**SQL Notes**

**What is SQL?**

SQL stands for Structured Query Language and allows you to access and manipulate databases. It became a standard of the American National Standards Institute (ANSI) in 1986 and of the International Organization for Standardization (ISO) in 1987.

**What Can SQL Do?**

* Execute queries against a database
* Retrieve data from a database
* Insert records into a database
* Update records in a database
* Delete records from a database
* Create new databases
* Create new tables in a database
* Create stored procedures in a database
* Create views in a database
* Set permissions on tables, procedures, and views

**SQL is a Standard - BUT...**

Although SQL is an ANSI/ISO standard, there are different versions of the SQL language. However, to be compliant with the ANSI standard, they all support at least the major commands (such as SELECT, UPDATE, DELETE, INSERT, WHERE) in a similar manner.

**Using SQL in Your Website**

To build a website that shows data from a database, you will need:

* An RDBMS database program (e.g., MS Access, SQL Server, MySQL)
* A server-side scripting language like PHP or ASP
* SQL to get the data you want

**Introduction to Data, Database, and DBMS**

**What is Data?**

Data is raw, unorganized facts that need to be processed. Data can be simple and seemingly random and useless until it is organized.

**Examples:** Numbers, words, measurements, observations, etc.

**What is a Database?**

A database is an organized collection of structured information or data, typically stored electronically in a computer system. It allows for efficient management, retrieval, and updating of data.

**Examples:**

* Customer Database: Stores customer information such as names, addresses, and purchase history.
* Library Database: Keeps track of books, members, and loans.
* Inventory System: Manages stock levels, orders, and product details.

**What is a Database Management System (DBMS)?**

A DBMS is software that interacts with end-users, applications, and the database itself to capture and analyze data. It provides a systematic way to create, retrieve, update, and manage data in a database.

**Examples:** MySQL, Oracle, Microsoft SQL Server, PostgreSQL, SQLite, etc.

**What is a Table in a Database?**

A table is a fundamental structure in a relational database used to store data. It is organized into rows and columns where each row represents a record and each column represents an attribute of the record.

**Components of a Table:**

* **Columns:** Define the attributes or properties of the data stored in the table.
* **Rows:** Represent individual records or entries in the table.
* **Primary Key:** A unique identifier for each row in the table.
* **Foreign Key:** A column in one table that uniquely identifies a row in another table.

**Example Table: Employees**

| **Employee ID** | **Name** | **Position** | **Salary** |
| --- | --- | --- | --- |
| 1 | Alice Johnson | Manager | 60000 |
| 2 | Bob Smith | Developer | 55000 |
| 3 | Charlie Brown | Analyst | 50000 |

**What is an Entity?**

An entity is a distinct, identifiable object in a database that represents a real-world concept or thing. Entities can be physical objects, people, places, or even abstract concepts.

**Examples of Entities:**

* **In a School Database:**
  + Students
  + Teachers
  + Courses
* **In a Library Database:**
  + Books
  + Members
  + Loans

**What are Attributes?**

Attributes are the properties or characteristics that describe an entity. They represent the data fields or columns in a table that hold specific pieces of information about each instance of the entity.

**Examples of Attributes:**

* **In a Student Table:**
  + Student ID
  + Name
  + Date of Birth
  + Grade
* **In an Employee Table:**
  + Employee ID
  + Name
  + Position
  + Salary

**Data Types in SQL**

In SQL, data types define the kind of data that can be stored in a column of a table.

**Common SQL Data Types:**

* **Numeric Data Types:**
  + INT or INTEGER
  + SMALLINT
  + BIGINT
  + FLOAT
  + DOUBLE
  + DECIMAL or NUMERIC
* **Character and String Data Types:**
  + CHAR(n)
  + VARCHAR(n)
  + TEXT
* **Date and Time Data Types:**
  + DATE
  + TIME
  + DATETIME
  + TIMESTAMP
  + YEAR
* **Boolean Data Type:**
  + BOOLEAN
* **Binary Data Types:**
  + BINARY(n)
  + VARBINARY(n)
  + BLOB
* **Special Data Types:**
  + UUID
  + JSON
  + XML

**Example Table with Different Data Types: Employees**

| **Column Name** | **Data Type** | **Description** |
| --- | --- | --- |
| EmployeeID | INT | Unique identifier for the employee |
| Name | VARCHAR(100) | Employee's full name |
| BirthDate | DATE | Employee's birth date |
| HireDate | DATETIME | Date and time of hiring |
| Salary | DECIMAL(10, 2) | Employee's salary |
| IsActive | BOOLEAN | Status indicating if the employee is currently active |

**What are Field Values?**

Field values are the actual data stored in the fields (or columns) of a table. Each field value represents a specific piece of information for a particular record (or row) in the table.

**Advantages of Using a Database System Over a File System**

Database systems offer several advantages over traditional file systems for managing data.

| **Aspect** | **File System** | **Database System** |
| --- | --- | --- |
| Data Management | Data is stored in flat files | Data is organized in tables with relationships and structures |
| Data Redundancy | High risk of redundancy | Minimizes redundancy through normalization |
| Data Integrity | Limited enforcement | Enforces data integrity constraints |
| Data Security | Basic file-level security | Advanced security features |
| Data Access | Access via custom programs | Efficient access through query languages like SQL |
| Concurrency Control | Limited control | Advanced mechanisms for concurrent access and transactions |
| Backup and Recovery | Manual processes | Built-in backup and recovery solutions |
| Data Relationships | No inherent support | Supports complex relationships through foreign keys |
| Data Retrieval | Manual and inefficient | Powerful querying capabilities with SQL |
| Scalability | Difficult to scale | Designed to handle large volumes of data |
| Data Consistency | Potential inconsistencies | Ensures consistency through transaction management |
| Data Modeling | Limited support | Supports complex data modeling and schema design |
| Management Tools | Fewer tools available | Wide range of tools for management, analysis, and reporting |

**What is ACID?**

ACID stands for Atomicity, Consistency, Isolation, and Durability. These properties ensure reliable transactions in a database system.

* **Atomicity:** Ensures a transaction is treated as a single unit of work.
* **Consistency:** Guarantees a transaction brings the database from one valid state to another.
* **Isolation:** Ensures operations of a transaction are isolated from other transactions.
* **Durability:** Ensures committed transactions remain permanent even in case of system failure.

**SQL Commands**

SQL commands are categorized into five main types based on their functionality: DDL, DML, DQL, DCL, and TCL.

* **DDL (Data Definition Language):** Used to define and manage database structures.
  + CREATE, ALTER, DROP, TRUNCATE, RENAME
* **DML (Data Manipulation Language):** Used to manipulate data in tables.
  + INSERT, UPDATE, DELETE
* **DQL (Data Query Language):** Used to query data from the database.
  + SELECT
* **DCL (Data Control Language):** Used to control access to data.
  + GRANT, REVOKE
* **TCL (Transaction Control Language):** Used to manage transactions.
  + COMMIT, ROLLBACK, SAVEPOINT, RELEASE SAVEPOINT

**Note:** MySQL does not support some advanced features found in other databases, such as certain types of TRIGGER and SEQUENCE. MySQL handles auto-incrementing columns with the AUTO\_INCREMENT attribute.

**Common DDL Commands:**

1. **CREATE**:
   * **Purpose**: Used to create new database objects such as tables, indexes, or schemas.

