**Basics of Database Management System**

**Overview of DBMS**

A **Database Management System (DBMS)** is software that enables users to create, manage, and interact with databases. It provides an interface for database users and applications to efficiently access, store, retrieve, and manipulate data while ensuring security, integrity, and consistency.

**Key Features:**

* **Data Abstraction**: Simplifies data access by hiding complexity.
* **Data Independence**: Schema changes don’t impact applications.
* **Data Security**: Prevents unauthorized access.
* **Data Integrity**: Ensures data accuracy and consistency.
* **Transaction Management**: Guarantees reliable and ACID-compliant transactions.
* **Backup & Recovery**: Protects data against loss.

**Types of DBMS:**

* **Relational DBMS**: Data in tables (e.g., MySQL, PostgreSQL).
* **NoSQL DBMS**: Handles unstructured data (e.g., MongoDB).
* **Hierarchical/Network DBMS**: Data in tree/graph structures.
* **Object-Oriented DBMS**: Data as objects.

**Example:**

In a library system, DBMS stores book details, member info, and borrowing records, allowing easy queries like finding overdue books

**MySQL**

**MySQL** is an open-source Relational Database Management System (RDBMS) that uses Structured Query Language (SQL) to manage and manipulate data. It is widely used for web applications due to its reliability, scalability, and compatibility with various platforms.

**Key Features:**

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

**Applications of MySQL**

* **Web Development**: Used in websites like WordPress and Joomla for dynamic content.
* **E-commerce**: Manages products, users, and orders in platforms like Shopify.
* **Social Media**: Stores user profiles, posts, and interactions.
* **Banking**: Handles secure transactions and account data.
* **Education**: Manages student records and course data in online learning systems.
* **Healthcare**: Stores patient data and billing information.
* **Mobile Apps**: Backends for user data and sync features.

**Overview of Workbench**

MySQL Workbench is a unified graphical tool for managing MySQL databases. It simplifies database design, administration, and querying with an intuitive interface.

**Key Features:**

1. **Database Design & Modeling**:
   * Create and visualize ER (Entity-Relationship) diagrams.
   * Generate and synchronize database schemas.
2. **SQL Development**:
   * Write, execute, and debug SQL queries.
   * Syntax highlighting and auto-completion for faster coding.
3. **Database Administration**:
   * Manage users and permissions.
   * Monitor server performance and configure settings.
4. **Data Import & Export**:
   * Easily import/export data in various formats like CSV and JSON.
5. **Backup & Recovery**:
   * Schedule automated backups for data safety.
   * Restore databases when needed.

**Interface of MySQL Workbench:**

* **Home Screen**: Quick access to server connections and tools.
* **SQL Editor**: Write, execute, and debug SQL queries.
* **Data Modeling**: Visualize and design database schemas using ER diagrams.
* **Administration Panel**: Manage users, monitor performance, and configure settings.
* **Performance Dashboard**: Analyze query performance and server status.

**Advantages of MySQL Workbench:**

* **User-Friendly**: Intuitive GUI for managing databases.
* **Time-Saving**: Automates tasks like schema design and query execution.
* **Cross-Platform**: Runs on Windows, macOS, and Linux.
* **Integrated Tools**: Combines design, development, and administration in one tool.
* **Free and Open Source**: Available at no cost for most use cases.

**Database Design**

**Database Design** refers to the process of structuring a database to efficiently store, organize, and retrieve data. It ensures data consistency, integrity, and scalability.

**Steps in Database Design:**

1. **Requirements Analysis**:  
   Understand the purpose of the database and identify the data requirements.
2. **Conceptual Design**:  
   Create an **Entity-Relationship (ER) Diagram** to represent entities, attributes, and relationships.
3. **Logical Design**:  
   Convert the ER diagram into a relational schema (tables with fields and relationships).
4. **Normalization**:  
   Organize data to minimize redundancy and improve integrity:
   * **1NF (First Normal Form)**: Eliminate duplicate columns and ensure atomic values.
   * **2NF (Second Normal Form)**: Remove partial dependencies.
   * **3NF (Third Normal Form)**: Eliminate transitive dependencies.
5. **Physical Design**:  
   Define how data is stored physically, including indexing, partitioning, and storage optimization.
6. **Implementation**:  
   Create the database and populate it with data.

