**Database Management System**

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

A Database Management System (DBMS) is software that allows users to create, read, update, and delete data. It provides a systematic and organized way to manage databases.

Key Functions of a DBMS:

1. Data Definition: Create and modify the structure of database objects like tables.
2. Data Manipulation: Insert, update, delete, and retrieve data using SQL.
3. Data Security: Ensure authorized access with user roles and permissions.
4. Data Integrity: Enforce rules like constraints to maintain data accuracy.
5. Concurrency Control: Allow multiple users to access data simultaneously.
6. Backup and Recovery: Restore the database in case of failure.

Types of DBMS:

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Relational DBMS (RDBMS)** | Stores data in tables (rows & columns). Uses SQL. | MySQL, PostgreSQL, Oracle, SQLite |
| **NoSQL DBMS** | Stores unstructured data (documents, key-value, etc.) | MongoDB, Cassandra |
| **Hierarchical DBMS** | Data stored in tree-like structure. | IBM IMS |
| **Network DBMS** | More flexible than hierarchical; allows many-to-many relationships. | Integrated Data Store (IDS) |

* **MySQL**
* **What is MySQL?**

**MySQL** is a widely used open-source **Relational Database Management System (RDBMS)**. It stores data in structured tables and uses **SQL (Structured Query Language)** to query and manage data.

MySQL is free and open-source. MySQL is ideal for both small and large applications.

Key Features of MySQL:

1. **Open Source**: Free to use and modify.
2. **High Performance**: Handles large volumes of data efficiently.
3. **Scalability**: Suitable for small applications and large systems.
4. **Platform Independence**: Runs on multiple operating systems like Windows, Linux, and macOS.
5. **Security**: Offers robust mechanisms for user authentication and access control.
6. **Multi-User Support**: Allows multiple users to work simultaneously.

* **Database Design**
* **What is Database Design?**

Database Design is the process of structuring a database in a way that minimizes data redundancy, ensures data integrity, supports efficient queries, reflects real-world entities and their relationships.

* **Database Design Process**

1. Requirements Analysis: Before designing the database, it is essential to understand the **business requirements** and **data needs**.

* Identify the types of data to be stored.
* Understand user workflows and system use cases.
* Define business rules, constraints, and goals.
* Collect requirements from stakeholders or clients.

1. Conceptual Design (ER Model): This stage focuses on modelling the system **at a high level**, independent of any specific database technology.

* Identify entities (real-world objects like Customer, Product, Order).
* Define attributes for each entity (e.g., name, email).
* Determine relationships between entities (e.g., a customer places an order).
* Create an Entity-Relationship Diagram (ERD) to visually represent entities, relationships, and cardinalities.

1. Logical Design: The logical design translates the conceptual model into a **relational schema** suitable for implementation in an RDBMS.

* Convert entities to tables, attributes to columns.
* Define primary keys to uniquely identify records.
* Establish foreign keys to model relationships.
* Set up constraints (e.g., NOT NULL, UNIQUE).
* Determine data types for each attribute (e.g., INT, VARCHAR, DATE).

1. Normalization: Normalization is the process of organizing data to minimize **redundancy** and ensure **data integrity**.

* 1NF (First Normal Form): Eliminate repeating groups; ensure atomic values.
* 2NF (Second Normal Form): Remove partial dependencies (on part of a composite key).
* 3NF (Third Normal Form): Remove transitive dependencies.

1. Physical Design: This phase involves optimizing the database for performance and scalability on a specific platform (e.g., MySQL, PostgreSQL).

* Define indexes on frequently searched columns.
* Optimize data types based on size and usage.
* Specify storage requirements (disk size, memory, partitioning).
* Set up backup, security, and user access control.
* Create scripts for database deployment.
* **SQL**

SQL (Structured Query Language) is a standard language used to communicate with **Relational Database Management Systems (RDBMS)**. SQL is used to insert, search, update, and delete database records.

* **Types of SQL Commands**

1. Data Definition Language (DDL): DDL commands are used to define or modify the structure of database objects like tables, schemas, or databases. These commands often result in permanent changes to the database structure and are automatically committed in most database systems.