**Example:**

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

Name VARCHAR(100),

Position VARCHAR(50),

Salary DECIMAL(10, 2)

);

1. **ALTER:**
   * **Purpose**: Used to modify the structure of an existing database object.

**Example**:

ALTER TABLE Employees

ADD BirthDate DATE;

1. **DROP:**
   * **Purpose:** Used to delete existing database objects such as tables, indexes, or schemas.

**Example:** DROP TABLE Employees;

1. **TRUNCATE:**
   * **Purpose:** Used to remove all records from a table without deleting the table itself. It is faster than the DELETE command because it does not generate individual row delete actions.

**Example**: TRUNCATE TABLE Employees;

1. **RENAME:**
   * **Purpose**: Used to rename an existing database object.

**Example:** RENAME TABLE Employees TO Staff;

### DML (Data Manipulation Language)

DML commands are used to manipulate data stored in database tables. These commands are essential for inserting, updating, deleting, and retrieving data from a database.

### Common DML Commands:

1. **INSERT**:
   * **Purpose:** Adds new records to a table.

**Example:**

INSERT INTO Employees (EmployeeID, Name, Position, Salary) VALUES (1, 'Alice Johnson', 'Manager', 60000);

1. **UPDATE**:
   * **Purpose:** Modifies existing records in a table.

**Example:**

UPDATE Employees

SET Salary = 65000

WHERE EmployeeID = 1;

1. **DELETE**:
   * **Purpose:** Removes records from a table.

**Example:**

DELETE FROM Employees WHERE EmployeeID = 1;

1. **SELECT**:

While **SELECT** is often categorized under **DQL**, it can be considered part of DML when used in conjunction with **INSERT, UPDATE, or DELETE** operations

* + **Purpose:** Retrieves data from one or more tables.

**Example:**

SELECT \* FROM Employees WHERE Salary > 50000;

### DCL (Data Control Language)

DCL commands are used to control access to data in a database. They manage permissions and ensure that users have the appropriate levels of access to the database objects and data.

### Common DCL Commands:

1. **GRANT**:
   * **Purpose:** Gives users specific privileges to database objects.

**Example:**

GRANT SELECT, INSERT, UPDATE ON Employees TO user\_name;

1. **REVOKE**:
   * **Purpose:** Removes specific privileges from users.

**Example:**

REVOKE SELECT, INSERT ON Employees FROM user\_name;

### TCL (Transaction Control Language)

TCL commands are used to manage transactions in a database. They ensure that database operations are completed successfully and maintain data integrity by allowing the user to control the transaction process.

### Common TCL Commands:

1. **COMMIT**:
   * **Purpose:** Saves all the changes made during the current transaction and makes them permanent in the database.

**Example:** COMMIT**;**

1. **ROLLBACK:**
   * **Purpose:** Undoes all the changes made during the current transaction, reverting the database to its previous state.

**Example :** ROLLBACK

1. **SAVEPOINT:**
   * **Purpose:** Sets a savepoint within a transaction, allowing the user to roll back to this specific point if needed.

**Example:** SAVEPOINT savepoint\_name;

1. **RELEASE SAVEPOINT**
   * **Purpose:** Removes a previously defined savepoint, making it no longer available for rollback.

**Example:** RELEASE SAVEPOINT savepoint\_name;

### Syntax for DDL Commands

### Creating a Table

To create a new table, use the CREATE TABLE statement. Here is the syntax:

CREATE TABLE table\_name (

column1 datatype constraints,

column2 datatype constraints,

...

columnN datatype constraints

);

**Example** :

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50),

Position VARCHAR(50),

DepartmentID INT,

BirthDate DATE,

HireDate DATE,

Salary DECIMAL(10, 2),

Email VARCHAR(100)

);

#### Renaming a Table Name

To rename an existing table, use the RENAME TABLE statement (MySQL) or ALTER TABLE RENAME TO (PostgreSQL, Oracle).

**Syntax** (MySQL):

RENAME TABLE old\_table\_name TO new\_table\_name;

**Syntax** (PostgreSQL, Oracle):

ALTER TABLE old\_table\_name RENAME TO new\_table\_name;

#### Add One Column to the Existing Table

To add a single column to an existing table, use the ALTER TABLE ADD statement.

**Syntax**:

ALTER TABLE table\_name ADD column\_name datatype constraints;

**Example**:

ALTER TABLE Employees ADD PhoneNumber VARCHAR(15);

#### Adding Multiple Columns

To add multiple columns to an existing table, use the ALTER TABLE ADD statement with multiple columns listed.

**Syntax:**

ALTER TABLE table\_name

ADD (

column1 datatype constraints,

column2 datatype constraints,

...

columnN datatype constraints

);

**Example**:

ALTER TABLE Employees

ADD (

Address VARCHAR(255),

City VARCHAR(100),

State VARCHAR(100)

);

#### Removing One Column and Multiple Columns

To remove columns from a table, use the ALTER TABLE DROP COLUMN statement. Note that some databases allow dropping multiple columns in a single statement.

**Syntax** for removing one column:

ALTER TABLE table\_name DROP COLUMN column\_name;

**Syntax for removing multiple columns (supported by some databases):**

ALTER TABLE table\_name

DROP COLUMN column1,

...,

DROP COLUMN columnN;

**Example:**

ALTER TABLE Employees DROP COLUMN PhoneNumber;

-- or

ALTER TABLE Employees

DROP COLUMN Address,

DROP COLUMN City,

DROP COLUMN State;

**Modifying Datatypes of Columns**

To modify the datatype of a column, use the ALTER TABLE MODIFY statement (MySQL) or ALTER TABLE ALTER COLUMN (PostgreSQL, SQL Server).

**Syntax** (MySQL):

ALTER TABLE table\_name MODIFY column\_name new\_datatype;

**Syntax** (PostgreSQL):

ALTER TABLE table\_name ALTER COLUMN column\_name TYPE new\_datatype;

**Syntax** (SQL Server):

ALTER TABLE table\_name ALTER COLUMN column\_name new\_datatype;

**Example**:

ALTER TABLE Employees MODIFY Salary DECIMAL(12, 2);

-- or

ALTER TABLE Employees ALTER COLUMN Salary TYPE DECIMAL(12, 2);

-- or

ALTER TABLE Employees ALTER COLUMN Salary DECIMAL(12, 2);

**Keywords and Clauses**

**WHERE Clause:**

The WHERE clause is used to filter records.

**Example:**

SELECT \* FROM Employees WHERE Salary > 50000;

**ORDER BY Clause:**

The ORDER BY clause is used to sort the result set.

**Example:**

SELECT \* FROM Employees ORDER BY Salary DESC;

**GROUP BY Clause:**

The GROUP BY clause is used to group rows sharing a property so that an aggregate function can be applied to each group.

**Example:**

SELECT Department, COUNT(\*) as NumberOfEmployees FROM Employees GROUP BY Department;

**HAVING Clause:**

The HAVING clause is used to filter records that work on aggregated data.

**Example:**

SELECT Department, COUNT(\*) as Num

`1234berOfEmployees FROM Employees GROUP BY Department HAVING COUNT(\*) > 1;

**DISTINCT Keyword:**

The DISTINCT keyword is used to return only distinct (different) values.

**Example:**

SELECT DISTINCT Department FROM Employees;

**LIMIT Clause:**

The LIMIT clause is used to specify the number of records to return.

