**7. Consider the following Database design**

**Customer (cid, custname, custstreet, custcity)**

**Account (accno, branchname, balance)**

**Loan (loanno, branchname, amount)**

**Borrower (cid, loanno)**

**Branch (branchname, branchcity, asset)**

**Depositor (cid, accno)**

CREATE TABLE Customer1 (

cid INT PRIMARY KEY,

custname VARCHAR(255),

custstreet VARCHAR(255),

custcity VARCHAR(255)

);

DROP TABLE Customer1;

CREATE TABLE Account1 (

accno INT PRIMARY KEY,

branchname VARCHAR(255),

balance DECIMAL(10, 2)

);

DROP TABLE Account1;

CREATE TABLE Loan1 (

loanno VARCHAR(10) PRIMARY KEY,

branchname VARCHAR(255),

amount DECIMAL(10, 2)

);

DROP TABLE Loan1;

CREATE TABLE Borrower1 (

cid INT,

loanno VARCHAR(10),

PRIMARY KEY (cid, loanno),

FOREIGN KEY (cid) REFERENCES Customer1(cid),

FOREIGN KEY (loanno) REFERENCES Loan1(loanno)

);

DROP TABLE Borrower1;

CREATE TABLE Branch1 (

branchname VARCHAR(255) PRIMARY KEY,

branchcity VARCHAR(255),

asset DECIMAL(15, 2)

);

CREATE TABLE Depositor1 (

cid INT,

accno INT,

PRIMARY KEY (cid, accno),

FOREIGN KEY (cid) REFERENCES Customer(cid),

FOREIGN KEY (accno) REFERENCES Account(accno)

);

INSERT INTO Customer1 (cid, custname, custstreet, custcity) VALUES

(1, 'John Doe', '123 Main St', 'New York');

INSERT INTO Customer1 (cid, custname, custstreet, custcity) VALUES

(2, 'Jane Smith', '456 Elm St', 'Los Angeles');

INSERT INTO Customer1 (cid, custname, custstreet, custcity) VALUES

(3, 'Bob Johnson', '789 Oak St', 'Chicago');

INSERT INTO Customer1 (cid, custname, custstreet, custcity) VALUES

(4, 'SHWETA K', '34 Oak St', 'CANADA');

SELECT \* FROM Customer1;

INSERT INTO Account1 (accno, branchname, balance) VALUES

(101, 'Main Branch', 5000.00);

INSERT INTO Account1 (accno, branchname, balance) VALUES

(102, 'Downtown Branch', 10000.00);

INSERT INTO Account1 (accno, branchname, balance) VALUES

(103, 'Uptown Branch', 7500.00);

INSERT INTO Loan1 (loanno, branchname, amount) VALUES

('L-101', 'Main Branch', 2000.00);

INSERT INTO Loan1 (loanno, branchname, amount) VALUES

('L-102', 'Downtown Branch', 3000.00);

INSERT INTO Loan1 (loanno, branchname, amount) VALUES

('L-103', 'Uptown Branch', 2500.00);

SELECT \* FROM Loan1;

INSERT INTO Borrower1 (cid, loanno) VALUES

(1, 'L-101');

INSERT INTO Borrower1 (cid, loanno) VALUES

(2, 'L-102');

INSERT INTO Borrower1 (cid, loanno) VALUES

(3, 'L-103');

INSERT INTO Branch (branchname, branchcity, asset) VALUES

('Main Branch', 'New York', 1000000.00);

INSERT INTO Branch (branchname, branchcity, asset) VALUES

('Downtown Branch', 'Los Angeles', 750000.00);

INSERT INTO Branch (branchname, branchcity, asset) VALUES

('Uptown Branch', 'Chicago', 500000.00);

INSERT INTO Depositor1 (cid, accno) VALUES

(1, 101);

INSERT INTO Depositor1 (cid, accno) VALUES

(2, 102);

INSERT INTO Depositor1 (cid, accno) VALUES

(3, 103);

**Solve the following queries in SQL**

**a. Display the name of customers who have both account and loan at the bank.**

SELECT DISTINCT c.custname

FROM Customer1 c

JOIN Depositor1 d ON c.cid = d.cid

JOIN Account1 a ON d.accno = a.accno

JOIN Borrower1 b ON c.cid = b.cid;

**b. Update amount of loan to 10000 where loan number is “L-101”.**

UPDATE Loan1

SET amount = 10000

WHERE loanno = 'L-101';

**c. Change the column name custcity to ccity.**

ALTER TABLE Customer1

RENAME COLUMN custcity TO ccity;

**d. Find all customers who an account but no loan at bank.**

SELECT c.\*

FROM Customer1 c

LEFT JOIN Borrower1 b ON c.cid = b.cid

WHERE b.cid IS NULL;

**4. List and explain aggregate functions of SQL with appropriate examples.**

## **SQL Aggregate Functions**

An aggregate function is a function that performs a calculation on a set of values, and returns a single value.

