# Unit 2: Relational Language: SQL & PL/SQL

## 1. Relational query languages:

Relational Query Languages – So far we have seen what a database is, what is the features of database, how to gather requirements and how to put them in ER diagrams, how to convert them into tables and their columns, set their constraints etc. Once we have database ready users will start using them. But how will they access the database? Most of the time they access the data by using some applications. These applications will communicate to database by SQL and DBMS is responsible for managing the application and SQL intact. SQL has its own querying methods to interact with database. But how these queries work in the database? These queries work similar to relational algebra that we have in mathematics. In database we have tables participating in relational algebra.

Relational query languages use relational algebra to break the user requests and instruct the DBMS to execute the requests. It is the language by which user communicates with the database. These relational query languages can be procedural or non-procedural.

**Procedural Query Language**

A procedural query language will have set of queries instructing the DBMS to perform various transactions in the sequence to meet the user request. For example, get_CGPA procedure will have various queries to get the marks of student in each subject, calculate the total marks, and then decide the CGPA based on his total marks. This procedural query language tells the database what is required from the database and how to get them from the database. Relational algebra is a procedural query language.

**Non-Procedural Query Language**

Non-procedural queries will have single query on one or more tables to get result from the database. For example, get the name and address of the student with particular ID will have single query on STUDENT table. Relational Calculus is a non-procedural language which informs what to do with the tables, but doesn't inform how to accomplish this.

These query languages basically will have queries on tables in the database. In the relational database, a table is known as relation. Records / rows of the table are referred as tuples. Columns of the table are also known as attributes. All these names are used interchangeably in relational database.

## 2. Relational algebra:

Relational algebra is a procedural query language. It takes one or more relations / tables and performs the operation and produce the result. This result is also considered as a new table or relation. Suppose we have to retrieve student name, address and class for the given ID. What a relational algebra will do in this case is, it filters the name, address and class from the STUDENT table for the input ID. In mathematical terms, relational algebra has produced a subset of STUDENT table for the given ID.

Relational algebra will have operators to indicate the operations. This algebra can be applied on single relation – called **unary** or can be applied on two tables – called **binary**. While applying the operations on the relation, the resulting subset of relation is also known as new relation. There can be multiple steps involved in some of the operations. The subsets of relations at the intermediary level are also known as relation. We will understand it better when we see different operations below.

Relational Algebra in DBMS has 6 fundamental operations. There are several other operations defined upon these fundamental operations.

## 2.1 Select (σ)

**Select (σ) –** This is a unary relational operation. This operation pulls the horizontal subset (subset of rows) of the relation that satisfies the conditions. This can use operators like <, >, <=, >=, = and != to filter the data from the relation. It can also use logical AND, OR and NOT operators to combine the various filtering conditions. This operation can be represented as below:

**σ p (r)**

Where σ is the symbol for select operation, r represents the relation/table, and p is the logical formula or the filtering conditions to get the subset. Let us see an example as below:

**σSTD_NAME = “James” (STUDENT)**

What does above relation algebra do? It selects the record/tuple from the STUDENT table with Student name as ‘James’

**σdept_id = 20 AND salary>=10000 (EMPLOYEE)**

Selects the records from EMPLOYEE table with department ID = 20 and employees whose salary is more than 10000.

## 2.2 Project (∏)

Project (∏) – This is a unary operator and is similar to select operation above. It creates the subset of relation based on the conditions specified. Here, it selects only selected columns/attributes from the relation- vertical subset of relation. The select operation above creates subset of relation but for all the attributes in the relation. It is denoted as below:

**∏a1, a2, a3 (r)**

Where ∏ is the operator for projection, r is the relation and a1, a2, a3 are the attributes of the relations which will be shown in the resultant subset.

**∏std_name, address, course (STUDENT)**

This will select all the records from STUDENT table but only selected columns – std_name, address and course. Suppose we have to select only these 3 columns for particular student then we have to combine both project and select operations.

$$\prod_{STD\_ID,\ address,\ course} (\sigma_{STD\_NAME = \text{“James”}}(STUDENT))$$

This selects the record for ‘James’ and displays only std_ID, address and his course columns. Here we can see two unary operators are combined, and it has two operations performing. First it selects the tuple from STUDENT table for ‘James’. The resultant subset of STUDENT is also considered as intermediary relation. But it is temporary and exists till the end of this operation. It then filters the 3 columns from this temporary relation.

## 2.3 Rename (ρ)

**Rename (ρ) –** This is a unary operator used to rename the tables and columns of a relation. When we perform self join operation, we have to differentiate two same tables. In such case rename operator on tables comes into picture. When we join two or more tables and if those tables have same column names, then it is always better to rename the columns to differentiate them. This occurs when we perform Cartesian product operation.

**ρ R(E)**

Where ρ is the rename operator, E is the existing relation name, and R is the new relation name.

**ρ STUDENT (STD_TABLE)**

Renames STD_TABLE table to STUDENT

Let us see another example to rename the columns of the table. If the STUDENT table has ID, NAME and ADDRESS columns and if they have to be renamed to STD_ID, STD_NAME, STD_ADDRESS, then we have to write as follows.

**ρ STD_ID, STD_NAME, STD_ADDRESS(STUDENT)**

It will rename the columns in the order the names appear in the table

## 2.4. Cartesian product (X)

Cartesian product (X): – This is a binary operator. It combines the tuples of two relations into one relation.

**RXS**

Where R and S are two relations and X is the operator. If relation R has m tuples and relation S has n tuples, then the resultant relation will have **mn** tuples. For example, if we perform cartesian product on EMPLOYEE (5 tuples) and DEPT relations (3 tuples), then we will have new tuple with 15 tuples.

**EMPLOYEE X DEPT**

This operator will simply create a pair between the tuples of each table. i.e.; each employee in the EMPLOYEE table will be mapped with each department in DEPT table. Below diagram depicts the result of cartesian product.

## 2.5. Union (U)

**Union (U) –** It is a binary operator, which combines the tuples of two relations. It is denoted by

**R U S**

Where R and S are the relations and U is the operator.

**DESIGN_EMPLOYEE U TESTING_EMPLOYEE**

Where DESIGN_EMPLOYEE and TESTING_EMPLOYEE are two relations.

It is different from cartesian product in:

- Cartesian product combines the attributes of two relations into one relation whereas Union combines the tuples of two relations into one relation.
- In Union, both relations should have same number of columns. Suppose we have to list the employees who are working for design and testing department. Then we will do the union on employee table. Since it is union on same table it has same number of attributes. Cartesian product does not concentrate on number of attribute or rows. It blindly combines the attributes.
- In Union, both relations should have same types of attributes in same order. In the above example, since union is on employee relation, it has same type of attribute in the same order.

It need not have same number of tuples in both the relation. If there is a duplicate tuples as a result of union, then it keeps only one tuple. If a tuple is present in any one relation, then it keeps that tuple in the new relation. In the above example, number of employees in design department need not be same as employees in testing department. Below diagram shows the same. We can observe that it combines the table data in the order they appear in the table.

We would not able to join both these tables if the order of columns or the number of columns were different.

## 2.6 Set-difference (-)

**Set-difference (-) –** This is a binary operator. This operator creates a new relation with tuples that are in one relation but not in other relation. It is denoted by '-'symbol.

**R – S**

Where R and S are the relations.

Suppose we want to retrieve the employees who are working in Design department but not in testing.

**DESIGN_EMPLOYEE −TESTING_EMPLOYEE**

There are additional relational operations based on the above fundamental operations. Some of them are:

## 2.7Set Intersection

**Set Intersection –** This operation is a binary operation. It results in a relation with tuples that are in both the relations. It is denoted by '∩ '.

**R∩S**

Where R and S are the relations. It picks all the tuples that are present in both R and S, and results it in a new relation.

Suppose we have to find the employees who are working in both design and testing department. If we have tuples as in above example, the new result relation will not have any tuples. Suppose we have tuples like below and see the new relation after set difference.

This set intersection can also be written as a combination of set difference operations.

$$R \cap S \quad R-(R-S)$$

i.e.; it evaluates R-S to get the tuples which are present only in R and then it gets the record which are present only in R but not in new resultant relation of R-S.

In above example of employees,

**DESIGN_EMPLOYEE – (DESIGN_EMPLOYEE – TESTING_EMPLOYEE)**

It first filters only those employees who are only design employees – (104, Kathy). This result is then used to find the difference with design employee. This will find those employees who are design employees but not in new result – (100, James). Thus it gives the result tuple which is both designer and tester. We can see here fundamental relational operator is used twice to get set intersection. Hence this operation is not fundamental operation.

## 3. Tuple relational calculus

Tuple Relational Calculus (TRC) is a non-procedural query language used in relational database management systems (RDBMS) to retrieve data from tables. TRC is based on the concept of tuples, which are ordered sets of attribute values that represent a single row or record in a database table.

TRC is a declarative language, meaning that it specifies what data is required from the database, rather than how to retrieve it. TRC queries are expressed as logical formulas that describe the desired tuples.

The basic syntax of TRC is as follows

$$\boxed{\{ t \mid P(t) \}}$$

Where t is a tuple variable and P(t) is a logical formula that describes the conditions that the tuples in the result must satisfy. The curly braces {} are used to indicate that the expression is a set of tuples.

For example, let’s say we have a table called “Employees” with the following attributes:

Employee ID, Name, Salary, Department ID

To retrieve the names of all employees who earn more than $50,000 per year, we can use the following TRC query:

$$\boxed{\{ t \mid Employees(t) \land t.Salary > 50000 \}}$$

In this query, the “Employees (t)” expression specifies that the tuple variable t represents a row in the “Employees” table. The “∧” symbol is the logical AND operator, which is used to combine the condition “t.Salary > 50000” with the table selection.

The result of this query will be a set of tuples, where each tuple contains the Name attribute of an employee who earns more than $50,000 per year.

TRC can also be used to perform more complex queries, such as joins and nested queries, by using additional logical operators and expressions.

While TRC is a powerful query language, it can be more difficult to write and understand than other SQL-based query languages, such as Structured Query Language (SQL). However, it is useful in certain applications, such as in the formal verification of database schemas and in academic research.

Tuple Relational Calculus is a non-procedural query language unlike relational algebra. Tuple Calculus provides only the description of the query but it does not provide the methods to solve it. Thus, it explains what to do but not how to do.

In Tuple Calculus, a query is expressed as

**{t| P(t)}**

Where t = resulting tuples,

P (t) = known as Predicate and these are the conditions that are used to fetch t

Thus, it generates set of all tuples t, such that Predicate P(t) is true for t.

P (t) may have various conditions logically combined with OR (∨), AND (∧), NOT(¬).

It also uses quantifiers:

∃ t ∈ r (Q(t))= ”there exists” a tuple in t in relation r such that predicate Q(t) is true.

