Module 3

Introduction to SQL

History of SQL:

- IBM **SEQUEL**(**S**tructured **E**nglish **QUE**ry **L**anguage) language developed as part of System R project at the IBM San Jose Research Laboratory.
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL and it is the standard language for commercial relational DBMS. The standardization of SQL is a joint effort by the American National Standards Institute (ANSI) and the International Standards Organization (ISO), Later,
  - SQL-86
  - SQL-89
  - SQL-92
  - SQL:1999 (language name became Y2K compliant!)
  - SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.

SQL Parts:

- DML -- provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
- integrity – the DDL includes commands for specifying integrity constraints.
- View definition -- The DDL includes commands for defining views.
- Transaction control –includes commands for specifying the beginning and ending of transactions.
- Embedded SQL and dynamic SQL -- define how SQL statements can be embedded within general-purpose programming languages.
- Authorization – includes commands for specifying access rights to relations and views.

Data Definition Language:

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The type of values associated with each attribute.
- The Integrity constraints
- The set of indices to be maintained for each relation.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Domain Types in SQL:

- **Character- string : a. char(n) or character(n).** Fixed length character string, with user-specified length *n.*

  b. varchar(n). Variable length character strings, with user-specified maximum length *n.*

  c. Another variable-length string data type called **CHARACTER LARGE OBJECT or CLOB** is also available to specify columns that have large text values, such as documents. The CLOB maximum length can be specified in kilobytes (K), megabytes (M), or gigabytes (G). For example, CLOB(20M) specifies a maximum length of 20 megabytes.
- **Numeric:** datatype includes integer numbers of various sizes.

  a. int. Integer (a finite subset of the integers that is machine-dependent).

  b. smallint. Small integer (a machine-dependent subset of the integer domain type).

  c. DECIMAL(p,d), dec(p,d) numeric(p,d). Fixed point number, with user-specified precision of *p* digits, with *d* digits to the right of decimal point. (ex., **numeric**(3,1), allows 44.5 to be stores exactly, but not 4444.5 or 0.32)

  d. real, double precision. Floating point and double-precision floating point numbers, with machine-dependent precision.
- **Bit-string** data types are either of fixed length n—BIT(n)—or varying length— BIT VARYING(n), where n is the maximum number of bits. The default for n, the length of a character string or bit string, is 1. Literal bit strings are placed between single quotes but preceded by a B to distinguish them from character strings; for example, B‘10101’.

- Another variable-length bitstring data type called **BINARY LARGE OBJECT or BLOB** is also available to specify columns that have large binary values, such as images.
- A **Boolean** data type has the traditional values of TRUE or FALSE.
- The **DATE** data type has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD.
- **TIME** data type has at least eight positions, with the components HOUR, MINUTE, and SECOND in the form HH:MM:SS
- A **timestamp** data type (TIMESTAMP) includes the DATE and TIME fields.
- The **INTERVAL** data type specifies an interval—a relative value that can be used to increment or decrement an absolute value of a date, time, or timestamp. Intervals are qualified to be either YEAR/MONTH intervals or DAY/TIME intervals

Database

- A database is an application that stores the collection of organized and interrelated data.

- Each database has its own structure, data types, and constraints of the data, their relation with another constraint, and the data or information about an object.

- **The data stored in a database would update regularly**. Hence it changes frequently. We can modify or change the data stored in the database using the **DML (data manipulation language) command**. The data in the database at a particular moment is called a database instance.

Creating a database:

- The CREATE DATABASE statement is a foundational SQL command used to **create new databases** in SQL-based Database Management Systems (DBMS), including **MySQL**, PostgreSQL, **SQL Server**, and others. Understanding how to use this command effectively is crucial for developers, database administrators, and anyone working with relational databases.

- The CREATE DATABASE command establishes a new **database** within your **SQL** ecosystem. A database is a repository that organizes data in structured formats through **tables**, views, **stored procedures**, and other components.

- **Syntax:**

- The syntax to use the CREATE DATABASE command in SQL is:

CREATE DATABASE database_name;

Example:

CREATE DATABASE college;

Schema:

- A schema is a **logical representation** of a database that describes the structural definition or description of an entire database. We must specify schema during the design of a database.
- Once we define the database schema, we should not change it frequently because it would disturb the organization of data in a database.
- We can display a database schema in the form of a diagram referred to as a **schema diagram**. This diagram indicates what data contains in a table, what variables are, and how they are associated with each other.
- We can specify the schema using the **DDL (Data Definition Language) statements**. The DDL statement sets the table name, the attributes and their types, constraints, and its relation with other tables in a database.

Creating a Schema:

- An **SQL schema** is identified by a **schema name** and includes an **authorization identifier** to indicate the user or account who owns the schema, as well as **descriptors** for each element in the schema.
- **Schema elements** include tables, types, constraints, views, domains, and other constructs (such as authorization grants) that describe the schema.
- A schema is created via the CREATE SCHEMA statement, which can include all the schema elements' definitions.
- Syntax: CREATE SCHEMA schema_name;
- For example,

  CREATE SCHEMA COMPANY **AUTHORIZATION** 'Jsmith';

  Create Table Construct:
- The CREATE TABLE command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints. The

attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and possibly attribute constraints, such as NOT NULL.

- An SQL relation is defined using the create table command:

```
        CREATE table table_name
(
Column1 datatype (size),
column2 datatype (size),
.
.
columnN datatype(size)
);
```

- table_name: The name you assign to the new table.
- column1, column2, … : The names of the columns in the table.
- datatype(size): Defines the data type and size of each column.

Example: create table *instructor* (

```
ID          char(5),
name        varchar(20),
dept_name   varchar(20),
salary      FLOAT(8,2));
```

Specifying Integrity Constraints in SQL:

- Types of integrity constraints

  1. primary key (A_1, ..., A_n): specifies one or more attributes that make up the primary key of a relation. If a primary key has a single attribute, the clause can follow the attribute directly.

  - Example: Dnumber INT PRIMARY KEY,
  - The UNIQUE clause specifies alternate (unique) keys, also known as candidate keys
  - Example: Dname VARCHAR(15) UNIQUE

  2. foreign key (A_m, ..., A_n) **references** *r:* a referential integrity constraint can be violated when tuples are inserted or deleted, or when a foreign key or primary key attribute value is updated.

- Example: **create table** *instructor* (

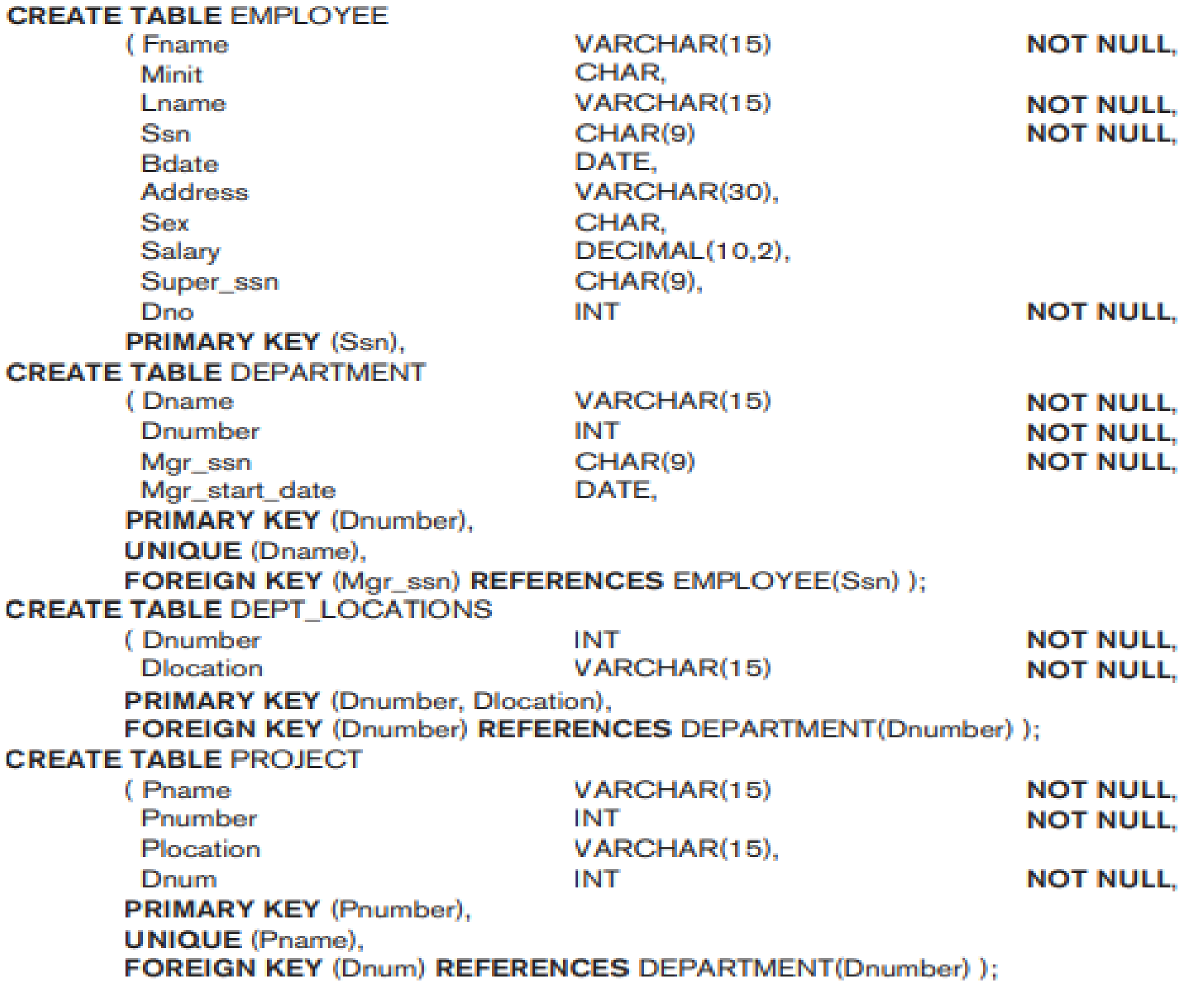
```
        ID          char(5),
name         varchar(20) not null,
dept_name  varchar(20),
salary         numeric(8,2),
primary key (ID),
foreign key (dept_name) references department);
```

3. An alternative action to be taken by attaching a referential triggered action clause to any foreign key constraint. The options include **SET NULL, CASCADE, and SET DEFAULT**.