Aggregate functions are often used with the GROUP BY clause of the SELECT statement. The GROUP BY clause splits the result-set into groups of values and the aggregate function can be used to return a single value for each group.

Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five built-in aggregate functions:

* MIN() - returns the smallest value within the selected column
* MAX() - returns the largest value within the selected column
* COUNT() - returns the number of rows in a set
* SUM() - returns the total sum of a numerical column
* AVG() - returns the average value of a numerical column

Aggregate functions ignore null values (except for COUNT()).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ProductID** | **ProductName** | **SupplierID** | **CategoryID** | **Unit** | **Price** |
| 1 | Chais | 1 | 1 | 10 boxes x 20 bags | 18 |
| 2 | Chang | 1 | 1 | 24 - 12 oz bottles | 19 |
| 3 | Aniseed Syrup | 1 | 2 | 12 - 550 ml bottles | 10 |
| 4 | Chef Anton's Cajun Seasoning | 2 | 2 | 48 - 6 oz jars | 22 |
| 5 | Chef Anton's Gumbo Mix | 2 | 2 | 36 boxes | 21.35 |

**1. COUNT FUNCTION**

* COUNT function is used to Count the number of rows in a database table. It can work on both numeric and non-numeric data types.
* COUNT function uses the COUNT(\*) that returns the count of all the rows in a specified table. COUNT(\*) considers duplicate and Null.

## **Syntax**

SELECT COUNT(column\_name)  
FROM table\_name  
WHERE condition;

**Example:**

## **Specify Column**

You can specify a column name instead of the asterix symbol (\*).

If you specify a column name instead of (\*), NULL values will not be counted.

### Example

Find the number of products where the ProductName is not null:

SELECT COUNT(ProductName)  
FROM Products;

## **Add a WHERE Clause**

You can add a WHERE clause to specify conditions:

### Example

Find the number of products where Price is higher than 20:

SELECT COUNT(ProductID)  
FROM Products  
WHERE Price > 20;

## **Use an Alias**

Give the counted column a name by using the AS keyword.

### Example

Name the column "Number of records":

SELECT COUNT(\*) AS [Number of records]  
FROM Products;

## **Use COUNT() with GROUP BY**

Here we use the COUNT() function and the GROUP BY clause, to return the number of records for each category in the Products table:

### Example

SELECT COUNT(\*) AS [Number of records], CategoryID  
FROM Products  
GROUP BY CategoryID;

**2. SUM Function**

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

## **Syntax**

SELECT SUM(column\_name)  
FROM table\_name  
WHERE condition;

**Example:**

## **Add a WHERE Clause**

You can add a WHERE clause to specify conditions:

### Example

Return the sum of the Price field for the product with ProductID 5:

SELECT SUM(Price)  
FROM Products  
WHERE ProductId = 5;

## **Use an Alias**

Give the summarized column a name by using the AS keyword.

### Example

Name the column "total":

SELECT SUM(Price) AS total  
FROM Products;

## **Use SUM() with GROUP BY**

Here we use the SUM() function and the GROUP BY clause, to return the Price for each ProductId in the Products table:

### Example

SELECT ProductId, SUM(Price) AS [Total Price]  
FROM Products  
GROUP BY ProductId;

**3. AVG function**

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

## **Syntax**

SELECT AVG(column\_name)  
FROM table\_name  
WHERE condition;

**Example:**

## **Add a WHERE Clause**

You can add a WHERE clause to specify conditions:

### Example

Return the average price of products in category 1:

SELECT AVG(Price)  
FROM Products  
WHERE CategoryID = 1;

## **Use an Alias**

Give the AVG column a name by using the AS keyword.

