UNIT-III: SQL

1) Write the syntax of basic SQL query. Explain briefly about selection and projection in SQL 5 M

Ans:

Basic SQL Query Syntax

```
SELECT column_list
FROM table_name
WHERE condition
GROUP BY column
HAVING condition
ORDER BY column;
```

- **SELECT** – specifies the columns to be displayed (attributes).
- **FROM** – specifies the table(s) from which data is retrieved.
- **WHERE** – filters rows based on given conditions.
- **GROUP BY** – groups rows that have the same values in specified columns.
- **HAVING** – filters groups based on a condition.
- **ORDER BY** – sorts the result in ascending or descending order.

Selection and Projection in SQL

- **Selection:**
  - It is used to retrieve **specific rows** from a table based on a condition.
  - Implemented using the **WHERE** clause.

    Example:

    SELECT * FROM employees WHERE department = 'Sales';

→ Retrieves all employees working in the Sales department.

- **Projection:**
  - It is used to retrieve **specific columns** from a table.
  - Implemented by specifying column names in the **SELECT** clause.

    Example:

    SELECT name, salary FROM employees;

→ Displays only the name and salary columns of all employees.

Summary:

- **Selection → filters rows (using WHERE)**
- **Projection → filters columns (using SELECT)**

2) Explain Aggregate functions with each one example. 5 M

Ans:

Aggregate functions are used to perform **calculations on a group of rows** and return a **single value**. They are often used with the GROUP BY clause.

Aggregate Functions:

1. **COUNT()**
    - Returns the total number of rows (or non-NULL values) in a column.

      Example:

      SELECT COUNT(*) AS total_employees FROM employees;

→ Counts the total number of employees.

2. **SUM()**
    - Calculates the total sum of a numeric column.

      Example:

      SELECT SUM(salary) AS total_salary FROM employees;

→ Returns the sum of all employee salaries.

3. **AVG()**
    - Returns the average value of a numeric column.

      Example:

      SELECT AVG(salary) AS average_salary FROM employees;

→ Calculates the average employee salary.

4. **MAX()**
    - Returns the maximum value from a column.

      Example:

      SELECT MAX(salary) AS highest_salary FROM employees;

→ Finds the highest salary in the company.

5. **MIN()**
    - Returns the minimum value from a column.

      Example:

SELECT MIN(salary) AS lowest_salary FROM employees;

→ Finds the lowest salary among employees.

Aggregate functions perform **calculations on sets of data** and return a **single summarized result** — useful for reports and data analysis.

3) Describe the implementation of key and integrity constraints 5 M

Ans:

Key Constraints:

Key constraints are the rules that uniquely identify records (rows) in a table. Key constraints are used to ensure that there are **no duplicate or missing values.**

Types of Key Constraint are:

- **Primary Key**
  - Uniquely identifies each record in a table.
  - Cannot have NULL values.

```
SQL> CREATE TABLE STUDENT (ROLLNO INT PRIMARY KEY,SNAME VARCHAR(30),AGE INT);
```

Example: RollNo in a Students table.

- **Unique Key**
  - Ensures all values in a column are unique.
  - Unlike primary key, it can have **one NULL.**

```
SQL> CREATE TABLE EMPLOYEE(EID INT PRIMARY KEY,ENAME VARCHAR(50), Email VARCHAR(100) UNIQUE);
```

Example: Email must be unique.

NULL is allowed only once

- **Foreign Key**
  - Links a column in one table to the **primary key** in another table.
  - Maintains **referential integrity** between two tables.

```
SQL>
SQL> CREATE TABLE COURSE(COURSEID INT PRIMARY KEY,COURSENAME VARCHAR(100));
```

```
SQL> CREATE TABLE ENROLLMENT(EnrollmentID INT PRIMARY KEY,ROLLNO INT,
     COURSEID INT,FOREIGN KEY (COURSEID) REFERENCES COURSE(COURSEID));
```

Example: CourseID in **Enrollment** table referencing CourseID in **Course** table.

Integrity Constraints:

Integrity constraints are the rules that ensure the **accuracy and consistency** of data in a database.

Types of Integrity Constraints are:

1. **Entity Integrity**
    - Every table must have a **primary key**, and it cannot be NULL.
    - Ensures each row is uniquely identifiable.

```
SQL> CREATE TABLE COLLEGE(CODE CHAR(10) PRIMARY KEY,CNAME VARCHAR(30),ADDRESS VARCHAR(100));
```

2. **Referential Integrity**
    - A foreign key must match a valid primary key in another table.
    - Prevents "orphan records" (e.g., student enrolled in a course that doesn't exist). An **orphan record** is a record in a child table that **does not have a matching parent record** in the parent table.

```
SQL>
SQL> CREATE TABLE COURSE(COURSEID INT PRIMARY KEY,COURSENAME VARCHAR(100));
```

```
SQL> CREATE TABLE ENROLLMENT(EnrollmentID INT PRIMARY KEY,ROLLNO INT,
     COURSEID INT,FOREIGN KEY (COURSEID) REFERENCES COURSE(COURSEID));
```

Ensures StudentID and CourseID in Enrollment exist in their respective tables.

3. **Domain Integrity**
    - Ensures values in a column are from a valid domain (data type, format, or range).

```
SQL> CREATE TABLE ST(ROLLNO INT PRIMARY KEY,NAME VARCHAR(50) NOT NULL,
AGE INT,CONSTRAINT CHK_RANGE CHECK (AGE BETWEEN 0 AND 100));

Table created.

SQL> INSERT INTO ST VALUES(101,'SAM',45);

1 row created.

SQL> INSERT INTO ST VALUES(101,'SAM',105);
INSERT INTO ST VALUES(101,'SAM',105)
*
ERROR at line 1:
ORA-02290: check constraint (SYSTEM.CHK_RANGE) violated
```

    Example: Age column values must be in between 0 and 100.

4. **User-defined Integrity**
    - User-defined integrity means additional rules defined by the user.

      Example: Mobile number must have 10 digits and should not non-numeric characters(alphabets).

```
SQL> CREATE TABLE USERS(USERID INT PRIMARY KEY,UNAME VARCHAR(50),MOBILE VARCHAR(10),
  2  CONSTRAINT CHK_MOB_NUM CHECK (LENGTH(MOBILE) = 10 AND MOBILE NOT LIKE '%[^0-9]%'));

Table created.

SQL> INSERT INTO USERS VALUES(101,'SAM','9876543210');

1 row created.
```

```
SQL> INSERT INTO USERS VALUES(102,'RAM','987654321');
INSERT INTO USERS VALUES(102,'RAM','987654321')
*
ERROR at line 1:
ORA-02290: check constraint (SYSTEM.CHK_MOB_NUM) violated
```

4) Discuss GROUPBY and HAVING clauses with an example. Also give the constraints related to their usage.

