Lab Manual

Database Lab (BCAMJ-406)
BCA (Hons.)- 4TH SEMESTER

ACADEMIC YEAR (2024-25)

Dr.Archana Sharma
Assistant Professor
P.G. Department of Computer Applications

Model Institute of Engineering & Technology

Approved by AICTE and permanently affiliated to University of Jammu NAAC "A" Grade

Accredited

www.mietjammu.in

LIST OF EXPERIMENTS

S.No.	Title
1	To execute the DDL commands • CREATE • ALTER • DROP • RENAME • TRUNCATE
2	To execute DML commands • INSERT • UPDATE • DELETE • SELECT
3	Creating tables with constraints: • NOT NULL • UNIQUE • PRIMARY KEY • FOREIGN KEY
4	Implementation of Number function-abs (), min (), max (), ceiling (), floor (), round (), mod (), pow ()
5	Implementation of Aggregate Function-count (), sum (), avg (), min (), max ()
6	Implementation of Conversion Function-cast (), convert (), TO_CHAR (), TO_DATE (), TO_NUMBER ()
7	Implementation of Character Function-length (), INITCAP (), LOWER (), UPPER (), TRIM (), CONCAT ()
8	Implementation of Date Function
9	Implementation of Group By & having clause
10	Implementation of Order by clause
11	Implementation of Views • Create Views • Insert data in views • Selecting a data from views • Filtering Data from a View • Updating Data of Views
12	Implementation of different types of Joins • Inner Join • Outer Join • Natural Join

EXPERIMENT NO: 1

OBJECTIVE: To execute the DDL commands

- **CREATE**
- **ALTER**
- **DROP**
- **RENAME**
- **TRUNCATE**

Sol: 1. CREATE

The CREATE command is used to create a new table, database, index, or other database objects.
Example (creating a table):

sql

```
CREATE TABLE Employees (
   EmployeeID INT PRIMARY KEY,
   FirstName VARCHAR(50),
   LastName VARCHAR(50),
   HireDate DATE
);
```

2. ALTER

The ALTER command is used to modify an existing database object, such as adding, deleting, or modifying columns in a table.
Example (adding a new column to a table):

sql

```
ALTER TABLE Employees
ADD Email VARCHAR(100);
```

3. DROP

The DROP command is used to delete a table, view, or other database objects. This action is irreversible.
Example (dropping a table):

sql
```
DROP TABLE Employees;
```

4. RENAME

The RENAME command is used to change the name of a database object, such as a table.
Example (renaming a table):

sql
```
RENAME TABLE Employees TO Staff;
```

5. TRUNCATE

The TRUNCATE command is used to delete all rows from a table, but unlike DROP, the table structure remains intact. It is faster than DELETE as it does not log individual row deletions.
Example (truncating a table):

sql
```
TRUNCATE TABLE Employees;
```

EXPERIMENT NO: 2

OBJECTIVE:

To execute DML commands

- **INSERT**
- **UPDATE**
- **DELETE**
- **SELECT**

Sol: 1. INSERT

The INSERT command is used to add new rows of data to a table.

Example (inserting a row into the Employees table):

sql

```
INSERT INTO Employees (EmployeeID, FirstName, LastName, HireDate, Email)
VALUES (1, 'John', 'Doe', '2025-01-29', 'john.doe@example.com');
```

You can also insert multiple rows at once:

sql

```
INSERT INTO Employees (EmployeeID, FirstName, LastName, HireDate, Email)
VALUES
   (2, 'Jane', 'Smith', '2025-01-28', 'jane.smith@example.com'),
   (3, 'Alice', 'Johnson', '2025-01-27', 'alice.johnson@example.com');
```

2. UPDATE

The UPDATE command is used to modify existing records in a table.

Example (updating a record in the Employees table):

sql

```
UPDATE Employees
SET Email = 'john.doe@newdomain.com'
WHERE EmployeeID = 1;
```

Be cautious with the UPDATE command. If the WHERE clause is omitted, all rows in the table will be updated!

3. DELETE

The DELETE command is used to remove rows from a table.
Example (deleting a record from the Employees table):

sql

```
DELETE FROM Employees
WHERE EmployeeID = 3;
```

Again, be careful with the DELETE command—if you don't specify a WHERE condition, all rows in the table will be deleted!

4. SELECT

The SELECT command is used to retrieve data from one or more tables.
Example (selecting all columns from the Employees table):
sql

```
SELECT * FROM Employees;
```

Example (selecting specific columns):

sql
```
SELECT FirstName, LastName, Email FROM Employees;
```

You can also filter results using the WHERE clause:

sql
```
SELECT * FROM Employees
WHERE HireDate > '2025-01-01';
```

To sort the results, use the ORDER BY clause:

sql

```
SELECT * FROM Employees
ORDER BY LastName ASC;
```

EXPERIMENT NO: 3

OBJECTIVE:

Creating tables with constraints:

- **NOT NULL**
- **UNIQUE**
- **PRIMARY KEY**
- **FOREIGN KEY**

Sol: When creating tables in SQL, constraints help enforce the integrity and validity of the data. Constraints ensure that data entered into the table follows specific rules. Here's how you can create tables with common constraints like NOT NULL, UNIQUE, PRIMARY KEY, and FOREIGN KEY:

1. NOT NULL

The NOT NULL constraint ensures that a column cannot contain a NULL value. It is used to ensure that a field must always have a value when inserting a new record.