### Example

Name the column "average price":

SELECT AVG(Price) AS [average price]  
FROM Products;

## **Use AVG() with GROUP BY**

Here we use the AVG() function and the GROUP BY clause, to return the average price for each category in the Products table:

### Example

SELECT AVG(Price) AS AveragePrice, CategoryID  
FROM Products  
GROUP BY CategoryID;

**4. MAX Function**

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

## **Syntax**

SELECT MAX(column\_name)  
FROM table\_name  
WHERE condition;

**Example:**

Find the highest price in the Price column:

SELECT MAX(Price)  
FROM Products;

## **Set Column Name (Alias)**

When you use MAX(), the returned column will not have a descriptive name. To give the column a descriptive name, use the AS keyword:

### Example

SELECT MAX(Price) AS LargestPrice  
FROM Products;

## **Use MAX() with GROUP BY**

Here we use the MAX() function and the GROUP BY clause, to return the smallest price for each category in the Products table:

### Example

SELECT MAX(Price) AS LargestPrice, CategoryID  
FROM Products  
GROUP BY CategoryID;

**5. MIN Function**

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

## **Syntax**

SELECT MIN(column\_name)  
FROM table\_name  
WHERE condition;

**Example:**

Find the lowest price in the Price column:

SELECT MIN(Price)  
FROM Products;

## **Set Column Name (Alias)**

When you use MIN(), the returned column will not have a descriptive name. To give the column a descriptive name, use the AS keyword:

### Example

SELECT MIN(Price) AS SmallestPrice  
FROM Products;

## **Use MIN() with GROUP BY**

Here we use the MIN() function and the GROUP BY clause, to return the smallest price for each category in the Products table:

### Example

SELECT MIN(Price) AS SmallestPrice, CategoryID  
FROM Products  
GROUP BY CategoryID;

**List and explain the types of Join in SQL.**

## **SQL JOIN**

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

SQL JOIN

SQL JOIN clause is used to query and access data from multiple tables by establishing logical relationships between them. It can access data from multiple tables simultaneously using common key values shared across different tables.

We can use SQL JOIN with multiple tables. It can also be paired with other clauses, the most popular use will be using JOIN with WHERE clause to filter data retrieval.

**SQL JOIN Example**

Consider the two tables below as follows:

**Student:**



**StudentCourse :**



Both these tables are connected by one common key (column) i.e ROLL\_NO.

We can perform a JOIN operation using the given SQL query:

***SELECT s.roll\_no, s.name, s.address, s.phone, s.age, sc.course\_id***

***FROM Student s***

***JOIN StudentCourse sc ON s.roll\_no = sc.roll\_no;***

Output:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ROLL\_NO | NAME | ADDRESS | PHONE | AGE | COURSE\_ID |
| 1 | HARSH | DELHI | XXXXXXXXXX | 18 | 1 |
| 2 | PRATIK | BIHAR | XXXXXXXXXX | 19 | 2 |
| 3 | RIYANKA | SILGURI | XXXXXXXXXX | 20 | 2 |
| 4 | DEEP | RAMNAGAR | XXXXXXXXXX | 18 | 3 |
| 5 | SAPTARHI | KOLKATA | XXXXXXXXXX | 19 | 1 |

## **Different Types of SQL JOINs**

Here are the different types of the JOINs in SQL:

* (INNER) JOIN: Returns records that have matching values in both tables
* LEFT (OUTER) JOIN: Returns all records from the left table, and the matched records from the right table
* RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table
* FULL (OUTER) JOIN: Returns all records when there is a match in either left or right table

## **INNER JOIN:**

The INNER JOIN keyword selects records that have matching values in both tables.

Syntax:

The syntax for SQL INNER JOIN is:

***SELECT table1.column1,table1.column2,table2.column1,....***

***FROM table1***

***INNER JOIN table2***

***ON table1.matching\_column = table2.matching\_column;***

Here,

table1: First table.

table2: Second table

matching\_column: Column common to both the tables.



**Note: We can also write JOIN instead of INNER JOIN. JOIN is same as INNER JOIN.**

**INNER JOIN Example**

Let’s look at the example of INNER JOIN clause, and understand it’s working.

This query will show the names and age of students enrolled in different courses.

***SELECT StudentCourse.COURSE\_ID, Student.NAME, Student.AGE***

***FROM Student***

***INNER JOIN StudentCourse***

***ON Student.ROLL\_NO = StudentCourse.ROLL\_NO;***

Output:



1. **SQL LEFT JOIN**

* LEFT JOIN returns all the rows of the table on the left side of the join and matches rows for the table on the right side of the join.
* For the rows for which there is no matching row on the right side, the result-set will contain null.
* LEFT JOIN is also known as LEFT OUTER JOIN.