∀ t ∈ r (Q(t)) = Q(t)is true “for all” tuples in relation r.

Example:

**Table: Customer**

| **Customer name** | **Street** | **City** |
|---|---|---|
| Saurabh | A7 | Patiala |
| Mehak | B6 | Jalandhar |
| Sumiti | D9 | Ludhiana |
| Ria | A5 | Patiala |

**Table: Branch**

| Branch name | Branch city |
|---|---|
| ABC | Patiala |
| DEF | Ludhiana |
| GHI | Jalandhar |

**Table: Account**

| Account number | Branch name | Balance |
|---|---|---|
| 1111 | ABC | 50000 |
| 1112 | DEF | 10000 |
| 1113 | GHI | 9000 |
| 1114 | ABC | 7000 |

**Table: Loan**

| Loan number | Branch name | Amount |
|---|---|---|
| L33 | ABC | 10000 |
| L35 | DEF | 15000 |
| L49 | GHI | 9000 |
| L98 | DEF | 65000 |

**Table: Borrower**

| Customer name | Loan number |
|---|---|
| Saurabh | L33 |
| Mehak | L49 |
| Ria | L98 |

**Table: Depositor**

| Customer name | Account number |
|---|---|
| Saurabh | 1111 |
| Mehak | 1113 |
| Sumiti | 1114 |

**Queries-1: Find the loan number, branch, amount of loans of greater than or equal to 10000 amount.**

{t| t ∈ loan ∧ t[amount]>=10000}

**Resulting relation:**

| Loan number | Branch name | Amount |
|---|---|---|
| L33 | ABC | 10000 |
| L35 | DEF | 15000 |
| L98 | DEF | 65000 |

In the above query, t[amount] is known as tuple variable.

**Queries-2: Find the loan number for each loan of an amount greater or equal to 10000.**

{t| ∃ s ∈ loan (t[loan number] = s[loan number]

∧ s[amount]>=10000)}

**Resulting relation:**

| **Loan number** |
|---|
| L33 |
| L35 |
| L98 |

**Queries-3: Find the names of all customers who have a loan and an account at the bank.**

{t | ∃ s ∈ borrower ( t[customer-name] = s[customer-name])

∧ ∃ u ∈ depositor ( t[customer-name] = u[customer-name])}

**Resulting relation:**

| **Customer name** |
|---|
| Saurabh |
| Mehak |

**Queries-4: Find the names of all customers having a loan at the "ABC" branch.**

{t | ∃ s ∈ borrower (t[customer-name] = s[customer-name]

∧ ∃ u ∈ loan (u [branch-name] = "ABC" ∧ u[loan-number] = s[loan-number]))}

**Resulting relation:**

| **Customer name** |
|---|
| Saurabh |

## 4. Domain relational calculus

Domain Relational Calculus is a non-procedural query language equivalent in power to Tuple Relational Calculus. Domain Relational Calculus provides only the description of the query but it does not provide the methods to solve it. In Domain Relational Calculus, a query is expressed as,

$\{ < x_1, x_2, x_3, ..., x_n > \mid P (x_1, x_2, x_3, ..., x_n ) \}$

where, $< x_1, x_2, x_3, \ldots, x_n >$ represents resulting domains variables and $P (x_1, x_2, x_3, \ldots, x_n )$ represents the condition or formula equivalent to the Predicate calculus.

**Predicate Calculus Formula:**

1. Set of all comparison operators
2. Set of connectives like and, or, not
3. Set of quantifiers

**Example:**

**Table: Customer**

| Customer name | Street | City |
|---|---|---|
| Debomit | Kadamtala | Alipurduar |
| Sayantan | Udaypur | Balurghat |
| Soumya | Nutanchati | Bankura |
| Ritu | Juhu | Mumbai |

**Table: Loan**

| Loan number | Branch name | Amount |
|---|---|---|
| L01 | Main | 200 |
| L03 | Main | 150 |
| L10 | Sub | 90 |
| L08 | Main | 60 |

**Table: Borrower**

| Customer name | Loan number |
|---|---|
| Ritu | L01 |
| Debomit | L08 |
| Soumya | L03 |

**Query-1: Find the loan number, branch, amount of loans of greater than or equal to 100 amount.**

$\{\prec l, b, a\succ \mid \prec l, b, a\succ \in loan \wedge (a \geq 100)\}$

**Resulting relation:**

| Loan number | Branch name | Amount |
|---|---|---|
| L01 | Main | 200 |
| L03 | Main | 150 |

**Query-2: Find the loan number for each loan of an amount greater or equal to 150.**

$\{\prec l\succ \mid \exists\ b, a\ (\prec l, b, a\succ \in loan \wedge (a \geq 150)\}$

**Resulting relation:**

| Loan number |
|---|
| L01 |
| L03 |

**Query-3: Find the names of all customers having a loan at the "Main" branch and find the loan amount.**

$\{\langle c, a\rangle \mid \exists\, l\, (\langle c, l\rangle \in \text{borrower} \land \exists\, b\, (\langle l, b, a\rangle \in \text{loan} \land (b = \text{"Main"})))\}$

**Resulting relation:**

| Customer Name | Amount |
|---|---|
| Ritu | 200 |
| Debomit | 60 |
| Soumya | 150 |

**Note:**

**The domain variables those will be in resulting relation must appear before | within ≺ and ≻ and all the domain variables must appear in which order they are in original relation or table.**

## 5.SQL3

### SQL (Structured Query Language)

SQL is a language that enables you to create and operate on relational databases, which are sets of related information stored in tables.

### RELATIONAL MODEL TERMINOLOGY

1. **Relation:** A table storing logically related data is called a Relation.

2. **Tuple:** A **row of a relation** is generally referred to as a tuple.

3. **Attribute:** A **column** of a relation is generally referred to as an attribute.

4. **Degree:** This refers to the **number of attributes** in a relation.

5. **Cardinality:** This refers to the **number of tuples** in a relation.
6. **Primary Key:** This refers to a set of one or more attributes that can uniquely identify tuples within the relation.
7. **Candidate Key:** All attribute combinations inside a relation that can serve as primary key are candidate keys as these are candidates for primary key position.

8. **Alternate Key:** A candidate key that is not primary key, is called an alternate key.

9. **Foreign Key:** A non-key attribute, whose values are derived from the primary key of some other table, is known as foreign key in its current table.

## REFERENTIAL INTEGRITY

- A referential integrity is a system of rules that a DBMS uses to ensure that relationships between records in related tables are valid, and that users don't accidentally delete or change related data. This integrity is ensured by foreign key.

## CLASSIFICATION OF SQL STATEMENTS

SQL commands can be mainly divided into following categories:

1. **Data Definition Language(DDL) Commands**

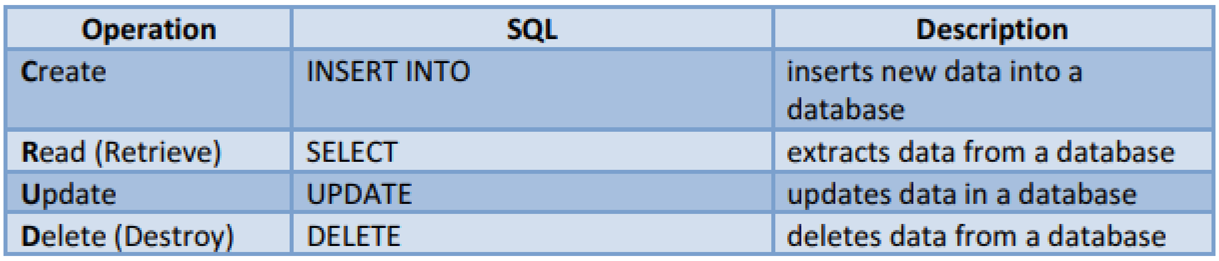
   Commands that allow you to perform task, related to data definition e.g;

   - Creating, altering and dropping.
   - Granting and revoking privileges and roles.
   - Maintenance commands.

2. **Data Manipulation Language(DML) Commands**

   Commands that allow you to perform data manipulation e.g., retrieval, insertion, deletion and modification of data stored in a database.

   The Data Manipulation Language (DML) is the subset of SQL used to add, update and delete data. The acronym CRUD refers to all of the major functions that need to be implemented in a relational database application to consider it complete. Each letter in the acronym can be mapped to a standard SQL statement:



| Operation | SQL | Description |
|---|---|---|
| Create | INSERT INTO | inserts new data into a database |
| Read (Retrieve) | SELECT | extracts data from a database |
| Update | UPDATE | updates data in a database |
| Delete (Destroy) | DELETE | deletes data from a database |

3. **Transaction Control Language(TCL) Commands**

   Commands that allow you to manage and control the transactions e.g.,

   - Making changes to database, permanent
   - Undoing changes to database, permanent
   - Creating savepoints
   - Setting properties for current transactions.

## SQL ELEMENTS

1. Literals
2. Data types
3. Nulls
4. Comments

## LITERALS

It refer to a fixed data value. This fixed data value may be of character type or numeric type. For example, ‘replay’, ‘Raj’, ‘8’ , ‘306’ are all character literals.

**Numbers not enclosed in quotation marks are numeric literals.** E.g. 22 , 18 , 1997 are all numeric literals.

Numeric literals can either be integer literals i.e., without any decimal or be real literals i.e. with a decimal point
e.g. 17 is an integer literal but 17.0 and 17.5 are real literals.

## DATA TYPES

Data types are means to identify the type of data and associated operations for handling it. MySQL data types are divided into three categories:

- Numeric
- Date and time
- String types

### Numeric Data Type

1. int – used for number without decimal.
2. Decimal (m,d) – used for floating/real numbers. m denotes the total length of number and d is number of decimal

   digits.

### Date and Time Data Type

1. date – used to store date in YYYY-MM-DD format.
2. Time – used to store time in HH:MM:SS format.

### String Data Types

1. char (m) – used to store a fixed length string. **m** denotes max. number of characters.
2. varchar (m) – used to store a variable length string. **m** denotes max. no. of characters.

## DIFFERENCE BETWEEN CHAR AND VARCHAR DATA TYPE

| S.NO. | Char Datatype | Varchar Datatype |
|---|---|---|
| 1. | It specifies a **fixed length** character String. | It specifies a **variable length** character string. |
| 2. | When a column is given datatype as CHAR(**n**), then MySQL ensures that all values stored in that column have this length i.e. **n** bytes. If a value is shorter than this length **n** then blanks are added, but the size of value remains **n** bytes. | When a column is given datatype as VARCHAR(**n**), then the maximum size a value in this column can have is **n** bytes. Each value that is stored in this column store exactly as you specify it i.e. no blanks are added if the length is shorter than maximum length **n**. |

## NULL VALUE

If a column in a row has no value, then column is said to be **null** , or to contain a null. **You should use a null value**
when the actual value is not known or when a value would not be meaningful.