**Best Practices for Database Design**

* **Use Meaningful Table and Column Names**: Names should clearly define their purpose.
* **Minimize Redundancy**: Apply normalization techniques to eliminate duplicate data.
* **Define Primary and Foreign Keys**: Use keys to maintain data integrity.
* **Optimize Indexing**: Create indexes for frequently queried columns to improve performance.
* **Consider Scalability**: Design with future data growth in mind.

**SQL Basics**

**SQL (Structured Query Language)** is the language used to interact with relational databases. It allows users to perform various operations, such as querying and manipulating data.

**Common SQL Operations:**

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

SELECT \* FROM students;

**INSERT**:  
Adds new rows to a table.

INSERT INTO students (id, name, age) VALUES (1, 'John', 20);

**UPDATE**:  
Modifies existing data in a table.

UPDATE students SET age = 21 WHERE id = 1;

**DELETE**:  
Removes rows from a table.

DELETE FROM students WHERE id = 1;

**WHERE**:  
Filters records based on specific conditions.

SELECT \* FROM students WHERE age > 18;

**ORDER BY**:  
Sorts the result set.

SELECT \* FROM students ORDER BY age DESC;

**LIMIT**:  
Limits the number of rows returned by a query.

SELECT \* FROM students LIMIT 5;

**DISTINCT**:  
Returns unique values by removing duplicates.

SELECT DISTINCT age FROM students;

**Data Sorting**

**Data Sorting** in MySQL allows you to arrange the result set of a query in a specific order, either ascending or descending, based on one or more columns.

**ORDER BY Clause**:  
The ORDER BY clause is used to sort data. By default, sorting is **ascending (ASC)**, but you can explicitly mention the sorting order.

* **Ascending Order**: From smallest to largest (default).
* **Descending Order**: From largest to smallest.

**Syntax**:

SELECT column1, column2

FROM table\_name

ORDER BY column1 [ASC|DESC], column2 [ASC|DESC];

**Sort by One Column (Ascending)**:  
Retrieve all students sorted by age from the lowest to the highest.

SELECT \* FROM students ORDER BY age ASC;

**Sort by One Column (Descending)**:  
Retrieve all students sorted by age from the highest to the lowest.

SELECT \* FROM students ORDER BY age DESC;

**Sort by Multiple Columns**:  
Retrieve students first by age (ascending), then by name (ascending).

SELECT \* FROM students ORDER BY age ASC, name ASC;

**Null Value & Keyword**

In SQL, **NULL** represents the absence of a value or an unknown value. It is not the same as an empty string ('') or a zero (0); it signifies that the value is missing or undefined.

**NULL Keyword**

**NULL in Data**:  
It can be used to insert or check for missing values in database columns.

INSERT INTO students (id, name, age) VALUES (1, 'John', NULL);

**Checking for NULL**:  
To check if a column value is NULL, you cannot use =. Instead, use IS NULL or IS NOT NULL.

**Example**: Check for students with no age specified.

SELECT \* FROM students WHERE age IS NULL;

**Example**: Check for students who have an age.

SELECT \* FROM students WHERE age IS NOT NULL;

**NULL Behavior in Operations**

**Arithmetic Operations**:  
Any arithmetic operation involving NULL results in NULL.

EX:

SELECT age + 10 FROM students WHERE id = 1; -- If age is NULL, result will be NULL

**Comparison Operations**:  
Comparing NULL with any value (including another NULL) using = or != does not work. Use IS NULL or IS NOT NULL instead.

**Aggregate Functions**:  
Functions like COUNT(), AVG(), SUM(), etc., ignore NULL values.

EX:

SELECT COUNT(\*) FROM students WHERE age IS NOT NULL; -- Counts only non-NULL ages

**Auto Increment**

**Auto Increment** is a feature in that automatically generates a unique value for a column, typically used for primary keys. This ensures that each row in a table has a unique identifier without needing manual input.