**Example:**

SELECT \* FROM Employees LIMIT 5;

**SQL-Specific Reserved Words**

In addition to the standard SQL reserved words, MySQL has some specific reserved words:

* ANALYZE
* CROSS
* DATABASE
* DESCRIBE
* EXPLAIN
* FORCE
* IGNORE
* INDEX
* KEY
* LOCK
* OUTFILE
* PROCEDURE
* READ
* SHOW
* SQL
* STATUS
* TABLE
* WRITE

**Basic Concepts and Commands in SQL**

**Comments**

* **Single-Line Comment:** Use -- for a single-line comment.
* **Multi-Line Comment:** Use /\* \*/ for a multi-line comment.

**Example:**

-- This is a single-line comment

SELECT \* FROM Employees; /\* This is a multi-line comment \*/

**Constraints**

* **NOT NULL:** Ensures that a column cannot have a NULL value.
* **UNIQUE:** Ensures that all values in a column are different.
* **PRIMARY KEY:** A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table.
* **FOREIGN KEY:** Ensures referential integrity for a record in another table.
* **CHECK:** Ensures that the value in a column meets a specific condition.
* **DEFAULT:** Sets a default value for a column when no value is specified.

**Example:**

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

Name VARCHAR(100) NOT NULL,

Position VARCHAR(50),

Salary DECIMAL(10, 2) CHECK (Salary > 0),

HireDate DATE DEFAULT CURRENT\_DATE

);

**Indexes**

* **INDEX:** Used to create and retrieve data from the database very quickly.
* **UNIQUE INDEX:** Ensures that the indexed values are unique.

**Example:**

CREATE INDEX idx\_employee\_name ON Employees (Name);

CREATE UNIQUE INDEX idx\_unique\_employee\_id ON Employees (EmployeeID);

**Aliases**

* **Column Alias:** Renames a column in the result set.
* **Table Alias:** Renames a table in the query.

**Example:**

SELECT Name AS EmployeeName, Salary AS EmployeeSalary FROM Employees;

SELECT e.Name, e.Salary FROM Employees e;

**Conditional Statements**

* **CASE:** Provides if-then-else functionality.

**Example:**

SELECT Name,

Salary,

CASE

WHEN Salary < 50000 THEN 'Low'

WHEN Salary BETWEEN 50000 AND 100000 THEN 'Medium'

ELSE 'High'

END AS SalaryRange

FROM Employees;

**Subqueries**

* **Subqueries:** A query within another query.

**Example:**

SELECT Name FROM Employees

WHERE Salary > (SELECT AVG(Salary) FROM Employees);

**Views**

* **VIEW:** A virtual table based on the result set of an SQL statement.

**Example:**

CREATE VIEW HighSalaryEmployees AS

SELECT Name, Position, Salary

FROM Employees

WHERE Salary > 100000;

**Transactions**

* **Transaction:** A sequence of one or more SQL operations treated as a unit.
* **COMMIT:** Saves the work done.
* **ROLLBACK:** Undoes the work done.

**Example:**

START TRANSACTION;

UPDATE Employees SET Salary = Salary \* 1.10 WHERE Department = 'Sales';

COMMIT;

START TRANSACTION;

DELETE FROM Employees WHERE EmployeeID = 10;

ROLLBACK;

**Stored Procedures and Functions**

* **Stored Procedure:** A prepared SQL code that can be saved and reused.
* **Function:** Similar to a stored procedure but can return a value.

**Example:**

CREATE PROCEDURE GetEmployeeDetails (IN emp\_id INT)

BEGIN

SELECT \* FROM Employees WHERE EmployeeID = emp\_id;

END;

CREATE FUNCTION CalculateBonus (salary DECIMAL(10, 2)) RETURNS DECIMAL(10, 2)

BEGIN

RETURN salary \* 0.10;

END;

**Triggers**

* **Trigger:** A set of SQL statements that automatically execute when an event occurs in the database.

**Example:**

CREATE TRIGGER before\_employee\_insert

BEFORE INSERT ON Employees

FOR EACH ROW

BEGIN

SET NEW.HireDate = CURRENT\_DATE;

END;

## **SQL Built-in Functions**

SQL provides a variety of built-in functions to manipulate string and character data. Here are some of the commonly used string functions along with examples.

* String/Char Functions
* Numeric/Math Functions
* Date / Time functions
* Aggregate Functions
* Conversion Functions
* Analytical Functions

### 1. String/Char Functions

**1.1. ASCII**

* **Explanation:** Returns the ASCII code of the first character of a string.
* **Syntax:**

SELECT ASCII(string) AS ASCII\_Value;

* **Example:**

SELECT ASCII('Hello') AS ASCII\_Value;

* **Output:**

ASCII\_Value

------------

72

**1.2. CHR**

* **Explanation:** Returns the character that corresponds to the given ASCII code.
* **Syntax:**

SELECT CHR(ascii\_code) AS Character;

* **Example:**

SELECT CHR(72) AS Character;

* **Output:**

Character

----------

**1.3. CONCAT (||)**

* **Explanation:** Concatenates two or more strings together.
* **Syntax:**

SELECT CONCAT(string1, string2) AS Concatenated\_String;

or

SELECT string1 || string2 AS Concatenated\_String;

* **Example:**

SELECT CONCAT('Hello', ' World') AS Concatenated\_String;

SELECT 'Hello' || ' World' AS Concatenated\_String;

* **Output:**

Concatenated\_String

-------------------

Hello World

**1.4. DUMP**

* **Explanation:** Returns a string that contains the datatype code, length in bytes, and internal representation of a value.
* **Syntax:**

SELECT DUMP(expression) AS Dump\_Value;

* **Example:**

SELECT DUMP('Hello') AS Dump\_Value;

* **Output:**

Dump\_Value

------------------------

Type=96 Len=5: 72,101,108,108,111

**1.5. INITCAP**

* **Explanation:** Returns a string with the first letter of each word capitalized.
* **Syntax:**

SELECT INITCAP(string) AS InitCap\_String;

* **Example:**

SELECT INITCAP('hello world') AS InitCap\_String;

* **Output:**

InitCap\_String

--------------

Hello World

**1.6. INSTR**

* **Explanation:** Returns the position of the first occurrence of a substring in a string.
* **Syntax:**

SELECT INSTR(string, substring, [start\_position], [nth\_occurrence]) AS Position;

* **Example:**

SELECT INSTR('Hello World', 'World') AS Position;

* **Output:**

Position

--------

7

**1.7. LENGTH**

* **Explanation:** Returns the length of a string.
* **Syntax:**

SELECT LENGTH(string) AS String\_Length;

* **Example:**

SELECT LENGTH('Hello World') AS String\_Length;

* **Output:**

String\_Length

-------------

11

**1.8. LOWER**

* **Explanation:** Converts a string to lowercase.
* **Syntax:**

SELECT LOWER(string) AS Lowercase\_String;

* **Example:**

SELECT LOWER('Hello World') AS Lowercase\_String;

* **Output:**

Lowercase\_String

----------------

hello world

**1.9. UPPER**

* **Explanation:** Converts a string to uppercase.
* **Syntax:**

SELECT UPPER(string) AS Uppercase\_String;

* **Example:**

SELECT UPPER('Hello World') AS Uppercase\_String;

* **Output:**

Uppercase\_String

----------------

HELLO WORLD

**1.10. LPAD**

* **Explanation:** Pads the left side of a string with a specified set of characters.
* **Syntax:**

SELECT LPAD(string, length, pad\_string) AS Padded\_String;

