELECT MIN(MARKS) AS MinMarks FROM Students;`

LAST() Function

The LAST() function returns the last value of the selected column.

syntax

`SELECT LAST(column_name) FROM table_name`

Example: Fetching marks of last student from the Students table.

`SELECT LAST(MARKS) AS MarksLast FROM Students;`

FIRST() Function

The FIRST() function returns the first value of the selected column.

Syntax

`SELECT FIRST(column_name) FROM table_name`

Example

Fetching marks of first student from the Students table.

`SELECT FIRST(MARKS) AS MarksFirst FROM Students;`

SUM() function

SUM() function returns the total sum of a numeric column

The GROUP BY Statement

The GROUP BY statement is used in conjunction with the aggregate functions to group the result-set by one or more columns.

SQL GROUP BY Syntax

```
SELECT column_name, aggregate_function(column_name) FROM table_name WHERE column_name operator value GROUP BY column_name
```

The HAVING Clause

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

SQL HAVING Syntax

```
SELECT column_name, aggregate_function(column_name) FROM table_name WHERE column_name operator value GROUP BY column_name HAVING aggregate_function(column_name) operator value
```

SQL NULL Values

A field with a NULL value is a field with no value.

If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

The IS NULL Operator

The IS NULL operator is used to test for empty values (NULL values).

The following SQL lists all customers with a NULL value in the "Address" field:

Example

```
SELECT CustomerName, ContactName, Address  
FROM Customers  
WHERE Address IS NULL;
```

IS NOT NULL Operator

The IS NOT NULL operator is used to test for non-empty values (NOT NULL values).

The following SQL lists all customers with a value in the "Address" field:

Example

```
SELECT CustomerName, ContactName, Address  
FROM Customers  
WHERE Address IS NOT NULL;
```

Relational Algebra

Relational algebra is one of the two formal query languages associated with the relational model. Queries in algebra are composed using a collection of operators. A fundamental property is that every operator in the algebra accepts one or more relational instances as arguments and returns a relation instance as the result. The relational algebra is *procedural*.

Consider the following instances to illustrate the relational algebra operators:

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

Fig: Instance S1 Sailors

sid	sname	rating	age
28	Yuppy	9	35.0
31	Lubber	8	55.5
44	Guppy	5	35.0
58	Rusty	10	35.0

Fig: Instance S2 of Sailors

sid	bid	Day
22	101	10/10/96
58	103	11/12/96

Fig: Instance R1 of Reserves

The various **operators** that form the relational algebraic query are:

1) Selection and Projection

Relational algebra includes operators to *select* rows from a relation (σ) and to *project* columns (π). These operations allow us to manipulate data in a single relation.

In general, the selection operator σ specifies the tuples to get through a *selection condition*. Here the selection condition is a Boolean combination of *terms* that have the form *attribute* op *constant* or *attribute1* op *attribute2*, where op is one of the relational operators $<$, $<=$, $=$, \neq , $>=$, or $>$.

Consider the instance S2 of the Sailors relation shown in figure, we can find the Sailors with rating above 8 by the following expression:

$\sigma_{\text{rating} > 8} (S2)$

This evaluates to the relation shown below:

Sid	sname	rating	Age
28	Yuppy	9	35.0
58	Rusty	10	35.0

The *projection* operator π allows us to extract columns from a relation. For example, we can find out all sailors names and ratings by the following expression:

$\Pi_{\text{sname}, \text{rating}} (S2)$

This evaluates to the relation shown below:

sname	rating
Yuppy	9
Lubber	8
Guppy	5
Rusty	10

Similarly, we can find the names and ratings of sailors with rating above 8 by the following expression:

$\Pi_{\text{sname, rating}} (\sigma_{\text{rating} > 8} (S2))$

This evaluates to the relation shown below:

sname	rating
Yuppy	9
Rusty	10

2) Set Operations

The following standard operations on sets are available in relational algebra: *union* (\cup), *intersection* (\cap), *set-difference* ($-$), and *cross-product* (\times).

Union:

$R \cup S$ returns a relation instance containing all tuples that occur in either relation instance R or relation instance S (or both).

R and S must be *union-compatible*, and the schema of the result is identical to the schema of R . Two relation instances are said to be *union-compatible* if the following conditions hold:

- They have the same number of fields, and
- Corresponding fields, taken in order from left to right, have the same *domains*

Ex: The *union* of $S1$ and $S2$ ($S1 \cup S2$) is shown below:

<i>Sid</i>	<i>Sname</i>	<i>rating</i>	<i>age</i>
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0
28	Yuppy	9	35.0
44	Guppy	5	35.0

Intersection:

$R \cap S$ returns a relation instance containing all tuples that occur in both R and S. The relations R and S must be *union-compatible*, and the schema of the result is same as the schema of R.

Ex: The *intersection* of S1 and S2 ($S1 \cap S2$) is shown below:

Sid	Sname	rating	age
31	Lubber	8	55.5
58	Rusty	10	35.0

Set-difference:

$R - S$ returns a relation instance containing all tuples that occur in R but not in S. The relations R and S must be *union-compatible*, and the schema of the result is identical to the schema of R.

Ex: The *Set-difference* of S1 and S2 ($S1 - S2$) is shown below:

Sid	Sname	rating	age
22	Dustin	7	45.0

Cross-product:

$R \times S$ returns a relation instance whose schema contains all the fields of R followed by all the fields of S. The cross product operation is sometimes called as ***Cartesian product***.

Ex: The *Cross-product* of S1 and R1 ($S1 \times R1$) is shown below:

(sid)	Sname	rating	age	(sid)	bid	day
22	Dustin	7	45.0	22	101	10/10/96
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	22	101	10/10/96
31	Lubber	8	55.5	58	103	11/12/96
58	Rusty	10	35.0	22	101	10/10/96
58	Rusty	10	35.0	58	103	11/12/96

3) Renaming

The result of the relational algebra expression includes the field names in such a way that naming conflicts can arise in some cases. For example, in $S1 \times R1$. Hence we have to rename the fields or rename the relation. Relation algebra provides **renaming** operator ρ for this purpose.

The expression $\rho (R (F), E)$ takes a relational algebra expression E and returns an instance of a relation R . R contains the same tuples as the result of E and has the same schema as E , but some fields are renamed. F is the list of fields renamed and is in the form *oldname* \rightarrow *newname* or *position* \rightarrow *newname*.

For example, the expression $\rho (C (1 \rightarrow sid1, 5 \rightarrow sid2), S1 \times R1)$ returns a relation with the following schema:

$C (sid1: integer, sname: string, rating: integer, age: real, sid2: integer, bid: integer, day: date)$

4) Joins

The *join* operation is used to combine the information from two or more relations. *Join* can be defined as a cross-product followed by selection and projection. There are several variants of join operation:

Conditional Join:

The most general version of the join operation accepts a *join condition* c and a pair of relational instances as arguments and returns a relation instance. The operation is defined as follows:

$$R \bowtie_c S = \sigma_c (R \times S)$$

Thus \bowtie is defined to be a cross-product followed by a

selection. For example, the result of $S1 \bowtie_{S1.sid < R1.sid} R1$ is

shown below:

(sid)	Sname	rating	age	(sid)	bid	day
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	58	103	11/12/96

Equijoin:

A common special case of the join operation $R \bowtie S$ is when the *join condition* consists solely of equalities of the form $R.name1 = S.name2$, that is, equalities between two fields in R and S . The join operation with such equality condition is called **equijoin**.

For example, the result of $S \bowtie_{sid = S.sid} R$ is shown below:

sid	Sname	rating	age	bid	day
22	Dustin	7	45.0	101	10/10/96
58	Rusty	10	35.0	103	11/12/96

Note that the fields in the *equijoin condition* appears only once in the resultant instance.

Natural Join:

A further special case of the join operation $R \bowtie S$ is an equijoin in which equalities are specified on *all common* fields of R and S. In this case, we can simply omit the join condition. By default, the equality condition is employed on all common fields.

We call this as *natural join* and can simply be denoted as $R \bowtie S$. The result of this expression is same as above, since the only common field is *sid*.

5) Division

We discuss division through an example. Consider two relation instances A and B in which A has two fields x and y and B has just one field y, with the same domain as in A. We define the *division* operation A/B as the set of all x values such that for *every* y value in B, there is a tuple $\langle x, y \rangle$ in A.

Division is illustrated in figure below. Consider the relation A listing the parts(pid) supplied by suppliers(sid) and the relation B listing the parts(pid). A/B computes suppliers who supply *all* parts listed in relation instance B.

Sno	pno
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4

A

Pno
P2
B1

Sno
B2
P2
P4

B3
Pno
P1
P2
P4

A/B1

Sno
S1
S2
S3
S4

A/B2

Sno
S1
S4

A/B3

Sno
S1

Relational Calculus

Relational calculus is an alternative to relational algebra. In contrast to the algebra, which is procedural, the calculus is non-procedural or declarative.

There are two variants of the calculus:

- 1) Tuple Relational Calculus (TRC): Variables in TRC take tuples as the values.
- 2) Domain Relational Calculus (DRC): Variables in DRC range over field values.

Tuple Relational Calculus (TRC)

A **tuple variable** is a variable that takes on tuples of a particular relation schema as values. A TRC query has the form $\{ T / p(T) \}$, where T is a tuple variable and $p(T)$ denotes a *formula* that describes T .