## DATABASE COMMNADS

### 1. VIEW EXISTING DATABASE

To view existing database names, the command is : **SHOW DATABASES ;**

### 2. CREATING DATABASE

For creating the database we write the following command : **CREATE DATABASE** <databasename> ;

e.g. In order to create a database Student, command is :

**CREATE DATABASE** Student ;

### 3. ACCESSING DATABASE

For accessing already existing database , we write :

**USE** <databasename> ;

e.g. to access a database named Student , we write command as :

**USE** Student ;

## 4. DELETING DATABASE

For deleting any existing database , the command is :

**DROP DATABASE** <databasename> ;

e.g. to delete a database , say student, we write command

as ; **DROP DATABASE** Student ;

## 5. VIEWING TABLE IN DATABASE

In order to view tables present in currently accessed database , command is : **SHOW TABLES ;**

## CREATING TABLES

- Tables are created with the CREATE TABLE command. When a table is created, its columns are named, data types and sizes are supplied for each column.

**Syntax of CREATE TABLE command is :**

```
CREATE TABLE <table-name>
        ( <column name> <data type> ,
        <column name> <data type> ,
                        ……… ) ;
```

**E.g.** in order to create table EMPLOYEE given below :

| ECODE | ENAME | GENDER | GRADE | GROSS |
|---|---|---|---|---|

We write the following ommand :

```
CREATE TABLE employee (   ECODE integer ,
            ENAME varchar(20) , GENDER char(1) ,
            GRADE char(2) , GROSS integer     ) ;
```

## INSERTING DATA INTO TABLE

- The rows are added to relations(table) using INSERT command of SQL. Syntax of INSERT is : INSERT INTO <tablename> [<column list>] VALUE ( <value1> , <value2> , …..) ;

e.g. to enter a row into EMPLOYEE table (created above), we write

command as : INSERT INTO employee
VALUES(1001 , ‘Ravi’ , ‘M’ , ‘E4’ , 50000);

**OR**

INSERT INTO employee (ECODE , ENAME , GENDER , GRADE , GROSS) VALUES(1001 , ‘Ravi’ , ‘M’ , ‘E4’ , 50000);

| **ECODE** | **ENAME** | **GENDER** | **GRADE** | **GROSS** |
|---|---|---|---|---|
| 1001 | Ravi | M | E4 | 50000 |

In order to insert another row in EMPLOYEE table , we write again INSERT command : INSERT INTO employee
VALUES(1002 , ‘Akash’ , ‘M’ , ‘A1’ , 35000);

| **ECODE** | **ENAME** | **GENDER** | **GRADE** | **GROSS** |
|---|---|---|---|---|
| 1001 | Ravi | M | E4 | 50000 |
| 1002 | Akash | M | A1 | 35000 |

## INSERTING NULL VALUES

- To insert value NULL in a specific column, we can type NULL without quotes and NULL will be inserted in that column. E.g. in order to insert NULL value in ENAME column of above table, we write INSERT command as :

  INSERT INTO EMPLOYEE
  VALUES (1004 , NULL , ‘M’ , ‘B2’ , 38965 ) ;

| **ECODE** | **ENAME** | **GENDER** | **GRADE** | **GROSS** |
|---|---|---|---|---|
| 1001 | Ravi | M | E4 | 50000 |
| 1002 | Akash | M | A1 | 35000 |
| 1004 | NULL | M | B2 | 38965 |

## SIMPLE QUERY USING SELECT COMMAND

- The SELECT command is used to pull information from a table.

  Syntax of SELECT command is : SELECT <column name>,<column name>
  FROM
  <tablename>

WHERE
<condition
name> ;

## SELECTING ALL DATA

- In order to retrieve everything (all columns) from a table, SELECT command is used as : **SELECT * FROM** <tablename> ;

e.g.

In order to retrieve everything from **Employee** table, we write SELECT command as :

**EMPLOYEE**

| **ECODE** | **ENAME** | **GENDER** | **GRADE** | **GROSS** |
|---|---|---|---|---|
| 1001 | Ravi | M | E4 | 50000 |
| 1002 | Akash | M | A1 | 35000 |
| 1004 | NULL | M | B2 | 38965 |

**SELECT * FROM** Employee ;

## SELECTING PARTICULAR COLUMNS

**EMPLOYEE**

| **ECODE** | **ENAME** | **GENDER** | **GRADE** | **GROSS** |
|---|---|---|---|---|
| 1001 | Ravi | M | E4 | 50000 |
| 1002 | Akash | M | A1 | 35000 |
| 1004 | Neela | F | B2 | 38965 |
| 1005 | Sunny | M | A2 | 30000 |
| 1006 | Ruby | F | A1 | 45000 |
| 1009 | Neema | F | A2 | 52000 |

- A particular column from a table can be selected by specifying column-names with SELECT command. E.g. in above table, if we want to select ECODE and ENAME column, then command is :

**SELECT** ECODE , ENAME
**FROM** EMPLOYEE ;

**E.g.2** in order to select only ENAME, GRADE and GROSS column, the command is :

**SELECT**
ENAME ,
GRADE ,
GROSS **FROM**
EMPLOYEE ;

## SELECTING PARTICULAR ROWS

We can select particular rows from a table by specifying a condition through **WHERE clause** along with SELECT statement. **E.g.** In employee table if we want to select rows where Gender is female, then command is :

SELECT * FROM **EMPLOYEE**
WHERE **GENDER = 'F' ;**

E.g.2. in order to select rows where salary is greater than 48000, then
command is : SELECT * FROM **EMPLOYEE**
WHERE **GROSS** > **48000 ;**

## ELIMINATING REDUNDANT DATA

The **DISTINCT** keyword eliminates duplicate rows from the results of a SELECT statement. For example ,

**SELECT** GENDER **FROM** EMPLOYEE ;

| GENDER |
|---|
| M |
| M |
| F |
| M |
| F |
| F |

**SELECT DISTINCT**(GENDER) **FROM** EMPLOYEE ;

| **DISTINCT**(GENDER) |
|---|
| M |
| F |

## VIEWING STRUCTURE OF A TABLE

- If we want to know the structure of a table, we can use DESCRIBE or DESC command, as per following syntax :

**DESCRIBE | DESC** <tablename> ;

**e.g.** to view the structure of table **EMPLOYEE**, command is : **DESCRIBE** EMPLOYEE ; **OR** **DESC** EMPLOYEE ;

## USING COLUMN ALIASES

- The columns that we select in a query can be given a different name, i.e. column alias name for output purpose.

**Syntax :**

SELECT <columnname> **AS** column alias , <columnname> **AS** column alias ….. FROM <tablename> ;

**e.g.** In output, suppose we want to display ECODE column as EMPLOYEE_CODE in output , then command is :

SELECT **ECODE AS "EMPLOYEE_CODE"**
FROM **EMPLOYEE ;**

## CONDITION BASED ON A RANGE

- The **BETWEEN** operator defines a range of values that the column values must fall in to make the condition true. The range include both lower value and upper value.

e.g. to display ECODE, ENAME and GRADE of those employees whose salary is between 40000 and 50000, command is:

SELECT ECODE ,
ENAME ,GRADE
FROM **EMPLOYEE**
WHERE GROSS **BETWEEN** 40000 AND 50000 ;

**Output will be :**

| ECODE | ENAME | GRADE |
|---|---|---|
| 1001 | Ravi | E4 |
| 1006 | Ruby | A1 |

## CONDITION BASED ON A LIST

- To specify a list of values, IN operator is used. The IN operator selects value that match any value in a given list of values. E.g.

  SELECT * FROM
  EMPLOYEE
  WHERE GRADE
  **IN** (‘A1’ , ‘A2’);

**Output will be :**

| ECODE | ENAME | GENDER | GRADE | GROSS |
|---|---|---|---|---|
| 1002 | Akash | M | A1 | 35000 |
| 1006 | Ruby | F | A1 | 45000 |
| 1005 | Sunny | M | A2 | 30000 |
| 1009 | Neema | F | A2 | 52000 |

- The **NOT IN** operator finds rows that do not match in the list. E.g.

  SELECT * FROM EMPLOYEE
  WHERE GRADE **NOT IN** (‘A1’ , ‘A2’);

  **Output will be :**

| ECODE | ENAME | GENDER | GRADE | GROSS |
|---|---|---|---|---|
| 1001 | Ravi | M | E4 | 50000 |
| 1004 | Neela | F | B2 | 38965 |

## CONDITION BASED ON PATTERN MATCHES

- LIKE operator is used for pattern matching in SQL. Patterns are described using two special wildcard characters:

  1. percent(%) – The % character matches any substring.
  2. underscore(_) – The _ character matches any character.

  **e.g.** to display names of employee whose name starts with R in EMPLOYEE table, the command is :

SELECT ENAME FROM **EMPLOYEE**

WHERE ENAME **LIKE 'R%';**

**Output will be :**

| ENAME |
|---|
| RAVI |
| RUBY |

**e.g.** to display details of employee whose second character in name is 'e'. SELECT *

FROM **EMPLOYEE**

WHERE **ENAME LIKE '_e%' ;**

**Output will be :**

| ECODE | ENAME | GENDER | GRADE | GROSS |
|---|---|---|---|---|
| 1004 | Neela | F | B2 | 38965 |
| 1009 | Neema | F | A2 | 52000 |

**e.g.** to display details of employee whose name ends with 'y'. SELECT *

FROM **EMPLOYEE**

WHERE **ENAME LIKE '%y' ;**

**Output will be :**

| ECODE | ENAME | GENDER | GRADE | GROSS |
|---|---|---|---|---|
| 1005 | Sunny | M | A2 | 30000 |
| 1006 | Ruby | F | A1 | 45000 |

## SEARCHING FOR NULL

- The NULL value in a column can be searched for in a table using IS NULL in the WHERE clause. E.g. to list employee details whose salary contain NULL, we use the command :

SELECT *

FROM **EMPLOYEE**

WHERE GROSS **IS NULL ;**

e.g.**STUDENT**

| **Roll_No** | **Name** | **Marks** |
|---|---|---|
| 1 | ARUN | NULL |
| 2 | RAVI | 56 |
| 4 | SANJAY | NULL |

to display the names of those students whose marks is NULL, we use the command :

```
SELECT Name
FROM EMPLOYEE
WHERE Marks IS NULL ;
```

**Output will be :**

| **Name** |
|---|
| ARUN |
| SANJAY |

## SORTING RESULTS

Whenever the SELECT query is executed , the resulting rows appear in a predecided order.The **ORDER BY clause** allow sorting of query result. The sorting can be done either in ascending or descending order, the default is ascending.