5 M

Ans:

1. GROUP BY Clause

- The GROUP BY clause is used to **group rows** that have the **same values** in one or more columns.
- It is generally used with **aggregate functions** like SUM(), AVG(), COUNT(), MAX(), or MIN() to perform calculations on each group of data.

Example:

```
SELECT department, COUNT(employee_id) AS total_employees
FROM Employees
GROUP BY department;
```

This query groups employees by their department and counts the number of employees in each department.

---

2. HAVING Clause

- The HAVING clause is used to **filter the groups** created by GROUP BY based on a condition.
- It works **like a WHERE clause**, but WHERE cannot be used with aggregate functions — hence HAVING is used.

Example:

```
SELECT department, AVG(salary) AS avg_salary
FROM Employees
GROUP BY department
HAVING AVG(salary) > 50000;
```

This query first groups employees by department, calculates the average salary for each, and then returns only those departments where the average salary is greater than ₹50,000.

Constraints / Rules Related to Usage

1. All **non-aggregated columns** in the SELECT list **must appear in the GROUP BY clause**.
   Correct:

```
SELECT dept, SUM(salary)
FROM Employees
GROUP BY dept;
```

Incorrect:

```
SELECT dept, name, SUM(salary)
FROM Employees
GROUP BY dept;
```

'name' is not in GROUP BY or an aggregate function

2. The **HAVING** clause is applied **after grouping**, while the **WHERE** clause is applied **before grouping**.
3. **ORDER BY** can be used **after HAVING** to sort the grouped results.

Hence:

- GROUP BY → groups rows.
- HAVING → filters groups (used with aggregate functions).

5) What is a View? How to create a View? Discuss in detail about Updatable & Non-Updatable Views. 5 M

Ans:

A **view** is a **virtual table** based on the result of an SQL query.

It does not store data itself but fetches data from one or more underlying tables.

Unlike regular tables, views do not store data themselves. Instead, they dynamically generate data by executing the SQL query defined in the view each time it is accessed.

Views provide security, simplicity, consistency, and abstraction in SQL.

A view is a saved SELECT query treated like a virtual table. You can SELECT from it. Some views also allow INSERT, UPDATE, DELETE – those are updatable views.

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 columns

FROM table_name

WHERE condition;
```

e.g. Lets consider the following students table:

```
SQL> SELECT * FROM STUDENTS;

ROL NAME                         DEPT           PERC
--- ---------------------------- ---------- ----------
101 AMIT                         CSE            85.1
121 NEHA                         IT             59.7
102 RAJ                          CSE            68.5
403 KIRAN                        ECE              60
301 SWAMY                        EEE            69.4
124 PRASAD                       IT             55.4
110 SAM                          CSE            80.4
103 RENU                         CSE              78
304 RAJU                         EEE            84.6
125 MURTHY                       IT             76.9
404 SRINIVAS                     ECE            72.6

ROL NAME                         DEPT           PERC
--- ---------------------------- ---------- ----------
104 DAVID                        CSE
401 ALICE                        ECE

13 rows selected.
```

From the above table, create a view CS_STUDENTS:

```
SQL> CREATE VIEW CS_STUDENTS AS SELECT ROLL,NAME, PERC
  2  FROM STUDENTS WHERE DEPT='CSE';

View created.
```

```
SQL> SELECT * FROM CS_STUDENTS;

ROL NAME                                PERC
--- ---------------------------- ----------
101 AMIT                              85.1
102 RAJ                               68.5
110 SAM                               80.4
103 RENU                                78
104 DAVID
```

1. Updatable View

A view is **updatable** if we can use INSERT, UPDATE, or DELETE operations on it, and changes are reflected in the underlying base tables.

- **Conditions for an updatable view:**
  - View must be created from **a single table.**
  - Must not contain **aggregates** (SUM, AVG, MAX, etc.).
  - Must not use **DISTINCT.**
  - Must not contain **GROUP BY / HAVING.**
  - Must not use **set operations** (UNION, INTERSECT, etc.).
  - Must not contain **subqueries in the SELECT list.**

2. Non-Updatable View

A view is **non-updatable** when modifications are **not allowed**, because the system cannot map the changes clearly to the base tables.

- **Non-updatable cases:**
  - View is based on **multiple tables (JOIN).**
  - Contains **aggregates** (SUM, AVG, etc.).
  - Uses **DISTINCT.**
  - Has **GROUP BY / HAVING.**
  - Uses **set operations.**

e.g. create a view DEPT_AVERAGE which contains department average percentage:

```
SQL> CREATE VIEW DEPT_AVERAGE AS SELECT DEPT, AVG(PERC) AS AVERAGE FROM STUDENTS GROUP BY DEPT;

View created.
```

display view:

```
SQL> SELECT * FROM DEPT_AVERAGE;

DEPT          AVERAGE
---------- ----------
IT                 64
EEE                77
CSE                75
ECE              66.3
```

Trying to update through view gives error:

```
SQL> UPDATE DEPT_AVERAGE SET AVERAGE=74 WHERE DEPT='IT';
UPDATE DEPT_AVERAGE SET AVERAGE=74 WHERE DEPT='IT'
       *
ERROR at line 1:
ORA-01732: data manipulation operation not legal on this view
```

6) What are the Relational SET operations in SQL. Exemplify using queries. 5 M

Ans:

Relational set operators in SQL combine the results of two or more SELECT statements into a single result set.

The relational set operators are based on the set theory operations from relational algebra.

They require the SELECT statements to be "union-compatible,"

Set Operations:

1) UNION
2) INTERSECT
3) EXCEPT (MINUS)

Set Operations automatically eliminates duplicates.

To retain all duplicates, use UNION ALL, INTERSECT ALL & EXCEPT/MINUS ALL.

Rules for Set Operations

- Both SELECT queries must have **same number of columns.**
- Data types of corresponding columns must be **compatible.**
- ORDER BY can only be used **once at the end** of the entire set operation.

Example:

Create a table with name CSE_STUDENTS:

```
SQL> CREATE TABLE CSE_STUDENTS (ROLL CHAR(3) PRIMARY KEY, NAME VARCHAR(10), DEPT VARCHAR(5));

Table created.
```

Insert values into CSE_STUDENTS:

```
SQL> INSERT INTO CSE_STUDENTS VALUES('501','ALICE','CSE');

1 row created.