Syntax of TRC Queries:

- A nonprocedural query language, where each query is of the form $\{ t \mid P(t) \}$
- It is the set of all tuples t such that predicate P is true for t
- t is a *tuple variable*, $t[A]$ denotes the value of tuple t on attribute A
- $t \in r$ denotes that tuple t is in relation r
- P is a *formula* similar to that of the predicate calculus

Let Rel be a relation, R and S be tuple variables, a be an attribute of R , and b be an attribute of S .

Let op denotes a relational operator. An **atomic formula** is one of the following:

- $R \in Rel$
- $R.a \text{ op } S.b$
- $R.a \text{ op } constant$, or $constant \text{ op } R.a$

A **formula** is one of the following:

- Any atomic formula

- $\neg p$, $p \wedge q$, $p \vee q$, or $p \Rightarrow q$
- $\exists R(p(R))$, where R is a tuple variable
- $\forall R(p(R))$, where R is a tuple variable

Where p and q are themselves are formulas and $p(R)$ denotes a formula with the variable R .

Predicate Calculus Formula

1. Set of attributes and constants
2. Set of comparison operators: (e.g., $<$, \leq , $=$, \neq , $>$, \geq)
3. Set of connectives: and (\wedge), or (\vee), not (\neg)
4. Implication (\Rightarrow): $x \Rightarrow y$, if x is true, then y is true

$$x \Rightarrow y \equiv \neg x \vee y$$

5. Set of quantifiers:
 - $\exists t \in r (Q(t)) \equiv$ "there exists" a tuple t in relation r such that predicate $Q(t)$ is true
 - $\forall t \in r (Q(t)) \equiv Q$ is true "for all" tuples t in relation r

Example Relations:

branch (branch-name, branch-city, assets)

customer (customer-name, customer-street, customer-city)

account (account-number, branch-name, balance)

loan (loan-number, branch-name, amount)

depositor (customer-name, account-number)

borrower (customer-name, loan-number)

1. Find the loan-number, branch-name, and amount for loans of over \$1200

$$\{t \mid t \in \text{loan} \wedge t[\text{amount}] > 1200\}$$

2. Find the loan number for each loan of an amount greater than \$1200

$$\{t \mid \exists s \in \text{loan} (t[\text{loan-number}] = s[\text{loan-number}] \wedge s[\text{amount}] > 1200)\}$$

3. Find the names of all customers having a loan, an account, or both at the bank

$$\{t \mid \exists s \in \text{borrower} (t[\text{customer-name}] = s[\text{customer-name}]) \\ \vee \exists u \in \text{depositor} (t[\text{customer-name}] = u[\text{customer-name}])\}$$

4. Find the names of all customers who have a loan and an account at the bank

$$\{t \mid \exists s \in \text{borrower} (t[\text{customer-name}] = s[\text{customer-name}]) \\ \wedge \exists u \in \text{depositor} (t[\text{customer-name}] = u[\text{customer-name}])\}$$

5. Find the names of all customers having a loan at the Perryridge branch

$$\{t \mid \exists s \in \text{borrower} (t[\text{customer-name}] = s[\text{customer-name}] \\ \wedge \exists u \in \text{loan} (u[\text{branch-name}] = \text{"Perryridge"} \\ \wedge u[\text{loan-number}] = s[\text{loan-number}]))\}$$

6. Find the names of all customers who have a loan at the Perryridge branch, but no account at any branch of the bank

$$\{t \mid \exists s \in \text{borrower} (t[\text{customer-name}] = s[\text{customer-name}] \\ \wedge \exists u \in \text{loan} (u[\text{branch-name}] = \text{"Perryridge"} \wedge u[\text{loan-number}] = s[\text{loan-number}])) \\ \wedge \text{not } \exists v \in \text{depositor} (v[\text{customer-name}] = t[\text{customer-name}]) \}$$

2) Domain Relational Calculus

A **domain variable** is a variable that ranges over the values in the domain of some attribute. A DRC query has the form $\{ \langle x_1, x_2, \dots, x_n \rangle \mid p(\langle x_1, x_2, \dots, x_n \rangle) \}$, where x_i is either a *domain variable* or a constant and $p(\langle x_1, x_2, \dots, x_n \rangle)$ denotes a **DRC formula**.

Syntax of DRC Queries:

Let Rel be a relation, X and Y be domain variables, x_i , $1 \leq i \leq n$, be an attribute of Rel . Let op denotes a relational operator. An **atomic formula** is one of the following:

- $\langle x_1, x_2, \dots, x_n \rangle \in Rel$
- $X \text{ op } Y$
- $X \text{ op } constant$, or $constant \text{ op } X$

A **formula** is one of the following:

- Any atomic formula
- $\neg p$, $p \wedge q$, $p \vee q$, or $p \Rightarrow q$
- $\exists X(p(X))$, where X is a domain variable
- $\forall X(p(X))$, where X is a domain variable

Example Queries:

1. Find the *loan-number*, *branch-name*, and *amount* for loans of over \$1200
 $\{ \langle l, b, a \rangle \mid \langle l, b, a \rangle \in loan \wedge a > 1200 \}$
2. Find the names of all customers who have a loan of over \$1200
 $\{ \langle c \rangle \mid \exists l, b, a (\langle c, l \rangle \in borrower \wedge \langle l, b, a \rangle \in loan \wedge a > 1200) \}$
3. Find the names of all customers who have a loan from the Perryridge branch and the loan amount:
 $\{ \langle c, a \rangle \mid \exists l (\langle c, l \rangle \in borrower \wedge \exists b (\langle l, b, a \rangle \in loan \wedge b = \text{"Perryridge"})) \}$
or
 $\{ \langle c, a \rangle \mid \exists l (\langle c, l \rangle \in borrower \wedge \langle l, \text{"Perryridge"}, a \rangle \in loan) \}$
4. Find the names of all customers having a loan, an account, or both at the Perryridge branch:
 $\{ \langle c \rangle \mid \exists l ((\langle c, l \rangle \in borrower \wedge \exists b, a (\langle l, b, a \rangle \in loan \wedge b = \text{"Perryridge"})) \vee \exists a (\langle c, a \rangle \in depositor \wedge \exists b, n (\langle a, b, n \rangle \in account \wedge b = \text{"Perryridge"})) \}$
5. Find the names of all customers who have an account at all branches located in Brooklyn:
 $\{ \langle c \rangle \mid \exists s, n (\langle c, s, n \rangle \in customer) \wedge \forall x, y, z (\langle x, y, z \rangle \in branch \wedge y = \text{"Brooklyn"}) \Rightarrow \exists a, b (\langle x, y, z \rangle \in account \wedge \langle c, a \rangle \in depositor) \}$

PL/SQL

PL/SQL (procedural language extension to Structured Query Language)

PL/SQL is a procedural language extension to Structured Query Language (SQL).

The purpose of PL/SQL is to combine database language and procedural programming language. The basic unit in PL/SQL is called a block and is made up of three parts: a declarative part, an executable part and an exception-building part.

PL/SQL enables users to mix SQL statements with procedural constructs, it is possible to use PL/SQL blocks and subprograms to group SQL statements before sending them to Oracle for execution. Without PL/SQL, Oracle must process SQL statements one at a time. In a network environment, this can affect traffic flow and slow down response time.

PL/SQL blocks can be compiled once and stored in executable form to improve response time. A PL/SQL program that is stored in a database in compiled form and can be called by name is referred to as a stored procedure. A PL/SQL stored procedure that is implicitly started when an INSERT, UPDATE or DELETE statement is issued against an associated table is called a trigger.

Disadvantages of SQL:

SQL doesn't provide the programmers with a technique of condition checking, looping and branching.

SQL statements are passed to Oracle engine one at a time which increases traffic and decreases speed.

SQL has no facility of error checking during manipulation of data.

Features of PL/SQL:

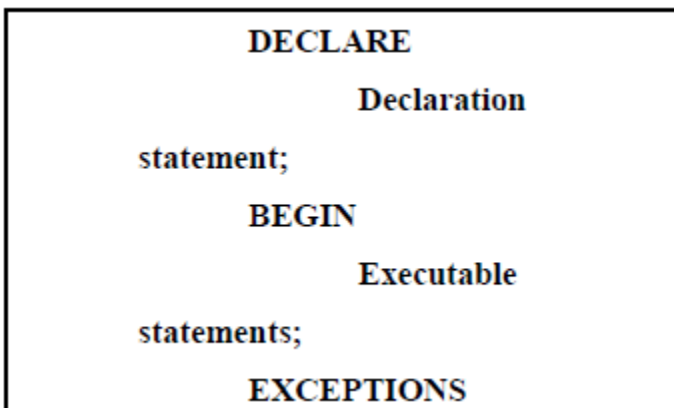
- PL/SQL is basically a procedural language, which provides the functionality of decision making, iteration and many more features of procedural programming languages.
- PL/SQL can execute a number of queries in one block using single command.
- One can create a PL/SQL unit such as procedures, functions, packages, triggers, and types, which are stored in the database for reuse by applications.
- PL/SQL provides a feature to handle the exception which occurs in PL/SQL block known as exception handling block.
- Applications written in PL/SQL are portable to computer hardware or operating system where Oracle is operational.
- PL/SQL Offers extensive error checking.