Example:

```
sql
CREATE TABLE Employees (
   EmployeeID INT NOT NULL,
   FirstName VARCHAR(50) NOT NULL,
   LastName VARCHAR(50) NOT NULL,
   HireDate DATE,
   Email VARCHAR(100) NOT NULL,
   PRIMARY KEY (EmployeeID)
);
```

In this example, EmployeeID, FirstName, LastName, and Email must have values when inserting records. They cannot be NULL.

2. UNIQUE

The UNIQUE constraint ensures that all values in a column are different. No two rows can have the same value for a column that is marked as UNIQUE.
Example:

```
sql
CREATE TABLE Employees (
   EmployeeID INT NOT NULL,
   FirstName VARCHAR(50),
   LastName VARCHAR(50),
   Email VARCHAR(100) UNIQUE,
   HireDate DATE,
   PRIMARY KEY (EmployeeID)
);
```

In this example, the Email column must have unique values. No two employees can have the same email address.

3. PRIMARY KEY
The PRIMARY KEY constraint uniquely identifies each record in the table. A table can only have one primary key, and it automatically implies that the columns involved are NOT NULL and UNIQUE.
Example:

```
sql
CREATE TABLE Employees (
   EmployeeID INT PRIMARY KEY,
   FirstName VARCHAR(50),
   LastName VARCHAR(50),
   Email VARCHAR(100),
   HireDate DATE
);
```

In this example, EmployeeID is the primary key, ensuring that each employee has a unique identifier. The primary key will automatically enforce uniqueness and non-nullability.

4. FOREIGN KEY

A FOREIGN KEY is a column (or a set of columns) in a table that creates a relationship between the data in two tables. It ensures that the value in the foreign key column matches a value in the referenced table's primary key or unique column.
Example (assuming there's another table Departments):

```
sql
CREATE TABLE Departments (
    DepartmentID INT PRIMARY KEY,
    DepartmentName VARCHAR(100)
);

CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    Email VARCHAR(100),
    DepartmentID INT,
    HireDate DATE,
    FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)
);
```

In this example:

- The Employees table has a DepartmentID column.
- The DepartmentID column in the Employees table is a foreign key that references the DepartmentID column in the Departments table.
  This ensures that every employee is associated with a valid department (i.e., one that exists in the Departments table).

Putting it all together:

Here's a combined example of a table creation with NOT NULL, UNIQUE, PRIMARY KEY, and FOREIGN KEY constraints:

sql

```
CREATE TABLE Departments (
   DepartmentID INT PRIMARY KEY,
   DepartmentName VARCHAR(100) NOT NULL
);

CREATE TABLE Employees (
   EmployeeID INT PRIMARY KEY,
   FirstName VARCHAR(50) NOT NULL,
   LastName VARCHAR(50) NOT NULL,
   Email VARCHAR(100) UNIQUE NOT NULL,
   HireDate DATE,
   DepartmentID INT,
   FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)
);
```

Explanation:

- Departments table has DepartmentID as the primary key, and DepartmentName cannot be NULL.
- Employees table has EmployeeID as the primary key, with FirstName, LastName, and Email being NOT NULL. The Email column is also UNIQUE, and DepartmentID is a FOREIGN KEY referencing the Departments table.

These constraints help maintain data integrity by ensuring that required fields are filled, there are no duplicate entries where they shouldn't be, and relationships between tables are correctly enforced.

EXPERIMENT NO: 4

OBJECTIVE: Implementation of Number function-abs (), min (), max (), ceiling (), floor (), round (), mod (), pow ()

Sol: In SQL, many databases support a set of built-in numeric functions to perform common mathematical operations. Here's an explanation and example of how to use each of the functions you mentioned: ABS(), MIN(), MAX(), CEILING(), FLOOR(), ROUND(), MOD(), and POW().

1. ABS( )

The ABS( ) function returns the absolute value of a number, meaning it removes the sign (positive or negative) of the number.

Example:

sql

```
SELECT ABS(-25);  -- Returns 25
SELECT ABS(15);   -- Returns 15
```

2. MIN( )

The MIN( ) function returns the smallest value in a set of values or column. It's typically used with GROUP BY or ORDER BY to find the minimum value.

Example:

sql

```
-- Finding the minimum value in a column
SELECT MIN(Salary) FROM Employees;

-- Finding the smallest value in a set of numbers
SELECT MIN(10, 20, 30, 40);  -- Returns 10
```

3. MAX( )

The MAX( ) function returns the largest value in a set of values or column. Like MIN(), it is used with GROUP BY or ORDER BY.

Example:

```
sql
-- Finding the maximum value in a column
SELECT MAX(Salary) FROM Employees;

-- Finding the largest value in a set of numbers
SELECT MAX(10, 20, 30, 40);  -- Returns 40
```

4. CEILING( )

The CEILING( ) function rounds a number up to the nearest integer. This means that if the number has a decimal part, it rounds up to the next whole number.

Example:

```
sql
SELECT CEILING(3.2);   -- Returns 4
SELECT CEILING(3.9);   -- Returns 4
SELECT CEILING(-3.2);  -- Returns -3
```

5. FLOOR( )

The FLOOR( ) function rounds a number down to the nearest integer. It truncates the decimal part and moves the number to the closest whole number less than or equal to the original number.

Example:

```
sql
SELECT FLOOR(3.7);    -- Returns 3
SELECT FLOOR(-3.7);   -- Returns -4
SELECT FLOOR(5.0);    -- Returns 5
```

6. ROUND( )

The ROUND( ) function rounds a number to a specified number of decimal places. If no decimal places are specified, it rounds to the nearest integer.