The **ORDER BY clause is used as :**

```
SELECT <column name> ,
<column name>.... FROM
<tablename>
WHERE <condition>
ORDER BY <column name> ;
```

**e.g.** to display the details of employees in EMPLOYEE table in alphabetical order, we

```
use command : SELECT *
FROM
EMPLOYE
E ORDER
BY ENAME
;
```

**Output will be :**

| ECODE | ENAME | GENDER | GRADE | GROSS |
|---|---|---|---|---|
| 1002 | Akash | M | A1 | 35000 |
| 1004 | Neela | F | B2 | 38965 |
| 1009 | Neema | F | A2 | 52000 |
| 1001 | Ravi | M | E4 | 50000 |
| 1006 | Ruby | F | A1 | 45000 |
| 1005 | Sunny | M | A2 | 30000 |

**e.g.** display list of employee in descending alphabetical order whose salary is greater than 40000.

```
SELECT ENAME
FROM
EMPLOYEE
WHERE
GROSS    >
40000
ORDER BY
ENAME
desc ;
```

Output Will be:

| ENAME |
|---|
| Ravi |
| Ruby |
| Neema |

## MODIFYING DATA IN TABLES

you can modify data in tables using UPDATE command of SQL. The UPDATE command specifies the rows to be changed using the WHERE clause, and the new data using the SET keyword. Syntax of update command is :

```
UPDATE <tablename>
SET          <columnname>=value          ,
<columnname>=value                    WHERE
<condition> ;
```

**e.g.** to change the salary of employee of those in EMPLOYEE table having employee code 1009 to 55000.

```
UPDATE EMPLOYEE  SET GROSS = 55000 WHERE
ECODE = 1009 ;
```

## UPDATING MORE THAN ONE COLUMNS

**e.g.** to update the salary to 58000 and grade to B2 for those employee whose employee code is 1001.

```
UPDATE EMPLOYEE
SET GROSS = 58000, GRADE='B2'
WHERE ECODE = 1009 ;
```

## OTHER EXAMPLES

**e.g.1.** Increase the salary of each employee by 1000 in the EMPLOYEE table.

```
UPDATE EMPLOYEE
SET GROSS = GROSS +100 ;
```

**e.g.2.** Double the salary of employees having grade as 'A1' or 'A2' .

```
UPDATE EMPLOYEE
SET GROSS = GROSS * 2 ;
WHERE GRADE='A1' OR GRADE='A2' ;
```

**e.g.3.** Change the grade to 'A2' for those employees whose employee code is 1004 and name is Neela.

```
UPDATE EMPLOYEE
SET GRADE='A2'
WHERE ECODE=1004 AND GRADE='NEELA' ;
```

## DELETING DATA FROM TABLES

To delete some data from tables, DELETE command is used. **The DELETE command removes rows from a table.** The syntax of DELETE command is :

```
DELETE FROM <tablename>
WHERE <condition> ;
```

For example, to remove the details of those employee from EMPLOYEE table whose grade is A1.

**DELETE** FROM EMPLOYEE

**WHERE GRADE ='A1'** ;

**<u>TO DELETE ALL THE CONTENTS FROM A TABLE</u>**

**DELETE FROM EMPLOYEE ;**

So if we do not specify any condition with WHERE clause, then all the rows of the table will be deleted. Thus above line will delete all rows from employee table.

## <u>DROPPING TABLES</u>

The DROP TABLE command lets you drop a table from the database. The **syntax of DROP TABLE** command is :

**DROP TABLE** <tablename> ;

**e.g.** to drop a table employee, we need to write :

**DROP TABLE** employee ;

Once this command is given, the table name is no longer recognized and no more commands can be given on that table.

After this command is executed, all the data in the table along with table structure will be deleted.

| S.NO. | DELETE COMMAND | DROP TABLE COMMAND |
|---|---|---|
| **1** | It is a DML command. | It is a DDL Command. |
| **2** | This command is used to delete only rows of data from a table | This command is used to delete all the data of the table along with the structure of the table. The table is no longer recognized when this command gets executed. |
| **3** | Syntax of DELETE command is:<br>**DELETE FROM** <tablename><br>**WHERE** <condition> ; | Syntax of DROP command is :<br>**DROP TABLE** <tablename> ; |

## <u>ALTER TABLE COMMAND</u>

The ALTER TABLE command is used to change definitions of existing tables.(adding columns,deleting columns etc.). The ALTER TABLE command is used for :

1. adding columns to a table
2. Modifying column-definitions of a table.

3. Deleting columns of a table.
4. Adding constraints to table.
5. Enabling/Disabling constraints.

## ADDING COLUMNS TO TABLE

To add a column to a table, ALTER TABLE command can be used as per following syntax:

**ALTER TABLE** <tablename>

**ADD** <Column name> <datatype> <constraint> ;

**e.g.** to add a new column ADDRESS to the EMPLOYEE table, we can write command as :

**ALTER TABLE** EMPLOYEE

**ADD** ADDRESS VARCHAR(50);

**A new column by the name ADDRESS will be added to the table, where each row will contain NULL value for the new column.**

| ECODE | ENAME | GENDER | GRADE | GROSS | ADDRESS |
|---|---|---|---|---|---|
| 1001 | Ravi | M | E4 | 50000 | NULL |
| 1002 | Akash | M | A1 | 35000 | NULL |
| 1004 | Neela | F | B2 | 38965 | NULL |
| 1005 | Sunny | M | A2 | 30000 | NULL |
| 1006 | Ruby | F | A1 | 45000 | NULL |
| 1009 | Neema | F | A2 | 52000 | NULL |

However **if you specify NOT NULL constraint while adding a new column**, MySQL adds the new column with the default value of that datatype e.g. for INT type it will add 0 , for CHAR types, it will add a space, and so on.

**e.g.** Given a table namely Testt with the following data in it.

| Col 1 | Col2 |
|---|---|
| 1 | A |
| 2 | G |

Now following commands are given for the table. Predict the table contents after each of the following statements:

(i) ALTER TABLE testt ADD col3 INT ;

(ii) ALTER TABLE testt ADD col4 INT NOT NULL ;

(iii) ALTER TABLE testt ADD col5 CHAR(3) NOT NULL ;

(iv) ALTER TABLE testt ADD col6 VARCHAR(3);

## MODIFYING COLUMNS

**Column name and data type of column** can be changed as per following syntax :

**ALTER TABLE** <table name>

**CHANGE** <old column name> <new column name> <new datatype>; If **Only data type of column need to be changed**, then

**ALTER TABLE** <table name>

**MODIFY** <column name> <new datatype>;

**e.g.1.** In table EMPLOYEE, change the column GROSS to SALARY.

```
ALTER TABLE EMPLOYEE
CHANGE GROSS SALARY INTEGER;
```

**e.g.2.** In table EMPLOYEE , change the column ENAME to EM_NAME and data type from VARCHAR(20) to VARCHAR(30).

```
ALTER TABLE EMPLOYEE
CHANGE ENAME EM_NAME VARCHAR(30);
```

**e.g.3.** In table EMPLOYEE , change the datatype of GRADE column from CHAR(2) to VARCHAR(2).

```
ALTER TABLE EMPLOYEE
MODIFY GRADE VARCHAR(2);
```

## DELETING COLUMNS

To delete a column from a table, the ALTER TABLE command takes the following form :

```
ALTER TABLE <table name>
DROP <column name>;
```

**e.g.** to delete column GRADE from table EMPLOYEE, we will write :

```
ALTER TABLE EMPLOYEE
DROP GRADE ;
```

## ADDING/REMOVING CONSTRAINTS TO A TABLE

ALTER TABLE statement can be used to add constraints to your existing table by using it in following manner:

### TO ADD PRIMARY KEY CONSTRAINT

```
ALTER TABLE <table name>
ADD PRIMARY KEY (Column name);
```

**e.g.** to add PRIMARY KEY constraint on column ECODE of table EMPLOYEE , the command is :

```
ALTER TABLE EMPLOYEE
ADD PRIMARY KEY (ECODE) ;
```

□

### TO ADD FOREIGN KEY CONSTRAINT

**ALTER TABLE** <table name>

**ADD FOREIGN KEY** (Column name) REFERENCES Parent Table

(Primary key of Parent Table);

## REMOVING CONSTRAINTS

- To remove primary key constraint from a table, we use ALTER TABLE command as : **ALTER TABLE** <table name>

  **DROP** PRIMARY KEY ;

- To remove foreign key constraint from a table, we use ALTER TABLE command as : **ALTER TABLE** <table name>

  **DROP** FOREIGN KEY ;

## ENABLING/DISABLING CONSTRAINTS

Only foreign key can be disabled/enabled in MySQL.

**To disable foreign keys :** **SET** FOREIGN_KEY_CHECKS = 0 ;

**To enable foreign keys :** **SET** FOREIGN_KEY_CHECKS = 1 ;

## INTEGRITY CONSTRAINTS/CONSTRAINTS

- A constraint is a condition or check applicable on a field(column) or set of fields(columns).
- Common types of constraints include :

| S.No. | Constraints | Description |
|---|---|---|
| 1 | NOT NULL | Ensures that a column cannot have NULL value |
| 2 | DEFAULT | Provides a default value for a column when none is specified |
| 3 | UNIQUE | Ensures that all values in a column are different |
| 4 | CHECK | Makes sure that all values in a column satisfy certain criteria |
| 5 | PRIMARY KEY | Used to uniquely identify a row in the table |
| 6 | FOREIGN KEY | Used to ensure referential integrity of the data |

## NOT NULL CONSTRAINT

By default, a column can hold NULL. It you not want to allow NULL value in a column, then NOT NULL constraint must be applied on that column. E.g.

**CREATE TABLE** Customer

( SID integer **NOT NULL** , Last_Name varchar(30) First_Name varchar(30) **NOT NULL** ,

) ;

Columns **SID** and **Last_Name** cannot include NULL, while **First_Name** can include NULL.

An attempt to execute the following SQL statement,

**INSERT INTO** Customer

**VALUES** (NULL , 'Kumar' , 'Ajay');

will result in an error because this will lead to column SID being NULL, which violates the NOT NULL constraint on that column.

**DEFAULT CONSTRAINT**

The DEFAULT constraint provides a default value to a column when the INSERT INTO statement does not provide a specific value. **E.g.**

```
CREATE
TABLE
Student     (
Student_ID
integer ,
   Name varchar(30) ,
   Score integer DEFAULT 80);
```

When following SQL statement is executed on table created above:

**INSERT INTO** Student

**VALUES** (10 , 'Ravi' ); ← **no value has been provided for score field.**

Then table **Student** looks like the following:

| Student_ID | Name | Score |
|---|---|---|
| 10 | Ravi | **80** |

← **score field has got the default value**

**UNIQUE CONSTRAINT**

- The UNIQUE constraint ensures that all values in a column are distinct. In other words, no two rows can hold the same value for a column with UNIQUE constraint.