The syntax of LEFT JOIN in SQL is:

***SELECT table1.column1,table1.column2,table2.column1,....***

***FROM table1***

***LEFT JOIN table2***

***ON table1.matching\_column = table2.matching\_column;***

Here,

table1: First table.

table2: Second table

matching\_column: Column common to both the tables.

**Note: We can also use LEFT OUTER JOIN instead of LEFT JOIN, both are the same.**



LEFT JOIN Example

Let’s look at the example of LEFT JOIN clause, and understand it’s working

***SELECT Student.NAME,StudentCourse.COURSE\_ID***

***FROM Student***

***LEFT JOIN StudentCourse***

***ON StudentCourse.ROLL\_NO = Student.ROLL\_NO;***

Output:



1. SQL RIGHT JOIN

RIGHT JOIN returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join.It is very similar to LEFT JOIN For the rows for which there is no matching row on the left side, the result-set will contain null. RIGHT JOIN is also known as RIGHT OUTER JOIN.

The syntax of RIGHT JOIN in SQL is:

***SELECT table1.column1,table1.column2,table2.column1,....***

***FROM table1***

***RIGHT JOIN table2***

***ON table1.matching\_column = table2.matching\_column;***

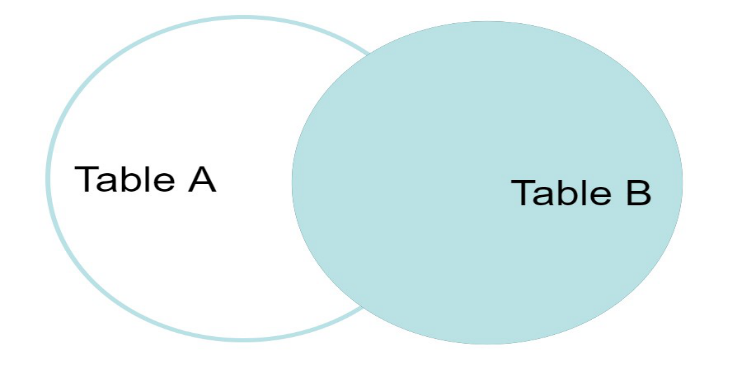
Here,

table1: First table.

table2: Second table

matching\_column: Column common to both the tables.

**Note: We can also use RIGHT OUTER JOIN instead of RIGHT JOIN, both are the same.**



RIGHT JOIN Example:

Let’s look at the example of RIGHT JOIN clause, and understand it’s working

***SELECT Student.NAME,StudentCourse.COURSE\_ID***

***FROM Student***

***RIGHT JOIN StudentCourse***

***ON StudentCourse.ROLL\_NO = Student.ROLL\_NO;***

Output:



1. **SQL FULL JOIN**

* FULL JOIN creates the result-set by combining results of both LEFT JOIN and RIGHT JOIN.
* The result-set will contain all the rows from both tables.
* For the rows for which there is no matching, the result-set will contain NULL values.

The syntax of SQL FULL JOIN is:

***SELECT table1.column1,table1.column2,table2.column1,....***

***FROM table1***

***FULL JOIN table2***

***ON table1.matching\_column = table2.matching\_column;***

Here,

table1: First table.

table2: Second table

matching\_column: Column common to both the tables.



FULL JOIN Example

Let’s look at the example of FULL JOIN clause, and understand it’s working

***SELECT Student.NAME, StudentCourse.COURSE\_ID***

***FROM Student***

***FULL JOIN StudentCourse***

***ON StudentCourse.ROLL\_NO = Student.ROLL\_NO;***

Output:

| **NAME** | **COURSE\_ID** |
| --- | --- |
| HARSH | 1 |
| PRATIK | 2 |
| RIYANKA | 2 |
| DEEP | 3 |
| SAPTARHI | 1 |
| DHANRAJ | NULL |
| ROHIT | NULL |
| NIRAJ | NULL |
| NULL | 4 |
| NULL | 5 |
| NULL | 4 |

**Explain the following SQL constructs with examples:**

**(1) order by:**

SQL ORDER BY clause sorts the result of the SELECT statement either in ascending or descending order.

In this article, we’ll explore the ORDER BY clause, exploring its syntax, functionality, and usage with detailed examples.

**ORDER BY in SQL**

The ORDER BY statement in SQL is used to sort the fetched data in either ascending or descending according to one or more columns. It is very useful to present data in a structured manner.

SQL ORDER BY default mode is sorting data into ascending order. To sort data in descending order use the DESC keyword with ORDER BY clause.