Example:

sql

```
SELECT ROUND(3.14159, 2);  -- Returns 3.14
SELECT ROUND(3.14159);     -- Returns 3
SELECT ROUND(3.6);         -- Returns 4 (since 3.6 is closer to 4)
SELECT ROUND(-3.6);        -- Returns -4 (since -3.6 is closer to -4)
```

7. MOD( )

The MOD( ) function returns the remainder of a division operation. It's commonly used to find the modulus between two numbers.

Example:

sql

```
SELECT MOD(10, 3);  -- Returns 1 (10 divided by 3 leaves a remainder of 1)
SELECT MOD(20, 6);  -- Returns 2 (20 divided by 6 leaves a remainder of 2)
SELECT MOD(-10, 3); -- Returns -1 (remainder of -10 divided by 3)
```

8. POW( )

The POW( ) function returns the result of raising a number to the power of another number (i.e., performs exponentiation).

Example:

sql

```
SELECT POW(2, 3);    -- Returns 8 (2 raised to the power of 3)
SELECT POW(5, 2);    -- Returns 25 (5 raised to the power of 2)
SELECT POW(9, 0.5);  -- Returns 3 (square root of 9, which is 3)
```

Summary of Functions and Their Use Cases:

1. ABS(): Converts a number to its absolute value.
2. MIN(): Finds the smallest value in a column or a list of numbers.
3. MAX(): Finds the largest value in a column or a list of numbers.
4. CEILING(): Rounds a number up to the nearest integer.
5. FLOOR(): Rounds a number down to the nearest integer.
6. ROUND(): Rounds a number to a specified number of decimal places or to the nearest integer.
7. MOD(): Returns the remainder of a division operation.
8. POW(): Raises a number to a specified power (exponentiation).

EXPERIMENT NO: 5

OBJECTIVE: Implementation of Aggregate Function-count (), sum (), avg (), min (), max ()

Sol: Aggregate functions in SQL are used to perform a calculation on a set of values and return a single value. These functions are typically used with the GROUP BY clause but can also be used without it. Here are the common aggregate functions you're asking about: **COUNT(), SUM(), AVG(), MIN(), MAX()**.

1. COUNT( )

The COUNT( ) function returns the number of rows that match a specified condition. It can count the number of non-NULL values in a column or the total number of rows.

Examples:

- **Counting total rows in a table:**

```
sql
SELECT COUNT(*) FROM Employees;  -- Returns the total number of rows in the Employees table
```

- **Counting non-NULL values in a column:**

```
sql
SELECT COUNT(Email) FROM Employees;  -- Counts only non-NULL values in the Email column
```

- **Counting with a condition:**

```
sql
SELECT COUNT(*) FROM Employees WHERE DepartmentID = 1;  -- Counts employees in Department 1
```

2. SUM( )

The SUM( ) function calculates the total sum of a numeric column. It only considers numeric data and ignores NULL values.

Example:

```
sql
SELECT SUM(Salary) FROM Employees;  -- Returns the total salary of all employees
```

- **Summing with a condition:**

sql

SELECT SUM(Salary) FROM Employees WHERE DepartmentID = 1; -- Total salary of employees in Department 1

3. AVG( )

The AVG( ) function returns the average value of a numeric column. It adds up the values in the column and divides by the number of non-NULL values.

Example:

sql

SELECT AVG(Salary) FROM Employees; -- Returns the average salary of all employees

- **Average with a condition:**

sql

SELECT AVG(Salary) FROM Employees WHERE DepartmentID = 1; -- Average salary of employees in Department 1

4. MIN( )

The MIN( ) function returns the smallest value in a column. It works for both numeric and string data types, returning the "lowest" value based on the column's data type.

Example:

sql

SELECT MIN(Salary) FROM Employees; -- Returns the lowest salary among employees

- **Finding the minimum value with a condition:**

sql

SELECT MIN(Salary) FROM Employees WHERE DepartmentID = 1; -- Lowest salary of employees in Department 1

5. MAX( )

The MAX( ) function returns the largest value in a column. Like MIN(), it works for numeric and string data types and returns the "highest" value.

Example:

sql

SELECT MAX(Salary) FROM Employees; -- Returns the highest salary among employees

- **Finding the maximum value with a condition:**

```
sql
SELECT MAX(Salary) FROM Employees WHERE DepartmentID = 1;  -- Highest salary of employees in Department 1
```

Combining Aggregate Functions with GROUP BY

You can use aggregate functions in combination with the GROUP BY clause to calculate results for each group of data.

Example: Using aggregate functions with GROUP BY:

```
sql
SELECT DepartmentID,
    COUNT(*) AS EmployeeCount,
    SUM(Salary) AS TotalSalary,
    AVG(Salary) AS AvgSalary,
    MIN(Salary) AS MinSalary,
    MAX(Salary) AS MaxSalary
FROM Employees
GROUP BY DepartmentID;
```

In this example:

- The COUNT(*) counts the number of employees in each department.
- The SUM(Salary) calculates the total salary in each department.
- The AVG(Salary) finds the average salary in each department.
- The MIN(Salary) finds the lowest salary in each department.
- The MAX(Salary) finds the highest salary in each department.

This query groups the data by DepartmentID and calculates the aggregate values for each department.

Summary of Aggregate Functions:

1. **COUNT( )**: Returns the number of rows in a set or the number of non-NULL values in a column.
2. **SUM( )**: Returns the sum of the values in a numeric column.