SQL> INSERT INTO CSE_STUDENTS VALUES('502','BOB','CSE');

1 row created.

SQL> INSERT INTO CSE_STUDENTS VALUES('503','CHARLIE','CSE');

1 row created.
```

Display table CSE_STUDENTS:

```
SQL> SELECT * FROM CSE_STUDENTS;

ROL NAME       DEPT
--- ---------- -----
501 ALICE      CSE
502 BOB        CSE
503 CHARLIE    CSE
```

Create another table ECE_STUDENTS:

```
SQL> CREATE TABLE ECE_STUDENTS (ROLL CHAR(3) PRIMARY KEY, NAME VARCHAR(10), DEPT VARCHAR(5));

Table created.
```

Insert values into ECE_STUDENTS:

```
SQL> INSERT INTO ECE_STUDENTS VALUES('401','DAVID','ECE');

1 row created.

SQL> INSERT INTO ECE_STUDENTS VALUES('402','EMMA','ECE');

1 row created.

SQL> INSERT INTO ECE_STUDENTS VALUES('403','BOB','ECE');

1 row created.
```

Display rows in ECE_STUDENTS:

```
SQL> SELECT * FROM ECE_STUDENTS;

ROL NAME       DEPT
--- ---------- -----
401 DAVID      ECE
402 EMMA       ECE
403 BOB        ECE
```

UNION:

Combines the result sets of two or more SELECT statements.

Removes duplicate rows from final result set.

```
SQL> (SELECT ROLL, NAME FROM CSE_STUDENTS) UNION (SELECT ROLL, NAME FROM ECE_STUDENTS);

ROL NAME
--- ----------
401 DAVID
402 EMMA
403 BOB
501 ALICE
502 BOB
503 CHARLIE

6 rows selected.
```

```
SQL> (SELECT NAME FROM CSE_STUDENTS) UNION (SELECT NAME FROM ECE_STUDENTS);

NAME
----------
ALICE
BOB
CHARLIE
DAVID
EMMA
```

UNION ALL:

- Same as UNION but **does not remove duplicates.**

```
SQL> (SELECT NAME FROM CSE_STUDENTS) UNION ALL (SELECT NAME FROM ECE_STUDENTS);

NAME
----------
ALICE
BOB
CHARLIE
DAVID
EMMA
BOB

6 rows selected.
```

INTERSECT

- Returns only the rows that are **common** to both queries.

```
SQL> (SELECT NAME FROM CSE_STUDENTS) INTERSECT (SELECT NAME FROM ECE_STUDENTS);

NAME
----------
BOB
```

EXCEPT / MINUS:

Returns rows from the **first query** that are **not in the second query.**

- **EXCEPT** is used in SQL Server/PostgreSQL.
- **MINUS** is used in Oracle.

```
SQL> (SELECT NAME FROM CSE_STUDENTS) MINUS (SELECT NAME FROM ECE_STUDENTS);

NAME
----------
ALICE
CHARLIE
```

This returns students who are **only in CSE** but not in ECE.

```
SQL> (SELECT NAME FROM ECE_STUDENTS) MINUS (SELECT NAME FROM CSE_STUDENTS);

NAME
----------
DAVID
EMMA
```

This returns students who are **only in ECE** but not in CSE.

7) How to create tables with relationship? Establish a relationship between two tables using primary key and foreign key. 5 M

Ans:

Two or more tables can be related by using Foreign Key.

When we **create tables with relationships** in SQL, we usually use **primary keys** and **foreign keys** to link them together.

Steps to Create Tables with Relationships:

First: Always define **PRIMARY KEY** in the parent table.

Then: Use **FOREIGN KEY** in the child table to create the relationship.

Primary Key

- Uniquely identifies each row in a table.
- Example: RollNo in the **Student** table.

Foreign Key

- Column(s) in one table that refers to the **Primary Key** of another table.
- Ensures **referential integrity**.
- Syntax:

```
FOREIGN KEY (column_name) REFERENCES parent_table(primary_key);
```

1. Create the Parent Table (with Primary Key)

```
SQL> CREATE TABLE DEPT(DEPTID INT PRIMARY KEY, DEPTNAME VARCHAR(30) NOT NULL);

Table created.
```

SCHEMA OF DEPT TABLE IS:

```
SQL> DESC DEPT;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 DEPTID                                    NOT NULL NUMBER(38)
 DEPTNAME                                  NOT NULL VARCHAR2(30)
```

2. Create the Child Table (with Foreign Key):

```
SQL> CREATE TABLE EMPLOYEE(EMPID INT PRIMARY KEY,ENAME VARCHAR(30) NOT NULL,SALARY DECIMAL(10,2),DEPTID INT,
  2  FOREIGN KEY (DEPTID) REFERENCES DEPT(DEPTID));

Table created.

SQL> DESC EMPLOYEE;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 EMPID                                     NOT NULL NUMBER(38)
 ENAME                                     NOT NULL VARCHAR2(30)
 SALARY                                             NUMBER(10,2)
 DEPTID                                             NUMBER(38)
```

Here, DEPTID in EMPLOYEE references DEPTID in DEPT

This enforces referential integrity (you can't assign an employee to a department that doesn't exist)

Here, the relationship between Department and Employee tables is One-Many Relationship

- **Department (One) → Employee (Many)**
  One department can have many employees, but each employee belongs to exactly one department

Inserting values in the DEPT Table(Parent Table):

```
SQL> INSERT INTO DEPT VALUES (1, 'HR');

1 row created.

SQL> INSERT INTO DEPT VALUES (2, 'SALES');

1 row created.

SQL> INSERT INTO DEPT VALUES (3, 'MARKETING');

1 row created.
```

Inserting values into Employee Table (Child Table):

```
SQL> INSERT INTO Employee VALUES (101, 'Alice', 50000, 1);

1 row created.

SQL> INSERT INTO Employee VALUES (102, 'Bob', 60000, 2);

1 row created.

SQL> INSERT INTO Employee VALUES (103, 'Charlie', 45000, 3);

1 row created.

SQL> INSERT INTO Employee VALUES (104, 'David', 35000, 4);
INSERT INTO Employee VALUES (104, 'David', 35000, 4)
*
ERROR at line 1:
ORA-02291: integrity constraint (SYSTEM.SYS_C004157) violated - parent key not
found
```

Joining Two Tables:

Example Join Query:

Show employee names along with their department names.

```
SQL> SELECT E.ENAME, D.DEPTNAME
  2  FROM EMPLOYEE E
  3  JOIN DEPT D ON E.DEPTID = D.DEPTID;

ENAME                          DEPTNAME
------------------------------ ------------------------------
Alice                          HR
Bob                            SALES
Charlie                        MARKETING
```

Since, the two tables are related using referential integrity, We cannot delete any Department (from Parent Table) that has an entry in the Employee Table (Child Table).

e.g.