Key DDL Commands:

1. CREATE: Creates a new database object, such as a table, database, or index.

CREATE TABLE Employees (

ID INT,

Name VARCHAR(100),

Department VARCHAR(50)

);

1. ALTER: Modifies an existing database object, such as adding, modifying, or dropping columns in a table.

ALTER TABLE Employees

ADD Salary DECIMAL(10, 2);

1. DROP: Deletes an entire database object, such as a table or database, permanently.

DROP TABLE Employees;

1. TRUNCATE: Removes all records from a table but retains the table structure.

TRUNCATE TABLE Employees;

1. RENAME: Changes the name of a database object.

RENAME TABLE Employees TO Staff;

1. Data Manipulation Language (DML): DML commands are used to manipulate the data within a database table. These commands allow insertion, updating, deletion, and retrieval of data.

Key DML Commands:

1. INSERT: Adds new records to a table.

INSERT INTO Employees (ID, Name, Department, Salary)

VALUES (1, 'Alice Johnson', 'HR', 55000);

1. UPDATE: Modifies existing records in a table.

UPDATE Employees

SET Salary = 60000

WHERE ID = 1;

1. DELETE: Removes records from a table based on a condition.

DELETE FROM Employees WHERE ID = 1;

1. Data Control Language (DCL): DCL commands manage access and permissions for database objects, ensuring security and proper user privileges.

Key DCL Commands:

1. GRANT: Assigns specific permissions to a user or role.

GRANT SELECT, INSERT ON Employees TO user123;

1. REVOKE: Removes previously granted permissions from a user or role.

REVOKE INSERT ON Employees FROM user123;

1. Transaction Control Language (TCL): TCL commands manage transactions to ensure data integrity and consistency. They control how changes made by DML commands are saved or undone.

Key TCL Commands:

1. COMMIT: Saves all changes made during the current transaction.

BEGIN;

UPDATE Employees

SET Salary = 70000

WHERE ID = 2;

COMMIT;

1. ROLLBACK: Undoes all changes made during the current transaction.

BEGIN;

DELETE FROM Employees

WHERE Department = 'IT';

ROLLBACK;

1. SAVEPOINT: Sets a point within a transaction to which you can roll back.

BEGIN;

UPDATE Employees

SET Salary = 80000

WHERE ID = 3;

SAVEPOINT BeforeBonus;

UPDATE Employees

SET Salary = 85000

WHERE ID = 3;

ROLLBACK TO BeforeBonus;

COMMIT;

1. Data Query Language (DQL): DQL commands are used to retrieve data from a database. While technically a subset of DML, DQL is often considered separately due to its focus on querying.

Key DQL Command:

1. SELECT: Retrieves data from one or more tables.

SELECT Name, Department, Salary

FROM Employees

WHERE Salary > 50000;

* **SQL Constraints**

SQL constraints are rules enforced on data columns in a table to ensure the accuracy, reliability, and integrity of the data within a relational database. They limit the type of data that can be inserted or updated in a table, preventing invalid data from entering the database. Constraints can be defined at the time of table creation or added later using the ALTER TABLE statement.

There are six primary types of SQL constraints:

1. NOT NULL Constraint: The NOT NULL constraint ensures that a column cannot contain NULL (empty) values. Every row must have a value for that column.

CREATE TABLE Employees (

ID INT NOT NULL,

Name VARCHAR(100) NOT NULL,

Department VARCHAR(50)

);

1. UNIQUE Constraint: The UNIQUE constraint ensures that all values in a column (or a set of columns) are distinct. No duplicates are allowed, but NULL values are permitted (unless combined with NOT NULL).

CREATE TABLE Employees (

ID INT,

Email VARCHAR(100) UNIQUE,

Name VARCHAR(100)

);

1. PRIMARY KEY Constraint: The PRIMARY KEY constraint uniquely identifies each record in a table. It is a combination of NOT NULL and UNIQUE constraints. A table can have only one primary key, which can consist of one or more columns (composite primary key).