3. **AVG( )**: Returns the average of the values in a numeric column.
4. **MIN( )**: Returns the smallest value in a column.
5. **MAX( )**: Returns the largest value in a column.

These functions are very helpful in generating summary data, such as calculating totals, averages, and ranges, and are often used in reporting and data analysis.

EXPERIMENT NO: 6

OBJECTIVE: Implementation of Conversion Function-cast (), convert (), TO_CHAR (), TO_DATE (), TO_NUMBER ()

Sol: In SQL, conversion functions are used to convert data from one data type to another. The functions are — **CAST(), CONVERT(), TO_CHAR(), TO_DATE(), and TO_NUMBER()** — are often used to ensure that data is in the correct format for comparison, display, or calculation. Below are details on how each of these functions is used in SQL

1. CAST( )

The CAST( ) function is used to convert an expression from one data type to another.

Syntax:

sql

CAST(expression AS target_data_type)

Example:

sql

SELECT CAST('123.45' AS DECIMAL(10, 2)); -- Converts the string '123.45' to a DECIMAL data type

SELECT CAST(150 AS VARCHAR(10)); -- Converts the integer 150 to a VARCHAR string

Usage:

- **Converting a string to a numeric type:**

sql

SELECT CAST('2025-01-29' AS DATE); -- Converts a string to a date

- **Converting a number to a string:**

sql

SELECT CAST(100 AS VARCHAR(5)); -- Converts number 100 to a string '100'

2. CONVERT( )

The CONVERT( ) function is similar to CAST() but provides additional formatting options, especially for date-time and string conversion. CONVERT() is more commonly used in **SQL Server**.

Syntax:

sql
```
CONVERT(target_data_type, expression, [style])
```
The third parameter, style, is optional and can be used to specify a format (particularly useful for dates).

Example:

sql
```
SELECT CONVERT(VARCHAR, 123.456, 1);  -- Converts the number 123.456 to a string '123.5' (rounded)
```

- **Converting a date to a string:**

sql
```
SELECT CONVERT(VARCHAR, GETDATE(), 103);  -- Converts current date to 'dd/mm/yyyy' format in SQL Server
```

- **Converting a string to a date:**

sql
```
SELECT CONVERT(DATE, '2025-01-29', 23);  -- Converts a string to a DATE
```

3. TO_CHAR( )

The TO_CHAR ( ) function is used to convert a **date**, **timestamp**, or **number** to a string. It is commonly used in **Oracle** and **PostgreSQL**.

Syntax:

sql
```
TO_CHAR(expression, format)
```

Example:

sql
```
-- Convert a date to a string with a specific format
SELECT TO_CHAR(SYSDATE, 'YYYY-MM-DD');  -- Converts the current date to a string in 'YYYY-MM-DD' format
```

- **Converting numbers to a string:**

sql
```
SELECT TO_CHAR(12345.6789, '99999.99');  -- Converts the number to a string with a specified format
```

- **Date formatting:**

sql
```
SELECT TO_CHAR(SYSDATE, 'Day, DD Mon YYYY');  -- Outputs the date as 'Monday, 29 Jan 2025'
```

4. TO_DATE( )

The TO_DATE() function is used to convert a string to a date in **Oracle**. You can specify the format of the string you're converting.

Syntax:

sql

TO_DATE(string, format)

Example:

sql

-- Converting a string to a date with a specific format

SELECT TO_DATE('2025-01-29', 'YYYY-MM-DD'); -- Converts the string '2025-01-29' to a date

- **Converting a string to a date with a different format:**

sql

SELECT TO_DATE('29-JAN-2025', 'DD-MON-YYYY'); -- Converts '29-JAN-2025' to a date

5. TO_NUMBER( )

The TO_NUMBER( ) function is used to convert a string or date to a numeric value. It is mostly used in **Oracle** and **PostgreSQL**.

Syntax:

sql

TO_NUMBER(expression, format)

Example:

sql

-- Converting a string to a number

SELECT TO_NUMBER('123.45', '999.99'); -- Converts the string '123.45' to a number

- **Converting a date to a number (typically used for extracting a numerical value from a date):**

sql

SELECT TO_NUMBER(TO_CHAR(SYSDATE, 'YYYYMMDD')); -- Converts the current date to a number '20250129'

Summary of Conversion Functions:

1. **CAST()**: Converts one data type to another, e.g., from a string to a number or date.

2. **CONVERT()**: Similar to CAST(), but it also allows for additional formatting, particularly useful for date-time conversions in SQL Server.
3. **TO_CHAR()**: Converts numbers, dates, or timestamps to a string in a specified format (commonly used in Oracle and PostgreSQL).
4. **TO_DATE()**: Converts a string to a date based on a specified format (commonly used in Oracle).
5. **TO_NUMBER()**: Converts a string to a numeric value (commonly used in Oracle).

These conversion functions are incredibly useful when dealing with data that needs to be in a specific format for comparisons, calculations, or display purposes.

EXPERIMENT NO: 7

OBJECTIVE: Implementation of Character Function-length (), INITCAP (), LOWER (), UPPER (), TRIM (), CONCAT ()

Sol:

1. **LENGTH()**: Returns the length of a string (in terms of the number of characters).

   Example:

   sql

   SELECT LENGTH('Hello World'); -- Output: 11

2. **INITCAP()**: Converts the first letter of each word in a string to uppercase and the remaining letters to lowercase.

   Example:

   sql

   SELECT INITCAP('hello world'); -- Output: 'Hello World'

3. **LOWER()**: Converts all characters in a string to lowercase.