* **Example:**

SELECT LPAD('Hello', 10, '\*') AS Padded\_String;

* **Output:**

Padded\_String

-------------

\*\*\*\*\*Hello

**1.11. RPAD**

* **Explanation:** Pads the right side of a string with a specified set of characters.
* **Syntax:**

SELECT RPAD(string, length, pad\_string) AS Padded\_String;

* **Example:**

SELECT RPAD('Hello', 10, '\*') AS Padded\_String;

* **Output:**

Padded\_String

-------------

Hello\*\*\*\*\*

**1.12. LTRIM**

* **Explanation:** Removes specified characters from the left side of a string.
* **Syntax:**

SELECT LTRIM(string, trim\_string) AS Trimmed\_String;

* **Example:**

SELECT LTRIM('\*\*\*\*\*Hello', '\*') AS Trimmed\_String;

* **Output:**

Trimmed\_String

--------------

Hello

**1.13. RTRIM**

* **Explanation:** Removes specified characters from the right side of a string.
* **Syntax:**

SELECT RTRIM(string, trim\_string) AS Trimmed\_String;

* **Example:**

SELECT RTRIM('Hello\*\*\*\*\*', '\*') AS Trimmed\_String;

* **Output:**

Trimmed\_String

--------------

Hello

**1.14. REPLACE**

* **Explanation:** Replaces all occurrences of a substring within a string with a new substring.
* **Syntax:**

SELECT REPLACE(string, old\_substring, new\_substring) AS Replaced\_String;

* **Example:**

SELECT REPLACE('Hello World', 'World', 'SQL') AS Replaced\_String;

* **Output:**

Replaced\_String

---------------

Hello SQL

**1.15. SUBSTR**

* **Explanation:** Extracts a substring from a string.
* **Syntax:**

SELECT SUBSTR(string, start\_position, [length]) AS Substring;

* **Example:**

SELECT SUBSTR('Hello World', 7, 5) AS Substring;

* **Output:**

Substring

---------

World

**1.16. TRANSLATE**

* **Explanation:** Replaces each character in a string with another character based on a translation map.
* **Syntax:**

SELECT TRANSLATE(string, from\_string, to\_string) AS Translated\_String;

* **Example:**

SELECT TRANSLATE('HELLO', 'HELO', '1234') AS Translated\_String;

* **Output:**

Translated\_String

-----------------

12334

### 2. Numeric/Math Functions

**2.1. ABS**

* **Explanation:** Returns the absolute value of a number.
* **Syntax:**

SELECT ABS(number) AS Absolute\_Value;

* **Example:**

SELECT ABS(-10) AS Absolute\_Value;

* **Output:**

Absolute\_Value

--------------

10

**2.2. CEIL / CEILING**

* **Explanation:** Returns the smallest integer greater than or equal to a number.
* **Syntax:**

SELECT CEIL(number) AS Ceiling\_Value;

* **Example:**

SELECT CEIL(4.3) AS Ceiling\_Value;

* **Output:**

Ceiling\_Value

-------------

5

**2.3. FLOOR**

* **Explanation:** Returns the largest integer less than or equal to a number.
* **Syntax:**

SELECT FLOOR(number) AS Floor\_Value;

* **Example:**

SELECT FLOOR(4.6) AS Floor\_Value;

* **Output:**

Floor\_Value

-----------

4

**2.4. ROUND**

* **Syntax:**

SELECT ROUND(number, decimal\_places) AS Rounded\_Value;

* **Example:**

SELECT ROUND(4.567, 2) AS Rounded\_Value;

* **Output:**

Rounded\_Value

-------------

4.57

**2.5. MOD**

* **Explanation:** Returns the remainder of a division operation.
* **Syntax:**

SELECT MOD(number1, number2) AS Remainder;

* **Example:**

SELECT MOD(10, 3) AS Remainder;

* **Output:**

Remainder

---------

1

**2.6. POWER**

* **Explanation:** Returns a number raised to the power of another number.
* **Syntax:**

SELECT POWER(number, exponent) AS Power\_Value;

* **Example:**

SELECT POWER(2, 3) AS Power\_Value;

* **Output:**

Power\_Value

-----------

8

**2.7. SQRT**

* **Explanation:** Returns the square root of a number.
* **Syntax:**

SELECT SQRT(number) AS Square\_Root;

* **Example:**

SELECT SQRT(16) AS Square\_Root;

* **Output:**

Square\_Root

------------

4

**2.8. LOG**

* **Explanation:** Returns the natural logarithm of a number.
* **Syntax:**

SELECT LOG(number) AS Log\_Value;

* **Example:**

SELECT LOG(2.718281828459045) AS Log\_Value;

* **Output:**

Log\_Value

---------

1

**2.9. PI**

* **Explanation:** Returns the value of π (pi).
* **Syntax:**

SELECT PI() AS Pi\_Value;

* **Example:**

SELECT PI() AS Pi\_Value;

* **Output:**

Pi\_Value

---------

3.141592653589793

### 3. Date/Time Functions

**3.1. SYSTIMESTAMP**

* **Explanation:** Returns the current system date and time with time zone.
* **Syntax:**

SELECT SYSTIMESTAMP AS Current\_Timestamp;

* **Example:**

SELECT SYSTIMESTAMP AS Current\_Timestamp;

* **Output:**

Current\_Timestamp

--------------------------

26-JUL-24 02:25:12.000000 PM +00:00

**3.2. CURRENT\_DATE**

* **Explanation:** Returns the current date in the session time zone.
* **Syntax:**

SELECT CURRENT\_DATE AS Current\_Date;

* **Example:**

SELECT CURRENT\_DATE AS Current\_Date;

* **Output:**

Current\_Date

------------

26-JUL-24

**3.3. CURRENT\_TIMESTAMP**

* **Explanation:** Returns the current date and time in the session time zone.
* **Syntax:**

SELECT CURRENT\_TIMESTAMP AS Current\_Timestamp;

* **Example:**

SELECT CURRENT\_TIMESTAMP AS Current\_Timestamp;

* **Output:**

Current\_Timestamp

--------------------------

26-JUL-24 02:25:12.000000 PM

**3.4. EXTRACT**

* **Explanation:** Extracts and returns a specified part (year, month, day, hour, etc.) from a date or timestamp.
* **Syntax:**

SELECT EXTRACT(part FROM date) AS Extracted\_Value;

* **Example:**

SELECT EXTRACT(YEAR FROM SYSDATE) AS Year;

* **Output:**

Year

----

2024

### 4. Aggregate Functions

**4.1. COUNT**

* **Explanation:** Returns the number of rows in a set.
* **Syntax:**

SELECT COUNT(column\_name) AS Count\_Value FROM table\_name;

* **Example:**

SELECT COUNT(\*) AS Total\_Records FROM employees;

* **Output:**

Total\_Records

-------------

150

**4.2. SUM**

* **Explanation:** Returns the sum of values in a numeric column.
* **Syntax:**

SELECT SUM(column\_name) AS Total\_Sum FROM table\_name;

* **Example:**

SELECT SUM(salary) AS Total\_Salary FROM employees;

* **Output:**

Total\_Salary

-------------

1200000

**4.3. AVG**

* **Explanation:** Returns the average value of a numeric column.
* **Syntax:**

SELECT AVG(column\_name) AS Average\_Value FROM table\_name;

* **Example:**

SELECT AVG(salary) AS Average\_Salary FROM employees;

* **Output:**

Average\_Salary

---------------

8000

**4.4. MIN**

* **Explanation:** Returns the minimum value in a set.
* **Syntax:**

SELECT MIN(column\_name) AS Minimum\_Value FROM table\_name;

* **Example:**

SELECT MIN(salary) AS Lowest\_Salary FROM employees;

* **Output:**

Lowest\_Salary

-------------

2500

**4.5. MAX**

* **Explanation:** Returns the maximum value in a set.
* **Syntax:**

SELECT MAX(column\_name) AS Maximum\_Value FROM table\_name;

* **Example:**

SELECT MAX(salary) AS Highest\_Salary FROM employees;

* **Output:**

Highest\_Salary

--------------

15000

**5. Conversion Functions**

**5.1. TO\_CHAR Equivalent**

In MySQL, you can use the DATE\_FORMAT() function to format dates as strings and the CAST() or CONVERT() function to convert other types to strings.