**Syntax:**

The syntax to use ORDER BY clause in SQL is:

***SELECT \****

***FROM table\_name***

***ORDER BY column\_name ASC | DESC***

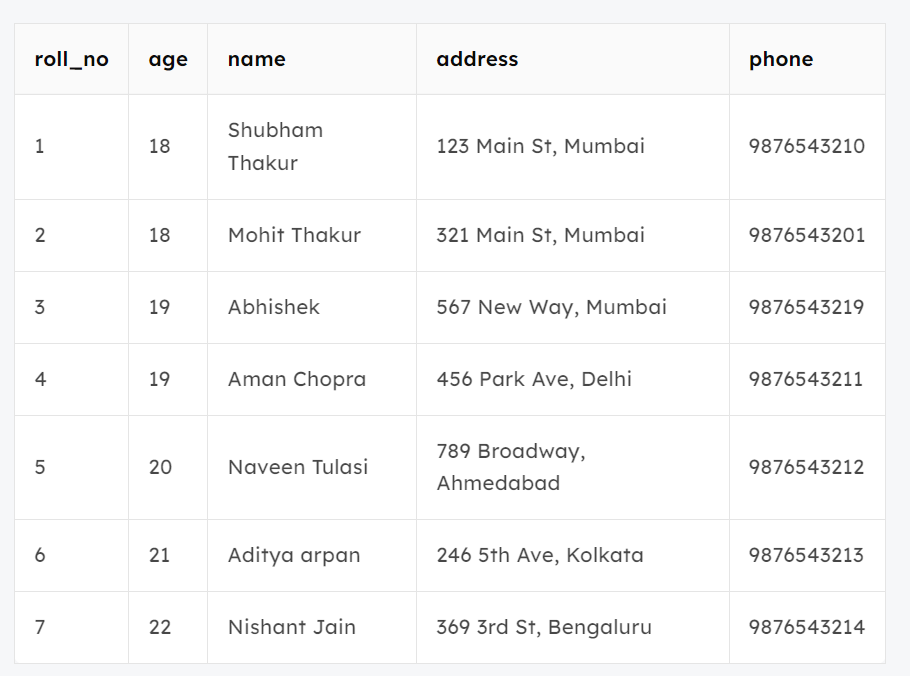
Key Terms:

table\_name: name of the table.

column\_name: name of the column according to which the data is needed to be arranged.

ASC: to sort the data in ascending order.

DESC: to sort the data in descending order.



**Example:**

**1. Sort According To a Single Column using ORDER BY Clause Example**

In this example, we will fetch all data from the table Student and sort the result in descending order according to the column ROLL\_NO.

Query:

***SELECT \****

***FROM students***

***ORDER BY ROLL\_NO DESC;***

**2. Sort According To Multiple Columns using ORDER BY Clause Example**

To sort according to multiple columns, separate the names of columns by the (,) operator.

Syntax:

***SELECT \****

***FROM table\_name***

***ORDER BY column1 ASC|DESC , column2 ASC|DESC***

In this example, we will fetch all data from the table Student and then sort the result in descending order first according to the column age. and then in ascending order according to the column name.

Query:

***SELECT \****

***FROM students***

***ORDER BY age DESC , name ASC;***

**(2) group by:**

The GROUP BY Statement in SQL is used to arrange identical data into groups with the help of some functions. i.e. if a particular column has the same values in different rows then it will arrange these rows in a group.

**Features:**

* GROUP BY clause is used with the SELECT statement.
* In the query, the GROUP BY clause is placed after the WHERE clause.
* In the query, the GROUP BY clause is placed before the ORDER BY clause if used.
* In the query, the Group BY clause is placed before the Having clause.
* Place condition in the having clause.

**Syntax:**

***SELECT column1, function\_name(column2)***

***FROM table\_name***

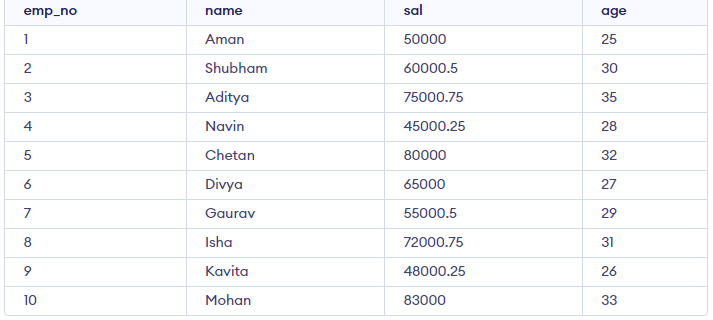
***WHERE condition***

***GROUP BY column1, column2***

***ORDER BY column1, column2;***

**Explanation:**

1. function\_name: Name of the function used for example, SUM() , AVG().
2. table\_name: Name of the table.
3. condition: Condition used.