   Example:

   sql

   SELECT LOWER('Hello World'); -- Output: 'hello world'

4. **UPPER()**: Converts all characters in a string to uppercase.

   Example:

   sql

   SELECT UPPER('Hello World'); -- Output: 'HELLO WORLD'

5. **TRIM()**: Removes leading and trailing spaces from a string.

   Example:

   sql

   SELECT TRIM('   Hello World   '); -- Output: 'Hello World'

6. **CONCAT()**: Combines two or more strings into one string.

   Example:

   sql

   ```
   SELECT CONCAT('Hello ', 'World');  -- Output: 'Hello World'
   ```

Combined Example:

sql

```
SELECT LENGTH('  Hello World  '),
       INITCAP('hello world'),
       LOWER('HELLO WORLD'),
       UPPER('hello world'),
       TRIM('   Hello World   '),
       CONCAT('Hello ', 'World');
```

Output:

- Length of ' Hello World ': 15 (including spaces)
- 'Hello World' (INITCAP applied)
- 'hello world' (LOWER applied)
- 'HELLO WORLD' (UPPER applied)
- 'Hello World' (TRIM applied)
- 'Hello World' (CONCAT applied)

These functions are essential for string manipulation in databases and other systems.

EXPERIMENT NO: 8

OBJECTIVE: Implementation of Date Function

Sol: Date functions are used to manipulate and perform operations on date and time values in SQL or programming languages. Here are some common date functions, along with examples of how to implement them:

1. CURRENT_DATE(): Returns the current date.

Example:

sql

```
SELECT CURRENT_DATE();  -- Output: 2025-01-29 (for example)
```

2. CURRENT_TIME(): Returns the current time.

Example:

sql

```
SELECT CURRENT_TIME();  -- Output: 14:30:00 (for example)
```

3. CURRENT_TIMESTAMP(): Returns the current date and time.

Example:

sql

```
SELECT CURRENT_TIMESTAMP();  -- Output: 2025-01-29 14:30:00 (for example)
```

4. DATE_ADD(): Adds a specific time interval to a date.

Example:

sql

```
SELECT DATE_ADD('2025-01-29', INTERVAL 10 DAY);  -- Output: 2025-02-08
```

5. DATE_SUB(): Subtracts a specific time interval from a date.

Example:

sql

```
SELECT DATE_SUB('2025-01-29', INTERVAL 10 DAY);  -- Output: 2025-01-19
```

6. DATEDIFF(): Returns the difference in days between two dates.

Example:

sql

```
SELECT DATEDIFF('2025-02-08', '2025-01-29');  -- Output: 10
```

7. EXTRACT(): Extracts a specific part (like year, month, day, etc.) from a date.

Example:

sql

```
SELECT EXTRACT(YEAR FROM '2025-01-29');  -- Output: 2025
SELECT EXTRACT(MONTH FROM '2025-01-29');  -- Output: 1
SELECT EXTRACT(DAY FROM '2025-01-29');  -- Output: 29
```

8. DATE_FORMAT(): Formats a date according to a specific format.

Example (MySQL):

sql

```
SELECT DATE_FORMAT('2025-01-29', '%Y-%m-%d');  -- Output: 2025-01-29
SELECT DATE_FORMAT('2025-01-29', '%M %d, %Y');  -- Output: January 29, 2025
```

9. NOW(): Returns the current date and time (similar to CURRENT_TIMESTAMP()).

Example:

sql

```
SELECT NOW();  -- Output: 2025-01-29 14:30:00 (for example)
```

10. YEAR(), MONTH(), DAY(): Extracts the year, month, and day from a date.

Example:

sql

```
SELECT YEAR('2025-01-29');  -- Output: 2025
SELECT MONTH('2025-01-29');  -- Output: 1
SELECT DAY('2025-01-29');  -- Output: 29
```

Combined Example of Date Functions:

sql

```
SELECT CURRENT_DATE(),
    CURRENT_TIME(),
    CURRENT_TIMESTAMP(),
    DATE_ADD('2025-01-29', INTERVAL 10 DAY),
    DATEDIFF('2025-02-08', '2025-01-29'),
    EXTRACT(YEAR FROM '2025-01-29'),
    EXTRACT(MONTH FROM '2025-01-29'),
    DATE_FORMAT('2025-01-29', '%M %d, %Y');
```

Output:

- Current Date: 2025-01-29
- Current Time: 14:30:00
- Current Timestamp: 2025-01-29 14:30:00
- Date after adding 10 days: 2025-02-08
- Difference between two dates: 10 days
- Extracted Year: 2025
- Extracted Month: 1
- Formatted Date: January 29, 2025

EXPERIMENT NO: 9

OBJECTIVE: Implementation of Group By & having clause

Sol: The GROUP BY and HAVING clauses in SQL are used to group rows that have the same values in specified columns into summary rows and filter those groups, respectively. These clauses are often used together to aggregate data and then filter based on those aggregations.

Here's a breakdown of how they work:

1. GROUP BY Clause

The GROUP BY clause is used to group rows based on one or more columns. It is typically used with aggregate functions (like SUM(), COUNT(), AVG(), etc.) to perform operations on each group.

Example:

Let's assume you have a sales table with the following columns:

- sale_id
- product_name
- sale_date
- amount

If you want to find the total sales (SUM(amount)) for each product, you can use the GROUP BY clause to group by product_name.

```
sql
SELECT product_name, SUM(amount) AS total_sales
FROM sales
GROUP BY product_name;
```

Explanation:

- This query groups the rows by the product_name column.
- The SUM(amount) calculates the total sales for each product.