* **Date Formatting:**

SELECT DATE\_FORMAT(NOW(), '%Y-%m-%d %H:%i:%s') AS FormattedDate;

* **Number to String Conversion:**

SELECT CAST(12345 AS CHAR) AS StringValue;

**5.2. TO\_NUMBER Equivalent**

MySQL does not have a direct TO\_NUMBER function, but you can use CAST() or CONVERT() to convert strings to numbers.

* **String to Number Conversion:**

SELECT CAST('12345' AS UNSIGNED) AS NumberValue;

-- Converts to integer

SELECT CAST('123.45' AS DECIMAL(5,2)) AS DecimalValue;

-- Converts to decimal

**5.3. TO\_DATE Equivalent**

In MySQL, you can use the STR\_TO\_DATE() function to convert strings to dates.

* **String to Date Conversion:**

SELECT STR\_TO\_DATE('2023-07-24', '%Y-%m-%d') AS DateValue;

**5.4. CAST Function**

The CAST() function in MySQL can be used to convert values from one data type to another.

* **General Syntax:**

CAST(value AS data\_type)

* **Examples:**

-- String to Integer

SELECT CAST('12345' AS UNSIGNED) AS NumberValue;

-- String to Decimal

SELECT CAST('123.45' AS DECIMAL(5,2)) AS DecimalValue;

-- Date to String

SELECT CAST(NOW() AS CHAR) AS StringDate;

-- String to Date

SELECT CAST('2023-07-24' AS DATE) AS DateValue;

**Examples Using Sample Data**

SELECT EmployeeID, DATE\_FORMAT(HireDate, '%W, %d %M %Y') AS FormattedHireDate FROM employees;

**6. Analytical Funcation**

**6.1. LISTAGG Equivalent: GROUP\_CONCAT**

The GROUP\_CONCAT() function in MySQL concatenates values from multiple rows into a single string result.

**Syntax:**

GROUP\_CONCAT(expression ORDER BY expression SEPARATOR separator)

**Example**:

SELECT Department, GROUP\_CONCAT(EmployeeName ORDER BY EmployeeName SEPARATOR ', ') AS EmployeeNames

FROM employees

GROUP BY Department;

### ****6.2. RANK****

The RANK() function assigns a rank to each row within a partition of a result set, with gaps in the ranking where there are ties.

**Syntax:**

RANK() OVER (PARTITION BY expression ORDER BY expression)

**Example:**

SELECT EmployeeID, Department, Salary,

RANK() OVER (PARTITION BY Department ORDER BY Salary DESC) AS Rank

FROM employees;

### 4. LEAD

The LEAD() function provides access to a row at a specified physical offset following the current row.

**Syntax:**

LEAD(expression, offset, default) OVER (PARTITION BY expression ORDER BY expression)

**Example:**

SELECT EmployeeID, Department, Salary,

LEAD(Salary, 1, 0) OVER (PARTITION BY Department ORDER BY Salary) AS NextSalary

FROM employees;

### 5. ****LAG****

The LAG() function provides access to a row at a specified physical offset before the current row.

**Syntax:**

LAG(expression, offset, default) OVER (PARTITION BY expression ORDER BY expression)

**Example:**

SELECT EmployeeID, Department, Salary,

LAG(Salary, 1, 0) OVER (PARTITION BY Department ORDER BY Salary) AS PreviousSalary

FROM employees;

**SQL Operators: LIKE, Wildcard, IN, BETWEEN**

#### 1. LIKE Operator

The LIKE operator is used to search for a specified pattern in a column. It's often used with wildcards.

##### Wildcards:

* % : Represents zero, one, or multiple characters.
* \_ : Represents a single character.

**Example:**

SELECT \* FROM employees WHERE EmployeeName LIKE 'A%'; -- Names starting with 'A'

SELECT \* FROM employees WHERE EmployeeName LIKE '%n'; -- Names ending with 'n'

SELECT \* FROM employees WHERE EmployeeName LIKE '\_a%'; -- Names with 'a' as the second character

#### 2. IN Operator

The IN operator allows you to specify multiple values in a WHERE clause. It is used to reduce the need for multiple OR conditions.

**Example:**

SELECT \* FROM employees WHERE Department IN ('HR', 'IT');

-- Equivalent to:

-- SELECT \* FROM employees WHERE Department = 'HR' OR Department = 'IT';

#### 3. BETWEEN Operator

The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates.

**Example:**

SELECT \* FROM employees WHERE Salary BETWEEN 50000 AND 70000;

SELECT \* FROM employees WHERE HireDate BETWEEN '2023-01-01' AND '2023-12-31';

### Using LIKE, Wildcards, IN, and BETWEEN with Examples

### LIKE Operator Example

-- Names starting with 'A'

SELECT \* FROM employees WHERE EmployeeName LIKE 'A%';

-- Output:

-- | EmployeeID | EmployeeName | Department | Salary | HireDate |

-- |------------|--------------|------------|----------|------------|

-- | 1 | Alice | HR | 50000.00 | 2023-07-01 |

### IN Operator Example

-- Employees in HR or IT department

SELECT \* FROM employees WHERE Department IN ('HR', 'IT');

-- Output:

-- | EmployeeID | EmployeeName | Department | Salary | HireDate |

-- |------------|--------------|------------|----------|------------|

-- | 1 | Alice | HR | 50000.00 | 2023-07-01 |

-- | 2 | Bob | HR | 60000.00 | 2023-01-15 |

-- | 3 | Charlie | IT | 55000.00 | 2024-03-22 |

-- | 4 | David | IT | 70000.00 | 2024-07-24 |

-- | 5 | Eve | HR | 75000.00 | 2022-12-10 |

-- | 6 | Frank | IT | 65000.00 | 2024-05-15 |

### BETWEEN Operator Example

-- Employees with salary between 50000 and 70000

SELECT \* FROM employees WHERE Salary BETWEEN 50000 AND 70000;

-- Output:

-- | EmployeeID | EmployeeName | Department | Salary | HireDate |

-- |------------|--------------|------------|----------|------------|

-- | 1 | Alice | HR | 50000.00 | 2023-07-01 |

-- | 2 | Bob | HR | 60000.00 | 2023-01-15 |

-- | 3 | Charlie | IT | 55000.00 | 2024-03-22 |

-- | 6 | Frank | IT | 65000.00 | 2024-05-15 |

### 8. SQL Joins

Joins are used in SQL to combine rows from two or more tables based on a related column between them. There are several types of joins, each serving different purposes.