➤ Differences between SQL and PL/SQL:

SQL	PL/SQL
SQL is a single query that is used to perform DML and DDL operations.	PL/SQL is a block of codes that used to write the entire program blocks/ procedure/ function, etc.
It is declarative, that defines what needs to be done, rather than how things need to be done.	PL/SQL is procedural that defines how the things needs to be done.
Execute as a single statement.	Execute as a whole block.
Mainly used to manipulate data.	Mainly used to create an application.
Cannot contain PL/SQL code in it.	It is an extension of SQL, so it can contain SQL inside it.

Structure of PL/SQL Block

- PL/SQL extends SQL by adding constructs found in procedural languages, resulting in a structural language that is more powerful than SQL. The basic unit in PL/SQL is a block. All PL/SQL programs are made up of blocks, which can be nested within each other. PL/SQL contains 4 main blocks o Declare o Begin o Exception o End
- Typically, each block performs a logical action in the program. A block has the following structure:



- Declare section starts with DECLARE keyword in which variables, constants, records as cursors can be declared which stores data temporarily. It basically consists definition of PL/SQL identifiers. This part of the code is optional.
- Execution section starts with BEGIN and ends with END keyword. This is a mandatory section and here the program logic is written to perform any task like loops and conditional statements. It supports all DML commands, DDL commands and SQL*PLUS built-in functions as well.

- Exception section starts with EXCEPTION keyword. This section is optional which contains statements that are executed when a run-time error occurs. Any exceptions can be handled in this section.

➤ PL/SQL Identifiers

- There are several PL/SQL identifiers such as variables, constants, procedures, cursors, triggers etc.
- Variables: Like several other programming languages, variables in PL/SQL must be declared prior to its use. They should have a valid name and data type as well.

Syntax for declaration of variables:

```
variable_name data type [NOT NULL := value];
```

- Example program to declare variables in PL/SQL :

```
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
    var1 INTEGER;
    var2 REAL;
    var3 varchar2(20) ;
BEGIN
    null;
END;
/
```

Output:

```
PL/SQL procedure successfully completed
```

- SET SERVEROUTPUT ON is used to display the buffer used by the dbms_output. var1 INTEGER is the declaration of variable, named var1 which is of integer type.
- There are many other data types that can be used like float, int, real, smallint, long etc.
- It also supports variables used in SQL as well like NUMBER (prec, scale), varchar, varchar2 etc.
- PL/SQL procedure successfully completed is displayed when the code is compiled and executed successfully. Slash (/) after END;
- The slash (/) tells the SQL*Plus to execute the block.

➤ INITIALISING VARIABLES

- The variables can also be initialised just like in other programming languages. Let us see an example for the same:

```
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
    var1 INTEGER := 2 ;
    var3 varchar2(20) := ' DBMS Book' ;

    BEGIN
        null;
    END;
    /
```

Output:
PL/SQL procedure successfully completed

- Assignment operator (:=) : It is used to assign a value to a variable.

➤ DISPLAYING OUTPUT

- The outputs are displayed by using DBMS_OUTPUT which is a built-in package that enables the user to display output, debugging information, and send messages from PL/SQL blocks, subprograms, packages, and triggers.
- Let us see an example to see how to display a message using PL/SQL:

```
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
    var varchar2(40) := 'I love
    GeeksForGeeks';
    BEGIN
        dbms_output.put_line(var);
    END;
    /
```

Output:
I love GeeksForGeeks
PL/SQL procedure successfully completed

dbms_output.put_line command is used to direct the PL/SQL output to a screen.

➤ USING COMMENTS

Like in many other programming languages, in PL/SQL also, comments can be put within the code which has no effect in the code. There are two syntaxes to create comments in PL/SQL:

- Single Line Comment: To create a single line comment, the symbol `--` is used.
- Multi Line Comment: To create comments that span over several lines, the symbol `/*` and `*/` is used. Example to show how to create comments in PL/SQL:

```
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
    -- I am a comment, so i will be ignored.
    var varchar2(40) := 'I love GeeksForGeeks';
BEGIN
    dbms_output.put_line(var);
END;
/
```

Output:

```
I love GeeksForGeeks
PL/SQL procedure successfully completed
```

➤ TAKING INPUT FROM USER

Just like in other programming languages, in PL/SQL also, we can take input from the user and store it in a variable.

Let us see an example to show how to take input from users in PL/SQL:

```
SQL> SET SERVEROUTPUT ON;
```

```
SQL> DECLARE
```

```
    -- taking input for variable a
```

```
    a number := &a;
```

```
    -- taking input for variable b
```

```
    b varchar2(30) := &b;
```

```
    BEGIN
```

```
    null;
```

```
    END;
```

```
    /
```

Output:

PL/SQL procedure successfully completed

Let us see an example on PL/SQL to demonstrate all above concepts in one single block of code.

```
--PL/SQL code to print sum of two numbers taken from the user.
```

```
SQL> SET SERVEROUTPUT ON;
```

```
SQL> DECLARE
```

```
    -- taking input for variable a
```

```
    a integer := &a ;
```

```
    -- taking input for variable b
```

```
    b integer := &b ;
```

```
    c integer ;
```

```
    BEGIN
```

```
    c := a + b ;
```

```
    dbms_output.put_line('Sum of '||a||' and '||b||' is = '||c);
```

```
    END;
```

```
    /
```

Output:

2

3

Sum of 2 and 3 is =5

PL/SQL procedure successfully completed

Let us see an example on PL/SQL to demonstrate all above concepts in one single block of code.

Syntax (IF-THEN-ELSIF)

The syntax for IF-THEN-ELSIF in Oracle/PLSQL is:

```
IF condition1 THEN
    {...statements to execute when condition1 is TRUE...}

ELSIF condition2 THEN
    {...statements to execute when condition2 is TRUE...}

END IF;
```

You use the IF-THEN-ELSIF syntax, when you want to execute one set of *statements* when *condition1* is TRUE or a different set of *statements* when *condition2* is TRUE.

Syntax (IF-THEN-ELSIF-ELSE)

The syntax for IF-THEN-ELSIF-ELSE in Oracle/PLSQL is:

```
IF condition1 THEN
    {...statements to execute when condition1 is TRUE...}

ELSIF condition2 THEN
    {...statements to execute when condition2 is TRUE...}

ELSE
    {...statements to execute when both condition1 and condition2 are FALSE...}

END IF;
```

You use the IF-THEN-ELSIF-ELSE syntax, when you want to execute one set of *statements* when *condition1* is TRUE, a different set of *statements* when *condition2* is TRUE, or a different set of *statements* when all previous conditions (ie: *condition1* and *condition2*) are FALSE

Iterative Statements in PL/SQL

Iterative control Statements are used when we want to repeat the execution of one or more statements for specified number of times.

There are three types of loops in PL/SQL:

- | | | |
|---|----------|------|
| • | Simple | Loop |
| • | While | Loop |
| • | For Loop | |

1) Simple Loop

A Simple Loop is used when a set of statements is to be executed at least once before the loop terminates. An EXIT condition must be specified in the loop, otherwise the loop will get into an infinite number of iterations. When the EXIT condition is satisfied the process exits from the loop.

General Syntax to write a Simple Loop is:

```
LOOP  
  statements;  
  EXIT;  
  {or EXIT WHEN condition;}  
END LOOP;
```

These are the important steps to be followed while using Simple Loop.

- 1) Initialise a variable before the loop body.
- 2) Increment the variable in the loop.
- 3) Use a EXIT WHEN statement to exit from the Loop. If you use a EXIT statement without WHEN condition, the statements in the loop is executed only once.

2) While Loop

A WHILE LOOP is used when a set of statements has to be executed as long as a condition is true. The condition is evaluated at the beginning of each iteration. The iteration continues until the condition becomes false.

The General Syntax to write a WHILE LOOP is:

```
WHILE <condition>  
  LOOP statements;  
END LOOP;
```

Important steps to follow when executing a while loop:

- 1) Initialise a variable before the loop body.
- 2) Increment the variable in the loop.
- 3) EXIT WHEN statement and EXIT statements can be used in while loops but it's not done often.

3) FOR Loop

A FOR LOOP is used to execute a set of statements for a predetermined number of times. Iteration occurs between the start and end integer values given. The counter is always incremented by 1. The loop exits when the counter reaches the value of the end integer.

The General Syntax to write a FOR LOOP is:

```
FOR counter IN val1..val2  
LOOP statements;  
END LOOP;
```

- val1 - Start integer value.
- val2 - End integer value.

Important steps to follow when executing a while loop:

- 1) The counter variable is implicitly declared in the declaration section, so it's not necessary to declare it explicitly.
- 2) The counter variable is incremented by 1 and does not need to be incremented explicitly.
- 3) EXIT WHEN statement and EXIT statements can be used in FOR loops but it's not done often.

CASE Statement

- A CASE statement is similar to IF-THEN-ELSIF statement that selects one alternative based on the condition from the available options.
- Unlike IF-THEN-ELSIF, the CASE statement can also be used in SQL statements.
- ELSE block in CASE statement holds the sequence that needs to be executed when none of the alternatives got selected.