```
SQL>
SQL> DELETE FROM DEPT WHERE DEPTNAME='HR';
DELETE FROM DEPT WHERE DEPTNAME='HR'
*
ERROR at line 1:
ORA-02292: integrity constraint (SYSTEM.SYS_C004157) violated - child record
found
```

Trying to delete HR department will throw an error as above.

8) How would you use the operators IN, EXISTS, UNIQUE, ANY and ALL in writing nested queries? Why are they useful? Explain with an example. 5 M

Ans:

1. IN Operator

Use: Checks if a value is **present in a list or in the result of a subquery.**

Example:

```
SELECT name
FROM Student
WHERE student_id IN (
   SELECT student_id
   FROM Marks
   WHERE subject = 'Maths');
```

Meaning:
Select students who have marks in **Maths**.

Why useful:
→ Simplifies checking if a value exists in a set of results.

2. EXISTS Operator

Use: Returns TRUE if the **subquery returns any row**.

Example:

```
SELECT name
FROM Student S
WHERE EXISTS (
   SELECT *
   FROM Attendance A
   WHERE A.student_id = S.student_id  AND A.status = 'Absent');
```

Meaning:
Select students who were **absent at least once**.

Why useful:
→ Good for checking the **existence** of related records.

3. UNIQUE Operator

Use: Tests whether all rows returned by a subquery are **unique** (no duplicates).

Example:

```
SELECT course_id
FROM Course C
WHERE UNIQUE (
   SELECT student_id
   FROM Enrollment E
   WHERE E.course_id = C.course_id);
```

Meaning:
Select courses where each student is enrolled **only once**.

Why useful:
→ Ensures there are **no duplicate** rows in the result.

4. ANY Operator

Use: Compares a value with **any value** in the subquery result.

Example:

```
SELECT name
FROM Employee
WHERE salary > ANY (
```

```
  SELECT salary
  FROM Employee
  WHERE department = 'HR');
```

Meaning:
Select employees whose salary is **greater than at least one** HR employee.

Why useful:
→ Used for **partial comparison** (greater than *some* values).

5. ALL Operator

Use: Compares a value with **all values** returned by a subquery.

Example:

```
SELECT name
FROM Employee
WHERE salary > ALL (
  SELECT salary
  FROM Employee
  WHERE department = 'Sales');
```

Meaning:
Select employees whose salary is **greater than every** Sales department employee.

Why useful:
→ Used for **universal comparison** (greater than *all* values).

9) Explain Arithmetic and Logical Operators with examples 5 M

Ans:

1. Arithmetic Operators

➤ **Definition:**

Arithmetic operators are used to perform **mathematical operations** on numeric data in SQL or programming.

➤ **Common Arithmetic Operators:**

Operator	Description	Example	Result
+	Addition	10 + 5	15
-	Subtraction	10 - 3	7
*	Multiplication	4 * 5	20

Operator	Description	Example	Result
/	Division	20 / 4	5
%	Modulus (remainder)	10 % 3	1

Example :

```
SELECT name, salary, salary + 500 AS New_Salary
FROM Employee;
```

Adds 500 to each employee's salary and displays it as New_Salary.

2. Logical Operators

➤ Definition:

Logical operators are used to **combine or compare** multiple conditions in a query.
They return **TRUE or FALSE** values.

Logical Operators:

Operator	Description	Example	Result
AND	Returns TRUE if **both** conditions are true	age > 18 AND city = 'Delhi'	TRUE if both match
OR	Returns TRUE if **any one** condition is true	city = 'Delhi' OR city = 'Mumbai'	TRUE if either matches
NOT	Reverses the logical result	NOT city = 'Delhi'	TRUE if city is not Delhi

Example :

```
SELECT name, age, city
FROM Student
WHERE age > 18 AND city = 'Hyderabad';
```

Selects students who are **older than 18** and **live in Hyderabad**.

Summary Table

Type	Operators	Used For	Example
Arithmetic	+, -, *, /, %	Mathematical operations	salary + 1000
Logical	AND, OR, NOT	Combine multiple conditions	age > 18 AND marks > 50

10) Why we need joins? Explain different types of joins with examples. 5 M

Ans:

In a relational database, data is usually stored in **multiple related tables** (to avoid duplication).
To **retrieve related data** from these tables **together**, we use **JOINs**.

JOIN is used to combine rows from two or more tables based on a related column.

Join Operation takes two relations as input and returns another relation.

A Join operation is primarily a cartesian product which requires that tuples in the two relations match(under some condition).

After joining two tables, it can also specify what are the attributes that are present in the resultant table

Join operations are typically used as sub-query in the ***"from clause"***.

There are different types of joins in SQL.

- Cross join
- Inner join
  - Equi-join
    - Natural join
- Outer join
  - Left outer join
  - Right outer join
  - Full outer join
- Self-join

Let us assume two relations "course table" and "prereq table"

Here, Course table contains four attributes course-id , course- title, department-name and credits for the course.

```
SQL> CREATE TABLE COURSE ( COURSEID VARCHAR(7), TITLE VARCHAR(12), DEPTNAME VARCHAR(8), CREDITS INT );

Table created.
```

“prereq” means pre-requisites: “specifies the pre-requisites for a course”.

“prereq” table contains two attributes course-id and prereq-id.

```
SQL> DESC PREREQ;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 COURSEID                                           VARCHAR2(7)
 PREREQID                                           VARCHAR2(7)
```

Insert values into course table:

```
SQL> INSERT INTO COURSE VALUES('BIO-301','GENETICS','BIOLOGY',4);

1 row created.

SQL> INSERT INTO COURSE VALUES('CS-190','GAME DESIGN','CSE',4);

1 row created.