**e.g.**

**CREATE TABLE** Customer

( SID integer

**Unique** ,

Last_Name

varchar(30) ,

First_Name

varchar(30) ) ;

Column SID has a unique constraint, and hence cannot include duplicate values. So, if the table already contains the following rows :

| **SID** | **Last_Name** | **First_Name** |
|---|---|---|
| 1 | Kumar | Ravi |
| 2 | Sharma | Ajay |
| 3 | Devi | Raj |

The executing the following SQL statement,

**INSERT INTO** Customer

**VALUES** (‘3’ , ‘Cyrus‘ , ‘Grace’) ;

will result in an error because the value 3 already exist in the SID column, thus trying to insert another row with that value violates the UNIQUE constraint.

## CHECK CONSTRAINT

- The CHECK constraint ensures that all values in a column satisfy certain conditions. Once defined, the table will only insert a new row or update an existing row if the new value satisfies the CHECK constraint.

  e.g.

  **CREATE TABLE** Customer

  ( SID integer **CHECK (SID > 0)**,

  Last_Name

  varchar(30) ,

  First_Name

  varchar(30) ) ;

  So, attempting to execute the following statement :

  **INSERT INTO** Customer

  **VALUES** (-2 , ‘Kapoor’ , ‘Raj’);

  will result in an error because the values for SID must be greater than 0.

## PRIMARY KEY CONSTRAINT

- A primary key is used to identify each row in a table. A primary key can consist of one or more fields(column) on a table. When multiple fields are used as a primary key, they are called a **composite key.**
- You can define a primary key in CREATE TABLE command through keywords PRIMARY KEY. e.g.

```
CREATE TABLE Customer
(       SID integer NOT NULL PRIMARY KEY,
Last_Name varchar(30) ,
First_Name   varchar(30)
) ;
```

**Or**

```
CREATE    TABLE    Customer    (
SID integer,
        Last_Name      varchar(30)      ,
        First_Name           varchar(30),
        PRIMARY KEY (SID) ) ;
```

- The second way is useful if you want to specify a composite primary key, **e.g.**

```
CREATE TABLE Customer
(       Branch
integer NOT
NULL, SID
integer NOT
NULL ,
Last_Name
varchar(30) ,
First_Name
varchar(30),
PRIMARY KEY (Branch , SID) ) ;
```

## FOREIGN KEY CONSTRAINT

- Foreign key is a non key column of a table (**child table**) that draws its values from **primary key** of another table(**parent table).**
- The table in which a foreign key is defined is called a **referencing table or child table.** A table to which a foreign key points is called **referenced table or parent table.**

**e.g.**

**TABLE: STUDENT**

**Parent Table**

**Primary key**

| **ROLL_NO** | NAME | CLASS |
|---|---|---|
| 1 | ABC | XI |
| 2 | DEF | XII |
| 3 | XYZ | XI |

**TABLE: SCORE**

**Child Table**

| ROLL_NO | MARKS |
|---|---|
| 1 | 55 |
| 2 | 83 |
| 3 | 90 |

Here column Roll_No is a foreign key in table SCORE(Child Table) and it is drawing its values from Primary key (ROLL_NO) of STUDENT table.(Parent Key).

```
CREATE TABLE STUDENT
(   ROLL_NO integer NOT
    NULL PRIMARY KEY ,
    NAME VARCHAR(30) ,
    CLASS VARCHAR(3) );
```

```
CREATE TABLE SCORE
( ROLL_NO integer ,
MARKS integer ,
FOREIGN KEY(ROLL_NO) REFERNCES STUDENT(ROLL_NO) ) ;
```

**** Foreign key is always defined in the child table.***

**Syntax for using foreign key**

FOREIGN KEY(column name) REFERENCES Parent_Table(PK of Parent Table);

**REFERENCING ACTIONS**

Referencing action with ON DELETE clause determines what to do in case of a DELETE occurs in the parent table. Referencing action with ON UPDATE clause determines what to do in case of a UPDATE occurs in the parent table.

**Actions:**

1. **CASCADE :** This action states that if a DELETE or UPDATE operation affects a row from the parent table, then automatically delete or update the matching rows in the child table i.e., cascade the action to child table.
2. **SET NULL :** This action states that if a DELETE or UPDATE operation affects a row from the parent table, then set the foreign key column in the child table to NULL.
3. **NO ACTION :** Any attempt for DELETE or UPDATE in parent table is not allowed.
4. **RESTRICT :** This action rejects the DELETE or UPDATE operation for the parent table.

**Q: Create two tables**

**Customer(<u>customer_id</u>, name)**

**Customer_sales(<u>transaction_id</u>, amount , customer_id)**

Underlined columns indicate primary keys and bold column names indicate foreign key.

Make sure that no action should take place in case of a DELETE or UPDATE in the parent table.

**Sol :** CREATE TABLE Customer (
customer_id int Not Null
Primary Key , name
varchar(30) ) ;

CREATE TABLE Customer_sales ( transaction_id Not Null
Primary Key , amount int ,
customer_id int ,
FOREIGN KEY(customer_id) REFERENCES Customer
(customer_id) ON DELETE NO ACTION
ON UPDATE NO ACTION );

**Q: Distinguish between a Primary Key and a Unique key in a table.**

| S.NO. | PRIMARY KEY | UNIQUE KEY |
|---|---|---|
| 1. | Column having Primary key can't contain NULL value | Column having Unique Key can contain NULL value |
| 2. | There can be only one primary key in Table. | Many columns can be defined as Unique key |

**Q: Distinguish between ALTER Command and UPDATE command of SQL.**

| S.NO. | ALTER COMMAND | UPDATE COMMAND |
|---|---|---|
| 1. | It is a DDL Command | It is a DML command |
| 2. | It is used to change the definition of existing table, i.e. adding column, deleting column, etc. | It is used to modify the data values present in the rows of the table. |
| 3. | Syntax for adding column in a table: ALTER TABLE <tablename> ADD <Column name><Datatype> ; | Syntax for using UPDATE command: UPDATE <Tablename> SET <Columnname>=value WHERE <Condition> ; |

## AGGREGATE / GROUP FUNCTIONS

Aggregate / Group functions work upon groups of rows , rather than on single row, and return one single output. Different aggregate functions are : COUNT( ) , AVG( ) , MIN( ) , MAX( ) , SUM ( )

**Table : EMPL**

| **EMPNO** | **ENAME** | **JOB** | **SAL** | **DEPTNO** |
|---|---|---|---|---|
| 8369 | SMITH | CLERK | 2985 | 10 |
| 8499 | ANYA | SALESMAN | 9870 | 20 |
| 8566 | AMIR | SALESMAN | 8760 | 30 |
| 8698 | BINA | MANAGER | 5643 | 20 |
| 8912 | SUR | NULL | 3000 | 10 |

1. **AVG( )**

   This function computes the average of given data. e.g. SELECT AVG(SAL)

   FROM EMPL ;

   **Output**

| AVG(SAL) |
|---|
| 6051.6 |

2. **COUNT( )**

   This function counts the number of rows in a given column.

   If you specify the COLUMN name in parenthesis of function, then this function returns rows where COLUMN is not null.

   If you specify the asterisk (*), this function returns all rows, including duplicates and nulls.

   e.g. SELECT

   COUNT(*)

   FROM

   EMPL ;

   **Output**

| COUNT(*) |
|---|
| 5 |

   e.g.2 SELECT COUNT(JOB) FROM EMPL ;

   **Output**

| COUNT(JOB) |
|---|
| 4 |

3. **MAX( )**

This function returns the maximum value from a given column or expression.

e.g.
```
SELECT
MAX(SAL)
FROM  EMPL
;
```

**Output**

| MAX(SAL) |
|---|
| 9870 |

4. **MIN( )**

This function returns the minimum value from a given column or expression.

e.g.
```
SELECT
MIN(SAL)
FROM  EMPL
;
```

**Output**

| MIN(SAL) |
|---|
| 2985 |

5. **SUM( )**

This function returns the sum of values in given column or expression.

e.g.
```
SELECT
SUM(SAL)
FROM   EMPL
;
```

**Output**

| SUM(SAL) |
|---|
| 30258 |

## GROUPING RESULT – GROUP BY

The GROUP BY clause combines all those records(row) that have identical values in a particular field(column) or a group of fields(columns).

GROUPING can be done by a column name, or with aggregate functions in which case the aggregate produces a value for each group.

**Table : EMPL**

| **EMPNO** | **ENAME** | **JOB** | **SAL** | **DEPTNO** |
|---|---|---|---|---|
| 8369 | SMITH | CLERK | 2985 | 10 |
| 8499 | ANYA | SALESMAN | 9870 | 20 |
| 8566 | AMIR | SALESMAN | 8760 | 30 |
| 8698 | BINA | MANAGER | 5643 | 20 |

**e.g. Calculate the number of employees in each grade.**

```
SELECT      JOB,
COUNT(*)
FROM EMPL
GROUP BY JOB ;
```

**Output**

| **JOB** | **COUNT(*)** |
|---|---|
| CLERK | 1 |
| SALESMAN | 2 |
| MANAGER | 1 |

**e.g.2. Calculate the sum of salary for each department.**

```
SELECT   DEPTNO   ,
SUM(SAL)       FROM
EMPL
GROUP BY DEPTNO ;
```

**Output**

| **DEPTNO** | **SUM(SAL)** |
|---|---|
| 10 | 2985 |
| 20 | 15513 |
| 30 | 8760 |

## NESTED GROUP

- To create a group within a group i.e., nested group, you need to specify multiple fields in the GROUP BY expression. e.g. To group records **job wise** within **Deptno wise**, you need to issue a query statement like :

```
SELECT DEPTNO , JOB ,
COUNT(EMPNO) FROM EMPL
GROUP BY DEPTNO , JOB ;
```

**Output**

| DEPTNO | JOB | COUNT(EMPNO) |
|---|---|---|
| 10 | CLERK | 1 |
| 20 | SALESMAN | 1 |
| 20 | MANAGER | 1 |
| 30 | SALESMAN | 1 |

## PLACING CONDITION ON GROUPS – HAVING CLAUSE

- The **HAVING clause places conditions on groups** in contrast to WHERE clause that places condition on individual rows. While **WHERE conditions cannot include aggregate functions, HAVING conditions can do so.**
- e.g. To display the jobs where the number of employees is

```
less than 2, SELECT JOB, COUNT(*)
FROM EMPL
GROUP BY JOB
HAVING COUNT(*) < 2 ;
```

**Output**

| **JOB** | **COUNT(*)** |
|---|---|
| CLERK | 1 |
| MANAGER | 1 |

## 6. Open source and Commercial DBMS - MYSQL, ORACLE, DB2, SQL server.

**Open Source Database:**

An open-source database is a database where anyone can easily view the source code and this is open and free to download. Also for the community version, some small additional and affordable costs are imposed. Open Source Database provides Limited technical support to end-users. Here Installation and updates are administered by the user. For example: MYSQL, PostgreSQL, MongoDB etc.