Syntax:

```
CASE (expression)  
WHEN <value1> THEN action_block1;  
WHEN <value2> THEN action_block2;  
WHEN <value3> THEN action_block3;  
ELSE action_block_default;  
END CASE;
```

Example:

```
DECLARE  
a NUMBER :=55;
```

```

b NUMBER :=5;
arth_operation VARCHAR2(20) :='MULTIPLY';
BEGIN
dbms_output.put_line('Program started.' );
CASE (arth_operation)
WHEN 'ADD' THEN dbms_output.put_line('Addition of the numbers are: '|| a+b );
WHEN 'SUBTRACT' THEN dbms_output.put_line('Subtraction of the numbers are: '||a-b );
WHEN 'MULTIPLY' THEN dbms_output.put_line('Multiplication of the numbers are: '|| a*b );
WHEN 'DIVIDE' THEN dbms_output.put_line('Division of the numbers are:'|| a/b);
ELSE dbms_output.put_line('No operation action defined. Invalid operation');
END case;
dbms_output.put_line('Program completed.' );
END;
/

```

PL/SQL Program to Reverse a String

```

declare
    str1 varchar2(50):='&str';
    str2 varchar2(50);
    len number;
    i number;

begin
    len:=length(str1);

    for i in reverse 1..len
    loop
        str2:=str2 || substr(str1,i,1);
    end loop;

    dbms_output.put_line('Reverse of String is:'||str2);
end;
/

```

PL/SQL Program for Palindrome Number

```

declare
    n number;
    m number;
    rev number:=0;
    r number;

begin
    n:=12321;

```

```

m:=n;

while n>0
loop
    r:=mod(n,10);
    rev:=(rev*10)+r;
    n:=trunc(n/10);
end loop;

if m=rev
then
    dbms_output.put_line('number is palindrome');
else
    dbms_output.put_line('number is not palindrome');
end if;
end;
/

```

PL/SQL Program to Find Factorial of a Number

```

declare
    n number;
    fac number:=1;
    i number;

begin
    n:=&n;

    for i in 1..n
    loop
        fac:=fac*i;
    end loop;

    dbms_output.put_line('factorial='||fac);
end;
/

```


PL/SQL Records

What are records?

Records are another type of datatypes which oracle allows to be defined as a placeholder. Records are composite datatypes, which means it is a combination of different scalar datatypes like char, varchar, number etc. Each scalar data types in the record holds a value. A record can be visualized as a row of data. It can contain all the contents of a row.

Declaring a record:

To declare a record, you must first define a composite datatype; then declare a record for that type.

The following table consolidates the different ways in which you can define and declare a pl/sql record.

Syntax	Usage
TYPE record_type_name IS RECORD (column_name1 datatype, column_name2 datatype, ...);	Define a composite datatype, where each field is scalar.
col_name table_name.column_name%type;	Dynamically define the datatype of a column based on a database column.
record_name record_type_name;	Declare a record based on a user-defined type.
record_name table_name%ROWTYPE;	Dynamically declare a record based on an entire row of a table. Each column in the table corresponds to a field in the record.

Example:

```
1.  
DECLARE  
TYPE employee_type IS RECORD  
(employee_id number(5),  
employee_first_name varchar2(25),  
employee_last_name employee.last_name%type,  
employee_dept employee.dept%type);
```

```
employee_salary employee.salary%type;  
employee_rec employee_type;
```

2.

```
DECLARE  
employee_rec employee%ROWTYPE;
```

Example:

Consider the table emp as below.

```
SQL> select * from emp;
```

ID	NAME	SALARY	DEPTNO
2	b	2500	20
1	a	1000	10

Example #1 – Program-Defined Record

```
record1.sql - Notepad
File Edit Format View Help
DECLARE

eid1 number(3);
ename1 varchar2(10);
esalary1 emp.salary%type;
edeptno1 emp.deptno%type;

TYPE emp_type IS RECORD
(eid number(3),
ename varchar2(10),
esalary emp.salary%type,
edeptno emp.deptno%type);

employee_rec emp_type;

BEGIN
select id,name,salary,deptno into eid1,ename1,esalary1,edeptno1 from emp where id=1;
dbms_output.put_line('employee no is '||eid1 );
dbms_output.put_line('employee name is '||ename1);
dbms_output.put_line('employee slary is '||esalary1);
dbms_output.put_line('employee department no is '||edeptno1);

select * into employee_rec from emp where id=2;
dbms_output.put_line('employee no is '||employee_rec.eid);
dbms_output.put_line('employee name is '||employee_rec.ename);
dbms_output.put_line('employee slary is '||employee_rec.esalary);
dbms_output.put_line('employee department no is '||employee_rec.edeptno);
END;
/

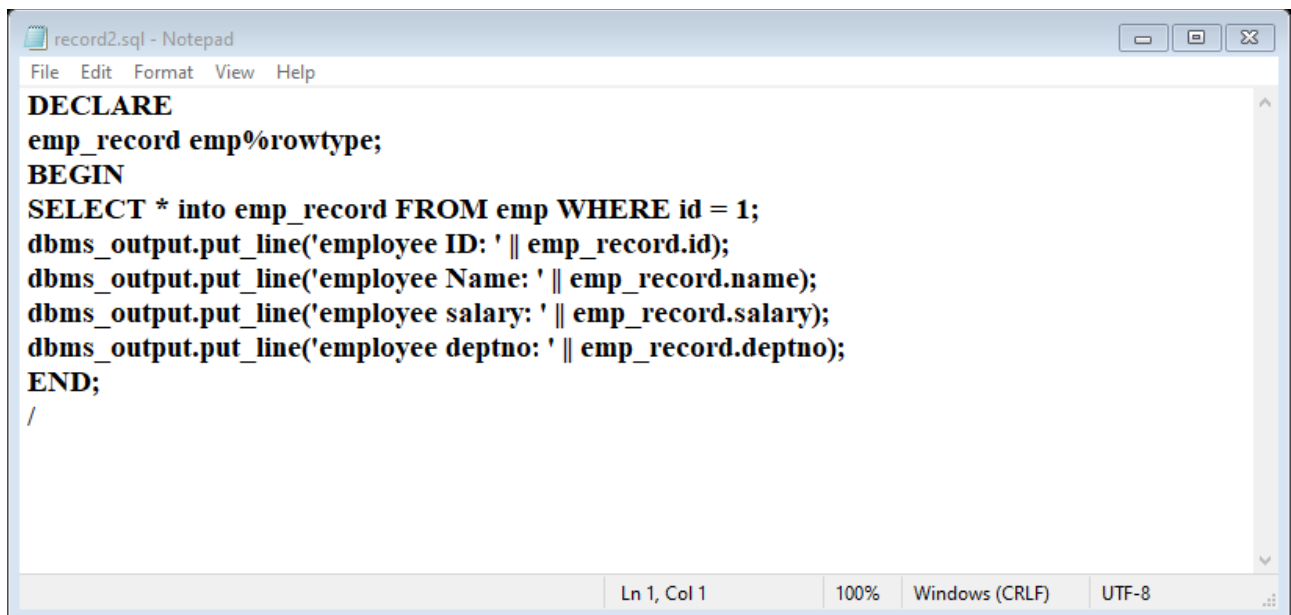
Ln 1, Col 1      100% Windows (CRLF) UTF-8
```

output:

```
SQL> set serverout on
SQL> @record1.sql
employee no is 1
employee name is a
employee slary is 1000
employee department no is 10
employee no is 2
employee name is b
employee slary is 2500
employee department no is 20

PL/SQL procedure successfully completed.
```

Example #2 – Table-based Record

A screenshot of a Notepad window titled 'record2.sql - Notepad'. The window has a menu bar with 'File', 'Edit', 'Format', 'View', and 'Help'. The text area contains the following SQL code:

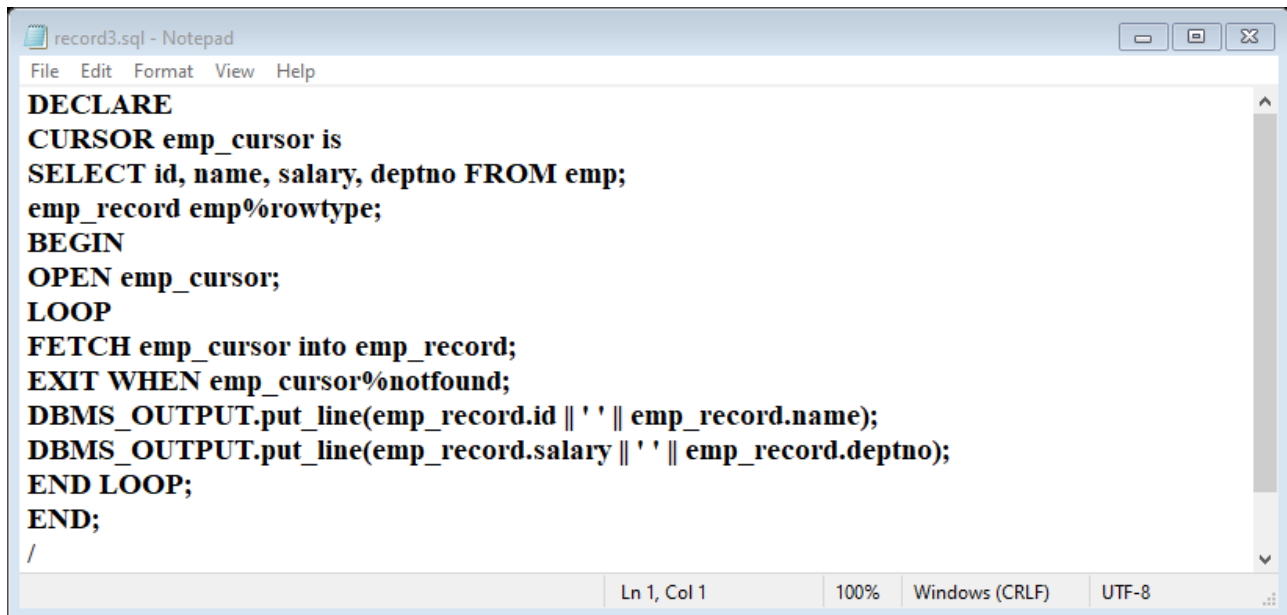
```
DECLARE
emp_record emp%rowtype;
BEGIN
SELECT * into emp_record FROM emp WHERE id = 1;
dbms_output.put_line('employee ID: ' || emp_record.id);
dbms_output.put_line('employee Name: ' || emp_record.name);
dbms_output.put_line('employee salary: ' || emp_record.salary);
dbms_output.put_line('employee deptno: ' || emp_record.deptno);
END;
/
```