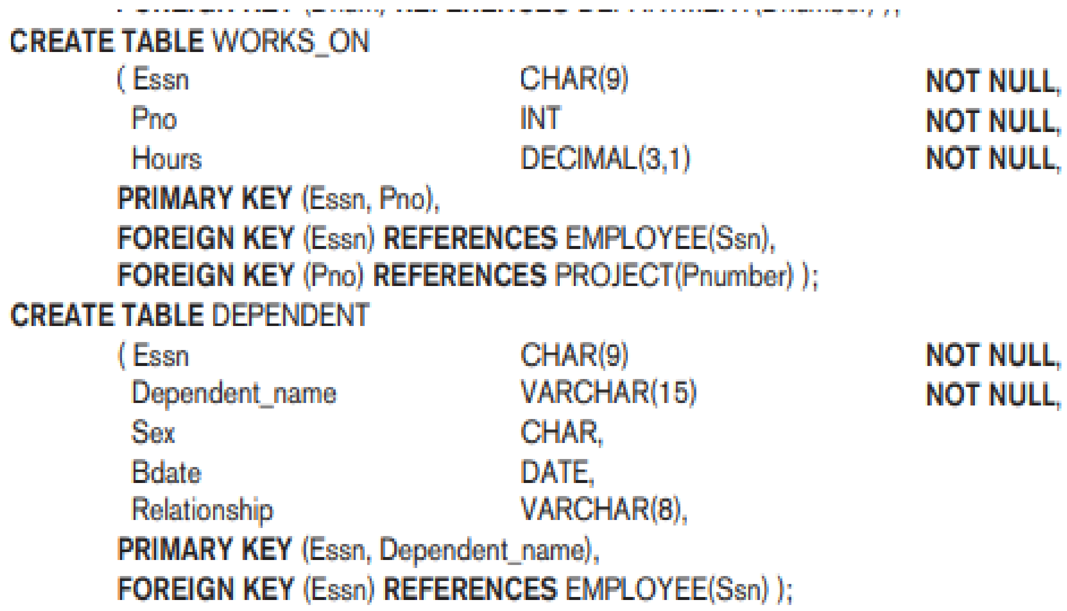
An option must be qualified with either **ON DELETE or ON UPDATE** : The value of the affected referencing attributes is changed to NULL for SET NULL and to the specified default value of the referencing attribute for SET DEFAULT. The action for **CASCADE ON DELETE** is to delete all the referencing tuples, whereas the action for **CASCADE ON UPDATE** is to change the value of the referencing foreign key attribute(s) to the updated (new) primary key value for all the referencing tuples.

4. Check: The table constraints can be specified through additional CHECK clauses at the end of a CREATE TABLE statement. These can be called row-based constraints because they apply to each row individually and are checked whenever a row is inserted or modified.

Example: CHECK (Dept_create_date <= Mgr_start_date);



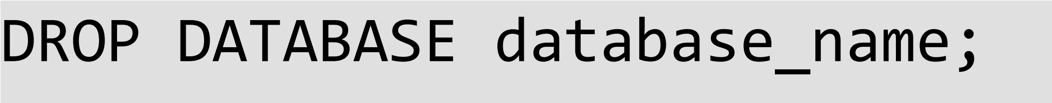
```
CREATE TABLE EMPLOYEE
        ( Fname                 VARCHAR(15)             NOT NULL,
          Minit                 CHAR,
          Lname                 VARCHAR(15)             NOT NULL,
          Ssn                   CHAR(9)                 NOT NULL,
          Bdate                 DATE,
          Address               VARCHAR(30),
          Sex                   CHAR,
          Salary                DECIMAL(10,2),
          Super_ssn             CHAR(9),
          Dno                   INT                     NOT NULL,
        PRIMARY KEY (Ssn),
CREATE TABLE DEPARTMENT
        ( Dname                 VARCHAR(15)             NOT NULL,
          Dnumber               INT                     NOT NULL,
          Mgr_ssn               CHAR(9)                 NOT NULL,
          Mgr_start_date        DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
CREATE TABLE DEPT_LOCATIONS
        ( Dnumber               INT                     NOT NULL,
          Dlocation             VARCHAR(15)             NOT NULL,
        PRIMARY KEY (Dnumber, Dlocation),
        FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE PROJECT
        ( Pname                 VARCHAR(15)             NOT NULL,
          Pnumber               INT                     NOT NULL,
          Plocation             VARCHAR(15),
          Dnum                  INT                     NOT NULL,
        PRIMARY KEY (Pnumber),
        UNIQUE (Pname),
        FOREIGN KEY (Dnum) REFERENCES DEPARTMENT(Dnumber) );
```



```
CREATE TABLE WORKS_ON
        ( Essn                  CHAR(9)                 NOT NULL,
          Pno                   INT                     NOT NULL,
          Hours                 DECIMAL(3,1)            NOT NULL,
        PRIMARY KEY (Essn, Pno),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Pno) REFERENCES PROJECT(Pnumber) );
CREATE TABLE DEPENDENT
        ( Essn                  CHAR(9)                 NOT NULL,
          Dependent_name        VARCHAR(15)             NOT NULL,
          Sex                   CHAR,
          Bdate                 DATE,
          Relationship          VARCHAR(8),
        PRIMARY KEY (Essn, Dependent_name),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn) );
```

Drop the database:

- The **SQL DROP DATABASE** statement is an important command used to permanently delete a **database** from the **Database Management System (DBMS)**. When executed, this command removes the **database** and all its **associated objects**, including **tables**, **views**, **stored procedures**, and other entities.
- Once executed, this operation cannot be undone, so it's crucial to **back up** the database before performing this action.
- This command is typically used in scenarios where a database is **no longer required** or needs to be removed for **reorganization**. Before running the command, ensure that we have appropriate **permissions** to drop the database, as only **database administrators** typically have the necessary rights.
- **Purpose**: To remove a database from the DBMS permanently.
- **Outcome**: All data, schema, and database objects are erased.
- **Syntax:**



```
DROP DATABASE database_name;
```

- **Example:**

  DROP DATABASE College;

Drop Schema, table:

- **Drop Command:** It can be used to drop named schema elements, such as tables, domains or constraints. Example ,
  - **Syntax:** **Drop schema schema_name**;
  - *Schema is dropped only if it has no elements in it otherwise DROP command will not be executed.*
  - *Example:* ***DROP schema*** *Dependent;*
- The DROP TABLE command in SQL is a powerful and essential tool used to permanently delete a table from a database, along with all of its data, structure, and associated constraints such as indexes, triggers, and so on. When executed, this command removes the table and all its contents, making it unrecoverable unless backed up.

- The DROP TABLE statement in SQL is used to delete a table and all of its data from the database permanently.
- This operation cannot be undone, and once the table is dropped all data in that table is lost.
- Once executed, this operation removes the table definition and all of its rows, so the table can no longer be accessed or used.
- This action is irreversible which means that once a table is dropped, it cannot be recovered unless there is a backup.
- Syntax:

  DROP TABLE Table-name;

- Example :

  DROP TABLE Instructor;

Insertion: insert a record:

- INSERT is used to add a single tuple (row) to a relation (table). We must specify the relation name and a list of values for the tuple. The values should be listed in the same order in which the corresponding attributes were specified in the CREATE TABLE command.
- Syntax:
- Insert into table_name values (val1, val2, val3,…., valn);
- Add a new tuple to *course*

  insert into *course*
  values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

  Or equivalently

  insert into *course* (*course_id*, *title*, *dept_name*, *credits*)
  values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

  Add a new tuple to *course* with *creds* set to null

  insert into *course*
  values ('3003', 'Green', 'Finance', *null*);

- Make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of $18,000.

insert into *instructor*
select *ID, name, dept_name, 18000*
from *student*
where *dept_name* = 'Music' **and** *total_cred* > 144;

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation.

  Otherwise queries like

  insert into *table2* **select** * **from** *table*1

  would cause problem

Update record:

- The UPDATE command is used to modify attribute values of one or more selected tuples. UPDATE command specifies the attributes to be modified and their new values.
- Give a 5% salary raise to all instructors

  update *instructor*
  set *salary* = *salary* * 1.05;

- Give a 5% salary raise to those instructors who earn less than 70000

  update *instructor*
  set *salary* = *salary* * 1.05
  where *salary* < 70000;

- Give a 5% salary raise to instructors whose salary is less than average

  update *instructor*
  set *salary* = *salary* * 1.05
  where *salary* < (**select avg** (salary)
  from *instructor*);

Deletion: delete a record

- The DELETE command removes tuples from a relation. It includes a WHERE clause, similar to that used in an SQL query, to select the tuples to be deleted. Tuples are explicitly deleted from only one table at a time.
- Delete all instructors

  delete from *instructor*

- Delete all instructors from the Finance department

  delete from *instructor*
  where *dept_name*= 'Finance';

- *Delete all tuples in the instructor relation for those instructors associated with a department located in the Watson building.*

  delete from *instructor*
  where *dept name* **in** (**select** *dept name*
  from *department*
  where *building* = 'Watson');

- Delete all instructors whose salary is less than the average salary of instructors
  - Problem: as we delete tuples from *instructor*, the average salary changes
  - Solution used in SQL:
    1. First, compute **avg** (salary) and find all tuples to delete
    2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

- **Alter:** the definition of a base table or other schema elements can be changed by this command. Actions like adding column, dropping column, changing column, adding or dropping table constraints and so on.