**Example:**

1. **Group By single column:**

Group By single column means, placing all the rows with the same value of only that particular column in one group. Consider the query as shown below:

**Query:**

***SELECT NAME, SUM(SALARY) FROM emp***

***GROUP BY NAME;***

1. **Group By Multiple Columns**

Group by multiple columns is say, for example, GROUP BY column1, column2. This means placing all the rows with the same values of columns column 1 and column 2 in one group. Consider the below query:

**Query:**

***SELECT SUBJECT, YEAR, Count(\*)***

***FROM Student***

***GROUP BY SUBJECT, YEAR;***

1. **HAVING Clause in GROUP BY Clause**

We know that the WHERE clause is used to place conditions on columns but what if we want to place conditions on groups? This is where the HAVING clause comes into use. We can use the HAVING clause to place conditions to decide which group will be part of the final result set. Also, we can not use aggregate functions like SUM(), COUNT(), etc. with the WHERE clause. So we have to use the HAVING clause if we want to use any of these functions in the conditions.

**Syntax:**

***SELECT column1, function\_name(column2)***

***FROM table\_name***

***WHERE condition***

***GROUP BY column1, column2***

***HAVING condition***

***ORDER BY column1, column2;***

**Explanation:**

1. function\_name: Name of the function used for example, SUM() , AVG().
2. table\_name: Name of the table.
3. condition: Condition used.

Example:

***SELECT NAME, SUM(sal) FROM Emp***

***GROUP BY name***

***HAVING SUM(sal)>3000;***

**(3) having:**

The HAVING clause was introduced in SQL to allow the filtering of query results based on aggregate functions and groupings, which cannot be achieved using the WHERE clause that is used to filter individual rows.

In simpler terms MSSQL, the HAVING clause is used to apply a filter on the result of GROUP BY based on the specified condition. The conditions are Boolean type i.e. use of logical operators (AND, OR). This clause was included in SQL as the WHERE keyword failed when we use it with aggregate expressions. Having is a very generally used clause in SQL. Similar to WHERE it helps to apply conditions, but HAVING works with groups. If you wish to filter a group, the HAVING clause comes into action.

Some important points:

* Having clause is used to filter data according to the conditions provided.
* Having a clause is generally used in reports of large data.
* Having clause is only used with the SELECT clause.
* The expression in the syntax can only have constants.
* In the query, ORDER BY is to be placed after the HAVING clause, if any.
* HAVING Clause is implemented in column operation.
* Having clause is generally used after GROUP BY.

**Syntax:**

***SELECT col\_1, function\_name(col\_2)***

***FROM tablename***

***WHERE condition***

***GROUP BY column1, column2***

***HAVING Condition***

***ORDER BY column1, column2;***

**Example:**

1. If we need to display the departments where the sum of salaries is 50,000 or more. In this condition, we will use the HAVING Clause.

***SELECT Department, sum(Salary) as Salary***

***FROM employee***

***GROUP BY department***

***HAVING SUM(Salary) >= 50000;***

2. If we want to returns the students who have more percentage than 95 and the sum of percentage is less than 1000.

***SELECT student***

***FROM Student***

***WHERE percentage > 90***

***GROUP BY student, percentage***

***HAVING SUM(percentage) < 1000 AND AVG(percentage) > 95;***

**(4) as:**

The AS keyword in SQL is used to create aliases for tables or columns within a query. Aliases act like temporary nicknames that make your queries easier to read and understand. They only exist for the duration of the specific query you're running.

Here are some examples of how AS is used:

* 1. **Renaming a Column:**

Suppose you have a table called Customers with a column named CustomerID. The following query selects the CustomerID and assigns it the alias Customer Number:

***SELECT CustomerID AS Customer\_Number***

***FROM Customers;***

In the results, the CustomerID column will be displayed as Customer Number.

* 1. **Aliasing Functions:**

You can also use AS to create aliases for functions used within your query. For instance, this query finds the number of customers and assigns the result to the alias Total Customers:

***SELECT COUNT(\*) AS Total\_Customers***

***FROM Customers;***

The result set will show the count under the name Total Customers.