**Key Concepts:**

**Purpose**:  
Auto increment is most commonly used for **primary key** columns to ensure uniqueness across rows.

**Syntax**:

**For MySQL**:

CREATE TABLE table\_name (

column\_name INT AUTO\_INCREMENT,

other\_column VARCHAR(255),

PRIMARY KEY (column\_name)

);

**Example**:

INSERT INTO students (name, age) VALUES ('John', 20);

In this example, if the id column is set as AUTO\_INCREMENT, the database automatically assigns the next available number to the id.

Changing auto\_increment start value

ALTER TABLE Student2 AUTO\_INCREMENT = 100;

Changing auto\_increment gap

SET @@auto\_increment\_increment = 3;

**DDL (Data Definition Language):**

DDL deals with the structure (schema) of the database, including creating, altering, and deleting database objects such as tables, indexes, and views.

**Key Commands**:

* + CREATE: Creates a new table, database, or index.

CREATE TABLE students (id INT, name VARCHAR(50));

* + ALTER: Modifies an existing database object.

ALTER TABLE students ADD age INT;

* + DROP: Deletes a table, database, or other objects.

DROP TABLE students;

* TRUNCATE: Truncate a table.

TRUNCATE TABLE Sample;

**DML (Data Manipulation Language):**

DML deals with the manipulation of data inside the database. It allows you to insert, update, delete, and retrieve data.

**Key Commands**:

* + INSERT: Adds new data.

INSERT INTO students (id, name, age) VALUES (1, 'John', 20);

* + UPDATE: Modifies existing data.

UPDATE students SET age = 21 WHERE id = 1;

* + DELETE: Removes data from a table.

DELETE FROM students WHERE id = 1;

* + SELECT: Retrieves data.

SELECT \* FROM students;

**DCL (Data Control Language):**

DCL is used to control access to data in the database, such as granting or revoking permissions.

**Key Commands**:

* + GRANT: Gives specific privileges to a user.

GRANT SELECT, INSERT ON students TO 'username';

* + REVOKE: Removes specific privileges from a user.

REVOKE SELECT, INSERT ON students FROM 'username';

**TCL (Transaction Control Language):**

TCL is used to manage transactions within a database. A transaction is a set of statements that are executed as a unit.

**Key Commands**:

* + COMMIT: Saves all changes made during the current transaction.

COMMIT;

* + ROLLBACK: Undoes changes made during the current transaction.

ROLLBACK;

* + SAVEPOINT: Sets a point in a transaction to which you can roll back.

SAVEPOINT save\_point\_name;

**DQL (Data Query Language):**

DQL is used for querying the database. It focuses on retrieving data.

**Key Command**:

* + SELECT: Retrieves data from a table.

SELECT \* FROM students;

**LIMIT**

The LIMIT keyword restricts the number of rows returned by a query.

**Syntax:**

SELECT column1, column2

FROM table\_name

LIMIT number\_of\_rows;

**Examples:**

**Limit Rows**:  
Get the first 5 students:

SELECT \* FROM students LIMIT 5;

**Limit with Offset**:  
Get 5 students starting from the 6th row:

SELECT \* FROM students LIMIT 5 OFFSET 5;

**Use Cases:**

* **Pagination**: Display records in pages (e.g., 10 records per page).
* **Testing**: Retrieve a subset of data during development.

**Aggregate Functions**

**Aggregate functions** perform a calculation on a set of values and return a single result. They are commonly used with the GROUP BY clause to summarize data.

**Common Aggregate Functions:**

1. **COUNT()**:  
   Returns the number of rows that match a specified condition.

SELECT COUNT(\*) FROM students;

1. **SUM()**:  
   Returns the sum of a numeric column.

SELECT SUM(age) FROM students;

1. **AVG()**:  
   Returns the average value of a numeric column.

SELECT AVG(age) FROM students;

1. **MIN()**:  
   Returns the smallest value in a column.

SELECT MIN(age) FROM students;

1. **MAX()**:  
   Returns the largest value in a column.

SELECT MAX(age) FROM students;

**Example with GROUP BY:**

To get the average age of students in each class:

SELECT class, AVG(age) FROM students GROUP BY class;

**Sub-Queries**

A **sub-query** is a query nested within another query. It is used to fetch data that will be used in the main query.