- **ALTER TABLE** COMPANY. EMPLOYEE **ADD COLUMN** JOB VARCHAR(12);
- **ALTER TABLE** COMPANY. EMPLOYEE **DROP COLUMN** Address CASCADE;
- **ALTER TABLE** COMPANY. DEPARTMENT **ALTER COLUMN mgr_ssn;**
- **ALTER TABLE** COMPANY. DEPARTMENT **ALTER COLUMN mgr_ssn SET DEFAULT** '334455';

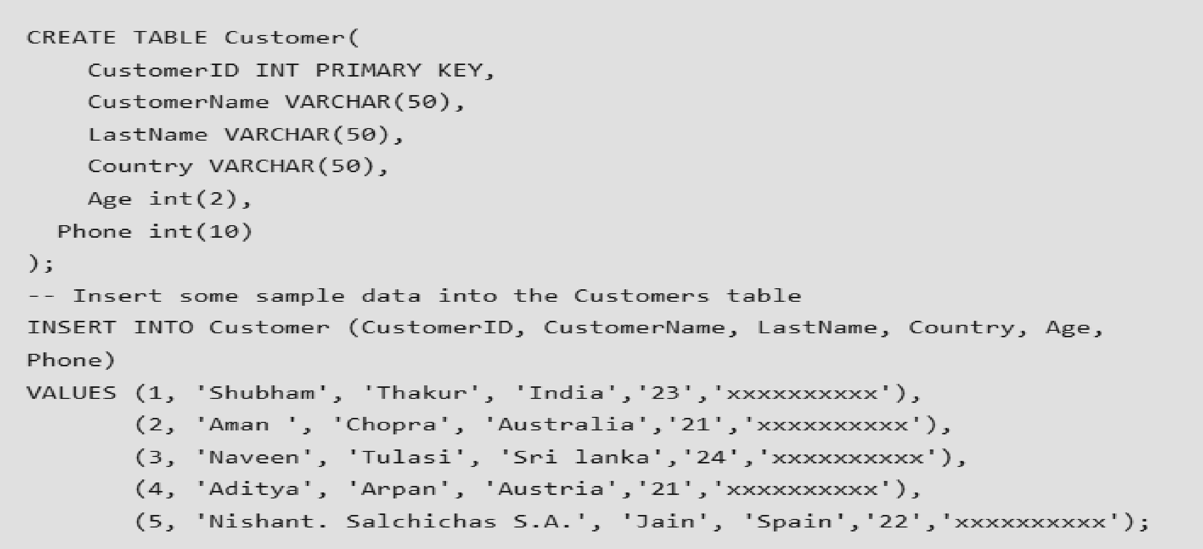
Basic Retrieval Queries in SQL : Select table

- The select query in SQL is one of the most commonly used **SQL** commands to retrieve data from a **database**. With the select command in SQL, users can access **data** and retrieve specific records based on various conditions, making it an essential tool for managing and analyzing data.
- The **SELECT statement** in **SQL** is used to fetch or retrieve data from a database. It allows users to access the data and retrieve specific data based on specific conditions.
- We can fetch either the entire table or according to some specified rules. The data returned is stored in a result table. With the **SELECT clause** of a SELECT command statement, we specify the columns that we want to be displayed in the query result and, optionally, which column headings we prefer to see above the result table.
- The SELECT clause is the first clause and is one of the last clauses of the select statement that the database server evaluates. The reason for this is that before we can determine what to include in the final result set, we need to know all of the possible columns that could be included in the final result set.
- *It is used to access records from one or more database tables and views.*
- *The SELECT statement retrieves selected data based on specified conditions.*
- *The result of a SELECT statement is stored in a result set or result table.*

- *The SELECT statement can be used to access specific columns or all columns from a table.*
- *It can be combined with clauses like WHERE, GROUP BY, HAVING, and ORDER BY for more refined data retrieval.*
- *The SELECT statement is versatile and allows users to fetch data based on various criteria efficiently.*

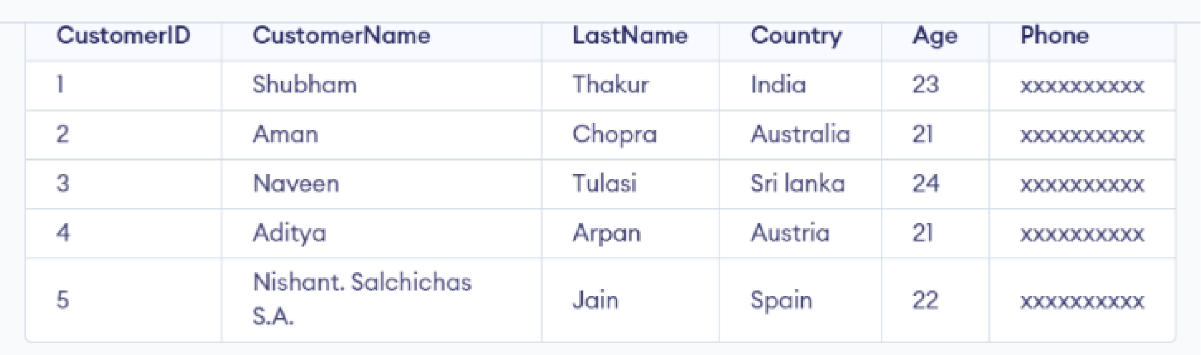
Syntax:

- The syntax for the SELECT statement is:
- *SELECT column1,column2…. FROM table_name ;*



```
CREATE TABLE Customer(
    CustomerID INT PRIMARY KEY,
    CustomerName VARCHAR(50),
    LastName VARCHAR(50),
    Country VARCHAR(50),
    Age int(2),
  Phone int(10)
);
-- Insert some sample data into the Customers table
INSERT INTO Customer (CustomerID, CustomerName, LastName, Country, Age,
Phone)
VALUES (1, 'Shubham', 'Thakur', 'India','23','xxxxxxxxxx'),
       (2, 'Aman ', 'Chopra', 'Australia','21','xxxxxxxxxx'),
       (3, 'Naveen', 'Tulasi', 'Sri lanka','24','xxxxxxxxxx'),
       (4, 'Aditya', 'Arpan', 'Austria','21','xxxxxxxxxx'),
       (5, 'Nishant. Salchichas S.A.', 'Jain', 'Spain','22','xxxxxxxxxx');
```

Output:

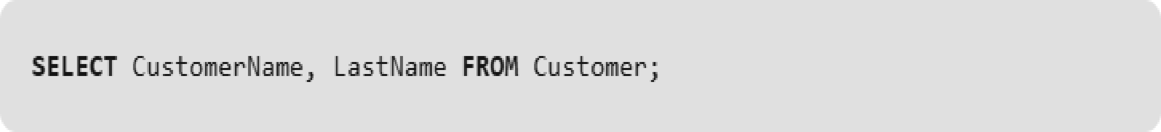


CustomerID	CustomerName	LastName	Country	Age	Phone
1	Shubham	Thakur	India	23	xxxxxxxxxx
2	Aman	Chopra	Australia	21	xxxxxxxxxx
3	Naveen	Tulasi	Sri lanka	24	xxxxxxxxxx
4	Aditya	Arpan	Austria	21	xxxxxxxxxx
5	Nishant. Salchichas S.A.	Jain	Spain	22	xxxxxxxxxx

Example 1: Retrieve Data using SELECT Query

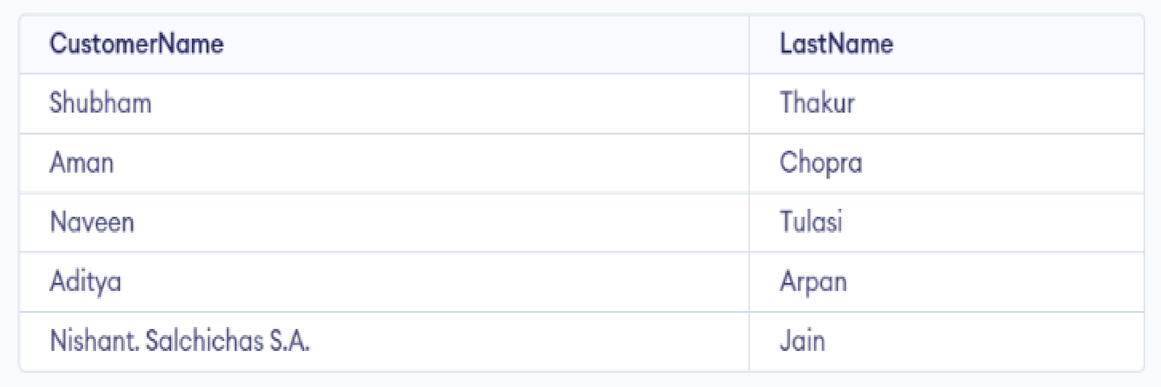
In this example, we will fetch CustomerName, LastName from the table Customer:

Query:



```
SELECT CustomerName, LastName FROM Customer;
```

Output:

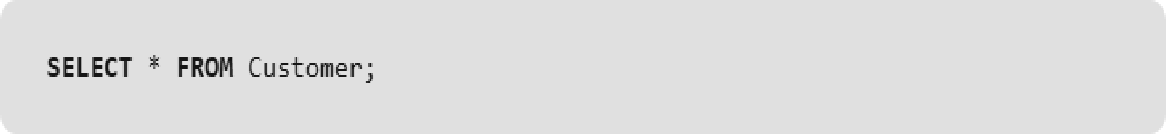


CustomerName	LastName
Shubham	Thakur
Aman	Chopra
Naveen	Tulasi
Aditya	Arpan
Nishant. Salchichas S.A.	Jain

Example 2: Fetch All Table using SELECT Statement.

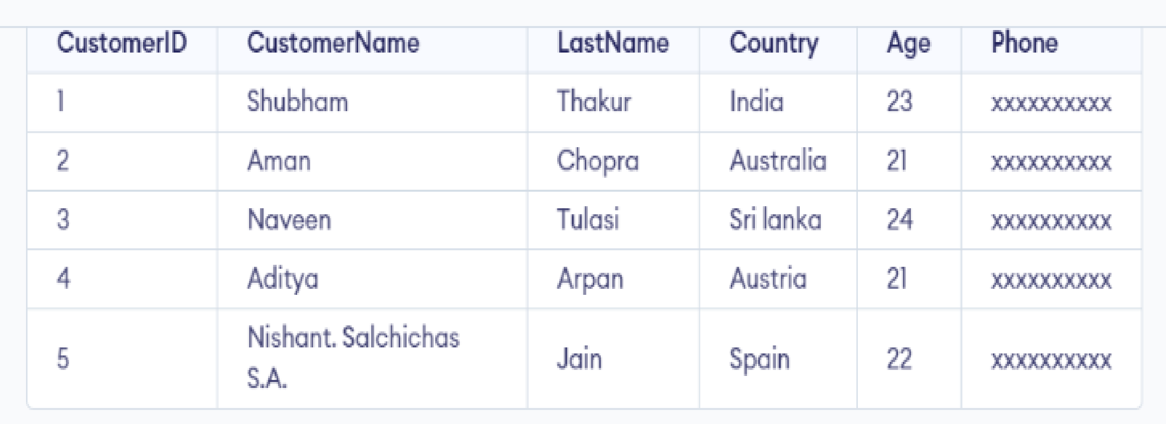
In this example, we will fetch all the fields from the table Customer:

Query:



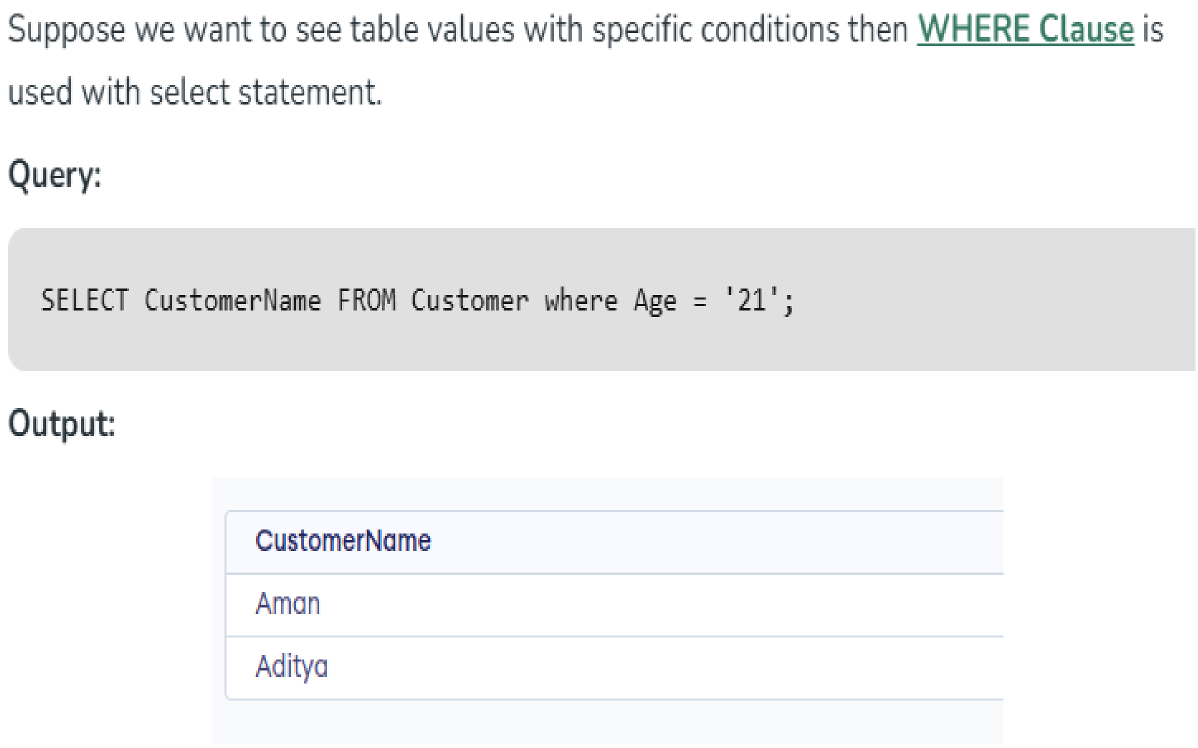
```
SELECT * FROM Customer;
```

Output:



CustomerID	CustomerName	LastName	Country	Age	Phone
1	Shubham	Thakur	India	23	xxxxxxxxxx
2	Aman	Chopra	Australia	21	xxxxxxxxxx
3	Naveen	Tulasi	Sri lanka	24	xxxxxxxxxx
4	Aditya	Arpan	Austria	21	xxxxxxxxxx
5	Nishant. Salchichas S.A.	Jain	Spain	22	xxxxxxxxxx

Example 3: SELECT Statement with WHERE Clause.



Suppose we want to see table values with specific conditions then WHERE Clause is used with select statement.

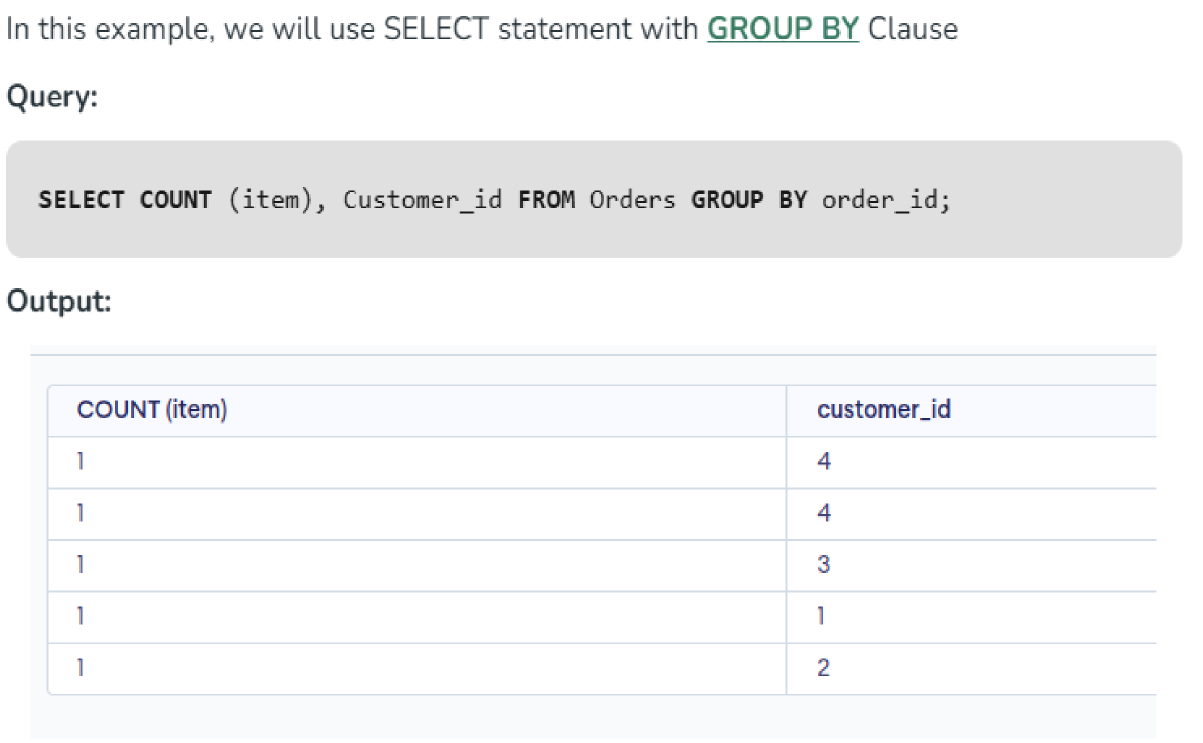
Query:

```
SELECT CustomerName FROM Customer where Age = '21';
```

Output:

CustomerName
Aman
Aditya

Example 4: SQL SELECT Statement with GROUP BY Clause.



In this example, we will use SELECT statement with GROUP BY Clause

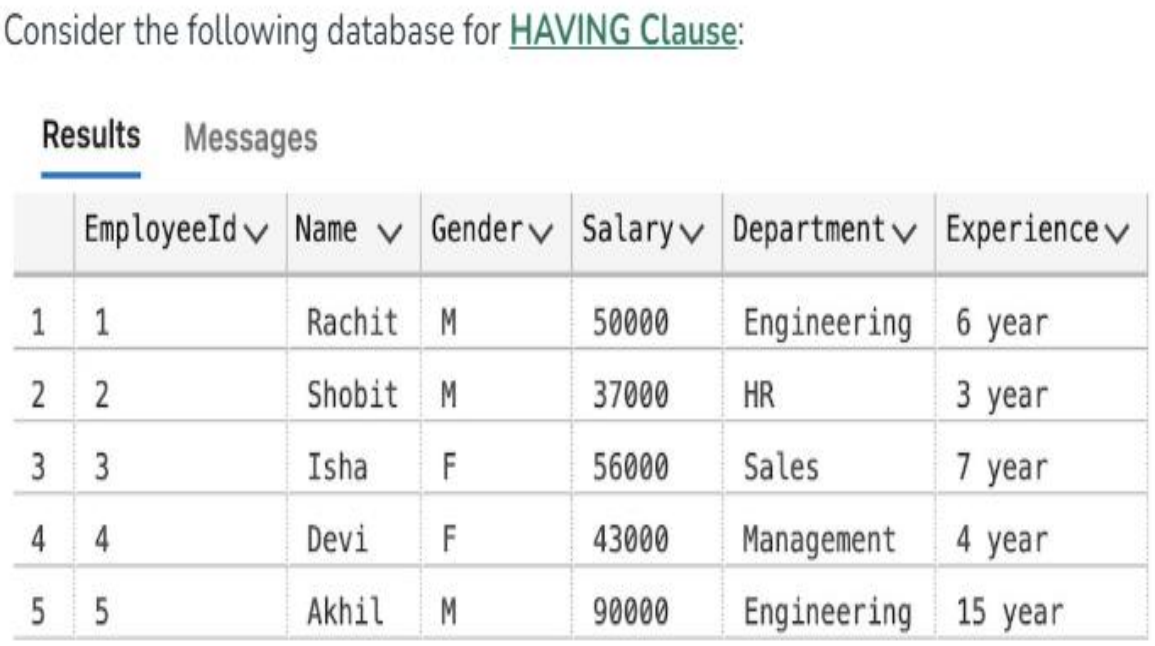
Query:

```
SELECT COUNT (item), Customer_id FROM Orders GROUP BY order_id;
```

Output:

COUNT (item)	customer_id
1	4
1	4
1	3
1	1
1	2

Example 5: SELECT Statement with HAVING Clause.

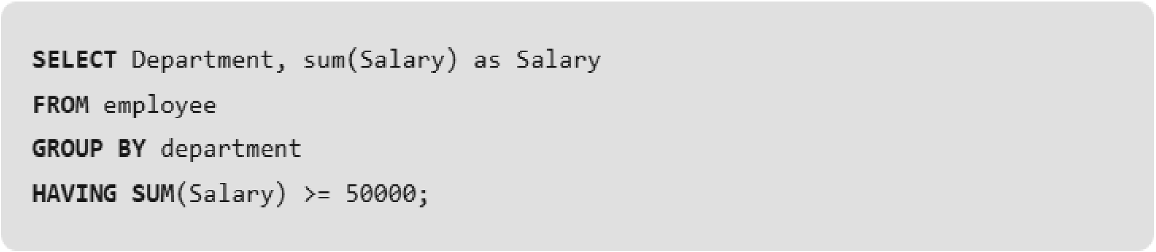


Consider the following database for HAVING Clause:

Results Messages

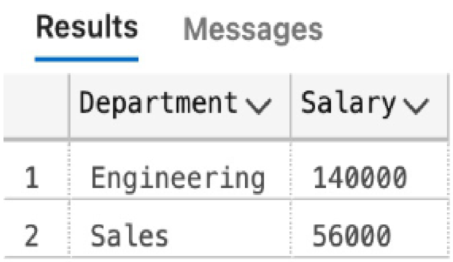
	EmployeeId	Name	Gender	Salary	Department	Experience
1	1	Rachit	M	50000	Engineering	6 year
2	2	Shobit	M	37000	HR	3 year
3	3	Isha	F	56000	Sales	7 year
4	4	Devi	F	43000	Management	4 year
5	5	Akhil	M	90000	Engineering	15 year