The status bar at the bottom shows 'Ln 1, Col 1', '100%', 'Windows (CRLF)', and 'UTF-8'.

```
SQL> @record2.sql
employee ID: 1
employee Name: a
employee salary: 1000
employee deptno: 10

PL/SQL procedure successfully completed.
```

Example #3 – Cursor Based Record



```
record3.sql - Notepad
File Edit Format View Help
DECLARE
CURSOR emp_cursor is
SELECT id, name, salary, deptno FROM emp;
emp_record emp%rowtype;
BEGIN
OPEN emp_cursor;
LOOP
FETCH emp_cursor into emp_record;
EXIT WHEN emp_cursor%notfound;
DBMS_OUTPUT.put_line(emp_record.id || ' ' || emp_record.name);
DBMS_OUTPUT.put_line(emp_record.salary || ' ' || emp_record.deptno);
END LOOP;
END;
/
```

Ln 1, Col 1 100% Windows (CRLF) UTF-8

```
SQL> @record3.sql
2 b
2500 20
1 a
1000 10

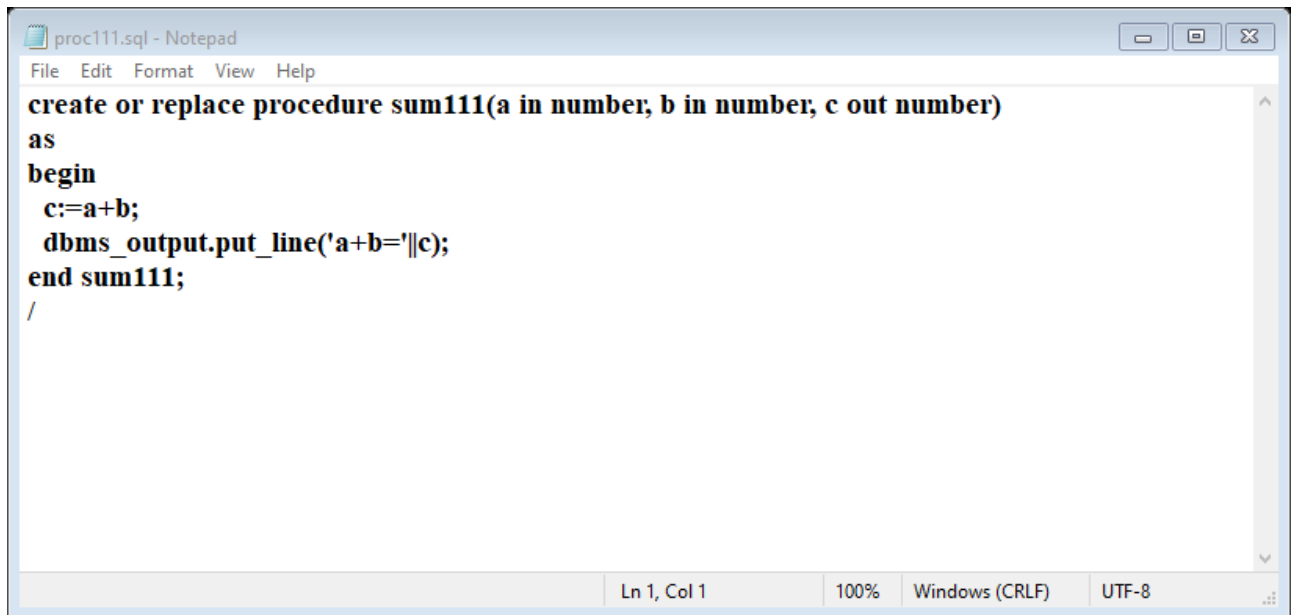
PL/SQL procedure successfully completed.
```

Procedures:

Procedures and functions allow code to be named and stored in the database, making code reuse simpler and more efficient. Procedures and functions still retain the block format, but the DECLARE keyword is replaced by PROCEDURE or FUNCTION definitions,

Stored procedures can also have parameters. These parameters have to be valid SQL types, and have one of three different modes: IN, OUT, or IN OUT. IN parameters are arguments to the stored procedure. OUT parameters are returned from the stored procedure; it assigns values to all OUT parameters that the user can process. INOUT parameters combine the properties of IN and OUT parameters: They contain values to be passed to the stored procedures, and the stored procedure can set their values as return values.

Example:

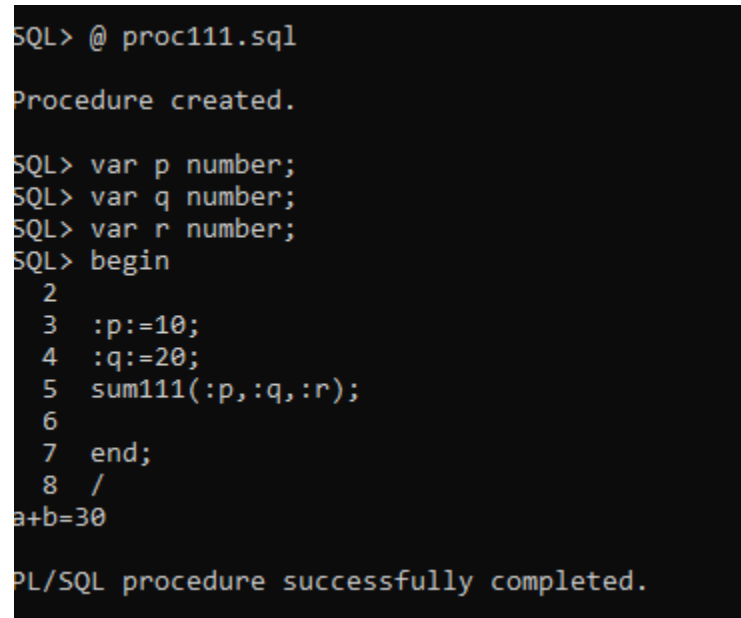


The screenshot shows a Notepad window titled 'proc111.sql - Notepad'. The menu bar includes 'File', 'Edit', 'Format', 'View', and 'Help'. The text area contains the following PL/SQL code:

```
create or replace procedure sum111(a in number, b in number, c out number)
as
begin
  c:=a+b;
  dbms_output.put_line('a+b='||c);
end sum111;
/
```

The status bar at the bottom indicates 'Ln 1, Col 1', '100%', 'Windows (CRLF)', and 'UTF-8'.

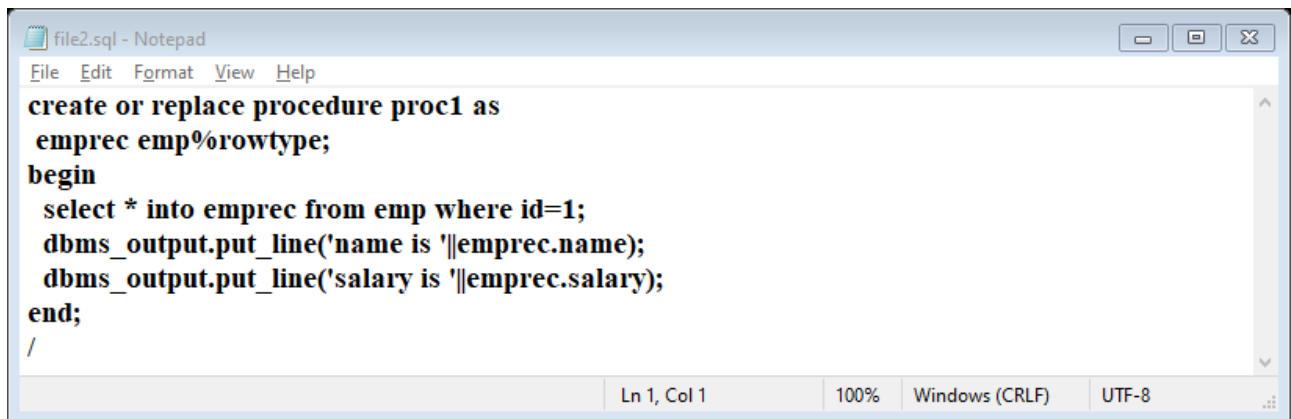
output:



The screenshot shows a SQL*Plus session with the following commands and output:

```
SQL> @ proc111.sql
Procedure created.

SQL> var p number;
SQL> var q number;
SQL> var r number;
SQL> begin
  2
  3  :p:=10;
  4  :q:=20;
  5  sum111(:p,:q,:r);
  6
  7  end;
  8  /
a+b=30
PL/SQL procedure successfully completed.
```

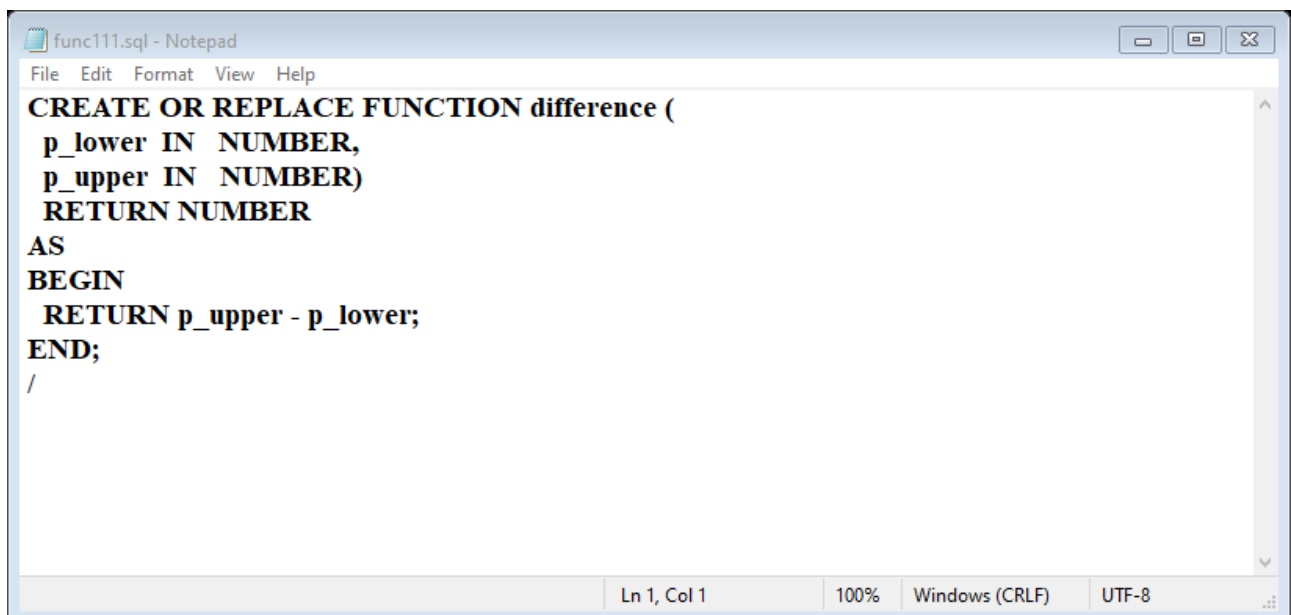


```
file2.sql - Notepad
File Edit Format View Help
create or replace procedure proc1 as
emprec emp%rowtype;
begin
  select * into emprec from emp where id=1;
  dbms_output.put_line('name is '||emprec.name);
  dbms_output.put_line('salary is '||emprec.salary);
end;
/

Ln 1, Col 1    100%    Windows (CRLF)    UTF-8
```

Functions

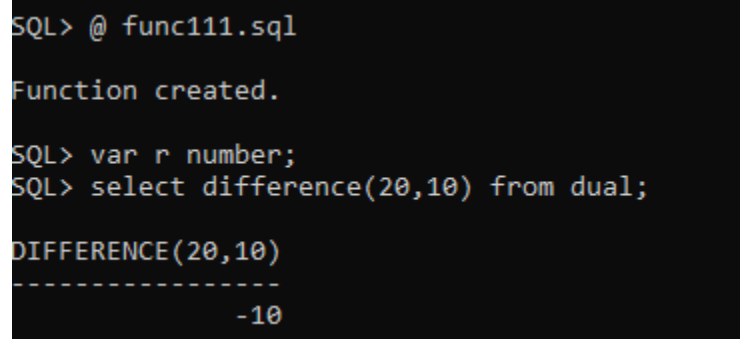
Functions are similar to procedures but return a single value.



```
func111.sql - Notepad
File Edit Format View Help
CREATE OR REPLACE FUNCTION difference (
  p_lower IN NUMBER,
  p_upper IN NUMBER)
RETURN NUMBER
AS
BEGIN
  RETURN p_upper - p_lower;
END;
/

Ln 1, Col 1    100%    Windows (CRLF)    UTF-8
```

output:



```
SQL> @ func111.sql

Function created.

SQL> var r number;
SQL> select difference(20,10) from dual;

DIFFERENCE(20,10)
-----
                -10
```

Example: Using %ROWTYPE with a PL/SQL Record

Proc2.sql

```
CREATE OR REPLACE PROCEDURE PROC2 AS
-- declare record variable that represents a row fetched from the ----
-- employees table
  emp_rec employees%ROWTYPE;
-- declare variable with %ROWTYPE attribute
BEGIN
  SELECT * INTO emp_rec FROM EMPLOYEES WHERE employee_id = 120;
-- retrieve record
  DBMS_OUTPUT.PUT_LINE('Employee name: ' || emp_rec.first_name || ' ' ||
emp_rec.last_name); -- display
END;
/
```

Proc1.sql

```
create or replace procedure proc1 as
  empid emp.empno%TYPE; -- employee_id datatype is NUMBER(6)
  emplname emp.ename%TYPE; -- last_name datatype is VARCHAR2(25)
BEGIN
  empid:= 7900; -- this is OK because it fits in NUMBER(6)
  -- empid := 3018907; -- this is too large and will cause an overflow
  emplname:= 'Patel'; -- this is OK because it fits in VARCHAR2(25)
  DBMS_OUTPUT.PUT_LINE('Employee ID: ' || empid); -- display data
  DBMS_OUTPUT.PUT_LINE('Employee name: ' || emplname); -- display dataet
  update emp set ename=emplname where empno=empid;
END;
/
```

CURSORS

A cursor is a pointer to a private SQL area that stores information about the processing of a SELECT or data manipulation language (DML) statement (INSERT, UPDATE, DELETE, or MERGE). Cursor management of DML statements is handled by Oracle Database, but PL/SQL offers several ways to define and manipulate cursors to execute SELECT statements.

Explicit Cursor vs Implicit Cursor

- Implicit cursors are automatically created and used every time a Select statement is issued in PL/SQL, when there is no explicitly defined cursor. Explicit cursors, as the name suggests, are defined explicitly by the developer. In PL/SQL an explicit cursor is actually a named query defined using the key word cursor.
- Implicit cursors are automatically created and used by Oracle each time a select statement is issued. If an implicit cursor is used, the Database Management System (DBMS) will perform the open, fetch and close operations automatically. An implicit cursor is automatically associated with each Data Manipulation Language (DML) statements, namely INSERT, UPDATE and DELETE statements. Also, an implicit cursor is used to process SELECT INTO statements. When fetching data using implicit cursors NO_DATA_FOUND exception can be raised when the SQL statement returns no data. Furthermore, implicit cursors can raise TOO_MANY_ROWS exceptions when the SQL statement returns more than one row.
- explicit cursors are queries defined using a name. An explicit cursor can be thought of as a pointer to a set of records and the pointer can be moved forward within the set of records. Explicit cursors provide the user the complete control over opening, closing and fetching data. Also, multiple rows can be fetched using an explicit cursor. Explicit cursors can also take parameters just like any function or procedure so that the variables in the cursor can be changed each time it is executed. In addition, explicit cursors allow you to fetch a whole row in to a PL/SQL record variable. When using an explicit cursor, first it needs to be declared using a name. Cursor attributes can be accessed using the name given to cursor. After declaring, cursor needs to be opened first. Then fetching can be started. If multiple rows need to be fetched, the fetching operation needs to be done inside a loop. Finally, the cursor needs to be closed.
- PL/SQL creates implicit cursor and manages automatically means implicit open & close takes place. It used when sql statement return only one row. It has 4 attributes SQL%ROWCOUNT, SQL%FOUND, SQL%NOTFOUND, SQL%ISOPEN.
- .EXPLICIT: It is created & managed by the programmer. It needs every time explicit open,fetch & close. It is used when sql statement returns more than one row. It has also 4 attributes CUR_NAME%ROWCOUNT, CUR_NAME%FOUND, CUR_NAME%NOTFOUND, CUR_NAME%ISOPEN. It process several rows by using loop. The programmer can pass the parameter too to explicit cursor

S.No	Attribute & Description
1	%FOUND

	Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	%NOTFOUND The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
3	%ISOPEN Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	%ROWCOUNT Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Implicit Cursor:

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

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

```
DECLARE
```

```
    total_rows number(2);
```

```
BEGIN
```

```
    UPDATE customers
```

```
    SET salary = salary + 500;
```

```
    IF sql%notfound THEN
```

```
        dbms_output.put_line('no customers selected');
```

```
    ELSIF sql%found THEN
```

```
        total_rows := sql%rowcount;
```

```
        dbms_output.put_line( total_rows || ' customers selected ');
```

```
END IF;
```

```
END;
```

```
/
```

When the above code is executed at the SQL prompt, it produces the following result –

6 customers selected

PL/SQL procedure successfully completed.