CREATE TABLE Employees (

ID INT PRIMARY KEY,

Name VARCHAR(100),

Department VARCHAR(50)

);

1. FOREIGN KEY Constraint: The FOREIGN KEY constraint maintains referential integrity by ensuring that a value in one table matches a value in another table (typically a primary key or unique key). It prevents actions that would destroy links between tables.

CREATE TABLE Departments (

DepartmentID INT PRIMARY KEY,

DepartmentName VARCHAR(50)

);

CREATE TABLE Employees (

ID INT PRIMARY KEY,

Name VARCHAR(100),

DepartmentID INT,

FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)

);

1. CHECK Constraint: The CHECK constraint ensures that all values in a column satisfy a specific condition or Boolean expression. It validates data based on custom rules.

CREATE TABLE Employees (

ID INT PRIMARY KEY,

Name VARCHAR(100),

Age INT CHECK (Age >= 18)

);

1. DEFAULT Constraint: The DEFAULT constraint provides a default value for a column when no value is specified during insertion. It simplifies data entry by auto-filling values.

CREATE TABLE Employees (

ID INT PRIMARY KEY,

Name VARCHAR(100),

Department VARCHAR(50) DEFAULT 'General'

);

* **SQL Clause**

SQL clauses are components of SQL queries that specify how data is retrieved, filtered, sorted, grouped, or combined from a database. They are primarily used within the SELECT statement but also appear in other SQL commands like INSERT, UPDATE, and DELETE. Clauses define the logic and structure of a query, enabling precise data manipulation and retrieval.

Basic SQL clauses:

1. SELECT Clause: The SELECT clause specifies the columns or expressions to retrieve from a database table. It is the starting point of most SQL queries and defines the result set.

SELECT Name, Salary

FROM Employees;

1. FROM Clause: The FROM clause specifies the table(s) from which data is retrieved. It is mandatory in a SELECT statement and can include multiple tables for joins.

SELECT Name

FROM Employees;

1. WHERE Clause: The WHERE clause filters rows based on a specified condition. Only rows that satisfy the condition are included in the result set.

SELECT Name, Salary

FROM Employees

WHERE Salary > 50000;

1. ORDER BY Clause: The ORDER BY clause sorts the result set based on one or more columns, either in ascending (ASC) or descending (DESC) order.

SELECT Name, Salary

FROM Employees

ORDER BY Salary DESC;

1. GROUP BY Clause: The GROUP BY clause groups rows with identical values in specified columns into summary rows, typically used with aggregate functions (e.g., COUNT, SUM, AVG, MAX, MIN).

SELECT Department, AVG(Salary) AS AvgSalary

FROM Employees

GROUP BY Department;

1. HAVING Clause: The HAVING clause filters grouped rows based on a condition, similar to WHERE but applied after GROUP BY. It is used with aggregate functions.

SELECT Department, COUNT(\*) AS TotalEmployees

FROM Employees

GROUP BY Department

HAVING COUNT(\*) > 5;

1. JOIN Clause: The JOIN clause combines rows from two or more tables based on a related column. It is used in the FROM clause and supports several types: INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN, and CROSS JOIN.

SELECT Employees.Name, Departments.DepartmentName

FROM Employees

JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;

1. DISTINCT Clause: The DISTINCT clause eliminates duplicate rows from the result set, returning only unique records.

SELECT DISTINCT Department

FROM Employees;

1. LIMIT Clause: The LIMIT clause restricts the number of rows returned in the result set. It is often used for pagination or sampling data.

SELECT Name, Salary

FROM Employees

LIMIT 5;

1. UNION Clause: The UNION clause combines the results of two or more SELECT statements into a single result set, removing duplicates. UNION ALL retains duplicates for better performance.

SELECT Name FROM Employees

UNION

SELECT Name FROM Managers;

1. WITH Clause (Common Table Expressions - CTE): The WITH clause defines temporary result sets (CTEs) that can be referenced within a SELECT, INSERT, UPDATE, or DELETE statement. It improves query readability and modularity.

WITH HighEarners AS (

SELECT Name, Salary

FROM Employees

WHERE Salary > 60000

)

SELECT \* FROM HighEarners;

1. CASE Clause: The CASE clause provides conditional logic within a query, allowing for dynamic values based on conditions.