Query:



```
SELECT Department, sum(Salary) as Salary
FROM employee
GROUP BY department
HAVING SUM(Salary) >= 50000;
```

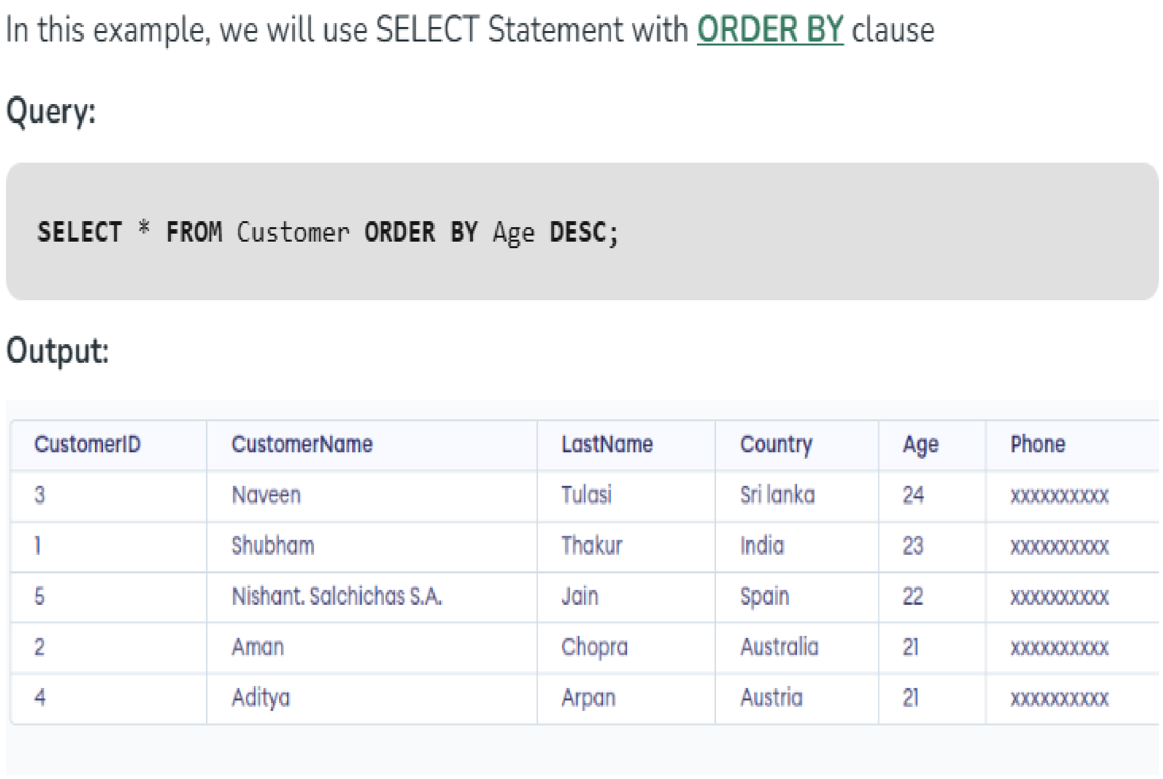
Output:



Results Messages

	Department	Salary
1	Engineering	140000
2	Sales	56000

Example 6: SELECT Statement with ORDER BY clause in SQL



In this example, we will use SELECT Statement with ORDER BY clause

Query:

```
SELECT * FROM Customer ORDER BY Age DESC;
```

Output:

CustomerID	CustomerName	LastName	Country	Age	Phone
3	Naveen	Tulasi	Sri lanka	24	xxxxxxxxxx
1	Shubham	Thakur	India	23	xxxxxxxxxx
5	Nishant. Salchichas S.A.	Jain	Spain	22	xxxxxxxxxx
2	Aman	Chopra	Australia	21	xxxxxxxxxx
4	Aditya	Arpan	Austria	21	xxxxxxxxxx

SELECT STATEMENT:

- In SQL, the basic logical comparison operators for comparing attribute values with one another and with literal constants are =, <=, >, >=, and <>.

- Example Query 0. Retrieve the birth date and address of the employee(s) whose name is ‘John B. Smith’.

Q0: SELECT Bdate, Address FROM EMPLOYEE WHERE Fname = ‘John’ AND Minit = ‘B’ AND Lname = ‘Smith’;

- This query involves only the EMPLOYEE relation listed in the FROM clause. The query selects the individual EMPLOYEE tuples that satisfy the condition of the WHERE clause, then projects the result on the Bdate and Address attributes listed in the SELECT clause.

- The SELECT clause of SQL specifies the attributes whose values are to be retrieved, which are called the **projection attributes** in relational algebra and the WHERE clause specifies the **Boolean** condition that must be true for any retrieved tuple, which is known as the **selection condition** in relational algebra.

Substring Pattern Matching and STRING Operators:

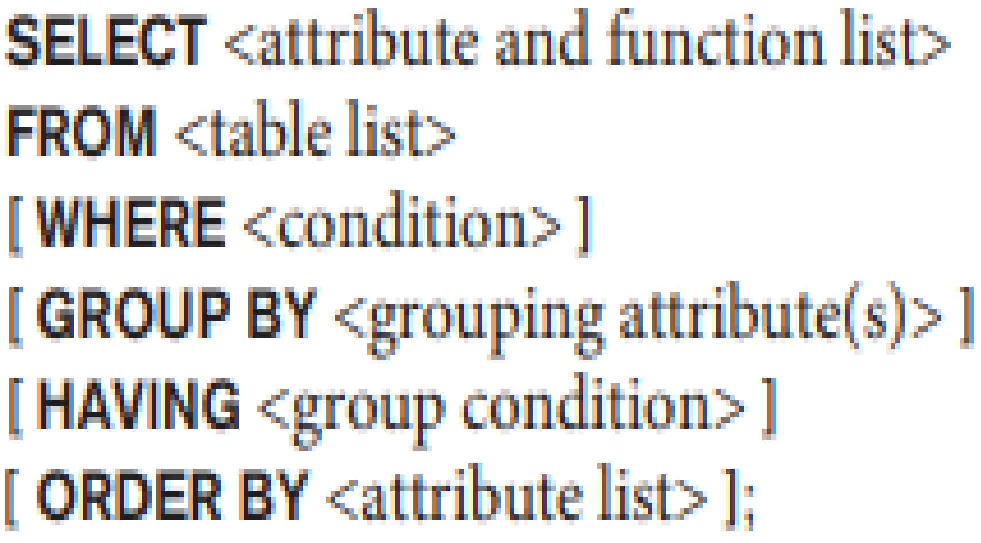
- The first feature allows comparison conditions on only parts of a character string, using the LIKE comparison operator. This can be used for string pattern matching. Partial strings are specified using two reserved characters: % replaces an arbitrary number of zero or more characters, and the underscore (_) replaces a single character.
- For example, consider the following query.
- Query 12. Retrieve all employees whose address is in Houston, Texas.
- Q12: SELECT Fname, Lname

  FROM EMPLOYEE

  WHERE Address LIKE '%Houston,TX%';

Summary of Basic SQL Retrieval Queries:

- A retrieval query in SQL can consist of up to six clauses, but only the first two— SELECT and FROM—are mandatory. The query can span several lines, and is ended by a semicolon. Query terms are separated by spaces, and parentheses can be used to group relevant parts of a query in the standard way. The clauses are specified in the following order, with the clauses between square brackets […] being optional:



```
SELECT <attribute and function list>
FROM <table list>
[ WHERE <condition> ]
[ GROUP BY <grouping attribute(s)> ]
[ HAVING <group condition> ]
[ ORDER BY <attribute list> ];
```

Set Operation:

- The SQL Set operation is used to combine the two or more SQL SELECT statements.
- Types of Set Operation

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

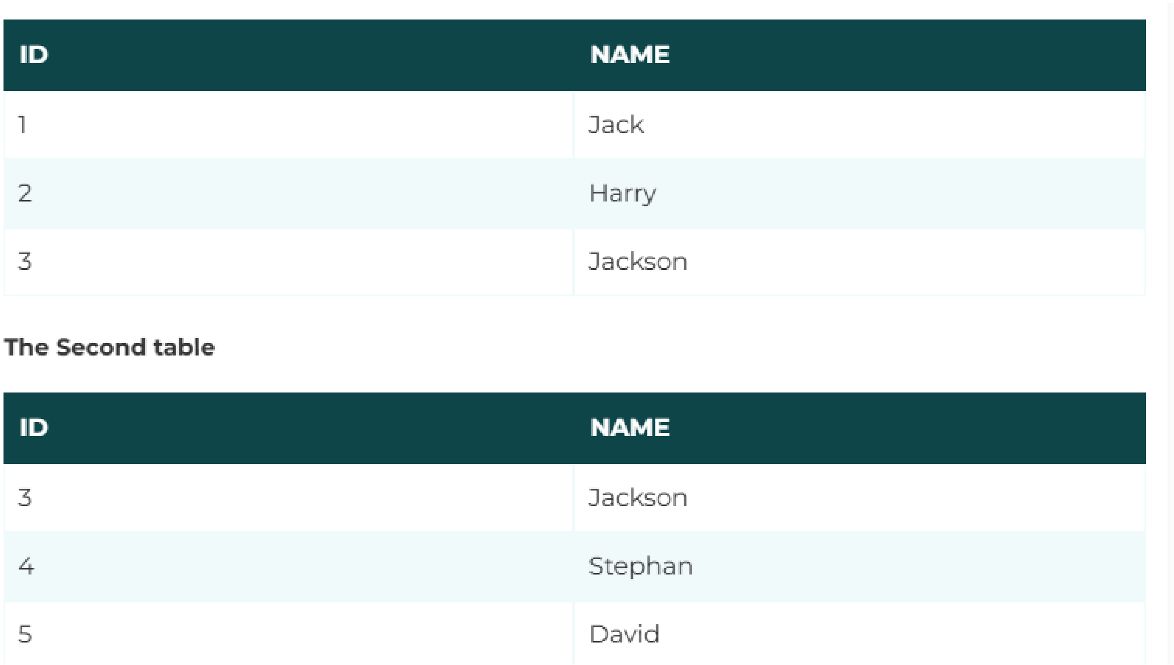
1. Union:

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

```
SELECT column_name FROM table1
UNION
SELECT column_name FROM table2;
```

Example:

The First table



ID	NAME
1	Jack
2	Harry
3	Jackson

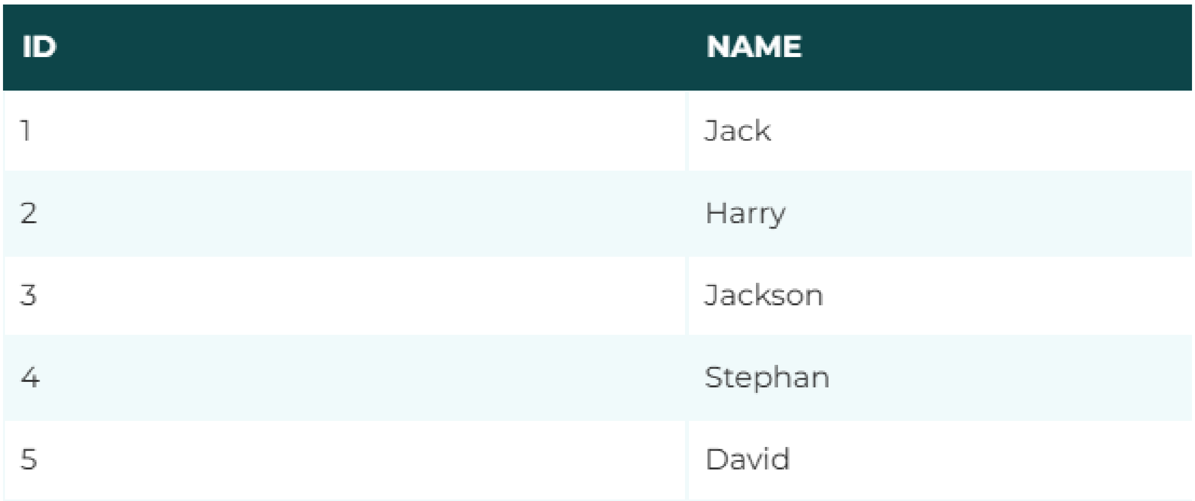
The Second table

ID	NAME
3	Jackson
4	Stephan
5	David

- Union SQL query will be:

```
SELECT * FROM First
UNION
SELECT * FROM Second;
```

- The resultset table will look like:



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

2. Union All

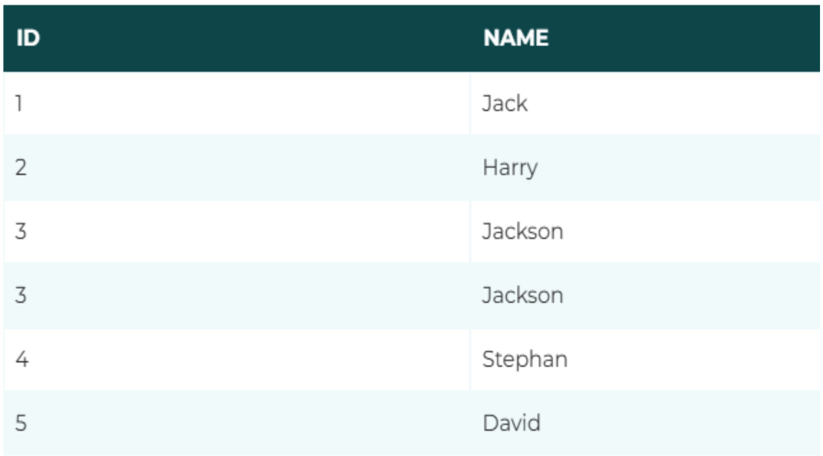
- Union All operation is equal to the Union operation. It returns the set without removing duplication and sorting the data.
- **Syntax:**

```
SELECT column_name FROM table1
UNION ALL
SELECT column_name FROM table2;
```

- **Example:** Using the above First and Second table.
- Union All query will be like:

```
SELECT * FROM First
UNION ALL
SELECT * FROM Second;
```

- The resultset table will look like:



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

3. Intersect:

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

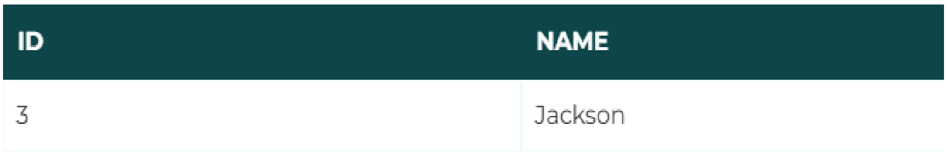
```
SELECT column_name FROM table1
INTERSECT
SELECT column_name FROM table2;
```

Example:

- **Using the above First and Second table.**
- Intersect query will be:

```
SELECT * FROM First
INTERSECT
SELECT * FROM Second;
```

- The resultset table will look like:



ID	NAME
3	Jackson

4. Minus

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

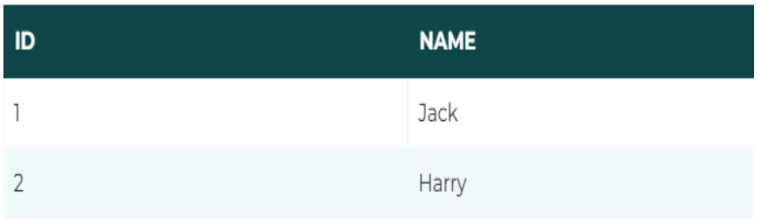
```
SELECT column_name FROM table1
```

```
MINUS
SELECT column_name FROM table2;
```

- **Example**
- **Using the above First and Second table.**
- Minus query will be:

```
SELECT * FROM First
MINUS
SELECT * FROM Second;
```

The resultset table will look like:



ID	NAME
1	Jack
2	Harry

Aggregate Functions:

- Aggregate functions are used to summarize information from multiple tuples into a single-tuple summary.
- **Grouping** is used to create subgroups of tuples before summarization. Grouping and aggregation are required in many database applications.
- These functions operate on the multiset of values of a column of a relation, and return a value

  avg: average value
  min: minimum value
  max: maximum value
  sum: sum of values
  count: number of values

- *Aggregate functions in SQL operate on a group of values and return a single result.*
- *They are often used with the GROUP BY clause to summarize the grouped data.*
- *Aggregate function operates on non-NULL values only (except COUNT).*

Aggregate Functions with Examples:

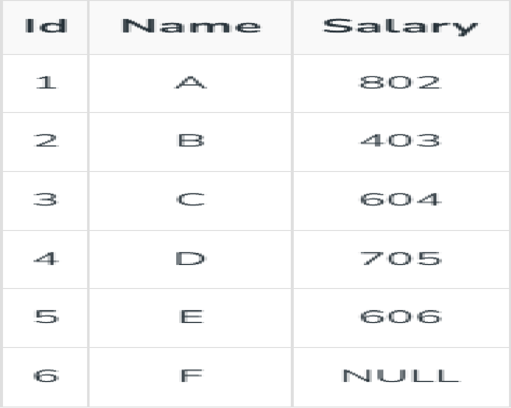
- Let's consider a demo Employee table for our examples. This table contains employee details such as their ID, Name, and Salary.
- **Query:**
- CREATE TABLE Employee (

```
Id INT PRIMARY KEY,
Name CHAR(1),
```

Salary DECIMAL(10,2)

-);
- INSERT INTO Employee (Id, Name, Salary)
  VALUES (1, 'A', 802),
  (2, 'B', 403),
  (3, 'C', 604),
  (4, 'D', 705),
  (5, 'E', 606),
  (6, 'F', NULL);

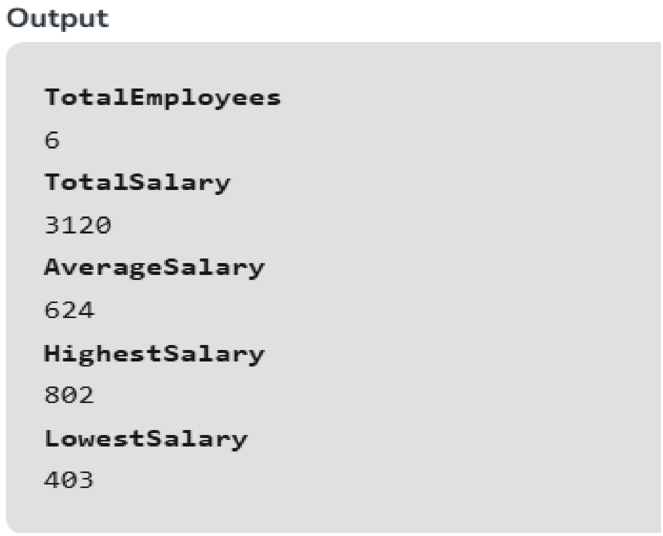
Output:



Id	Name	Salary
1	A	802
2	B	403
3	C	604
4	D	705
5	E	606
6	F	NULL

Queries:

- --Count the number of employees
  SELECT COUNT(*) AS TotalEmployees FROM Employee;
- -- Calculate the total salary
  SELECT SUM(Salary) AS TotalSalary FROM Employee;
- -- Find the average salary
  SELECT AVG(Salary) AS AverageSalary FROM Employee;
- -- Get the highest salary
  SELECT MAX(Salary) AS HighestSalary FROM Employee;
- -- Determine the lowest salary
  SELECT MIN(Salary) AS LowestSalary FROM Employee;



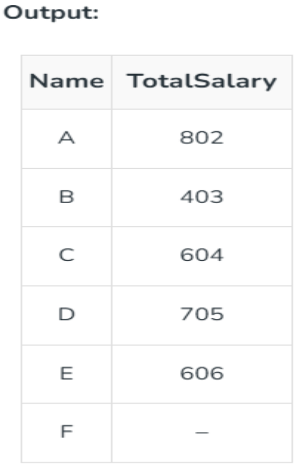
Output

```
TotalEmployees
6
TotalSalary
3120
AverageSalary
624
HighestSalary
802
LowestSalary
403
```

Using Aggregate Functions with GROUP BY:

- GROUP BY allows you to group rows that share a property, enabling you to perform **aggregate calculation**s on each group. This is commonly used with the COUNT(), SUM(), AVG(), MIN(), and MAX() functions.
- **Example: Total Salary by Each Employee**
- **Query:**

```
SELECT Name, SUM(Salary) AS TotalSalary
FROM Employee
GROUP BY Name;
```



Output:

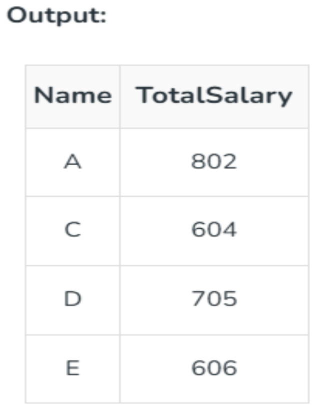
Name	TotalSalary
A	802
B	403
C	604
D	705
E	606
F	–

Using HAVING with Aggregate Functions:

- The HAVING clause is used to filter results after applying aggregate functions. Unlike WHERE, which filters rows before aggregation, HAVING filters groups after aggregation.