**Advantages of Open Source Databases:**

- Cost: Open source databases are generally free, which means they can be used without any licensing fees.
- Customization: Since the source code is available, developers can modify and customize the database to meet specific requirements.
- Community Support: Open source databases have a large community of users who contribute to documentation, bug fixes, and improvements.
- Security: With open source databases, security vulnerabilities can be detected and fixed quickly by the community.
- Scalability: Open source databases are typically designed to be scalable, which means they can handle large amounts of data and traffic.

**Disadvantages of Open Source Databases:**

- Limited Technical Support: While there is a large community of users who can help troubleshoot issues, there is no guarantee of professional technical support.
- Complexity: Open source databases can be more difficult to set up and configure than commercial databases, especially for users who are not experienced in database administration.
- Lack of Features: Open source databases may not have all the features that are available in commercial databases, such as advanced analytics and reporting tools.

2. Commercial Database :

Commercial database are that which has been created for Commercial Purpose only. They are premium and are not free like Open Source Database. In Commercial Database it is **guaranteed** that technical support is provided. In this Installation and updates are Administrated by software Vendor. For examples: Oracle, IBM DB2 etc.

Advantages of Commercial Databases:

- Technical Support: Commercial databases usually come with professional technical support, which can be helpful for organizations that need assistance with setup, configuration, or troubleshooting.
- Features: Commercial databases typically have more features than open source databases, including advanced analytics, reporting, and data visualization tools.
- Security: Commercial databases often have built-in security features and can provide better protection against cyber threats.
- Integration: Commercial databases are often designed to work seamlessly with other enterprise software, making integration with existing systems easier.

Disadvantages of Commercial Databases:

- Cost: Commercial databases can be expensive, with licensing fees and maintenance costs that can add up over time.
- Vendor Lock-In: Organizations that use commercial databases may become dependent on the vendor and find it difficult to switch to another database.
- Limited Customization: Commercial databases may not be as customizable as open source databases, which can be a disadvantage for organizations with specific requirements.

Similarities between Open Source Database and Commercial Database:

- Both can handle large amounts of data and support complex data structures.
- Both can be used to store and retrieve data in a structured manner.
- Both can be used to support mission-critical applications and services.
- Both use SQL (Structured Query Language) to perform queries and manipulate data.
- Both can be accessed and managed remotely using a variety of tools and interfaces.
- Both can be optimized for performance, scalability, and security.

**Difference between Open Source Database and Commercial Database:**

| **Sr. No.** | **Basis of Comparison** | **Open Source Database** | **Commercial Database** |
|---|---|---|---|
| 1. | Focus | In Open Source Database anyone | Commercial Database are that |

| Sr. No. | Basis of Comparison | Open Source Database | Commercial Database |
|---|---|---|---|
| | | can easily view the Source code of it. | which has been created for Commercial purpose only. |
| 2. | Examples | Examples: MYSQL, PostgreSQL, MongoDB etc. | Examples: Oracle, DB2, Splunk etc. |
| 3. | Cost | They are free or have additional and affordable cost. | They are premium and are not free like open source database. |
| 4. | Community | The community can see, share, and modify the code of open-source DBMS software. | The community cannot see, exchange, or modify the code of commercial DBMS software. |
| 5. | Source Code | Because the source code is open, there is a risk of coding malfunction. | The code is not accessible to unauthorized users and has a high level of protection. |
| 6. | Technical support | It provide limited technical support. | It provide guaranteed technical support. |
| 7. | License | In this software is available under free licensing. | In this Software is available under high licensing cost. |
| 8. | Support | In this User’s needs to rely on Community Support. | In this user’s get dedicated support from Vendor’s from where one’s buy. |

| **Sr. No.** | **Basis of Comparison** | **Open Source Database** | **Commercial Database** |
|---|---|---|---|
| 9. | Installation and Updates | In this Installation and Updates are administrated by user. | In this Installation and updates are administrated by Software Vendor. |
| 10. | Benefits | • Bug fixes are simple to implement without having to go through the approval process at corporate.<br>• For any premium solution, a free open source alternative with the same or more functionality is always accessible.<br>• Because it is more visible by nature, it can be inspected for security concerns, which is a big benefit. | • There is a single point of contact for any issues that arise. That implies, you pay for specific needs, and there is a party to blame if difficulties develop.<br>• The licensing is usually obvious, and it comes with a guarantee.<br>• Developers typically have a thorough plan in place for the programme and release updates as needed. This allows businesses to save money on the costs of technical outages and failures. |
| 11. | Drawbacks | • Volunteer Technical support<br>• Because of Compatibility difficulties, it cannot be assured that it will work in each user's environment | • Strict Licensing guidelines<br>• Source code cannot be modified so extra cost is incurred for getting more functionality of premium features |

| Sr. No. | Basis of Comparison | Open Source Database | Commercial Database |
|---|---|---|---|
| | | due to compatibility difficulties.<br>• Professionals are required to manage and even install the essential infrastructure because some of these difficulties vary from software to hardware.<br>• Open source also poses a significant security risk because some of it can easily contain security exploits. | |

| Property | SQL Server | MySQL | ORACLE | DB2 |
|---|---|---|---|---|
| Purpose | A Relational database of Microsoft | A Widely used open source database | Hugely used RDBMS | Common in IBM host environments, 2 different versions for host and Windows/Linux |
| The Basic model of Database | RDBMS | RDBMS | RDBMS | RDBMS |
| Secondary model of Database | Graph DBMS Key-value store Document Store | Key-value store Document store | Graph DBMS Key-value store Document Store RDF store | Document Store RDF Store Spatial Dbms |

| | | | | |
|---|---|---|---|---|
| Website | www.microsoft.com/en-us/sql-server | www.mysql.com | www.oracle.com/database/index.html | www.ibm.com/-analytics/db2 |
| Developer | Microsoft | Oracle | Oracle | IBM |
| Initial Release Year | 1989 | 1995 | 1980 | 1983 |
| License | Commercial | Open Source | Commercial | Commercial |
| Cloud Support | No | No | No | No |
| Implementation Language | C++ | C and C++ | C and C++ | C and C++ |
| Data Scheme | Yes | Yes | Yes | Yes |

## 7. PL/SQL:

PL/SQL is a block structured language that enables developers to combine the power of SQL with procedural statements.All the statements of a block are passed to oracle engine all at once which increases processing speed and decreases the traffic.

**Basics of PL/SQL**

- PL/SQL stands for Procedural Language extensions to the Structured Query Language (SQL).
- PL/SQL is a combination of SQL along with the procedural features of programming languages.
- Oracle uses a PL/SQL engine to processes the PL/SQL statements.

- PL/SQL includes procedural language elements like conditions and loops. It allows declaration of constants and variables, procedures and functions, types and variable of those types and triggers.

**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:**

1. PL/SQL is basically a procedural language, which provides the functionality of decision making, iteration and many more features of procedural programming languages.
2. PL/SQL can execute a number of queries in one block using single command.
3. 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.
4. PL/SQL provides a feature to handle the exception which occurs in PL/SQL block known as exception handling block.
5. Applications written in PL/SQL are portable to computer hardware or operating system where Oracle is operational.
6. PL/SQL Offers extensive error checking.

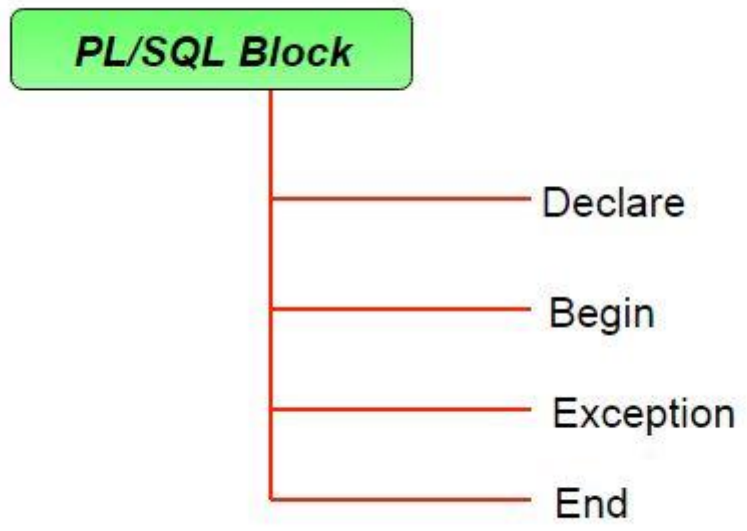
**Differences between SQL and PL/SQL:**

| 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 | PL/SQL is procedural that defines how the things needs to be done. |

| | |
|---|---|
| things need 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.

Typically, each block performs a logical action in the program. A block has the following structure:

**DECLARE**

  declaration statements;

**BEGIN**
    executable statements
**EXCEPTIONS**
    exception handling statements
**END;**

- 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.

1. 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:
2. variable_name datatype [NOT NULL := value ];

**Example to show how 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.**

1. Explanation:
   - SET SERVEROUTPUT ON: It is used to display the buffer used by the dbms_output.
   - var1 INTEGER : It 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.: It is displayed when the code is compiled and executed successfully.
   - Slash (/) after END;: The slash (/) tells the SQL*Plus to execute the block.
   - Assignment operator (:=) : It is used to assign a value to a variable.
2. 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) := 'Hello World' ;
   BEGIN
     dbms_output.put_line(var);
   END;
  /
```

**Output:**

**Hello World**

PL/SQL procedure successfully completed.

1. Explanation:
    - dbms_output.put_line : This command is used to direct the PL/SQL output to a screen.
2. 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.
3. 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 := &amp;a;
             -- taking input for variable b
      b varchar2(30) := &amp;b;
   BEGIN
      null;
   END;
  /
```

**Output:**

```
Enter value for a: 24
old   2: a number := &a;
new   2: a number := 24;
```

```
Enter value for b: 'Hello World'
old   3: b varchar2(30) := &b;
new   3: b varchar2(30) := 'Hello World'
```