SQL> INSERT INTO COURSE VALUES('CS-315','ROBOTICS','CSE',3);

1 row created.
```

After Inserting values into “course” table:

```
SQL> SELECT * FROM COURSE;

COURSEI TITLE        DEPTNAME    CREDITS
------- ------------ -------- ----------
BIO-301 GENETICS     BIOLOGY           4
CS-190  GAME DESIGN  CSE               4
CS-315  ROBOTICS     CSE               3
```

After Inserting values into “prereq” table:

```
SQL> SELECT * FROM PREREQ;

COURSEI PREREQI
------- -------
BIO-301 BIO-101
CS-190  CS-101
CS-347  CS-101
```

Observe that

prereq information is missing for CS-315 and

course information is missing for CS-347

1) CROSS JOIN

- Returns the Cartesian product of the given two tables.
- Every row from the first table is combined with every row from the second table.
- No join condition is required.
- Keyword is “CROSS JOIN”

```
SQL> SELECT * FROM COURSE CROSS JOIN PREREQ;

COURSEI TITLE        DEPTNAME    CREDITS COURSEI PREREQI
------- ------------ -------- ---------- ------- -------
BIO-301 GENETICS     BIOLOGY           4 BIO-301 BIO-101
BIO-301 GENETICS     BIOLOGY           4 CS-190  CS-101
BIO-301 GENETICS     BIOLOGY           4 CS-347  CS-101
CS-190  GAME DESIGN  CSE               4 BIO-301 BIO-101
CS-190  GAME DESIGN  CSE               4 CS-190  CS-101
CS-190  GAME DESIGN  CSE               4 CS-347  CS-101
CS-315  ROBOTICS     CSE               3 BIO-301 BIO-101
CS-315  ROBOTICS     CSE               3 CS-190  CS-101
CS-315  ROBOTICS     CSE               3 CS-347  CS-101

9 rows selected.
```

We can also do implicit cross join as below:

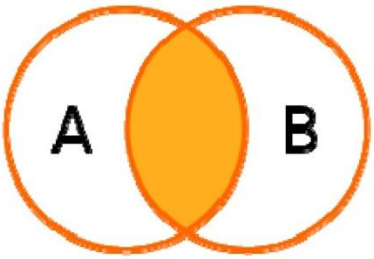
```
SQL> SELECT * FROM COURSE, PREREQ;

COURSEI TITLE        DEPTNAME    CREDITS COURSEI PREREQI
------- ------------ -------- ---------- ------- -------
BIO-301 GENETICS     BIOLOGY           4 BIO-301 BIO-101
BIO-301 GENETICS     BIOLOGY           4 CS-190  CS-101
BIO-301 GENETICS     BIOLOGY           4 CS-347  CS-101
CS-190  GAME DESIGN  CSE               4 BIO-301 BIO-101
CS-190  GAME DESIGN  CSE               4 CS-190  CS-101
CS-190  GAME DESIGN  CSE               4 CS-347  CS-101
CS-315  ROBOTICS     CSE               3 BIO-301 BIO-101
CS-315  ROBOTICS     CSE               3 CS-190  CS-101
CS-315  ROBOTICS     CSE               3 CS-347  CS-101

9 rows selected.
```

2) INNER JOIN (or simply JOIN):

- Returns only the rows where there is a match in both tables based on the join condition.
- Rows that do not have a match in both tables are excluded from the result set.



INPUT:

```
SQL> SELECT * FROM COURSE;

COURSEI TITLE        DEPTNAME    CREDITS
------- ------------ -------- ----------
BIO-301 GENETICS     BIOLOGY           4
CS-190  GAME DESIGN  CSE               4
CS-315  ROBOTICS     CSE               3
```

```
SQL> SELECT * FROM PREREQ;

COURSEI PREREQI
------- -------
BIO-301 BIO-101
CS-190  CS-101
CS-347  CS-101
```

OUTPUT:

```
SQL> SELECT * FROM COURSE C INNER JOIN PREREQ P ON C.COURSEID=P.COURSEID;

COURSEI TITLE        DEPTNAME    CREDITS COURSEI PREREQI
------- ------------ -------- ---------- ------- -------
BIO-301 GENETICS     BIOLOGY           4 BIO-301 BIO-101
CS-190  GAME DESIGN  CSE               4 CS-190  CS-101
```

3) EQUI JOIN: An Equi Join is a type of join that combines rows from two or more tables based on the equality of values in specified columns.

Equi Join is a type of INNER Join that displays the output by performing a join operation between two or more tables based on the common column between them.

It uses the equality (=) symbol to compare the data between two columns.

```
SQL> SELECT * FROM COURSE JOIN  PREREQ ON COURSE.COURSEID=PREREQ.COURSEID;

COURSEI TITLE        DEPTNAME    CREDITS COURSEI PREREQI
------- ------------ -------- ---------- ------- -------
BIO-301 GENETICS     BIOLOGY           4 BIO-301 BIO-101
CS-190  GAME DESIGN  CSE               4 CS-190  CS-101
```

```
SQL> SELECT * FROM COURSE,PREREQ WHERE COURSE.COURSEID=PREREQ.COURSEID;

COURSEI TITLE        DEPTNAME    CREDITS COURSEI PREREQI
------- ------------ -------- ---------- ------- -------
BIO-301 GENETICS     BIOLOGY           4 BIO-301 BIO-101
CS-190  GAME DESIGN  CSE               4 CS-190  CS-101
```

4) **NATURAL JOIN**

- Automatically joins tables based on columns with the **same name and compatible data types.**
- No need of explicit “ON clause”.

```
SQL> SELECT * FROM COURSE NATURAL JOIN PREREQ;

COURSEI TITLE        DEPTNAME    CREDITS PREREQI
------- ------------ -------- ---------- -------
BIO-301 GENETICS     BIOLOGY           4 BIO-101
CS-190  GAME DESIGN  CSE               4 CS-101
```

```
SQL> SELECT * FROM COURSE  NATURAL INNER JOIN PREREQ ;

COURSEI TITLE        DEPTNAME    CREDITS PREREQI
------- ------------ -------- ---------- -------
BIO-301 GENETICS     BIOLOGY           4 BIO-101
CS-190  GAME DESIGN  CSE               4 CS-101
```

OUTER JOIN

In Inner join there may be a loss of information.

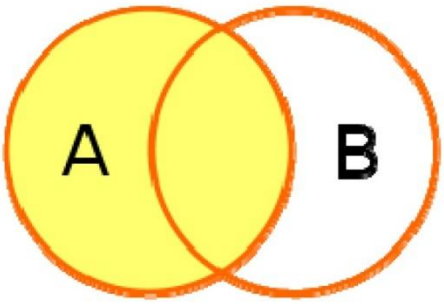
Outer join is an extension of Join that avoids loss of information.

Outer join computes the join and then add tuples from one relation that does not match tuples in the other relation to the result.

Outer join uses NULL values.

5) **LEFT OUTER JOIN (LEFT JOIN)**

- Returns all rows from the **left table** and matching rows from the right table.
- If no match, NULL values are returned for the right table’s columns.



INPUT:

```
SQL> SELECT * FROM COURSE;