- Example: Find Employees with Salary Greater Than 600
- Query:

```
SELECT Name, SUM(Salary) AS TotalSalary
FROM Employee
GROUP BY Name
HAVING SUM(Salary) > 600;
```



Output:

Name	TotalSalary
A	802
C	604
D	705
E	606

Nested Sub Queries:

- An SQL Subquery, is a SELECT query within another query. It is also known as Inner query or Nested query and the query containing it is the outer query.
- The outer query can contain the SELECT, INSERT, UPDATE, and DELETE statements. We can use the subquery as a column expression, as a condition in SQL clauses, and with operators like =, >, <, >=, <=, IN, BETWEEN, etc.

Rules to be followed:

- A subquery can be placed in a number of SQL clauses like WHERE clause, FROM clause, HAVING clause.
- You can use Subquery with SELECT, UPDATE, INSERT, DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.
- A subquery is a query within another query. The outer query is known as the main query, and the inner query is known as a subquery.
- Subqueries are on the right side of the comparison operator.
- A subquery is enclosed in parentheses.
- In the Subquery, ORDER BY command cannot be used. But GROUP BY command can be used to perform the same function as ORDER BY command.

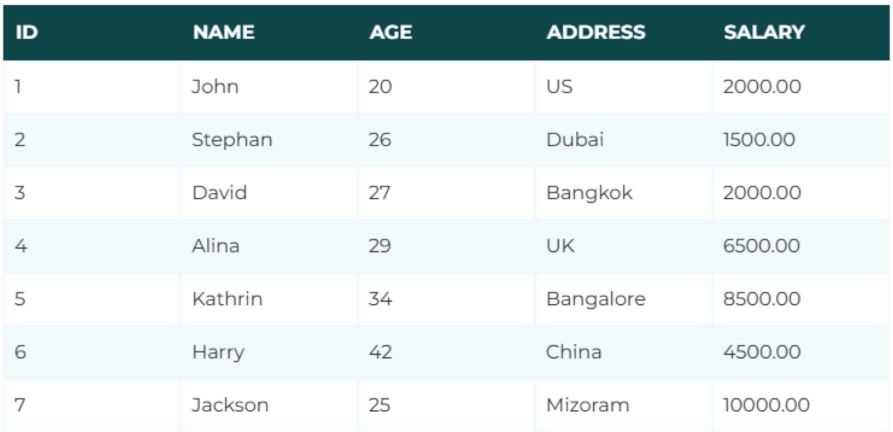
Subqueries with the Select Statement:

- SQL subqueries are most frequently used with the Select statement.
- **Syntax**

```
SELECT column_name
FROM table_name
WHERE column_name expression operator
( SELECT column_name  from table_name WHERE ... );
```

Example

Consider the EMPLOYEE table have the following records:

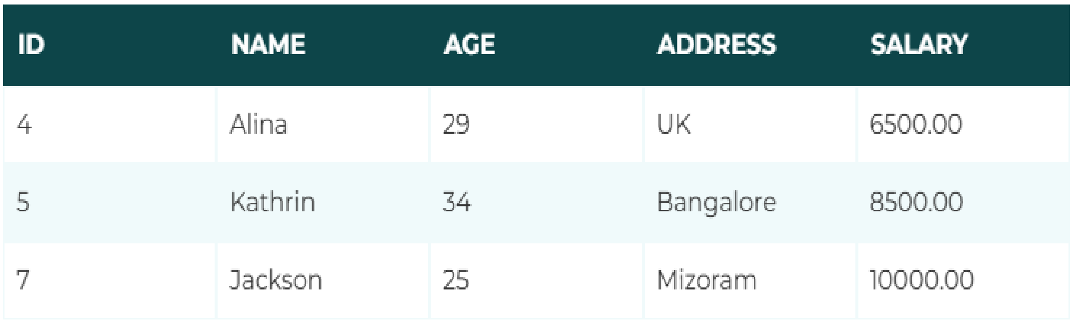


ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
4	Alina	29	UK	6500.00
5	Kathrin	34	Bangalore	8500.00
6	Harry	42	China	4500.00
7	Jackson	25	Mizoram	10000.00

- The subquery with a SELECT statement will be:

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

- This would produce the following result:



ID	NAME	AGE	ADDRESS	SALARY
4	Alina	29	UK	6500.00
5	Kathrin	34	Bangalore	8500.00
7	Jackson	25	Mizoram	10000.00

Views:

- Views in SQL are a type of **virtual table** that simplifies how users interact with data across one or more tables. Unlike **traditional tables**, a view in **SQL** does not store data on disk; instead, it dynamically retrieves data based on a pre-defined query each time it's accessed.

- SQL views are particularly useful for managing complex queries, enhancing security, and presenting data in a simplified format. In this guide, we will cover the **SQL** create view statement, updating and deleting views, and using the WITH CHECK OPTION clause.
- **What is a View in SQL?**
- A view in SQL is a saved SQL query that acts as a virtual table. It can fetch data from one or more tables and present it in a customized format, allowing developers to:

- **Simplify Complex Queries:** Encapsulate complex joins and conditions into a single object.
- **Enhance Security:** Restrict access to specific columns or rows.
- **Present Data Flexibly:** Provide tailored data views for different users.

CREATE VIEWS in SQL:

- We can create a view using CREATE VIEW statement. A View can be created from a single table or multiple tables.
- Syntax:

```
CREATE VIEW view_name AS
SELECT column1, column2…..
FROM table_name
WHERE condition;
```

- **Parameters:**

- **view_name**: Name for the View
- **table_name**: Name of the table
- **condition**: Condition to select rows

Example 1: Creating View From Table

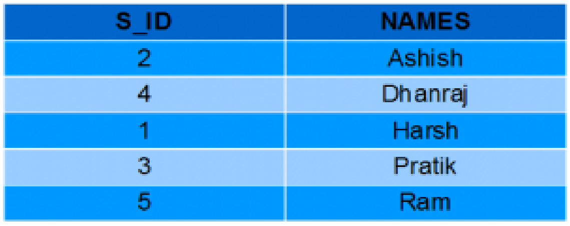
- In this example, we will create a view named StudentNames from the table StudentDetails. Query:

```
CREATE VIEW StudentNames AS
SELECT S_ID, NAME
FROM StudentDetails
ORDER BY NAME;
```

- If we now query the view as,

```
SELECT * FROM StudentNames;
```

Output:



S_ID	NAMES
2	Ashish
4	Dhanraj
1	Harsh
3	Pratik
5	Ram

Delete View in SQL:

- SQL allows us to delete an existing View. We can delete or drop View using the DROP statement.
- **Syntax:**

```
DROP VIEW view_name;
```

- **Example**
- In this example, we are deleting the View MarksView.

```
DROP VIEW MarksView;
```

Update View in SQL:

- If you want to update the existing data within the view, use the UPDATE statement.
- **Syntax:**

```
UPDATE view_name
SET column1 = value1, column2 = value2...., columnN = valueN
WHERE [condition];
```

- **Rules to Update Views in SQL:**
- Certain conditions need to be satisfied to update a view. If any of these conditions are **not** met, the view can not be updated.

1. The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
2. The SELECT statement should not have the DISTINCT keyword.
3. The View should have all NOT NULL values.
4. The view should not be created using nested queries or complex queries.
5. The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.

Uses of a View:

- A good database should contain views for the given reasons:

1. **Restricting data access** – Views provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table.

2. **Hiding data complexity –** A view can hide the complexity that exists in multiple joined tables.
3. **Simplify commands for the user –** Views allow the user to select information from multiple tables without requiring the users to actually know how to perform a join.
4. **Store complex queries –** Views can be used to store complex queries.
5. **Rename Columns –** Views can also be used to rename the columns without affecting the base tables provided the number of columns in view must match the number of columns specified in a select statement. Thus, renaming helps to hide the names of the columns of the base tables.
6. **Multiple view facility –** Different views can be created on the same table for different users.

Procedures:

- A stored procedure in SQL is a group of SQL queries that can be saved and reused multiple times. It is very useful as it reduces the need for rewriting SQL queries.
- It enhances efficiency, reusability, and security in database management.
- Users can also pass parameters to stored procedures so that the stored procedure can act on the passed parameter values.
- Stored Procedures are created to perform one or more DML operations on the Database. It is nothing but a group of **SQL statements** that accepts some input in the form of parameters, performs some task, and may or may not return a value.

Syntax:

- Syntax to Create a Stored Procedure

```
CREATE PROCEDURE procedure_name
(parameter1 data_type, parameter2 data_type, …)
AS
BEGIN
  — SQL statements to be executed
END
```

- Syntax to Execute the Stored Procedure

```
EXEC procedure_name parameter1_value, parameter2_value, ..
```

- **Parameter**
- The most important part is the parameters. Parameters are used to pass values to the Procedure. There are different types of parameters, which are as follows:

1. **Stored Procedure With One Parameter**

2. Stored Procedure With Multiple Parameters

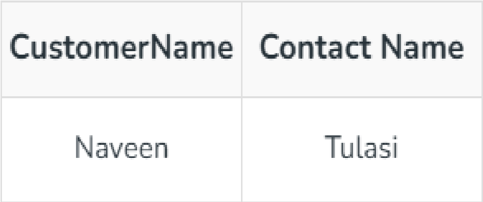
SQL Stored Procedure Example:

-
  ```
  -- Create a new database named "SampleDB"
  CREATE DATABASE SampleDB;
  ```

-
  ```
  -- Create a new table named "Customers"
  CREATE TABLE Customers (
      CustomerID INT PRIMARY KEY,
      CustomerName VARCHAR(50),
      ContactName VARCHAR(50),
      Country VARCHAR(50)
  );
  ```

-
  ```
  -- Insert some sample data into the Customers table
  INSERT INTO Customers (CustomerID, CustomerName, ContactName, Country)
  VALUES (1, 'Shubham', 'Thakur', 'India'),
         (2, 'Aman ', 'Chopra', 'Australia'),
         (3, 'Naveen', 'Tulasi', 'Sri lanka'),
         (4, 'Aditya', 'Arpan', 'Austria'),
         (5, 'Nishant. Salchichas S.A.', 'Jain', 'Spain');
  -- Create a stored procedure named "GetCustomersByCountry"
  CREATE PROCEDURE GetCustomersByCountry
      @Country VARCHAR(50)
  AS
  BEGIN
      SELECT CustomerName, ContactName
      FROM Customers
      WHERE Country = @Country;
  END;
  -- Execute the stored procedure with parameter "Sri lanka"
  EXEC GetCustomersByCountry @Country = 'Sri lanka';
  ```

Output:



CustomerName	Contact Name
Naveen	Tulasi

Examples:

1> **Stored Procedure With One Parameter:**

The following SQL statement creates a stored procedure that selects Customers from a particular City from the "Customers" table:

```
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30)
     AS
     SELECT * FROM Customers WHERE City = @City
    GO;
```

Execute the stored procedure above as follows:

- Example

```
        EXEC SelectAllCustomers @City = 'London’;
```

2) **Stored Procedure With Multiple Parameters:**

- Setting up multiple parameters is very easy. Just list each parameter and the data type separated by a comma as shown below.
- The following SQL statement creates a stored procedure that selects Customers from a particular City with a particular PostalCode from the "Customers" table:
- Example

```
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30),
@PostalCode nvarchar(10)
AS
SELECT * FROM Customers WHERE City = @City AND PostalCode =
@PostalCode
GO;
```

- Execute the stored procedure above as follows:
- Example

```
EXEC SelectAllCustomers @City = 'London', @PostalCode = 'WA1
1DP';
```

Important Points About SQL Stored Procedures:

- A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.
- Stored procedures allow for code that is used repeatedly to be saved on the database and run from there, rather than from the client. This provides a more **modular approach** to database design.
- Since stored procedures are compiled and stored in the database, they are highly efficient. SQL Server compiles each stored procedure once and then

reutilizes the execution plan. This leads to tremendous performance boosts when stored procedures are called repeatedly.