SELECT Name, Salary,

CASE

WHEN Salary >= 80000 THEN 'High'

WHEN Salary >= 50000 THEN 'Medium'

ELSE 'Low'

END AS SalaryLevel

FROM Employees;

* **SQL Logical and Query Operators**

SQL logical and query operators are essential for constructing conditions and filtering data in SQL queries. These operators are primarily used within clauses such as WHERE, HAVING, and CASE to define conditions for selecting, updating, or deleting data. They allow users to combine conditions, perform pattern matching, check ranges, assign aliases, and test for the existence of data.

Common SQL operators:

1. AND Operator: The AND operator combines multiple conditions in a query, requiring all conditions to be true for a row to be included in the result set.

SELECT \* FROM Employees

WHERE Department = 'HR' AND Salary > 50000;

1. OR Operator: The OR operator combines multiple conditions, requiring at least one condition to be true for a row to be included in the result set.

SELECT \* FROM Employees

WHERE Department = 'HR' OR Department = 'Finance';

1. NOT Operator: The NOT operator negates a condition, returning rows where the condition is false.

SELECT \* FROM Employees

WHERE NOT Department = 'HR';

1. IN Operator: The IN operator checks if a value matches any value in a specified list or subquery result.

SELECT \* FROM Employees

WHERE Department IN ('HR', 'Finance', 'IT');

1. LIKE Operator: The LIKE operator performs pattern matching on string data, using wildcards to match patterns.

SELECT \* FROM Employees

WHERE Name LIKE 'A%'; -- Names starting with 'A'

1. BETWEEN Operator: The BETWEEN operator selects values within a specified inclusive range, including numbers, dates, or strings.

SELECT \* FROM Employees

WHERE Salary BETWEEN 40000 AND 60000;

1. AS Operator: The AS operator assigns an alias to a column, table, or expression, improving query readability or renaming output columns.

SELECT Name AS EmployeeName, Salary AS MonthlyPay

FROM Employees;

1. ALL Operator: The ALL operator compares a value against all values returned by a subquery, requiring the condition to be true for all values.

SELECT \* FROM Employees

WHERE Salary > ALL (

SELECT Salary FROM Employees WHERE Department = 'HR'

);

1. ANY Operator: The ANY operator compares a value against any value returned by a subquery, requiring the condition to be true for at least one value.

SELECT \* FROM Employees

WHERE Salary > ANY (

SELECT Salary FROM Employees WHERE Department = 'HR'

);

1. EXISTS Operator: The EXISTS operator tests for the existence of rows in a subquery, returning true if the subquery returns at least one row.

SELECT DepartmentName

FROM Departments d

WHERE EXISTS (

SELECT 1 FROM Employees e

WHERE e.DepartmentID = d.DepartmentID

);

* **SQL Aggregate Functions**

SQL aggregate functions perform calculations on a set of values in a column and return a single value. These functions are commonly used in SELECT statements, often in conjunction with the GROUP BY clause, to summarize data, such as calculating totals, averages, or counts. Aggregate functions are essential for data analysis and reporting in relational databases.

Common Aggregate functions:

1. COUNT Function: The COUNT function returns the number of rows that match a specified condition or the total number of rows in a table.

-- Count all employees

SELECT COUNT(\*) AS TotalEmployees

FROM Employees;

-- Count employees in the HR department

SELECT COUNT(\*) AS HREmployees

FROM Employees

WHERE Department = 'HR';

1. SUM Function: The SUM function calculates the total sum of numeric values in a column.

SELECT SUM(Salary) AS TotalSalaries

FROM Employees;

1. AVG Function: The AVG function calculates the average (arithmetic mean) of numeric values in a column.

SELECT AVG(Salary) AS AverageSalary

FROM Employees;

1. MIN Function: The MIN function returns the smallest value in a column.

SELECT MIN(Salary) AS LowestSalary

FROM Employees;

1. MAX Function: The MAX function returns the largest value in a column.

SELECT MAX(Salary) AS HighestSalary

FROM Employees;

* **SQL Joins**

SQL joins are used to combine rows from two or more tables based on a related column, allowing data retrieval from multiple tables in a single query. Joins are essential for querying relational databases where data is distributed across normalized tables. They are typically specified in the FROM clause of a SELECT statement and use the ON keyword to define the relationship between tables.