COURSEI TITLE        DEPTNAME    CREDITS
------- ------------ -------- ----------
BIO-301 GENETICS     BIOLOGY           4
CS-190  GAME DESIGN  CSE               4
CS-315  ROBOTICS     CSE               3
```

```
SQL> SELECT * FROM PREREQ;

COURSEI PREREQI
------- -------
BIO-301 BIO-101
CS-190  CS-101
CS-347  CS-101
```

OUTPUT:

```
SQL> SELECT * FROM COURSE NATURAL LEFT OUTER JOIN PREREQ;

COURSEI TITLE        DEPTNAME    CREDITS PREREQI
------- ------------ -------- ---------- -------
BIO-301 GENETICS     BIOLOGY           4 BIO-101
CS-190  GAME DESIGN  CSE               4 CS-101
CS-315  ROBOTICS     CSE               3
```

6) **RIGHT OUTER JOIN (RIGHT JOIN)**

- Returns all rows from the **right table** and matching rows from the left table.
- If no match, NULL values are returned for the left table's columns.

INPUT:

```
SQL> SELECT * FROM COURSE;

COURSEI TITLE        DEPTNAME    CREDITS
------- ------------ -------- ----------
BIO-301 GENETICS     BIOLOGY           4
CS-190  GAME DESIGN  CSE               4
CS-315  ROBOTICS     CSE               3
```

```
SQL> SELECT * FROM PREREQ;

COURSEI PREREQI
------- -------
BIO-301 BIO-101
CS-190  CS-101
CS-347  CS-101
```

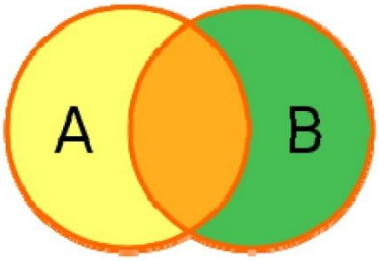
OUTPUT:

```
SQL> SELECT * FROM COURSE NATURAL RIGHT OUTER JOIN PREREQ;

COURSEI TITLE        DEPTNAME    CREDITS PREREQI
------- ------------ -------- ---------- -------
BIO-301 GENETICS     BIOLOGY           4 BIO-101
CS-190  GAME DESIGN  CSE               4 CS-101
CS-347                                   CS-101
```

7) FULL JOIN (FULL OUTER JOIN):

- **Returns all rows when there is a match in either left or right table.**
- **Non-matching rows from both tables will contain NULL.**



INPUT:

```
SQL> SELECT * FROM COURSE;

COURSEI TITLE        DEPTNAME    CREDITS
------- ------------ -------- ----------
BIO-301 GENETICS     BIOLOGY           4
CS-190  GAME DESIGN  CSE               4
CS-315  ROBOTICS     CSE               3
```

```
SQL> SELECT * FROM PREREQ;

COURSEI PREREQI
------- -------
BIO-301 BIO-101
CS-190  CS-101
CS-347  CS-101
```

OUTPUT:

Returns all rows from both **COURSE** and **PREREQ**, even if no match.

```
SQL> SELECT * FROM COURSE NATURAL FULL OUTER JOIN PREREQ;

COURSEI TITLE        DEPTNAME    CREDITS PREREQI
------- ------------ -------- ---------- -------
BIO-301 GENETICS     BIOLOGY           4 BIO-101
CS-190  GAME DESIGN  CSE               4 CS-101
CS-315  ROBOTICS     CSE               3
CS-347                                   CS-101
```

8) **SELF JOIN**

- A table joined with itself.
- Useful for hierarchical or recursive data.

```
SQL> SELECT * FROM PREREQ P1 JOIN PREREQ P2 ON P1.PREREQID=P2.COURSEID;

no rows selected
```

Join Type	returns
CROSS JOIN	Cartesian product
INNER JOIN	Matching rows only
NATURAL JOIN	Auto-join on same-named columns
LEFT JOIN	All left rows + matching right

Join Type	returns
RIGHT JOIN	All right rows + matching left
FULL JOIN	All rows from both tables
SELF JOIN	Table joined with itself

11) Write in detail about SQL functions(Date and Time, Numeric, String conversion). 10 M

Ans:

SQL Functions

SQL functions are similar to SQL operators in that both manipulate data items and both return a result.

Aggregate functions are the set functions: AVG, MIN, MAX, SUM, and COUNT. You must provide them with an alias that can be used by the GROUP BY function.

Types of Functions:

1) Number Functions (Numeric)

Number functions accept numeric input and return numeric values.

2) Character Functions

Single row character functions accept character input and can return both character and number values.

3) Character Functions Returning Number Values

Some character functions return only number values.

4) Date Functions

Date functions operate on values of the DATE datatype. All date functions return a value of the DATE datatype, except the MONTHS_BETWEEN function which returns a number.

5) Conversion Functions

Conversion functions convert a value from one datatype to another. Generally, the form of the function name follows the convention *datatype* TO *datatype*. The first datatype is the input

DATE AND TIME FUNCTIONS:

SQL provides a rich set of functions for manipulating and formatting date and time values. Here are some commonly used date functions with examples:

1. Getting Current Date and Time:

- **SYSDATE:** Returns the current date and time of the database server.

```
SQL> SELECT SYSDATE FROM DUAL;

SYSDATE
---------
14-SEP-25
```

2. SYSTIMESTAMP: Returns the system date and time, including fractional seconds and time zone.

```
SQL> SELECT SYSTIMESTAMP FROM DUAL;

SYSTIMESTAMP
---------------------------------------------------------------------------
14-SEP-25 01.54.01.194000 PM +05:30
```

3.Extracting Date Components:

- **EXTRACT(component FROM date):** Extracts a specific component (e.g., YEAR, MONTH, DAY, HOUR, MINUTE, SECOND) from a date or timestamp.

```
SQL> SELECT SYSTIMESTAMP FROM DUAL;

SYSTIMESTAMP
---------------------------------------------------------------------------
14-SEP-25 01.58.24.157000 PM +05:30

SQL> SELECT EXTRACT(YEAR FROM SYSDATE) AS THIS_YEAR FROM DUAL;

 THIS_YEAR
----------
      2025
```

```
SQL> SELECT EXTRACT(MONTH FROM SYSDATE) AS THIS_MONTH FROM DUAL;

THIS_MONTH
----------
         9

SQL> SELECT EXTRACT(DAY FROM SYSDATE) AS THIS_DAY FROM DUAL;

  THIS_DAY
----------
        14
```

4. Adding/Subtracting Date Components:

- **ADD_MONTHS(date, n):** Adds or subtracts n months from a date.