**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;
  /
```

```
Enter value for a: 2
Enter value for b: 3
Sum of 2 and 3 is = 5
PL/SQL procedure successfully completed
```

**PL/SQL Execution Environment:**

The PL/SQL engine resides in the Oracle engine.The Oracle engine can process not only single SQL statement but also block of many statements.The call to Oracle engine needs to be made only once to execute any number of SQL statements if these SQL statements are bundled inside a PL/SQL block.

## 7.1.concept of Stored Procedures & Functions

A stored procedure in PL/SQL is nothing but a series of declarative SQL statements which can be stored in the database catalogue. A procedure can be thought of as a function or a method. They can be invoked through triggers, other procedures, or applications on Java, PHP etc.
All the statements of a block are passed to Oracle engine all at once which increases processing speed and decreases the traffic.

**Advantages:**

- They result in performance improvement of the application. If a procedure is being called frequently in an application in a single connection, then the compiled version of the procedure is delivered.
- They reduce the traffic between the database and the application, since the lengthy statements are already fed into the database and need not be sent again and again via the application.
- They add to code reusability, similar to how functions and methods work in other languages such as C/C++ and Java.

**Disadvantages:**

- Stored procedures can cause a lot of memory usage. The database administrator should decide an upper bound as to how many stored procedures are feasible for a particular application.
- MySQL does not provide the functionality of debugging the stored procedures.

**The procedure contains a header and a body.**

- **Header:** The header contains the name of the procedure and the parameters or variables passed to the procedure.
- **Body:** The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

**How to pass parameters in procedure:**

When you want to create a procedure or function, you have to define parameters .There is three ways to pass parameters in procedure:

1. **IN parameters**: The IN parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or the function.
2. **OUT parameters**: The OUT parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **INOUT parameters**: The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

**PL/SQL Create Procedure**

**Syntax for creating procedure:**

```
CREATE [OR REPLACE] PROCEDURE procedure_name
    [ (parameter [,parameter]) ]
IS
    [declaration_section]
BEGIN
    executable_section
[EXCEPTION
    exception_section]
END [procedure_name];
```

**Create procedure example**

In this example, we are going to insert record in user table. So you need to create user table first.

**Table creation:**

```
create table user(id number(10) primary key,name varchar2(100));
```

Now write the procedure code to insert record in user table.

**Procedure Code:**

```
create or replace procedure "INSERTUSER"
(id IN NUMBER,
name IN VARCHAR2)
is
begin
insert into user values(id,name);
end;
/
```

Output:

```
Procedure created.
```

PL/SQL program to call procedure

Let's see the code to call above created procedure.

```
BEGIN
   insertuser(101,'Rahul');
   dbms_output.put_line('record inserted successfully');
END;
/
```

Now, see the "USER" table, you will see one record is inserted.

| ID | Name |
|---|---|
| 101 | Rahul |

**PL/SQL Drop Procedure**

Syntax for drop procedure

**DROP PROCEDURE procedure name;**

Example of drop procedure

**DROP PROCEDURE pro1;**

The PL/SQL Function is very similar to PL/SQL Procedure. The main difference between procedure and a function is, a function must always return a value, and on the other hand a procedure may or may not return a value. Except this, all the other things of PL/SQL procedure are true for PL/SQL function too.

**Syntax to create a function:**

1. **CREATE** [OR REPLACE] **FUNCTION** function_name [parameters]
2. [(parameter_name [IN | **OUT** | IN **OUT**] type [, ...])]
3. **RETURN** return_datatype
4. {**IS** | **AS**}
5. **BEGIN**
6.   < function_body >
7. **END** [function_name];

Here:

- Function_name: specifies the name of the function.
- [OR REPLACE] option allows modifying an existing function.
- The optional parameter list contains name, mode and types of the parameters.
- IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.

The function must contain a r**eturn** statement.

- **RETURN** clause specifies that data type you are going to return from the function.
- Function_body contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

**PL/SQL Function Example**

Let's see a simple example to create a function.

```
create or replace function adder(n1 in number, n2 in number)
return number
is
n3 number(8);
begin
n3 :=n1+n2;
return n3;
end;
/
```

Now write another program to **call the function**.

```
DECLARE
   n3 number(2);
BEGIN
   n3 := adder(11,22);
   dbms_output.put_line('Addition is: ' || n3);
END;
/
```

**Output:**

```
Addition is: 33
Statement processed.
0.05 seconds
```

## Another PL/SQL Function Example

Let's take an example to demonstrate Declaring, Defining and Invoking a simple PL/SQL function which will compute and return the maximum of two values.

```
DECLARE
   a number;
   b number;
   c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
    z number;
BEGIN
   IF x > y THEN
      z:= x;
   ELSE
      Z:= y;
   END IF;

   RETURN z;
END;
BEGIN
   a:= 23;
   b:= 45;

   c := findMax(a, b);
   dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/
```

**Output:**

```
Maximum of (23,45): 45
Statement processed.
0.02 seconds
```

## PL/SQL function example using table

Let's take a customer table. This example illustrates creating and calling a standalone function. This function will return the total number of CUSTOMERS in the customers table.

| Customers | | | |
|---|---|---|---|
| **Id** | **Name** | **Department** | **Salary** |
| 1 | alex | web developer | 35000 |
| 2 | ricky | program developer | 45000 |
| 3 | mohan | web designer | 35000 |
| 4 | dilshad | database manager | 44000 |

**Create Function:**

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
   total number(2) := 0;
BEGIN
   SELECT count(*) into total
   FROM customers;
    RETURN total;
END;
/
```

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

```
Function created.
```

**Calling PL/SQL Function:**

While creating a function, you have to give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. Once the function is called, the program control is transferred to the called function.

After the successful completion of the defined task, the call function returns program control back to the main program.

To call a function you have to pass the required parameters along with function name and if function returns a value then you can store returned value. Following program calls the function totalCustomers from an anonymous block:

```
DECLARE
  c number(2);
BEGIN
  c := totalCustomers();
  dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

After the execution of above code in SQL prompt, you will get the following result.

```
Total no. of Customers: 4
PL/SQL procedure successfully completed.
```

## PL/SQL Recursive Function

You already know that a program or a subprogram can call another subprogram. When a subprogram calls itself, it is called recursive call and the process is known as recursion.

### Example to calculate the factorial of a number

Let's take an example to calculate the factorial of a number. This example calculates the factorial of a given number by calling itself recursively.

```
DECLARE
   num number;
   factorial number;

FUNCTION fact(x number)
RETURN number
IS
   f number;
BEGIN
   IF x=0 THEN
      f := 1;
   ELSE
      f := x * fact(x-1);
   END IF;
RETURN f;
END;

BEGIN
   num:= 6;
   factorial := fact(num);
   dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
END;
/
```

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

```
Factorial 6 is 720
PL/SQL procedure successfully completed.
```

PL/SQL Drop Function

**Syntax for removing your created function:**

If you want to remove your created function from the database, you should use the following syntax.

1. **DROP FUNCTION** function_name;

## 7.2. Cursors

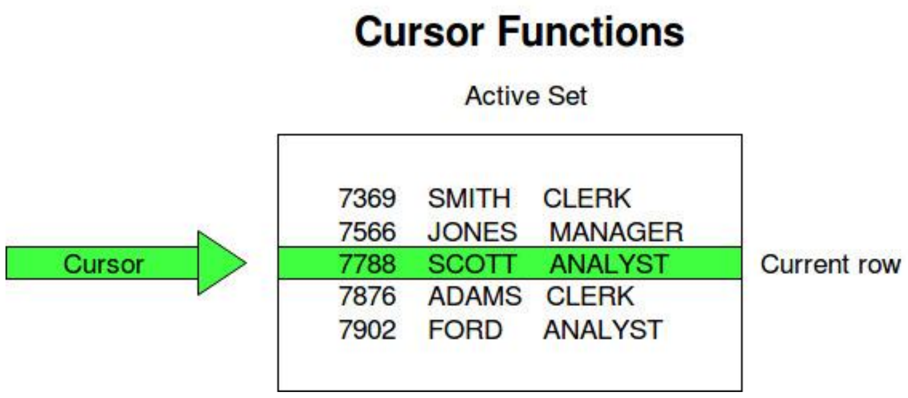
Cursor in SQL

To execute SQL statements, a work area is used by the Oracle engine for its internal processing and storing the information. This work area is private to SQL's operations. The 'Cursor' is the PL/SQL construct that allows the user to name the work area and access the stored information in it.

**Use of Cursor**

The major function of a cursor is to retrieve data, one row at a time, from a result set, unlike the SQL commands which operate on all the rows in the result set at one time. Cursors are used when the user needs to update records in a singleton fashion or in a row by row manner, in a database table.

The Data that is stored in the Cursor is called the Active Data Set. Oracle DBMS has another predefined area in the main memory Set, within which the cursors are opened. Hence the size of the cursor is limited by the size of this pre-defined area.



**Cursor Actions**

**Declare Cursor**: A cursor is declared by defining the SQL statement that returns a result set.
**Open:** A Cursor is opened and populated by executing the SQL statement defined by the cursor.
**Fetch:** When the cursor is opened, rows can be fetched from the cursor one by one or in a block to perform data manipulation.
**Close:** After data manipulation, close the cursor explicitly.
**Deallocate:** Finally, delete the cursor definition and release all the system resources associated with the cursor.

**Types of Cursors**

Cursors are classified depending on the circumstances in which they are opened.

- **Implicit Cursor:** If the Oracle engine opened a cursor for its internal processing it is known as an Implicit Cursor. It is created "automatically" for the user by Oracle when a query is executed and is simpler to code.
- **Explicit Cursor:** A Cursor can also be opened for processing data through a PL/SQL block, on demand. Such a user-defined cursor is known as an Explicit Cursor.

**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.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT**. The SQL cursor has additional attributes, **%BULK_ROWCOUNT** and **%BULK_EXCEPTIONS**, designed for use with the **FORALL** statement. The following table provides the description of the most used attributes −

| S.No | Attribute & Description |
|---|---|
| 1 | **%FOUND**<br>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**<br>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**<br>Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement. |
| 4 | **%ROWCOUNT**<br>Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement. |

**Example**

We will be using the CUSTOMERS table we had created and used in the previous chapters.

**Select * from customers;**

| ID | NAME | AGE | ADDRESS | SALARY |
|---|---|---|---|---|
| 1 | Ramesh | 32 | Ahmedabad | 2000 |
| 2 | Khilan | 25 | Delhi | 1500 |
| 3 | Kaushik | 23 | Kota | 2000 |
| 4 | Chailtali | 25 | Mumbai | 6500 |

|  |  |  |  |  |
|---|---|---|---|---|
| 5 | Hardik | 27 | Bhopal | 8500 |
| 6 | Komal | 22 | MP | 4500 |

The following program will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected −

```
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.

**Explicit cursor**

An explicit cursor is defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row. A suitable name for the cursor.

General syntax for creating a cursor:

**CURSOR cursor_name IS select_statement;**

**cursor_name –** A suitable name for the cursor.

**select_statement –** A select query which returns multiple rows

**How to use Explicit Cursor?**

There are four steps in using an Explicit Cursor.

1. **DECLARE** the cursor in the Declaration section.
2. **OPEN** the cursor in the Execution Section.
3. **FETCH** the data from the cursor into PL/SQL variables or records in the Execution Section.
4. **CLOSE** the cursor in the Execution Section before you end the PL/SQL Block.