2. HAVING Clause

The HAVING clause is used to filter groups after the GROUP BY operation. Unlike the WHERE clause, which filters rows before grouping, the HAVING clause filters the result of the aggregation.

Example:

Let's say you only want to see products with total sales greater than 1000. You can use the HAVING clause to filter groups based on the aggregated value.

sql

```
SELECT product_name, SUM(amount) AS total_sales
FROM sales
GROUP BY product_name
HAVING SUM(amount) > 1000;
```

Explanation:

- This query first groups the sales by product_name.
- Then, it filters out any groups where the total sales (SUM(amount)) is less than or equal to 1000.

Combined Example with GROUP BY and HAVING

Let’s assume the sales table has the following data:

sale_id	product_name	sale_date	amount
1	Widget	2025-01-01	200
2	Widget	2025-01-02	300
3	Gadget	2025-01-01	500
4	Gadget	2025-01-02	400
5	Widget	2025-01-03	400
6	Widget	2025-01-04	150

Now, let’s find the total sales for each product and filter out those with total sales greater than 1000.

sql

```
SELECT product_name, SUM(amount) AS total_sales
FROM sales
GROUP BY product_name
HAVING SUM(amount) > 1000;
```

Output:

product_name	total_sales
Widget	1050
Gadget	900

Explanation:

- The GROUP BY clause groups sales by product_name.
- The SUM(amount) calculates the total sales for each product.

- The HAVING clause filters out the Gadget group because its total sales are less than or equal to 1000.

Summary:

- GROUP BY: Groups rows based on column values.
- HAVING: Filters the result of the aggregation (after the GROUP BY).

Note: HAVING can only be used with aggregate functions (e.g., COUNT(), SUM(), AVG(), MAX(), MIN()), while WHERE is used to filter rows before aggregation.

EXPERIMENT NO: 10

OBJECTIVE: Implementation of Order by clause

Sol: The ORDER BY clause in SQL is used to sort the result set based on one or more columns, either in ascending or descending order. It can be used with both numeric and string columns. By default, the sorting is in **ascending order** (ASC), but you can explicitly specify the order (ascending or descending) for each column.

Syntax of the ORDER BY Clause:

```
sql
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...;
```

- ASC: Sorts the result in ascending order (this is the default).
- DESC: Sorts the result in descending order.

Examples of ORDER BY Clause:

1. Simple Sorting (Ascending Order)

If you want to sort the records by a single column in ascending order (alphabetically or numerically), you can do the following:

```
sql
SELECT product_name, price
FROM products
ORDER BY price;
```

- This will sort the products by their price in **ascending order**.

2. Descending Order

If you want to sort the records in descending order, you can specify DESC:

```
sql
SELECT product_name, price
FROM products
ORDER BY price DESC;
```

- This will sort the products by their price in **descending order** (from highest to lowest).

3. Sorting by Multiple Columns

You can also sort by multiple columns. If you want to sort products first by their category and then by price within each category:

```
sql
```

```
SELECT product_name, category, price
FROM products
ORDER BY category ASC, price DESC;
```

- This will sort the products first by category in **ascending order**, and for each category, it will sort by price in **descending order**.

4. Sorting by Text (Alphabetical Order)

For text-based fields, the sorting will follow alphabetical order. Here's an example of sorting by product_name alphabetically:

```
sql
SELECT product_name, price
FROM products
ORDER BY product_name;
```

- This will sort the products by product_name in **ascending alphabetical order**.

5. Sorting with NULL Values

In SQL, NULL values are treated as either the lowest or highest value depending on the database. By default:

- NULL values are treated as **the lowest** when sorting in **ascending order**.
- NULL values are treated as **the highest** when sorting in **descending order**.

For example, if you're sorting products by price and there are some NULL values in the price column, the results will vary depending on the order:

```
sql
SELECT product_name, price
FROM products
ORDER BY price;
```

- In ascending order, NULL values will be shown at the top of the list (treated as the lowest value).

If you want to explicitly control how NULL values are sorted, you can use the IS NULL condition or use specific database features to handle NULLs.

6. Using ORDER BY with Aggregate Functions

When combined with GROUP BY, you can use ORDER BY to sort the result of aggregated data. For example, sorting the total sales per product:

```
sql
SELECT product_name, SUM(amount) AS total_sales
FROM sales
GROUP BY product_name
ORDER BY total_sales DESC;
```

- This will display the total sales per product and order the products by total_sales in **descending order**.

Example with Data:

Let's say you have the following employees table:

employee_id	**name**	**department**	**salary**
1	Alice	HR	5000
2	Bob	IT	6000
3	Charlie	IT	4500
4	David	HR	5500
5	Eve	Sales	7000

Example 1: Order by salary in Ascending Order

```
sql
SELECT name, salary
FROM employees
ORDER BY salary;
```

- **Output** (ascending order by salary):

name	**salary**
Charlie	4500
Alice	5000
David	5500
Bob	6000
Eve	7000

Example 2: Order by salary in Descending Order

```
sql
SELECT name, salary
FROM employees
ORDER BY salary DESC;
```

- **Output** (descending order by salary):

name	salary
Eve	7000
Bob	6000
David	5500
Alice	5000
Charlie	4500

Example 3: Order by department and then by salary

```
sql
SELECT name, department, salary
FROM employees
ORDER BY department ASC, salary DESC;
```

- **Output** (first sorted by department in ascending order, then by salary in descending order within each department):

name	department	salary
Alice	HR	5000
David	HR	5500
Bob	IT	6000
Charlie	IT	4500
Eve	Sales	7000

Summary:

- ORDER BY is used to sort the result set by one or more columns.
- You can specify sorting order using ASC (ascending) or DESC (descending).
- It can be used with text, numbers, and dates.
- The default sorting order is ascending.
- You can sort by multiple columns, and you can specify different sorting directions for each column.