Syntax:

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

FROM table1

[JOIN TYPE] JOIN table2

ON table1.common\_column = table2.common\_column;

Primary SQL join types:

1. INNER JOIN: The INNER JOIN returns only the rows where there is a match in both tables based on the specified condition.
2. LEFT JOIN (LEFT OUTER JOIN): The LEFT JOIN returns all rows from the left table and the matched rows from the right table. If there is no match, NULL values are returned for columns from the right table.
3. RIGHT JOIN (RIGHT OUTER JOIN): The RIGHT JOIN returns all rows from the right table and the matched rows from the left table. If there is no match, NULL values are returned for columns from the left table.
4. FULL JOIN (FULL OUTER JOIN): The FULL JOIN returns all rows from both tables, with NULLs in places where there is no match in either table.
5. CROSS JOIN: The CROSS JOIN produces a Cartesian product, combining every row from the first table with every row from the second table, without requiring a matching condition.

* **SQL Index**

An index is a database object that improves the speed of data retrieval operations on a table by providing quick access to rows. It works like an index in a book, allowing the database to locate data without scanning the entire table.

Key Features:

* Enhances performance of SELECT queries and WHERE clauses.
* Can slow down INSERT, UPDATE, and DELETE operations due to index maintenance.
* Automatically created for PRIMARY KEY and UNIQUE constraints.
* Types include clustered (affects physical data order, one per table) and non-clustered (separate structure, multiple allowed).
* Supports unique indexes to enforce uniqueness.

Syntax:

-- Create an index

CREATE [UNIQUE] INDEX index\_name

ON table\_name (column\_name1, column\_name2, ...);

-- Drop an index

DROP INDEX index\_name ON table\_name;

Key Considerations:

* Performance Trade-off: Indexes improve read performance but increase write overhead.
* Storage: Indexes consume additional disk space.
* When to Use: On frequently queried columns (e.g., in WHERE, JOIN, or ORDER BY) or columns with high selectivity (unique or nearly unique values).
* Database Variations: Syntax may vary (e.g., SQL Server uses DROP INDEX index\_name ON table\_name, while MySQL uses DROP INDEX index\_name).
* **SQL View**

A view is a virtual table based on the result set of a SELECT query. It does not store data physically but provides a simplified or customized perspective of the data.

Key Features:

* Simplifies complex queries by encapsulating them.
* Enhances security by restricting access to specific columns or rows.
* Can be queried like a regular table.
* Some views (updatable views) allow INSERT, UPDATE, or DELETE operations if they meet certain conditions (e.g., single-table queries without aggregations).

Syntax:

-- Create a view

CREATE VIEW view\_name AS

SELECT column1, column2, ...

FROM table\_name

[WHERE condition];

-- Drop a view

DROP VIEW view\_name;

Key Considerations:

* Performance: Views do not inherently improve performance; they rely on the underlying query.
* Updatability: Views are generally read-only unless they meet specific criteria (e.g., no joins or aggregations).
* Security: Views can hide sensitive columns (e.g., Salary) from users with limited permissions.
* Materialized Views: Some databases (e.g., PostgreSQL, Oracle) support materialized views, which store data physically for better performance.
* **SQL Function**

A function (or user-defined function, UDF) is a reusable database object that encapsulates a set of SQL statements to perform a specific task and return a value. Functions are often used for calculations or data transformations.

Key Features:

* Returns a single value (scalar) or a table (table-valued function).
* Can be used in SELECT, WHERE, or other clauses.
* Cannot modify database state (e.g., no INSERT, UPDATE, or DELETE in most systems).
* Deterministic (same input, same output) or non-deterministic (output may vary).