**Syntax:**

```
DECLARE variables;
 records;
 create a cursor;
 BEGIN
OPEN cursor;
FETCH cursor;
 process the records;
 CLOSE cursor;
 END;
```

Following is a complete example to illustrate the concepts of explicit cursors &minua;

```
DECLARE
   c_id customers.id%type;
   c_name customers.name%type;
   c_addr customers.address%type;
```

```
  CURSOR c_customers is
    SELECT id, name, address FROM customers;
BEGIN
  OPEN c_customers;
  LOOP
  FETCH c_customers into c_id, c_name, c_addr;
    EXIT WHEN c_customers%notfound;
    dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
  END LOOP;
  CLOSE c_customers;
END;
/
```

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

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP

PL/SQL procedure successfully completed.
```

## 7.3.Triggers

**Trigger** is a statement that a system executes automatically when there is any modification to the database. In a trigger, we first specify when the trigger is to be executed and then the action to be performed when the trigger executes. Triggers are used to specify certain integrity constraints and referential constraints that cannot be specified using the constraint mechanism of SQL.

**Example –**

Suppose, we are adding a tuple to the 'Donors' table that is some person has donated blood. So, we can design a trigger that will automatically add the value of donated blood to the 'Blood_record' table.

**Types of Triggers –**

We can define 6 types of triggers for each table:

1. **AFTER INSERT** activated after data is inserted into the table.

2. **AFTER UPDATE:** activated after data in the table is modified.

3. **AFTER DELETE:** activated after data is deleted/removed from the table.

4. **BEFORE INSERT:** activated before data is inserted into the table.

5. **BEFORE UPDATE:** activated before data in the table is modified.

6. **BEFORE DELETE:** activated before data is deleted/removed from the table.

Examples showing implementation of Triggers:

**1. Write a trigger to ensure that no employee of age less than 25 can be inserted in the database.**

```
delimiter $$
```

```
CREATE TRIGGER  Check_age  BEFORE INSERT ON employee
FOR EACH ROW
BEGIN
IF NEW.age < 25 THEN
SIGNAL SQLSTATE '45000'
SET MESSAGE_TEXT = 'ERROR:
        AGE MUST BE ATLEAST 25 YEARS!';
END IF;
END; $$
delimiter;
```

**Explanation:** Whenever we want to insert any tuple to table ’employee’, then before inserting this tuple to the table, trigger named ‘Check_age’ will be executed. This trigger will check the age attribute. If it is greater than 25 then this tuple will be inserted into the tuple otherwise an error message will be printed stating “ERROR: AGE MUST BE ATLEAST 25 YEARS!”

**2. Create a trigger which will work before deletion in employee table and create a duplicate copy of the record in another table employee_backup.**

Before writing trigger, we need to create table employee_backup.

```
create table employee_backup (employee_no int,
      employee_name varchar(40), job varchar(40),
      hiredate date, salary int,
      primary key(employee_no));
delimiter $$
CREATE TRIGGER Backup BEFORE DELETE ON employee
FOR EACH ROW
BEGIN
INSERT INTO employee_backup
VALUES (OLD.employee_no, OLD.name,
        OLD.job, OLD.hiredate, OLD.salary);
END; $$
delimiter;
```

**Explanation:** We want to create a backup table that holds the value of those employees who are no more the employee of the institution. So, we create a trigger named Backup that will be executed before the deletion of any Tuple from the table employee. Before deletion, the values of all the attributes of the table employee will be stored in the table employee_backup.

**3. Write a trigger to count number of new tuples inserted using each insert statement.**

```
Declare count int
Set count=0;
delimiter $$
CREATE TRIGGER Count_tupples
        AFTER INSERT ON employee
FOR EACH ROW
BEGIN
SET count = count + 1;
END; $$
delimiter;
```

**Explanation:** We want to keep track of the number of new Tuples in the employee table. For that, we first create a variable ‘count’ and initialize it to 0. After that, we create a trigger named Count_tuples that will increment the value of count after insertion of any new Tuple in the table employee.

## 7.4. Assertions

**What are Assertions?**

When a constraint involves 2 (or) more tables, the table constraint mechanism is sometimes hard and results may not come as expected. To cover such situation SQL supports the creation of assertions that are constraints not associated with only one table. And an assertion statement should ensure a certain condition will always exist in the database. DBMS always checks the assertion whenever modifications are done in the corresponding table.

**Syntax –**

```
CREATE ASSERTION  [ assertion_name ]
CHECK ( [ condition ] );
```

**Example –**

```
CREATE TABLE sailors (sid int,sname varchar(20), rating int,primary key(sid),
CHECK(rating >= 1 AND rating <=10)
CHECK((select count(s.sid) from sailors s) + (select count(b.bid)from boats b)<100) );
```

In the above example, we enforcing CHECK constraint that the number of boats and sailors should be less than 100. So here we are able to CHECK constraints of two tablets simultaneously.

**Difference between Assertions and Triggers :**

| **S.No** | **Assertions** | **Triggers** |
|---|---|---|
| 1. | We can use Assertions when we know that the given particular condition is always true. | We can use Triggers even particular condition may or may not be true. |
| 2. | When the SQL condition is not met then there are chances to an entire table or even Database to get locked up. | Triggers can catch errors if the condition of the query is not true. |
| 3. | Assertions are not linked to specific table or event. It performs task | It helps in maintaining the integrity constraints in the database tables, especially when the |

| S.No | Assertions | Triggers |
|---|---|---|
| | specified or defined by the user. | primary key and foreign key constraint are not defined. |
| 4. | Assertions do not maintain any track of changes made in table. | Triggers maintain track of all changes occurred in table. |
| 5. | Assertions have small syntax compared to Triggers. | They have large Syntax to indicate each and every specific of the created trigger. |
| 6. | Modern databases do not use Assertions. | Triggers are very well used in modern databases. |
| 7. | Purpose of assertions is to Enforces business rules and constraints. | Purpose of triggers is to Executes actions in response to data changes. |
| 8. | Activation is checked after a transaction completes | Activation is activated by data changes during a transaction |
| 9. | Granularity applies to the entire database | Granularity applies to a specific table or view |
| 10. | Syntax Uses SQL statements | Syntax Uses procedural code (e.g. PL/SQL, T-SQL) |
| 11. | Error handling Causes transaction to be rolled back. | Error handling can ignore errors or handle them explicitly |
| 12. | Assertions may slow down | Triggers Can impact performance of data |

| S.No | Assertions | Triggers |
|---|---|---|
| | performance of queries. | changes. |
| 13. | Assertions are Easy to debug with SQL statements. | Triggers are more difficult to debug procedural code |
| 14. | Examples- CHECK constraints, FOREIGN KEY constraints | Examples – AFTER INSERT triggers, INSTEAD OF triggers |

Assertions can't modify the data and they are not linked to any specific tables or events in the database but Triggers are more powerful because they can check conditions and also modify the data within the tables inside a database, unlike assertions.

## 7.5.roles

A role is created to ease setup and maintenance of the security model. It is a named group of related privileges that can be granted to the user. When there are many users in a database it becomes difficult to grant or revoke privileges to users. Therefore, if you define roles:

- You can grant or revoke privileges to users, thereby automatically granting or revoking privileges.
- You can either create Roles or use the system roles pre-defined.

Some of the privileges granted to the system roles are as given below:

| System Roles | Privileges granted to the Role |
|---|---|
| Connect | Create table, Create view, Create synonym, Create sequence, Create session etc. |
| Resource | Create Procedure, Create Sequence, Create Table, Create Trigger etc. The |

| | |
|---|---|
| | primary usage of the Resource role is to restrict access to database objects. |
| DBA | All system privileges |

**Creating and Assigning a Role –**

First, the (Database Administrator)DBA must create the role. Then the DBA can assign privileges to the role and users to the role.

**Syntax –**

```
CREATE ROLE manager;
Role created.
```

In the syntax:
'manager' is the name of the role to be created.

- Now that the role is created, the DBA can use the GRANT statement to assign users to the role as well as assign privileges to the role.
- It's easier to GRANT or REVOKE privileges to the users through a role rather than assigning a privilege directly to every user.
- If a role is identified by a password, then GRANT or REVOKE privileges have to be identified by the password.

**Grant privileges to a role –**

```
GRANT create table, create view
TO manager;
Grant succeeded.
```

**Grant a role to users**

```
GRANT manager TO SAM, STARK;
Grant succeeded.
```

**Revoke privilege from a Role :**

```
REVOKE create table FROM manager;
```

**Drop a Role :**

DROP ROLE manager;

**Explanation** –

Firstly it creates a manager role and then allows managers to create tables and views. It then grants Sam and Stark the role of managers. Now Sam and Stark can create tables and views. If users have multiple roles granted to them, they receive all of the privileges associated with all of the roles. Then create table privilege is removed from role 'manager' using Revoke.The role is dropped from the database using drop.

## 8. Embedded SQL

**Static or Embedded** SQL are SQL statements in an application that do not change at runtime and, therefore, can be hard-coded into the application.

**Limitations of Static SQL:** They do not change at runtime thus are hard-coded into applications.

9. Dynamic SQL.

**Dynamic** SQL is SQL statements that are constructed at runtime; for example, the application may allow users to enter their own queries. **Dynamic** SQL is a programming technique that enables you to build SQL statements dynamically at runtime. You can create more general purpose, flexible applications by using dynamic SQL because the full text of a SQL statement may be unknown at compilation.

**Limitation of Dynamic SQL:** We cannot use some of the SQL statements dynamically. Performance of these statements is poor as compared to Static SQL.

Following are some of the important differences between Static Routing and Dynamic Routing.

| Sr. No. | Key | Static SQL | Dynamic SQL |
|---|---|---|---|
| 1 | Database Access | In Static SQL, database access procedure is predetermined in the statement. | In Dynamic SQL, how a database will be accessed, can be determine only at run time. |
| 2 | Efficiency | Static SQL statements are more faster and efficient. | Dynamic SQL statements are less efficient. |
| 3 | Compilation | Static SQL statements are compiled at compile time. | Dynamic SQL statements are compiled at run time. |

| Sr. No. | Key | Static SQL | Dynamic SQL |
|---|---|---|---|
| 4 | Application Plan | Application Plan parsing, validation, optimization and generation are compile time activities. | Application Plan parsing, validation, optimization and generation are run time activities. |
| 5 | Use Cases | Static SQL is used in case of uniformly distributed data. | Dynamic SQL is used in case of non-uniformly distributed data. |
| 6 | Dynamic Statements | Statements like EXECUTE IMMEDIATE, EXECUTE, PREPARE are not used. | Statements like EXECUTE IMMEDIATE, EXECUTE, PREPARE are used |
| 7 | Flexibility | Static SQL is less flexible. | Dynamic SQL is highly flexible. |