#### SQL INNER JOIN  SQL LEFT JOIN  SQL RIGHT JOIN  SQL FULL OUTER JOIN

#### 8.1 Inner Join

An INNER JOIN returns records that have matching values in both tables.

**Example:**

Let's assume we have two tables, employees and departments.

**Query:**

SELECT employees.EmployeeID, employees.EmployeeName, departments.DepartmentName

FROM employees

INNER JOIN departments ON employees.DepartmentID = departments.DepartmentID;

**Result:**

| **EmployeeID** | **EmployeeName** | **DepartmentName** |
| --- | --- | --- |
| 1 | Alice | HR |
| 2 | Bob | HR |
| 3 | Charlie | IT |
| 4 | David | Finance |

#### 8.2 Left Join

A LEFT JOIN (or LEFT OUTER JOIN) returns all records from the left table (table1), and the matched records from the right table (table2). The result is NULL from the right side if there is no match.

**Query:**

SELECT employees.EmployeeID, employees.EmployeeName, hjvhdepartments.DepartmentName

FROM employees

LEFT JOIN departments ON employees.DepartmentID = departments.DepartmentID;

**Result:**

| **EmployeeID** | **EmployeeName** | **DepartmentName** |
| --- | --- | --- |
| 1 | Alice | HR |
| 2 | Bob | HR |
| 3 | Charlie | IT |
| 4 | David | Finance |

#### 8.3 Right Join

A RIGHT JOIN (or RIGHT OUTER JOIN) returns all records from the right table (table2), and the matched records from the left table (table1). The result is NULL from the left side when there is no match.

**Query:**

SELECT employees.EmployeeID, employees.EmployeeName, departments.DepartmentName

FROM employees

RIGHT JOIN departments ON employees.DepartmentID = departments.DepartmentID;

**Result:**

| **EmployeeID** | **EmployeeName** | **DepartmentName** |
| --- | --- | --- |
| 1 | Alice | HR |
| 2 | Bob | HR |
| 3 | Charlie | IT |
| 4 | David | Finance |
| NULL | NULL | Sales |

#### 8.4 Full Join

A FULL JOIN (or FULL OUTER JOIN) returns all records when there is a match in either left (table1) or right (table2) table records. Unfortunately, MySQL does not support FULL JOIN directly. However, you can achieve this by combining LEFT JOIN and RIGHT JOIN using UNION.

**Query:**

SELECT employees.EmployeeID, employees.EmployeeName, departments.DepartmentName

FROM employees

LEFT JOIN departments ON employees.DepartmentID = departments.DepartmentID

UNION

SELECT employees.EmployeeID, employees.EmployeeName, departments.DepartmentName

FROM employees

RIGHT JOIN departments ON employees.DepartmentID = departments.DepartmentID;

**Result:**

| **EmployeeID** | **EmployeeName** | **DepartmentName** |
| --- | --- | --- |
| 1 | Alice | HR |
| 2 | Bob | HR |
| 3 | Charlie | IT |
| 4 | David | Finance |
| NULL | NULL | Sales |

### Union

The UNION operator is used to combine the result set of two or more SELECT statements. Each SELECT statement within the UNION must have the same number of columns in the result sets with similar data types.

**Example:**

SELECT EmployeeID, EmployeeName, DepartmentID FROM employees

UNION

SELECT DepartmentID, DepartmentName, NULL FROM departments;

**Result:**

| **EmployeeID** | **EmployeeName** | **DepartmentID** |
| --- | --- | --- |
| 1 | Alice | 1 |
| 2 | Bob | 1 |
| 3 | Charlie | 2 |
| 4 | David | 3 |
| 1 | HR | NULL |
| 2 | IT | NULL |
| 3 | Finance | NULL |

### Summary

* **INNER JOIN**: Returns records that have matching values in both tables.
* **LEFT JOIN**: Returns all records from the left table, and the matched records from the right table. Result is NULL from the right side if there is no match.
* **RIGHT JOIN**: Returns all records from the right table, and the matched records from the left table. Result is NULL from the left side if there is no match.
* **FULL JOIN**: Returns all records when there is a match in either left or right table records.
* **UNION**: Combines the result sets of two or more SELECT statements into a single result set.

**Difference Between FULL JOIN and CROSS JOIN in MySQL**

In MySQL, FULL JOIN and CROSS JOIN serve different purposes and produce different results. Here's a tabular comparison to illustrate their differences:

| **Feature** | **FULL JOIN (FULL OUTER JOIN)** | **CROSS JOIN** |
| --- | --- | --- |
| **Definition** | Combines results of both LEFT JOIN and RIGHT JOIN, returns all records from both tables with matched records where available. If there is no match, the result is NULL from the side with no match. | Returns the Cartesian product of the two tables, meaning every row from the first table is combined with every row from the second table. |
| **MySQL Support** | Not directly supported in MySQL. It can be emulated using UNION of LEFT JOIN and RIGHT JOIN. | Directly supported in MySQL. |
| **Usage** | Used when you need all rows from both tables, regardless of matching rows. | Used when you need all possible combinations of rows from both tables. |
| **Result Rows** | Number of rows is the maximum of the number of rows in the two tables. | Number of rows is the product of the number of rows in the two tables. |
| **NULL Values** | Contains NULLs for unmatched rows. | Does not contain NULLs for unmatched rows because every possible combination is included. |
| **Example Query** | SELECT \* FROM table1 LEFT JOIN table2 ON table1.id = table2.id UNION SELECT \* FROM table1 RIGHT JOIN table2 ON table1.id = table2.id; | SELECT \* FROM table1 CROSS JOIN table2; |
| **Example Output** | Combines all rows from both tables, matching rows together and filling with NULLs where there are no matches. | Combines every row of the first table with every row of the second table, creating a large result set. |

**Examples**

**FULL JOIN Example**

Let's say we have two tables, employees and departments.

**Employees Table:**

| **EmployeeID** | **EmployeeName** | **DepartmentID** |
| --- | --- | --- |
| 1 | Alice | 1 |
| 2 | Bob | 2 |
| 3 | Charlie | 3 |

**Departments Table:**

| **DepartmentID** | **DepartmentName** |
| --- | --- |
| 1 | HR |
| 2 | IT |
| 4 | Marketing |

**FULL JOIN Emulation in MySQL:**

SELECT employees.EmployeeID, employees.EmployeeName, departments.DepartmentID, departments.DepartmentName

FROM employees

LEFT JOIN departments ON employees.DepartmentID = departments.DepartmentID

UNION

SELECT employees.EmployeeID, employees.EmployeeName, departments.DepartmentID, departments.DepartmentName

FROM employees

RIGHT JOIN departments ON employees.DepartmentID = departments.DepartmentID;

**Result:**

| **EmployeeID** | **EmployeeName** | **DepartmentID** | **DepartmentName** |
| --- | --- | --- | --- |
| 1 | Alice | 1 | HR |
| 2 | Bob | 2 | IT |
| 3 | Charlie | 3 | NULL |
| NULL | NULL | 4 | Marketing |

**CROSS JOIN Example**

**Employees Table:**

| **EmployeeID** | **EmployeeName** | **DepartmentID** |
| --- | --- | --- |
| 1 | Alice | 1 |
| 2 | Bob | 2 |
| 3 | Charlie | 3 |

**Departments Table:**

| **DepartmentID** | **DepartmentName** |
| --- | --- |
| 1 | HR |
| 2 | IT |
| 4 | Marketing |

**CROSS JOIN Query:**

SELECT employees.EmployeeID, employees.EmployeeName, departments.DepartmentID, departments.DepartmentName

FROM employees

CROSS JOIN departments;

**Result:**

| **EmployeeID** | **EmployeeName** | **DepartmentID** | **DepartmentName** |
| --- | --- | --- | --- |
| 1 | Alice | 1 | HR |
| 1 | Alice | 2 | IT |
| 1 | Alice | 4 | Marketing |
| 2 | Bob | 1 | HR |
| 2 | Bob | 2 | IT |
| 2 | Bob | 4 | Marketing |
| 3 | Charlie | 1 | HR |
| 3 | Charlie | 2 | IT |
| 3 | Charlie | 4 | Marketing |

**Summary**

* **FULL JOIN** combines rows from both tables, including all records from both sides with NULLs where there are no matches. Not directly supported in MySQL, needs to be emulated.
* **CROSS JOIN** produces a Cartesian product of both tables, resulting in all possible combinations of rows. Directly supported in MySQL.