If you check the records in customers table, you will find that the rows have been updated –

```
Select * from customers;
```

```
+---+-----+---+-----+-----+
| ID | NAME   | AGE | ADDRESS | SALARY |
+---+-----+---+-----+-----+
| 1 | Ramesh | 32 | Ahmedabad | 2500.00 |
| 2 | Khilan | 25 | Delhi    | 2000.00 |
| 3 | kaushik | 23 | Kota     | 2500.00 |
| 4 | Chaitali | 25 | Mumbai   | 7000.00 |
| 5 | Hardik | 27 | Bhopal   | 9000.00 |
| 6 | Komal | 22 | MP       | 5000.00 |
+---+-----+---+-----+-----+
```

Example 1 of an Implicit cursors

In the following PL/SQL code block, the select statement makes use of an implicit cursor:

```
Begin
```

```
Update emp Where 1=2;
```

```
Dbms_output.put_line (sql%rowcount || ' ' || ' rows are affected by the update statement');
```

```
End;
```

Another Example of an Implicit cursor

The following single-row query calculates and returns the total salary for a department. PL/SQL creates an implicit cursor for this statement:

```
SELECT SUM (salary) INTO department_total
```

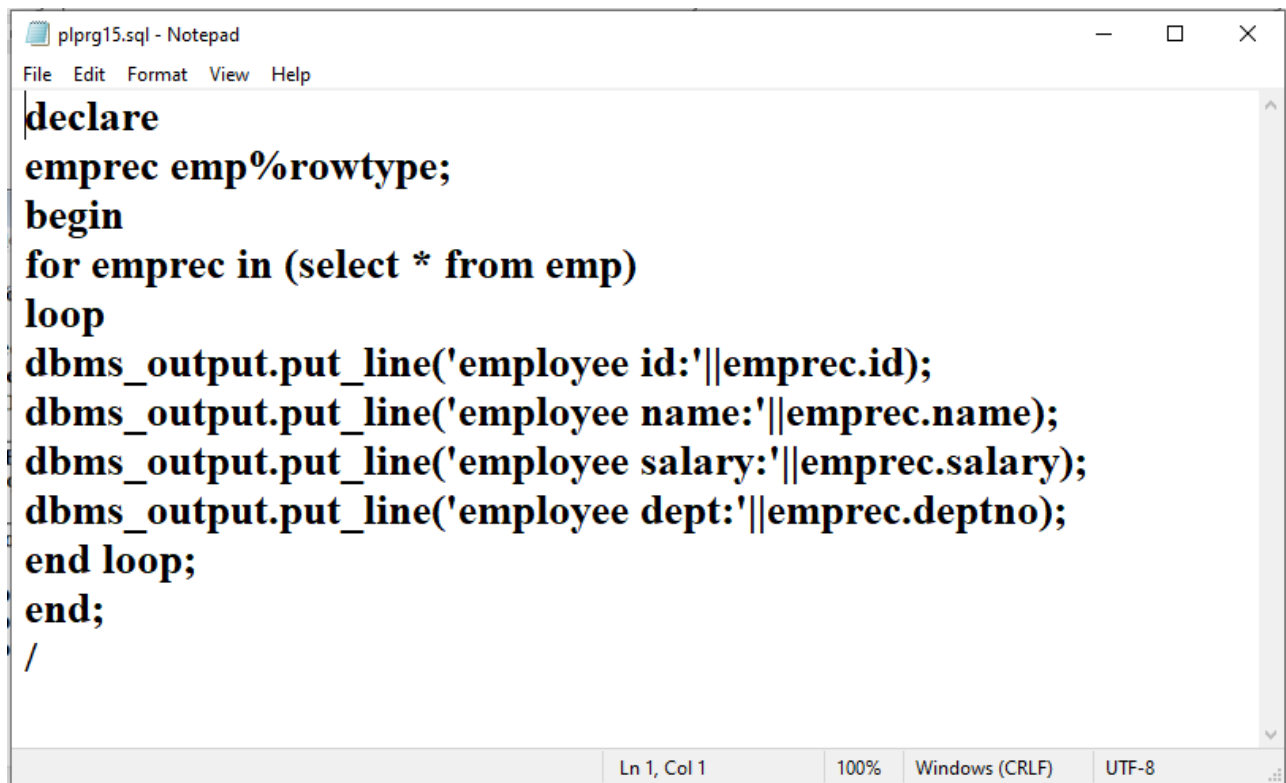
```
FROM employee
```

```
WHERE department_number = 10;
```

An Example of PL/SQL Attribute

```
DECLARE
rows_deleted NUMBER;
BEGIN
DELETE * FROM emp;
rows_deleted := SQL%ROWCOUNT;
END;
```

Example:



```
plprg15.sql - Notepad
File Edit Format View Help
declare
emprec emp%rowtype;
begin
for emprec in (select * from emp)
loop
dbms_output.put_line('employee id: '||emprec.id);
dbms_output.put_line('employee name: '||emprec.name);
dbms_output.put_line('employee salary: '||emprec.salary);
dbms_output.put_line('employee dept: '||emprec.deptno);
end loop;
end;
/
```

Ln 1, Col 1 100% Windows (CRLF) UTF-8

Explicit Cursors

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is –

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps –

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example –

```
CURSOR c_customers IS  
  
    SELECT id, name, address FROM customers;
```

Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows –

```
OPEN c_customers;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows –

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows –

```
CLOSE c_customers;
```

Example

1. Write a cursor program to retrieve the details of all the employees using cursors.

SQL> ed cursor1.sql;

```

declare
cursor cemp is select empno, ename, job, sal from emp;
vemp cemp%rowtype;
begin
open cemp;
dbms_output.put_line(' Details of the employees :- ');
loop
fetch cemp into vemp ;
exit when (cemp% notfound);
dbms_output.put_line(vempno.empno || ' ' || vempno.ename || ' ' ||
vemp.job || ' ' || vempno.sal);
end loop;
close cemp;
end;

```

3. Write a PL/SQL block to update the salary of each employee in department 20 by raising the salary by 20% and insert a record in emprise table.

```

SQL> create table emprise(empno number(5), raisedate date, raiseamt
number(5));

```

Table created.

```

SQL> ed cursor3.sql;

```

```

declare
cursor cempr is select empno, sal from emp where deptno=20;
vemp cempr%rowtype;
begin
open cempr;
loop
fetch cempr into vemp;
exit when cempr% notfound;
update emp set sal = vemp.sal+(vemp.sal*.20) where
empno=vemp.empno;
insert into emprise values(vemp.empno, sysdate, vemp.sal*0.20);
end loop;
commit;
close cempr;
end;

```

Triggers

What are triggers?

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

Events Which Fires the Triggers

These events can be anything such as

1. **A DML Statement.** For example Update, Insert or Delete, 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 statement. Or you can create a trigger which will get triggered after the execution of your INSERT DML statement.
2. 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 such as who created the object or which user. Just like some cool spy tricks.
3. **A system event.** Yes, you can create a trigger on a system event and by system event I mean shut down or startup of your database.
4. 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 and record the information such as time of event occur, the username who created it.

Differences between Procedures and Triggers

A procedure is executed explicitly from another block via a procedure call with passing arguments, while a trigger is executed (or fired) **implicitly** whenever the triggering event (**DML**: INSERT, UPDATE, or DELETE) happens, and a trigger doesn't accept arguments.

When triggers are used?

- Auditing information in a table by recording the changes.
- Automatically signaling other programs that action needs to take place when changes are made to a table
- Collecting/maintaining statistical data.

How to Create Trigger

Below is the syntax for creating a trigger.

Syntax:

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