- Stored procedures provide better security to your data. Users can execute a stored procedure without needing to execute any of the statements directly. Therefore, a user can be granted permission to execute a stored procedure without having any permissions on the underlying tables.
- Stored procedures can reduce network traffic and latency, boosting application performance. A single call to a stored procedure can execute many statements.
- Stored procedures have better support for error handling.
- Stored procedures can be used to provide advanced database functionality, such as modifying data in tables, and encapsulating these changes within database transactions.

SQL Trigger:

- **SQL triggers** are a critical feature in **database management systems (DBMS)** that provide automatic execution of a set of SQL statements when specific database events, such as **INSERT**, **UPDATE**, or **DELETE** operations, occur.
- Triggers are commonly used to maintain **data integrity**, **track changes**, and **enforce business rules** automatically, without needing manual input.
- A trigger is a stored procedure in a **database** that automatically invokes whenever a special event in the database occurs.
- By using **SQL triggers**, developers can **automate tasks**, ensure **data consistency**, and keep accurate records of **database activities**.
- For example, a trigger can be invoked when a row is inserted into a specified table or when specific table columns are updated.
- In simple words, a **trigger** is a collection of SQL statements with particular names that are stored in system memory. It belongs to a specific class of **stored procedures** that are automatically invoked in response to database server events. Every **trigger** has a table attached to it.

Key Features of SQL Triggers:

- **Automatic Execution**: Triggers fire automatically when the defined event occurs (e.g., INSERT, UPDATE, DELETE).
- **Event-Driven**: Triggers are tied to specific events that take place within the database.

- **Table Association**: A trigger is linked to a specific table or view, and operates whenever changes are made to the table’s data.

BEFORE and AFTER Triggers

- SQL triggers can be specified to run **BEFORE** or **AFTER** the triggering event.
- **BEFORE Triggers:** Execute before the actual SQL statement is executed. Useful for validating data before insertion or updating.
- **AFTER Triggers:** Execute after the SQL statement completes. Useful for logging or cascading updates to other tables.

Syntax:

-
  ```
  Create trigger [trigger_name]
  [before | after]
  {insert | update | delete}
  on [table_name]
  FOR EACH ROW
  AS
  BEGIN
  SQL Statements
  END;
  ```

Key Terms:

- **trigger_name:** The name of the trigger to be created.
- **BEFORE | AFTER:** Specifies whether the trigger is fired **before** or **after** the triggering event (INSERT, UPDATE, DELETE).
- **{INSERT | UPDATE | DELETE}:** Specifies the operation that will activate the trigger.
- **table_name:** The name of the table the trigger is associated with.
- **FOR EACH ROW:** Indicates that the trigger is row-level, meaning it executes once for each affected row.
- **trigger_body:** The SQL statements to be executed when the trigger is fired.

Examples:

```
1.CREATE TRIGGER prevent_table_creation
ON DATABASE
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
BEGIN
PRINT 'you can not create, drop and alter table
```

```
2. CREATE TRIGGER prevent_update
ON students
FOR UPDATE
AS
BEGIN
PRINT 'You can not insert, update and delete
this table i';
ROLLBACK;
```

records when certain conditions are met.

- **Audit Trail**: Triggers can track changes in a database, providing an audit trail of **INSERT**, **UPDATE**, and **DELETE** operations.
- **Performance**: By automating repetitive tasks, triggers improve **SQL query performance** and reduce manual workload.

Cursors:

- Cursor is a Temporary Memory or Temporary Work Station. It is Allocated by Database Server at the Time of Performing DML(Data Manipulation Language) operations on the Table by the User. Cursors are used to store Database Tables.
- There are 2 types of Cursors: Implicit Cursors, and Explicit Cursors. These are explained as following below.

- **Implicit Cursors:** Implicit Cursors are also known as Default Cursors of SQL SERVER. These Cursors are allocated by SQL SERVER when the user performs DML operations.
- **Explicit Cursors**: Explicit Cursors are Created by Users whenever the user requires them. Explicit Cursors are used for Fetching data from Table in Row-By-Row Manner.
- How To Create Explicit Cursor?
- **Declare Cursor Object**
- Syntax:

DECLARE cursor_name CURSOR FOR SELECT * FROM table_name

- Query:

DECLARE s1 CURSOR FOR SELECT * FROM studDetails

2. **Open Cursor Connection**

- Syntax:

OPEN cursor_connection

Query:

OPEN s1

- **Fetch Data from the Cursor** There is a total of 6 methods to access data from the cursor. They are as follows:

1. **FIRST** is used to fetch only the first row from the cursor table.
2. **LAST** is used to fetch only the last row from the cursor table.
3. **NEXT** is used to fetch data in a forward direction from the cursor table.
4. **PRIOR** is used to fetch data in a backward direction from the cursor table.
5. **ABSOLUTE n** is used to fetch the exact n^th^ row from the cursor table.
6. **RELATIVE n** is used to fetch the data in an incremental way as well as a decremental way.

- **Syntax:**

FETCH NEXT/FIRST/LAST/PRIOR/ABSOLUTE n/RELATIVE n FROM cursor_name

- **Query:**

FETCH FIRST FROM s1

FETCH LAST FROM s1

FETCH NEXT FROM s1

FETCH PRIOR FROM s1

FETCH ABSOLUTE 7 FROM s1

FETCH RELATIVE -2 FROM s1

- **Close cursor connection**
- **Syntax:**

CLOSE cursor_name

- Query:

CLOSE s1

- **Deallocate cursor memory**
- **Syntax:**

DEALLOCATE cursor_name

Query:

DEALLOCATE s1

PL/SQL

- PL/SQL (Procedural Language/Structured Query Language) is a **block-structured language** developed by **Oracle** that allows developers to combine the power of **SQL** with procedural programming constructs. The PL/SQL language enables efficient data manipulation and control-flow logic, all within the **Oracle Database**.
- PL/SQL is a combination of SQL along with the procedural features of programming languages.
- Oracle uses a PL/SQL engine to process the PL/SQL statements.

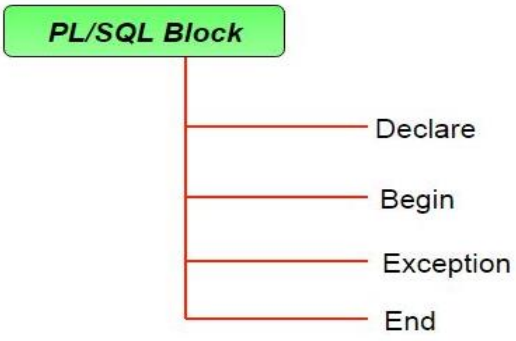
- PL/SQL includes procedural language elements like conditions and loops. It allows declaration of constants and variables, procedures and functions, types and variable of those types and triggers.

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.

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.

An example on PL/SQL to demonstrate:

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

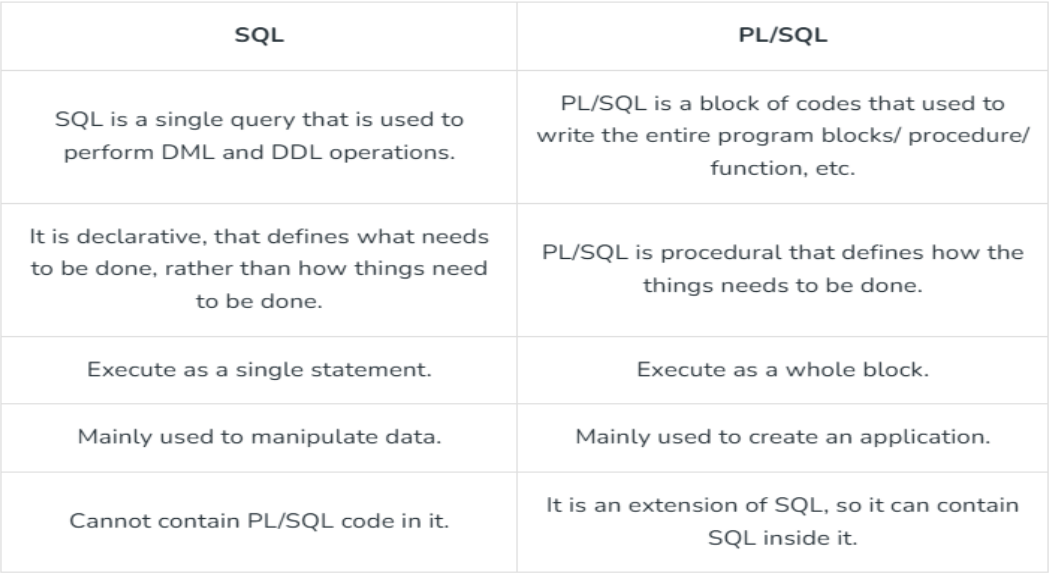
```
SQL> SET SERVEROUTPUT ON;
SQL> DECLARE
          -- taking input for variable a
    a integer := &a ;

    -- taking input for variable b
    b integer := &b ;
    c integer ;
BEGIN
    c := a + b ;
    dbms_output.put_line('Sum of '||a||' and '||b||' is = '||c);
  END;
```

OUTPUT:

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

Differences Between SQL and PL/SQL:



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