Syntax:

-- Create a function

CREATE FUNCTION function\_name (@parameter datatype)

RETURNS return\_datatype

AS

BEGIN

RETURN expression;

END;

-- Drop a function

DROP FUNCTION function\_name;

Key Considerations:

* Scope: Functions are scoped to a database or schema.
* Determinism: Deterministic functions can be used in indexed views or computed columns.
* Database Variations: Syntax and capabilities vary (e.g., MySQL uses CREATE FUNCTION with different syntax, PostgreSQL supports multiple languages like PL/pgSQL).
* Limitations: Functions typically cannot modify data (use stored procedures for that).
* **SQL Stored Procedure**

A stored procedure is a precompiled collection of SQL statements stored in the database that can be executed repeatedly. It can accept parameters, perform complex operations, and return results.

Key Features:

* Can include INSERT, UPDATE, DELETE, and control-of-flow statements (e.g., IF, WHILE).
* Improves performance by reducing network traffic and reusing execution plans.
* Enhances security by restricting direct table access.
* Can return multiple result sets or output parameters.

Syntax:

DELIMITER $$

CREATE PROCEDURE procedure\_name ([parameter\_list])

BEGIN

-- SQL statements

END $$

DELIMITER ;

Key Considerations:

* Reusability: Stored procedures are ideal for repetitive tasks (e.g., batch updates).
* Security: Can grant execute permissions without exposing underlying tables.
* Performance: Precompiled execution plans improve efficiency for complex operations.
* Database Variations: Syntax and features vary (e.g., MySQL uses DELIMITER, PostgreSQL uses CREATE OR REPLACE PROCEDURE).
* **SQL Trigger**

A MySQL trigger is a named database object associated with a specific table. It automatically activates (or "fires") in response to a defined event, such as an INSERT, UPDATE, or DELETE operation on that table. Triggers are useful for enforcing data integrity, performing automatic calculations, validating data, or auditing changes without requiring application-level logic.

**Types of Triggers**:

Triggers are classified by timing and event:

* Timing:
* BEFORE: Executes before the triggering event (e.g., before data is inserted or updated). Useful for validation or modifying incoming data.
* AFTER: Executes after the triggering event (e.g., after data is successfully inserted). Useful for post-operation tasks like logging or updating related tables.
* Events:
* INSERT: Triggers on row insertions.
* UPDATE: Triggers on row updates.
* DELETE: Triggers on row deletions.

Syntax:

DELIMITER $$

CREATE TRIGGER trigger\_name

{ BEFORE | AFTER } { INSERT | UPDATE | DELETE }

ON table\_name FOR EACH ROW

BEGIN

trigger\_action;

END $$

DELIMITER ;

* **SQL Cursor**

A MySQL cursor is a database object used within stored procedures, functions, or triggers to iterate over the rows of a result set from a SELECT query. Cursors allow row-by-row processing of query results, which is useful for complex logic that cannot be handled by set-based SQL operations.

Key cursor operations:

* DECLARE: Defines the cursor and its associated SELECT query.
* OPEN: Executes the query and prepares the result set.
* FETCH: Retrieves the next row’s data into variables.
* CLOSE: Releases the cursor and its resources.

Syntax:

-- Declaring a Cursor

DECLARE cursor\_name CURSOR FOR select\_statement;

-- Declaring a Handler for End of Cursor

DECLARE CONTINUE HANDLER FOR NOT FOUND SET variable\_name = value;

-- Opening a Cursor

OPEN cursor\_name;

-- Fetching Data

FETCH cursor\_name INTO variable\_list;

-- Closing a Cursor

CLOSE cursor\_name;

* **SQL Backup and Restore**

Backup and recovery in MySQL refer to the processes of creating copies of database data and structures (backups) and restoring them in case of data loss, corruption, or disasters (recovery). Effective backup strategies ensure data availability, integrity, and minimal downtime. MySQL supports various backup methods suited to different scenarios, such as full backups for complete data snapshots and incremental backups for efficient updates.

Tool for backup in MySQL:

* **mysqldump:** A command-line tool for logical backups.

Syntax:

mysqldump [options] database [tables] > backup.sql

Key Options:

--all-databases: Backup all databases.

--routines: Includes stored procedures and functions.

--triggers: Includes triggers.

--events: Includes scheduled events.

Restore Methods: Restore involves restoring data from backups, potentially followed by applying logs for updates.

Use mysql client to import SQL dump.

Syntax:

mysql -u root -p database < backup.sql