EXPERIMENT NO: 11

<u>**OBJECTIVE:**</u> **Implementation of Views**

- **Create Views**
- **Insert data in views**
- **Selecting a data from views**
- **Filtering Data from a View**
- **Updating Data of Views**

Sol: Views in SQL are virtual tables that represent the result of a query. They are essentially stored queries that can be referenced just like a regular table. Views provide a way to simplify complex queries, improve security, and allow easier management of data.

1. Creating a View

A view is created using the CREATE VIEW statement. You define the view by writing a SELECT query that specifies the data you want to include in the view.

Syntax:

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

Example:

Let's create a view employee_salary_view that shows employee names and their salaries from the employees table.

```
CREATE VIEW employee_salary_view AS
SELECT name, salary
FROM employees;
```

This creates a view called employee_salary_view, which includes the name and salary columns from the employees table.

2. Inserting Data into Views

You can insert data into a view if the view is updatable (i.e., it is created from a single table without joins or aggregation functions). However, it's important to note that you cannot insert data into views that involve multiple tables, aggregation, or complex joins.

Syntax:

```
INSERT INTO view_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

Example:

Let’s insert data into the employee_salary_view view (assuming the view is updatable):

```
INSERT INTO employee_salary_view (name, salary)
VALUES ('John Doe', 5500);
```

This will insert a new record into the underlying employees table (assuming employee_salary_view maps directly to the employees table).

3. Selecting Data from Views

You can select data from a view just like you would from a table. The view represents a stored query, so the data returned by a SELECT query from a view will be based on the logic defined when the view was created.

```
Syntax:
SELECT column1, column2, ...
FROM view_name;
```

Example:

To get the employee names and salaries from the employee_salary_view:

```
SELECT * FROM employee_salary_view;
```

This will return all employee names and their corresponding salaries from the view.

4. Filtering Data from a View

You can filter the data returned from a view by using a WHERE clause, just as you would filter data from a regular table.

```
Syntax:
SELECT column1, column2, ...
FROM view_name
WHERE condition;
```

Example:

Let’s say we want to filter out employees with salaries greater than 5000:

```
SELECT * FROM employee_salary_view
WHERE salary > 5000;
```

This will return all rows from the employee_salary_view where the salary is greater than 5000.

5. Updating Data in Views

You can update data in a view if the view is updatable. A view is typically updatable if it directly maps to a single table without any complex logic like joins or aggregations. However, if the view involves joins, groupings, or other complex operations, it may not be directly updatable.

```
Syntax:
UPDATE view_name
SET column1 = value1, column2 = value2, ...
```

```
WHERE condition;
```

Example:

To update the salary of an employee in the employee_salary_view:

```
UPDATE employee_salary_view
SET salary = 6000
WHERE name = 'John Doe';
```

This will update the underlying employees table, changing the salary of "John Doe" to 6000. Note that the view must be simple (e.g., based on a single table) for this to work.

Summary of View Operations:

1. Creating Views:
   ```
   CREATE VIEW view_name AS
   SELECT column1, column2, ...
   FROM table_name;
   ```
2. Inserting Data into Views:
   ```
   INSERT INTO view_name (column1, column2, ...)
   VALUES (value1, value2, ...);
   ```
3. Selecting Data from Views:
   ```
   SELECT * FROM view_name;
   ```
4. Filtering Data from Views:
   ```
   SELECT * FROM view_name
   WHERE condition;
   ```
5. Updating Data in Views:
   ```
   UPDATE view_name
   SET column1 = value1, column2 = value2, ...
   WHERE condition;
   ```

Restrictions with Views:

- You cannot use a view to insert data if the view involves multiple tables or complex operations like aggregation.
- Views cannot contain DISTINCT or GROUP BY clauses if you want to insert data into them.
- Not all views are updatable (e.g., views based on multiple tables or complex operations like joins may not allow updates).

Example Walkthrough:

Given the following employees table:

employee_id	name	department	salary
1	Alice	HR	5000

employee_id	name	department	salary
2	Bob	IT	6000
3	Charlie	IT	4500
4	David	HR	5500
5	Eve	Sales	7000

1. Create View to see employee names and salaries:
   ```
   CREATE VIEW employee_salary_view AS
   SELECT name, salary
   FROM employees;
   ```
2. Select Data from the view:
   ```
   SELECT * FROM employee_salary_view;
   ```
3. Insert Data into the view (assuming it's updatable):
   ```
   INSERT INTO employee_salary_view (name, salary)
   VALUES ('John', 4500);
   ```
4. Filter Data (e.g., show only salaries greater than 5000):
   ```
   SELECT * FROM employee_salary_view
   WHERE salary > 5000;
   ```
5. Update Data in the view:
   ```
   UPDATE employee_salary_view
   SET salary = 7500
   WHERE name = 'Bob';
   ```

In conclusion, views are very useful for simplifying queries and encapsulating business logic.

EXPERIMENT NO: 11

<u>OBJECTIVE:</u> Implementation of different types of Joins

- **Inner Join**
- **Outer Join**
- **Natural Join**

Sol: Joins in SQL allow you to combine rows from two or more tables based on a related column between them. The most common types of joins are INNER JOIN, OUTER JOIN (which includes LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN), and NATURAL JOIN. Below, I'll explain each type of join with examples:

1. INNER JOIN

An INNER JOIN returns only the rows that have matching values in both tables. If a row in one table doesn't have a corresponding match in the other table, it will not appear in the result.

Syntax:

```
sql
SELECT column1, column2, ...
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

Example:

Let's assume we have two tables:

employees table:

employee_id	name	department_id
1	Alice	101
2	Bob	102
3	Charlie	101

departments table:

department_id	department_name
101	HR
102	IT

To retrieve the names of employees along with their department names using an INNER JOIN:

```
sql
```

```
SELECT employees.name, departments.department_name
FROM employees
INNER JOIN departments
ON employees.department_id = departments.department_id;
```

Output:

name	department_name
Alice	HR
Bob	IT
Charlie	HR

In this case, we only get employees who have a corresponding department in the departments table.

---

2. OUTER JOIN

An OUTER JOIN includes rows that do not have a match in one of the tables. There are three types of OUTER JOINs: LEFT OUTER JOIN, RIGHT OUTER JOIN, and FULL OUTER JOIN.

a. LEFT JOIN (LEFT OUTER JOIN)

A LEFT JOIN returns all rows from the left table and the matching rows from the right table. If there's no match, the result is NULL on the side of the right table.

Syntax:

sql

```
SELECT column1, column2, ...
FROM table1
LEFT JOIN table2
ON table1.column = table2.column;
```

Example:

To get all employees and their corresponding department names, even if they don't belong to a department:

sql

```
SELECT employees.name, departments.department_name
FROM employees
LEFT JOIN departments
ON employees.department_id = departments.department_id;
```

Output:

name	department_name
Alice	HR
Bob	IT
Charlie	HR

If an employee has no corresponding department, the department name would be NULL for that employee.

b. RIGHT JOIN (RIGHT OUTER JOIN)

A RIGHT JOIN is the opposite of a LEFT JOIN. It returns all rows from the right table and the matching rows from the left table. If there’s no match, the result is NULL on the left table side.

Syntax:

```
sql
SELECT column1, column2, ...
FROM table1
RIGHT JOIN table2
ON table1.column = table2.column;
```

Example:

To get all departments and their corresponding employees, even if a department has no employees:

```
sql
SELECT employees.name, departments.department_name
FROM employees
RIGHT JOIN departments
ON employees.department_id = departments.department_id;
```

Output:

name	department_name
Alice	HR
Bob	IT
Charlie	HR
NULL	Marketing

In this case, Marketing department has no employees, so the employee column will be NULL for that row.

c. FULL OUTER JOIN

A FULL OUTER JOIN returns all rows from both the left and right tables. If there is no match, the result is NULL for non-matching rows from either table.

Syntax:

```
sql
SELECT column1, column2, ...
FROM table1
FULL OUTER JOIN table2
ON table1.column = table2.column;
```

Example:

To get all employees and departments, even if there are employees without departments or departments without employees:

```
sql
SELECT employees.name, departments.department_name
FROM employees
FULL OUTER JOIN departments
ON employees.department_id = departments.department_id;
```

Output:

name	department_name
Alice	HR
Bob	IT
Charlie	HR
NULL	Marketing

This includes all employees and all departments, even if some employees don't belong to a department or some departments don't have employees.

---

3. NATURAL JOIN

A NATURAL JOIN automatically joins tables based on columns with the same name and compatible data types. It eliminates the need to explicitly specify the ON clause. It's a shorthand for matching columns with the same names in both tables.

Syntax:

```
sql
SELECT column1, column2, ...
```

```
FROM table1
NATURAL JOIN table2;
```

Example:

Let’s assume the employees table now has the following additional column:

employees table:

employee_id	name	department_id
1	Alice	101
2	Bob	102
3	Charlie	101

departments table:

department_id	department_name
101	HR
102	IT

To perform a NATURAL JOIN:

```
sql
SELECT name, department_name
FROM employees
NATURAL JOIN departments;
```

Output:

name	department_name
Alice	HR
Bob	IT
Charlie	HR

Here, the department_id column is automatically used for the join because it has the same name in both tables.

Summary of Joins:

1. INNER JOIN: Returns rows where there is a match in both tables.
   - Syntax: SELECT ... FROM table1 INNER JOIN table2 ON condition;
2. LEFT JOIN (LEFT OUTER JOIN): Returns all rows from the left table, and the matched rows from the right table. If no match, NULL is returned for the right table.

    - Syntax: SELECT ... FROM table1 LEFT JOIN table2 ON condition;
3. RIGHT JOIN (RIGHT OUTER JOIN): Returns all rows from the right table, and the matched rows from the left table. If no match, NULL is returned for the left table.
    - Syntax: SELECT ... FROM table1 RIGHT JOIN table2 ON condition;
4. FULL OUTER JOIN: Returns all rows from both tables. If no match, NULL is returned for non-matching rows from either table.
    - Syntax: SELECT ... FROM table1 FULL OUTER JOIN table2 ON condition;
5. NATURAL JOIN: Joins tables based on columns with the same name in both tables.
    - Syntax: SELECT ... FROM table1 NATURAL JOIN table2;

Joins are crucial for combining data from multiple tables based on related columns. The choice of join type depends on the specific data you want to retrieve.