* 1. **Table Aliases:**

AS can also be used to create aliases for entire tables in queries involving joins. Imagine you have tables named Customers and Orders. This query joins them and assigns aliases C and O for better readability:

***SELECT C.CustomerID, C.CustomerName, O.OrderID, O.OrderAmount***

***FROM Customers AS C***

***JOIN Orders AS O ON C.CustomerID = O.CustomerID;***

By using aliases, you can shorten table names and make complex joins easier to follow.

**(5) in**

The SQL IN Operator:

* The IN operator allows you to specify multiple values in a WHERE clause.
* The IN operator is a shorthand for multiple OR conditions.
* The IN operator can be used with any data type in SQL. It is used to filter data from a database table based on specified values.
* The IN operator is useful when you want to select all rows that match one of a specific set of values. While the OR operator is useful when you want to select all rows that match any one of multiple conditions.

Syntax:

***SELECT column\_name(s)***

***FROM table\_name***

***WHERE column\_name IN (value1, value2, ...);***

**Example:**

**1. NOT IN**

By using the NOT keyword in front of the IN operator, you return all records that are NOT any of the values in the list.

Example

Return all customers that are NOT from 'Germany', 'France', or 'UK':

***SELECT \* FROM Customers***

***WHERE Country NOT IN ('Germany', 'France', 'UK');***

**2. IN (SELECT)**

You can also use IN with a subquery in the WHERE clause.

With a subquery you can return all records from the main query that are present in the result of the subquery.

Example

Return all customers that have an order in the Orders table:

***SELECT \* FROM Customers***

***WHERE CustomerID IN (SELECT CustomerID FROM Orders);***

**3. NOT IN (SELECT)**

The result in the example above returned 74 records, that means that there are 17 customers that haven't placed any orders.

Let us check if that is correct, by using the NOT IN operator.

Example

Return all customers that have NOT placed any orders in the Orders table:

***SELECT \* FROM Customers***

***WHERE CustomerID NOT IN (SELECT CustomerID FROM Orders);***

**1. Assume the Relations given below.**

**Student( Enrno, name, courseId, emailId, cellno)**

**Course(courseId, course\_nm, duration)**

CREATE TABLE Course1 (  
  courseId INT PRIMARY KEY,  
  course\_nm VARCHAR(50),  
  duration INT   
);

CREATE TABLE Student12 (  
  Enrno INT,  
  name VARCHAR(50),  
  courseId INT,  
  FOREIGN KEY (courseId) REFERENCES Course1(courseId),  
  emailId VARCHAR(50),  
  cellno VARCHAR(15),  
  marks INT  
);

INSERT INTO Course1 (courseId, course\_nm, duration)  
VALUES (101, 'Computer Science', 180);

INSERT INTO Course1 (courseId, course\_nm, duration)  
VALUES (102, 'Mechanical Engineering', 240);

INSERT INTO Course1 (courseId, course\_nm, duration)  
VALUES (103, 'Mathematics', 120);

INSERT INTO Student12 (Enrno, name, courseId, emailId, cellno, marks)  
VALUES (1, 'Alice', 101, 'alice@example.com', '1234567890', 85);

INSERT INTO Student12 (Enrno, name, courseId, emailId, cellno, marks)  
VALUES (2, 'Bob', 102, 'bob@example.com', '9876543210', 78);

INSERT INTO Student12 (Enrno, name, courseId, emailId, cellno, marks)  
VALUES (3, 'Charlie', 103, 'charlie@example.com', '0123456789', 90);

INSERT INTO Student12 (Enrno, name, courseId, emailId, cellno, marks)  
VALUES (4, 'David', 101, 'david@example.com', '1230987654', 65);

INSERT INTO Student12 (Enrno, name, courseId, emailId, cellno, marks)  
VALUES (5, 'Eve', 102, 'eve@example.com', '9870123456', 82);

**Write SQL statements for following:**

**a. Find out list of students who have enrolled in “computer” course.**

SELECT Student12.name  
FROM Student12  
INNER JOIN Course1 ON Student12.courseId = Course1.courseId  
WHERE Course1.course\_nm = 'Computer Science';

**b. List name of all courses with their duration.**

SELECT course\_nm, duration  
FROM Course1;

**c. List name of all students start with “a”.**

SELECT name  
FROM Student12  
WHERE name LIKE 'A%';

**d. List email Id and cell no of all mechanical engineering students.**

SELECT emailId, cellno  
FROM Student12  
INNER JOIN Course1 ON Student12.courseId = Course1.courseId  
WHERE Course1.course\_nm = 'Mechanical Engineering';