**Views**

A view in SQL is a virtual table based on the result set of an SQL query. It contains rows and columns, just like a real table, but it does not store the data itself. Instead, it dynamically retrieves data from one or more tables whenever it is queried. Views can simplify complex queries, improve security by limiting data access, and provide a level of abstraction.

**Advantages of Using Views**

* **Simplify Complex Queries**: By encapsulating complex joins, filters, and calculations in a view, you can simplify SQL queries.
* **Security**: Restrict access to specific rows or columns.
* **Consistency**: Ensure that a complex query logic is used consistently across the application.
* **Maintainability**: Easier to update the query logic in one place (the view) instead of multiple locations in your application.

**Creating a View**

You can create a view using the CREATE VIEW statement followed by the view name and the AS keyword with the query.

**Syntax**:

CREATE VIEW view\_name AS

SELECT columns

FROM tables

WHERE conditions;

**Example:** Let's create a view to show employees along with their department names.

CREATE VIEW employee\_department AS

SELECT employees.EmployeeID, employees.EmployeeName, departments.DepartmentName

FROM employees

JOIN departments ON employees.DepartmentID = departments.DepartmentID;

**Querying a View**

Querying a view is similar to querying a regular table.

**Example:**

SELECT \* FROM employee\_department;

**Result:**

| **EmployeeID** | **EmployeeName** | **DepartmentName** |
| --- | --- | --- |
| 1 | Alice | HR |
| 2 | Bob | IT |
| 3 | Charlie | Finance |

**Updating a View**

Views can be updated using the ALTER VIEW statement.

**Syntax:**

ALTER VIEW view\_name AS

SELECT columns

FROM tables

WHERE conditions;

**Example:**

ALTER VIEW employee\_department AS

SELECT employees.EmployeeID, employees.EmployeeName, departments.DepartmentName, employees.Salary

FROM employees

JOIN departments ON employees.DepartmentID = departments.DepartmentID;

**Dropping a View**

You can drop a view using the DROP VIEW statement.

**Syntax:**

DROP VIEW view\_name;

**Example:**

DROP VIEW employee\_department;

**Read-Only Views**

You can create a read-only view by using the WITH READ ONLY option. This ensures that no INSERT, UPDATE, or DELETE operations can be performed on the view.

**Example:**

CREATE VIEW employee\_department\_readonly AS

SELECT employees.EmployeeID, employees.EmployeeName, departments.DepartmentName

FROM employees

JOIN departments ON employees.DepartmentID = departments.DepartmentID

WITH READ ONLY;

**Example Usage**

Let's assume we have the following employees and departments tables:

CREATE TABLE employees (

EmployeeID INT PRIMARY KEY,

EmployeeName VARCHAR(100),

DepartmentID INT,

Salary DECIMAL(10, 2),

HireDate DATE

);

CREATE TABLE departments (

DepartmentID INT PRIMARY KEY,

DepartmentName VARCHAR(100)

);

INSERT INTO employees (EmployeeID, EmployeeName, DepartmentID, Salary, HireDate) VALUES

(1, 'Alice', 1, 50000.00, '2023-07-01'),

(2, 'Bob', 2, 60000.00, '2023-01-15'),

(3, 'Charlie', 3, 55000.00, '2024-03-22');

INSERT INTO departments (DepartmentID, DepartmentName) VALUES

(1, 'HR'),

(2, 'IT'),

(3, 'Finance');

**Creating a View:**

CREATE VIEW employee\_department AS

SELECT employees.EmployeeID, employees.EmployeeName, departments.DepartmentName, employees.Salary

FROM employees

JOIN departments ON employees.DepartmentID = departments.DepartmentID;

**Querying the View:**

SELECT \* FROM employee\_department;

**Result:**

| **EmployeeID** | **EmployeeName** | **DepartmentName** | **Salary** |
| --- | --- | --- | --- |
| 1 | Alice | HR | 50000.00 |
| 2 | Bob | IT | 60000.00 |
| 3 | Charlie | Finance | 55000.00 |

**Updating the View:**

ALTER VIEW employee\_department AS

SELECT employees.EmployeeID, employees.EmployeeName, departments.DepartmentName, employees.Salary, employees.HireDate

FROM employees

JOIN departments ON employees.DepartmentID = departments.DepartmentID;

**Dropping the View:**

DROP VIEW employee\_department;

**Creating a Read-Only View:**

CREATE VIEW employee\_department\_readonly AS

SELECT employees.EmployeeID, employees.EmployeeName, departments.DepartmentName

FROM employees

JOIN departments ON employees.DepartmentID = departments.DepartmentID

WITH READ ONLY;

#### 

#### Difference Between Normal View and Materialized View and Advantages of Views

| **Feature** | **Normal View** | **Materialized View** |
| --- | --- | --- |
| **Data Storage** | Does not store data physically | Stores data physically |
| **Performance** | Slow if the underlying query is complex | Faster retrieval of data |
| **Refresh** | Always retrieves the latest data | Needs to be refreshed |
| **Use Case** | Suitable for dynamic, real-time data | Suitable for static or infrequently changing data |
| **Creation** | Simple and quick to create | More complex, involves additional storage |

**Indexes in MySQL/SQL**

**Why Indexes?**

Indexes are special data structures that improve the speed of data retrieval operations on a database table. They work similarly to the index in a book, allowing quick access to the rows in a table. Indexes significantly enhance the performance of database queries, especially for large datasets.

**Types of Indexes**

1. **Normal Index (Non-Unique)**
2. **Unique Index**
3. **Primary Key Index**
4. **Composite Index**
5. **Full-Text Index**

**Syntax for Creating Indexes**

**Creating a Normal Index**

**Syntax:**

CREATE INDEX index\_name ON table\_name(column\_name);

**Example:**

CREATE INDEX idx\_employee\_name ON employees(EmployeeName);

**Creating a Unique Index**

**Syntax:**

CREATE UNIQUE INDEX index\_name ON table\_name(column\_name);

**Example:**

CREATE UNIQUE INDEX idx\_employee\_email ON employees(Email);

**How to Access Index**

Indexes are used implicitly by the database management system (DBMS) when executing queries. You don't need to do anything special to use them, other than ensuring they exist and are correctly defined.