```
SQL>
SQL> SELECT ADD_MONTHS(SYSDATE,3) AS PLUS3 FROM DUAL;

PLUS3
---------
14-DEC-25

SQL> SELECT ADD_MONTHS(SYSDATE,-3) AS MINUS3 FROM DUAL;

MINUS3
---------
14-JUN-25
```

- **Adding/Subtracting Days:** Directly add or subtract numbers from a DATE value.

```
SQL> SELECT SYSDATE + 7 AS PLUS7 FROM DUAL;

PLUS7
---------
21-SEP-25

SQL> SELECT SYSDATE - 5 AS MINUS5 FROM DUAL;

MINUS5
---------
09-SEP-25
```

5. Formatting Dates:

- **TO_CHAR(date, format_model):** Converts a date to a character string using a specified format model.

```
SQL> SELECT TO_CHAR(SYSDATE, 'DD-MM-YYYY') AS NEW1 FROM DUAL;

NEW1
----------
14-09-2025

SQL> SELECT TO_CHAR(SYSDATE, 'YYYY/MM/DD') AS NEW2 FROM DUAL;

NEW2
----------
2025/09/14

SQL> SELECT TO_CHAR(SYSDATE, 'DD/MM/YYYY HH24:MI:SS') AS NEW3 FROM DUAL;

NEW3
-------------------
14/09/2025 14:14:33

SQL> SELECT TO_CHAR(SYSDATE, 'DAY DD/MONTH/YYYY HH12:MI:SS') AS NEW4 FROM DUAL;

NEW4
------------------------------------------------------------------------------
SUNDAY    14/SEPTEMBER/2025 02:14:42
```

6. Converting Strings to Dates:

- **TO_DATE(string, format_model):** Converts a character string to a date value using a specified format model. The format model must match the input string.

```
SQL> SELECT TO_DATE('15-SEP-2025','DD-MON-YYYY') AS NEW FROM DUAL;

NEW
---------
15-SEP-25
```

7. Other Useful Functions:

- **LAST_DAY(date):** Returns the last day of the month for a given date.

```
SQL> SELECT LAST_DAY(SYSDATE) AS LAST FROM DUAL;

LAST
---------
30-SEP-25
```

- **NEXT_DAY(date, day_of_week):** Returns the date of the next specified day of the week following the given date.

```
SQL> SELECT NEXT_DAY(SYSDATE,'MONDAY') AS NEXTMONDAY FROM DUAL;

NEXTMONDA
---------
15-SEP-25
```

- **ROUND(date, format_model):** Rounds a date to the nearest unit specified by the format model (e.g., 'YEAR', 'MONTH', 'DAY').

```
SQL> SELECT ROUND(SYSDATE,'YEAR') AS NEW FROM DUAL;

NEW
---------
01-JAN-26

SQL> SELECT ROUND(SYSDATE,'MONTH') AS NEW FROM DUAL;

NEW
---------
01-SEP-25
```

- **TRUNC(date, format_model):** Truncates a date to the unit specified by the format model.

```
SQL> SELECT TRUNC(SYSDATE,'MONTH') AS NEW FROM DUAL;

NEW
---------
01-SEP-25

SQL> SELECT TRUNC(SYSDATE,'YEAR') AS NEW FROM DUAL;

NEW
---------
01-JAN-25
```

SQL NUMERIC FUNCTIONS:

SQL Numeric Functions are built-in tools that allow you to perform mathematical and arithmetic operations on numeric data.

Used to:

- Operate on numeric data types like INT, FLOAT, DECIMAL, DOUBLE.
- Perform basic arithmetic operations (addition, subtraction, multiplication, division).
- Round numbers to desired precision.
- Format numeric values for better readability.
- Aggregate and analyze numeric data efficiently.

1. ABS() – Absolute Value

The ABS() function returns the absolute value of a number, which is the number without its sign (i.e., it converts negative numbers to positive).

```
SELECT ABS(number);
```

```
SQL> SELECT ABS(-15) FROM DUAL;

  ABS(-15)
----------
        15
```

2. CEIL() or CEILING() – Round Number Up

The CEIL() (or CEILING()) function rounds a number up to the nearest integer, regardless of whether the decimal part is greater than or less than 0.5.

```
SELECT CEIL(number);
```

```
SQL> SELECT CEIL(123.78) FROM DUAL;

CEIL(123.78)
------------
         124

SQL> SELECT CEIL(123.4) FROM DUAL;

CEIL(123.4)
-----------
        124
```

3. FLOOR() – Round Number Down

The FLOOR() function rounds a number down to the nearest integer, ignoring the decimal part.

```
SELECT FLOOR(number);
```

```
SQL> SELECT FLOOR(37.9) FROM DUAL;

FLOOR(37.9)
-----------
         37

SQL> SELECT FLOOR(123.1) FROM DUAL;

FLOOR(123.1)
------------
         123
```

4. ROUND() – Round a Number to a Specified Decimal Place

The ROUND() function rounds a number to a specified number of decimal places. It is very useful for financial calculations or whenever precise rounding is necessary.

Syntax:

```
SELECT ROUND(number, decimal_places);
```

```
SQL> SELECT ROUND(15.6789, 2) FROM DUAL;

ROUND(15.6789,2)
----------------
           15.68
```

5. TRUNCATE() – Remove Decimal Places

The TRUNCATE() function is used to remove the decimal portion of a number without rounding. It truncates the number to the specified number of decimal places.

Syntax:

```
SELECT TRUNC(number, decimal_places);
```

```
SQL> SELECT TRUNC(12.98765, 2) FROM DUAL;

TRUNC(12.98765,2)
-----------------
            12.98
```

6. MOD() – Modulo or Remainder

The MOD() function returns the remainder of a division operation (i.e., it computes the modulus). This function is useful for tasks like determining even/odd numbers or finding remainders in mathematical operations.

Syntax:

```
SELECT MOD(dividend, divisor);
```

```
SQL> SELECT MOD(10, 3) FROM DUAL;

 MOD(10,3)
----------
         1

SQL> SELECT MOD(1000, 3) FROM DUAL;

MOD(1000,3)
-----------
          1

SQL> SELECT MOD(10000, 3) FROM DUAL;

MOD(10000,3)
------------
           1
```

7. POWER() – Raise a Number to the Power of Another

The POWER() function is used to raise a number to the power of another number.

Syntax:

```
SELECT POWER(base, exponent);
```

```
SQL> SELECT POWER(2, 3) FROM DUAL;

POWER(2,3)
----------
         8

SQL> SELECT POWER(10, 5) FROM DUAL;

POWER(10,5)
-----------
     100000
```

8. SQRT() – Square Root

The SQRT() function returns the square root of a number. This is useful for mathematical calculations.

Syntax:

```
SELECT SQRT(number);
```

```
SQL> SELECT SQRT(16) FROM DUAL;

  SQRT(16)
----------
         4

SQL> SELECT SQRT(2) FROM DUAL;

   SQRT(2)
----------
1.41421356
```

9. EXP() – Exponential Function

The EXP() function returns the value of e raised to the power of a specified number, where e is the base of the natural logarithm (approximately 2.71828).

Syntax:

```
SELECT EXP(number);
```

```
SQL> SELECT EXP(1) FROM DUAL;

    EXP(1)
----------
2.71828183

SQL> SELECT EXP(10) FROM DUAL;

   EXP(10)
----------
22026.4658
```

10. LOG() – Logarithm

The LOG() function returns the natural logarithm (base e) of a number. You can also use LOG(base, number) to calculate the logarithm of a number with a custom base.

Syntax:

```
SELECT LOG(number);
SELECT LOG(base, number);
```

```
SQL> SELECT LOG(10,2) FROM DUAL;

 LOG(10,2)
----------
.301029996

SQL> SELECT LOG(10,1) FROM DUAL;

 LOG(10,1)
----------
         0
```

SQL STRING FUNCTIONS

SQL String Functions are powerful tools that allow us to manipulate, format, and extract specific parts of text data in our database. These functions are essential for tasks like cleaning up data, comparing strings, and combining text fields.

String functions are used to perform an operation on input string and return an output string. Below are some of the most commonly used SQL string functions:

1. LENGTH()

Returns the length of a string.

```
SQL> SELECT LENGTH('HELLO WORLD!') FROM DUAL;

LENGTH('HELLOWORLD!')
---------------------
                   12
```

2. UPPER()

Converts all characters in the string to uppercase.

```
SQL> SELECT UPPER('Hello World!') FROM DUAL;

UPPER('HELLO
------------
HELLO WORLD!
```

3. LOWER()

Converts all characters in the string to lowercase.

```
SQL> SELECT LOWER('HELLO WORLD!') FROM DUAL;

LOWER('HELLO
------------
hello world!
```

4. SUBSTRING() / SUBSTR()

Extracts a part of a string starting at a specified position.

```
SQL> SELECT SUBSTR('Hello World', 7, 5) FROM DUAL;

SUBST
-----
World

SQL> SELECT SUBSTR('Hello World', 5, 4) FROM DUAL;

SUBS
----
o Wo
```

5. CONCAT()

Concatenates two or more strings.

```
SQL> SELECT CONCAT('Swarnandhra','College') FROM DUAL;

CONCAT('SWARNANDHR
------------------
SwarnandhraCollege

SQL> SELECT CONCAT('Hello','World') FROM DUAL;

CONCAT('HE
----------
HelloWorld
```

6. TRIM()

Removes leading and/or trailing spaces (or specified characters) from a string.

```
SQL> SELECT TRIM('   Hello World   ') FROM DUAL;

TRIM('HELLO
-----------
Hello World
```

7. REPLACE()

Replaces all occurrences of a substring with another substring.

```
SQL> SELECT REPLACE('Hello World', 'World', 'SQL')FROM DUAL;

REPLACE('
---------
Hello SQL
```

8. INSTR()

Finds the position of a substring within a string.

```
SQL> SELECT INSTR('Hello World', 'World') FROM DUAL;

INSTR('HELLOWORLD','WORLD')
---------------------------
                          7
```

9. LPAD() / RPAD()

Pads the left or right side of a string with a specified character to a certain length.

```
SQL> SELECT LPAD('123', 5, '0') FROM DUAL;

LPAD(
-----
00123

SQL> SELECT RPAD('123', 5, '0') FROM DUAL;

RPAD(
-----
12300
```

11. ASCII()

Returns the ASCII code of the first character or converts an ASCII code to a character.

```
SQL> SELECT ASCII('A') FROM DUAL;

ASCII('A')
----------
        65
```

12.REVERSE()

The REVERSE() function reverses the characters in a string

```
SQL> SELECT REVERSE('SWARNANDHRA') FROM DUAL;

REVERSE('SW
-----------
ARHDNANRAWS
```

AGGEREGATE FUNCTIONS

SQL aggregate functions perform calculations on a set of rows and return a single summary value.

These functions are commonly used with the GROUP BY clause to perform calculations on groups of data.

the most common SQL aggregate functions with examples:

- **COUNT()**: Returns the number of rows that match a specified criterion.

```
SQL> SELECT * FROM STUDENTS;

ROL NAME                           DEPT           PERC
--- ------------------------------ ---------- ----------
101 AMIT                           CSE              85.1
121 NEHA                           IT               59.7
102 RAJ                            CSE              68.5
402 MEENA                          ECE                70
302 ROHIT                          EEE              65.2
105 ANKIT                          CSE                75
403 KIRAN                          ECE                60
123 KUMAR                          IT               67.9
301 SWAMY                          EEE              69.4
124 PRASAD                         IT               55.4
110 SAM                            CSE              80.4

ROL NAME                           DEPT           PERC
--- ------------------------------ ---------- ----------
303 KUMAR                          EEE              68.3
103 RENU                           CSE                78
304 RAJU                           EEE              84.6
125 MURTHY                         IT               76.9
404 SRINIVAS                       ECE              72.6

16 rows selected.
```

```
SQL> SELECT COUNT(*) FROM STUDENTS;

  COUNT(*)
----------
        16

SQL>  SELECT COUNT(DISTINCT DEPT) FROM STUDENTS;

COUNT(DISTINCTDEPT)
-------------------
                  4
```

SUM(): Calculates the sum of a numeric column.

```
SQL>  SELECT SUM(PERC) FROM STUDENTS;

 SUM(PERC)
----------
      1137
```

AVG(): Calculates the average value of a numeric column

```
SQL>  SELECT AVG(PERC) FROM STUDENTS;

 AVG(PERC)
----------
   71.0625
```

MIN(): Returns the minimum value in a column.

MAX(): Returns the maximum value in a column.

```
SQL> SELECT MIN(PERC) FROM STUDENTS;

 MIN(PERC)
----------
      55.4

SQL> SELECT MAX(PERC) FROM STUDENTS;

 MAX(PERC)
----------
      85.1
```

GROUP BY:

To calculate the average salary for each department:

```
SQL> SELECT DEPT, AVG(PERC) FROM STUDENTS GROUP BY DEPT;

DEPT       AVG(PERC)
---------- ----------
IT             64.975
EEE            71.875
CSE              77.4
ECE        67.5333333
```

*****