**8. The following relations keep track of Library Management system.**

**Book\_info( bookid, bname, bauthor, price, edition, publication, pur\_date,)**

**Student( lib\_car\_num, stud\_name, class, branch, roll\_no)**

**Issue\_table( issue\_date, sub\_date, bookid, lib\_car\_num, due)**

CREATE TABLE Book\_info (

bookid INT PRIMARY KEY,

bname VARCHAR(255),

bauthor VARCHAR(255),

price DECIMAL(10,2),

edition INT,

publication VARCHAR(255),

pur\_date DATE

);

CREATE TABLE Student (

lib\_car\_num INT PRIMARY KEY,

stud\_name VARCHAR(255),

class VARCHAR(50),

branch VARCHAR(100),

roll\_no INT

);

DROP TABLE Student;

CREATE TABLE Issue\_table (

issue\_date DATE,

sub\_date DATE,

bookid INT,

lib\_car\_num INT,

due DECIMAL(10,2),

FOREIGN KEY (bookid) REFERENCES Book\_info(bookid),

FOREIGN KEY (lib\_car\_num) REFERENCES Student(lib\_car\_num)

);

-- Inserting values into Book\_info table

INSERT INTO Book\_info (bookid, bname, bauthor, price, edition, publication, pur\_date) VALUES

(1, 'Book1', 'Author1', 20.00, 10, 'Publication1', '17SEP2003');

INSERT INTO Book\_info (bookid, bname, bauthor, price, edition, publication, pur\_date) VALUES

(2, 'Book2', 'Author2', 25.50, 20, 'Publication2', '17SEP2003');

INSERT INTO Book\_info (bookid, bname, bauthor, price, edition, publication, pur\_date) VALUES

(3, 'Book3', 'Author3', 18.75, 31, 'Publication3', '17SEP2003');

-- Inserting values into Student table

INSERT INTO Student (lib\_car\_num, stud\_name, class, branch, roll\_no) VALUES

(1, 'Student1', 'Class A', 'Branch1', 101);

INSERT INTO Student (lib\_car\_num, stud\_name, class, branch, roll\_no) VALUES

(2, 'Student2', 'Class B', 'Branch2', 102);

INSERT INTO Student (lib\_car\_num, stud\_name, class, branch, roll\_no) VALUES

(3, 'Student3', 'Class C', 'Branch3', 103);

-- Inserting values into Issue\_table

INSERT INTO Issue\_table (issue\_date, sub\_date, bookid, lib\_car\_num, due) VALUES

('20OCT2003', '17OCT2003', 1, 1, 0);

INSERT INTO Issue\_table (issue\_date, sub\_date, bookid, lib\_car\_num, due) VALUES

('20OCT2003', '17OCT2003', 2, 2, 5.00);

INSERT INTO Issue\_table (issue\_date, sub\_date, bookid, lib\_car\_num, due) VALUES

('20OCT2003', '17OCT2003', 3, 1, 0);

INSERT INTO Issue\_table (issue\_date, sub\_date, bookid, lib\_car\_num, due) VALUES

('20OCT2003', '17OCT2003', 2, 3, 7.50);

**Write the following SQL queries:**

**a. Find the details of the books issued to the library card number 1.**

SELECT Book\_info.\*

FROM Book\_info

JOIN Issue\_table ON Book\_info.bookid = Issue\_table.bookid

WHERE Issue\_table.lib\_car\_num = 1;

**b. Give all the information about student and the book issued with ascending order of library card number**

SELECT Student.\*, Book\_info.\*

FROM Student

JOIN Issue\_table ON Student.lib\_car\_num = Issue\_table.lib\_car\_num

JOIN Book\_info ON Issue\_table.bookid = Book\_info.bookid

ORDER BY Student.lib\_car\_num ASC;

**c. Find the author, edition, price of book.**

SELECT bauthor, edition, price

FROM Book\_info;

**d. Find the names of the students with dues on the book issue.**

SELECT DISTINCT Student.stud\_name

FROM Student

JOIN Issue\_table ON Student.lib\_car\_num = Issue\_table.lib\_car\_num

WHERE Issue\_table.due > 0;

* Domain Constraints: A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as a constraint on the values that it can take. Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered into the database.
* Referential Integrity: There are cases where we wish to ensure that a value that appears in one relation for a given set of attributes also appears in a certain set of attributes in another relation (referential integrity). For example, the department listed for each course must be one that actually exists. More precisely, the dept name value in a course record must appear in the dept name attribute of some record of the department relation. Database modifications can cause violations of referential integrity. When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.