**Why Using Index in Queries**

Indexes are used to:

1. **Improve Query Performance**: By providing a quick lookup path to the data.
2. **Speed Up Search Operations**: Especially for large datasets.
3. **Enhance Sorting and Grouping**: Queries with ORDER BY and GROUP BY clauses benefit from indexes.
4. **Enforce Uniqueness**: Unique indexes ensure that all values in a column are unique.

**How Index is Helping in Real World Data**

In real-world applications, indexes help to:

* **Quickly retrieve customer data** from a large customer database.
* **Speed up search queries** on large text fields like product descriptions.
* **Enhance performance** of analytical queries on big data.

**Checking Performance Through Execution Plan**

An execution plan shows how the DBMS executes a query and uses indexes. You can analyze execution plans to understand and optimize query performance.

**Analyzing Execution Plan in MySQL**

**Syntax:**

EXPLAIN SELECT \* FROM table\_name WHERE condition;

**Example:**

EXPLAIN SELECT \* FROM employees WHERE EmployeeName = 'John Doe';

**Sample Output:**

+----+-------------+-----------+------------+------+---------------+------+---------+------+-------+----------+-------+

| id | select\_type | table | partitions | type | possible\_keys | key | key\_len | ref | rows | filtered | Extra |

+----+-------------+-----------+------------+------+---------------+------+---------+------+-------+----------+-------+

| 1 | SIMPLE | employees | NULL | ref | idx\_emp\_name | idx\_emp\_name | 402 | const | 1 | 100.00 | NULL |

+----+-------------+-----------+------------+------+---------------+------+---------+------+-------+----------+-------+

**Explanation of Columns:**

* **id**: The identifier of the SELECT.
* **select\_type**: The type of SELECT.
* **table**: The table accessed by the query.
* **type**: The join type.
* **possible\_keys**: The possible indexes the query could use.
* **key**: The actual index used by the query.
* **key\_len**: The length of the key used.
* **ref**: The columns or constants used to select rows.
* **rows**: The number of rows MySQL estimates it needs to examine.
* **filtered**: The percentage of rows filtered by the condition.
* **Extra**: Additional information about the query.

**Subqueries and Correlated Queries**

Subqueries and correlated queries are powerful tools in SQL that allow you to perform complex data retrieval operations. They enable you to nest queries within other queries, making it possible to perform more advanced and precise data manipulations.

**Examples of Subqueries**

**Subquery (Nested Query)**: A subquery is a query nested inside another query. The subquery executes first, and its result is used by the outer query.

**Types of Subqueries:**

* **Single-row subquery**: Returns a single row.
* **Multi-row subquery**: Returns multiple rows.
* **Scalar subquery**: Returns a single value.
* **IN subquery**: Used with the IN operator to compare a value against a set of values.

**Example 1: Single-Row Subquery** Find the employees who have a salary higher than the average salary in the company.

SELECT EmployeeName, Salary

FROM employees

WHERE Salary > (SELECT AVG(Salary) FROM employees);

**Example 2: Scalar Subquery** Find the employee with the highest salary.

SELECT EmployeeName, Salary

FROM employees

WHERE Salary = (SELECT MAX(Salary) FROM employees);

**Examples of Correlated Queries**

**Correlated Query**: A correlated query is a subquery that references columns from the outer query. It executes once for each row selected by the outer query.

**Example 1: Find Employees Who Earn More Than the Average Salary in Their Department** This query uses a correlated subquery to calculate the average salary for each department and compare it to the salary of each employee.

SELECT EmployeeName, Salary, DepartmentID

FROM employees e1

WHERE Salary > (SELECT AVG(Salary)

FROM employees e2

WHERE e1.DepartmentID = e2.DepartmentID);

**Explanation**:

* The subquery (SELECT AVG(Salary) FROM employees e2 WHERE e1.DepartmentID = e2.DepartmentID) calculates the average salary for the department of the current employee (e1).
* The outer query (e1) compares each employee’s salary to this average.

**Example 2: List All Employees Along with Their Rank Within Their Department Based on Salary** This example ranks employees within their departments based on salary.

SELECT EmployeeName, Salary, DepartmentID,

(SELECT COUNT(\*)

FROM employees e2

WHERE e2.Salary > e1.Salary AND e2.DepartmentID = e1.DepartmentID) + 1 AS Rank

FROM employees e1

ORDER BY DepartmentID, Rank;

**Explanation**:

* The subquery counts the number of employees in the same department with a higher salary than the current employee.
* The rank is determined by adding 1 to this count.

**Example 3: Find Departments Where All Employees Earn More Than $5000** This query checks if all employees in a department earn more than $5000.

SELECT DepartmentID

FROM departments d

WHERE NOT EXISTS (SELECT \*

FROM employees e

WHERE e.DepartmentID = d.DepartmentID AND e.Salary <= 5000);

**Explanation**:

* The subquery checks if there are any employees in the department with a salary less than or equal to $5000.
* The NOT EXISTS condition ensures that only departments where all employees earn more than $5000 are selected.

**Common Table Expressions (CTE) in SQL**

A **Common Table Expression (CTE)** is a temporary result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement. CTEs are particularly useful for improving the readability and organization of complex queries.

**Syntax of a CTE**

WITH cte\_name AS (

-- CTE query

SELECT column1, column2, ...

FROM table\_name

WHERE condition

)

-- Main query using the CTE

SELECT column1, column2, ...

FROM cte\_name

WHERE condition;

**Key Points About CTEs:**

1. **Temporary Scope**: CTEs exist only for the duration of the query and are not stored in the database.
2. **Modularity**: CTEs allow you to break down complex queries into more manageable pieces.
3. **Recursiveness**: CTEs can be recursive, meaning they can reference themselves, which is useful for hierarchical or tree-structured data.

**Examples of CTEs**

**Example 1: Basic CTE** Suppose you want to find all employees who earn more than the average salary in their department.

WITH DepartmentAvg AS (

SELECT DepartmentID, AVG(Salary) AS AvgSalary

FROM employees

GROUP BY DepartmentID

)

SELECT e.EmployeeName, e.Salary, e.DepartmentID

FROM employees e

JOIN DepartmentAvg d ON e.DepartmentID = d.DepartmentID

WHERE e.Salary > d.AvgSalary;

**Explanation:**

* The DepartmentAvg CTE calculates the average salary for each department.
* The main query joins the employees table with the CTE to find employees earning more than their department's average salary.

**Example 2: CTE with Multiple Definitions** You can define multiple CTEs within a single query by separating them with commas.

WITH DepartmentSales AS (

SELECT DepartmentID, SUM(Sales) AS TotalSales

FROM sales

GROUP BY DepartmentID

),

TopDepartments AS (

SELECT DepartmentID

FROM DepartmentSales

WHERE TotalSales > 100000

)

SELECT e.EmployeeName, e.DepartmentID

FROM employees e

JOIN TopDepartments t ON e.DepartmentID = t.DepartmentID;

**Explanation:**

* The first CTE, DepartmentSales, calculates total sales per department.
* The second CTE, TopDepartments, filters departments with sales over 100,000.
* The main query selects employees in those top-performing departments.

**Summary**

* **CTEs** are temporary result sets that enhance query readability and modularity.
* **Syntax** involves defining a CTE with WITH and then referencing it in the main query.
* **Recursive CTEs** allow for querying hierarchical data structures.

CTEs are widely supported in most SQL databases, including MySQL (from version 8.0), PostgreSQL, SQL Server, and Oracle. They are ideal for structuring complex queries, improving readability, and solving recursive problems.