```
[ BEFORE | AFTER | INSTEAD OF ]
```

Trigger Timing

```
[ INSERT | UPDATE | DELETE.....]
```

Event

```
ON <name of underlying object>
```

```
[ FOR EACH ROW ]
```

Row Level

```
[ WHEN <condition for trigger to get execute> ]
```

Conditional Clause

```
DECLARE
```

```
<Declaration part>
```

```
BEGIN
```

```
<Execution part>
```

```
EXCEPTION
```

```
<Exception handling part>
```

```
END;
```

Trigger names

Triggers exist in a separate namespace from procedure, package, tables (that share the same namespace), which means that a trigger can have the same name as a table or procedure.

Triggering Event or Statement

A triggering event or statement is the SQL statement that causes a trigger to be fired. A triggering event can be an INSERT, UPDATE, or DELETE statement on a table, a DML event or a DDL event or system event or a user event

Trigger Restriction or condition

A trigger restriction specifies a Boolean (logical) expression that must be TRUE for the trigger to fire. The trigger action is not executed if the trigger restriction evaluates to FALSE or UNKNOWN.

A trigger restriction is an option available for triggers that are fired for each row. Its function is to control the execution of a trigger conditionally. You specify a trigger restriction using a WHEN clause.

Trigger Action

A trigger action is the procedure (PL/SQL block) that contains the SQL statements and PL/SQL code to be executed when a triggering statement is issued and the trigger restriction evaluates to TRUE.

Similar to stored procedures, a trigger action can contain SQL and PL/SQL statements, define PL/SQL language constructs (variables, constants, cursors, exceptions, and so on), and call stored procedures. Additionally, for row trigger, the statements in a trigger action have access to column values (new and old) of the current row being processed by the trigger. Two correlation names provide access to the old and new values for each column.

Types of triggers

There are two types of triggers in Oracle including row-level triggers and statement-level triggers

Row-level triggers for data-related activities

- Row-level triggers execute **once for each row** in a transaction.
- Row-level triggers are the most common type of triggers; they are often used in data auditing applications.
- Row-level trigger is identified by the **FOR EACH ROW** clause in the CREATE TRIGGER command.

Statement-level triggers for transaction-related activities

- Statement-level triggers execute **once for each transaction**. For example, if a single transaction inserted 500 rows into the Customer table, then a statement-level trigger on that table would only be executed once.
- Statement-level triggers therefore are not often used for *data-related* activities; they are normally used to enforce additional security measures on the types of transactions that may be performed on a table.
- Statement-level triggers are the default type of triggers created and are identified by **omitting** the **FOR EACH ROW** clause in the CREATE TRIGGER command.

Before and After Triggers

- Since triggers occur because of events, they may be set to occur immediately before or after those events. The events that execute triggers are database transactions, triggers can be executed immediately BEFORE or AFTER the statements INSERTs, UPDATEs, DELETEs.
 - AFTER row-level triggers are frequently used in auditing applications, since they do not fire until the row has been modified.
- Clearly, there is a great deal of flexibility in the design of a trigger.

Valid trigger types (possible combination of triggers)

- **Statement** (INSERT, DELETE, UPDATE), **Timing** (BEFORE, AFTER), **Level** (Row-level, Statement-level)
- The values for the statement, timing, and level determine the types of the triggers. There are total of 12 possible types of triggers: $3 \times 2 \times 2 = 12$

Combining Trigger Types

- Triggers for multiple INSERT, UPDATE, DELETE commands on a table can be combined into a single trigger (using **OR**), provided they are all at the same level (row-level or statement-level), e.g.,
 INSERT OR UPDATE OR DELETE.
- However, you can not combine BEFORE or AFTER, e.g., BEFORE OR AFTER is illegal.

:NEW and :OLD Clause

In a row level trigger, the trigger fires for each related row. And sometimes it is required to know the value before and after the DML statement.

Oracle has provided two clauses in the RECORD-level trigger to hold these values. We can use these clauses to refer to the old and new values inside the trigger body.

:NEW – It holds a new value for the columns of the base table/view during the trigger execution

:OLD – It holds old value of the columns of the base table/view during the trigger execution

This clause should be used based on the DML event. Below table will specify which clause is valid for which DML statement (INSERT/UPDATE/DELETE).

	INSERT	UPDATE	DELETE
:NEW	VALID	VALID	INVALID. There is no new value in delete case.
:OLD	INVALID. There is no old value in insert case	VALID	VALID

Examples

TRIGGERS's PROGRAMS :

1. Trigger on Insertion.

```
SQL> create table log(access_date date, operation varchar2(40), table
varchar(10));
```

Table created.

```
SQL> create or replace trigger trigger2 after insert on class
2 begin
3 insert into log values(sysdate,' 1 row successfully inserted ','
CLASS ');
4 dbms_output.put_line('CONGRATULATIONS!!!');
5 end;
6 /
```

Trigger created.

```
SQL> insert into class values(1260,'Vikram');
```

Output :-

```
CONGRATULATIONS!!!
83
```

1 row created.

```
SQL> select * from log;
ACCESS_DATE OPERATION TABLE
02 – APR – 2005 1 row successfully inserted CLASS
```

2. Trigger on updation.

```
SQL> create or replace trigger trigger3 after update on class
2 begin
3 insert into log values(sysdate,' 1 row successfully updated ',' CLASS ');
4 dbms_output.put_line('Success!!!');
5 end;
6 /
```

Trigger created.

```
SQL> update class set name='Uday' where roll=1260;
```

Output :-

```
Success!!!
1 row updated.
SQL> select * from log;
ACCESS_DATE OPERATION TABLE
02 – APR – 2005 1 row successfully updated CLASS
02 – APR – 2005 1 row successfully inserted CLASS
84
```

3. Trigger on updation (Statement Level Trigger).

```
SQL> create or replace trigger trigger3 after update on class
2 begin
3 dbms_output.put_line ('Success!!!');
4 insert into log values(sysdate, 'One row successfully updated ',' CLASS ');
5 end;
6 /
```

Trigger created.

```
SQL> update class set grade='A' where marks>90;
```

Output :-

```
Success!!!
3 rows updated.
```

```
SQL> select * from log;
```

```
ACCES DATE OPERATION TABLENAME
```

```
85
```

```
02 – APR – 2005 One row successfully updated CLASS
```

```
02 – APR – 2005 1 row successfully updated CLASS
```

```
02 – APR – 2005 1 row successfully inserted CLASS
```

4. Trigger on updation (Row Level Trigger).

```
SQL> create or replace trigger trigger4 after update on class for each row
```

```
2 begin
```

```
3 dbms_output.put_line ('Success!!!');
```

```
4 insert into log values(sysdate, '1 row Updated ', ' CLASS ');
```

```
5 end;
```

```
6 /
```

Trigger created.

```
SQL> update class set grade='A' where marks>90;
```

Output :-

```
Success!!!
```

```
Success!!!
```

```
Success!!!
```

```
3 rows updated.
```

```
SQL> select * from log;
```

```
ACCES DATE OPERATION TABLENAME
```

```
86
```

```
02 – APR – 2005 1 row updated CLASS
```

```
02 – APR – 2005 1 row updated CLASS
```

```
02 – APR – 2005 1 row updated CLASS
```

```
02 – APR – 2005 One row successfully updated CLASS
```

```
02 – APR – 2005 1 row successfully updated CLASS
```

```
02 – APR – 2005 1 row successfully inserted CLASS
```

5. Trigger on Deletion.

```
SQL> create or replace trigger trigger4 after update on class
```

```
2 begin
```

```
3 dbms_output.put_line ('A row deleted!!!');
```

```
4 insert into log values(sysdate, '1 row successfully deleted ', ' CLASS ');
```

```
5 end;
```

```
6 /
```

Trigger created.

```
SQL> delete from class where roll=1201;
```

Output :-

```
A row deleted!!!
```

```
1 row deleted.
```

```
SQL> select * from log;
```

```
ACCES DATE OPERATION TABLENAME
```

```
87
```

```
02 – APR – 2005 1 row successfully deleted CLASS
```

```
02 – APR – 2005 1 row updated CLASS
```

02 – APR – 2005 1 row updated CLASS
02 – APR – 2005 1 row updated CLASS
02 – APR – 2005 One row successfully updated CLASS
02 – APR – 2005 1 row successfully updated CLASS
02 – APR – 2005 1 row successfully inserted CLASS

Example2: Create an update trigger that displays salary changes for every salary updation in customer table

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

```
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
```

Assertions

- An **assertion** is a predicate expressing a condition we wish the database to always satisfy.
- Domain constraints, functional dependency and referential integrity are special forms of assertion.
- Where a constraint cannot be expressed in these forms, we use an assertion, e.g.
 - Ensuring the sum of loan amounts for each branch is less than the sum of all account balances at the branch.

- Ensuring every loan customer keeps a minimum of \$1000 in an account.
- An assertion takes the form,
create assertion assertion-name **check** predicate

Example1: to create an assertion called “salary_assertion” that checks that no employee in the “employees” table has a salary greater than \$100,000, you could use the following statement:

```
CREATE ASSERTION salary_assertion  
CHECK (salary <= 100000);
```

Example 2: to create an assertion named nomanager which checks that all the tuples in manager relation with department_id being NULL as not a manager.

```
CREATE ASSERTION nomanager  
CHECK( NOT EXISTS  
( SELECT * FROM MANAGER  
WHERE Department_id is NULL));
```

Example3: to check the sum of all loan amounts for each branch must be less than the sum of all account balances at the branch. This can be satisfied by creating assertion as below:

```
create assertion sum-constraint check  
(not exists (select * from branch  
where (select sum(amount) from loan  
where loan.branch-name = branch.branch-name)  
>= (select sum(amount) from account  
where loan.branch-name = branch.branch-name))))
```

Example 4:

Ensuring every loan customer keeps a minimum of \$1000 in an account.

create assertion *balance-constraint* **check**

```
(not exists (select * from loan L
              where not exists (select * from borrower B, depositor D, account A
                                where L.loan# = B.loan# and B.cname = D.cname
                                and D.account# = A.account# and A.balance >=
1000 )))
```

To drop an assertion, you can use the following statement:

DROP ASSERTION *assertion_name*;

Assertions are useful for enforcing data integrity and ensuring that the data in the database meets certain conditions. They can be used to enforce business rules or to ensure the consistency of the data. However, they can also be time-consuming to create and maintain, so they are not always used in practice.