**Key Points:**

**Types of Sub-Queries**:

* **Single-row Sub-Query**: Returns one row.
* **Multi-row Sub-Query**: Returns multiple rows.
* **Correlated Sub-Query**: Depends on the main query for its execution.

**Syntax**:

SELECT column1 FROM table1 WHERE column2 = (SELECT column FROM table2 WHERE condition);

**Examples:**

1. **Single-row Sub-Query**:  
   Get students older than the oldest teacher.

SELECT name FROM students WHERE age > (SELECT MAX(age) FROM teachers);

1. **Multi-row Sub-Query**:  
   Get students in classes where teachers have more than 5 years of experience.

SELECT name FROM students WHERE class IN (SELECT class FROM teachers WHERE experience > 5);

1. **Correlated Sub-Query**:  
   Find students with above-average marks in their class.

SELECT name

FROM students s1

WHERE marks > (SELECT AVG(marks) FROM students s2 WHERE s1.class = s2.class);

**Joins**

**Joins** are used to combine rows from two or more tables based on a related column. They help retrieve meaningful data by merging tables.

**Types of Joins:**

**INNER JOIN**:  
Returns rows where there is a match in both tables.

SELECT students.name, classes.class\_name

FROM students

INNER JOIN classes

ON students.class\_id = classes.id;

**LEFT OUTER JOIN**:  
Returns all rows from the left table and matching rows from the right table. If no match, NULL is returned.

SELECT students.name, classes.class\_name

FROM students

LEFT JOIN classes

ON students.class\_id = classes.id;

**RIGHT OUTER JOIN**:  
Returns all rows from the right table and matching rows from the left table. If no match, NULL is returned.

SELECT students.name, classes.class\_name

FROM students

RIGHT JOIN classes

ON students.class\_id = classes.id;

**FULL OUTER JOIN**:  
Returns all rows when there is a match in either table. Non-matching rows will have NULLs.

SELECT S.Name AS StudentName, S.Class, Sub.SubjectName

FROM Student S

LEFT JOIN Subject Sub

ON S.SubjectId = Sub.SubjectId

UNION

SELECT S.Name AS StudentName, S.Class, Sub.SubjectName

FROM Student S

RIGHT JOIN Subject Sub

ON S.SubjectId = Sub.SubjectId;

**CROSS JOIN**:  
Returns the Cartesian product of two tables (every row from the first table is paired with every row from the second table).

SELECT students.name, classes.class\_name

FROM students

CROSS JOIN classes;

**SELF JOIN**:  
A table joins with itself.

SELECT a.name AS student1, b.name AS student2

FROM students a, students b

WHERE a.class\_id = b.class\_id;

**NATURAL JOIN:**

SELECT \* FROM

student

NATURAL JOIN

subject;

**Unions**

A **UNION** combines the results of two or more SELECT queries into a single result set, eliminating duplicate rows by default.

**Key Points:**

**Structure**:

* All SELECT queries must have the same number of columns.
* The columns must have the same data type and order.

**Syntax**:

SELECT column1, column2 FROM table1

UNION

SELECT column1, column2 FROM table2;

**UNION ALL**: Includes duplicate rows in the result.

SELECT column1, column2 FROM table1

UNION ALL

SELECT column1, column2 FROM table2;

**Index**

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

**Key Points:**

**Purpose**:

* Speeds up SELECT queries and data lookups.
* Slows down INSERT, UPDATE, and DELETE operations due to index maintenance.

**Syntax**:

* **Create Index**:

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

* **Drop Index**:

DROP INDEX index\_name;

**View**

A **View** is a virtual table based on the result of a query. It does not store data itself but dynamically fetches it from the underlying tables.

**Key Points:**

**Purpose**:

* Simplifies complex queries.
* Enhances security by restricting direct access to table data.
* Provides a consistent interface for frequently used queries.

**Syntax**:

* **Create a View**:

CREATE VIEW view\_name AS

SELECT column1, column2

FROM table\_name

WHERE condition;

* **Query a View**:

SELECT \* FROM view\_name;

* **Drop a View**:

DROP VIEW view\_name;

**Advantages:**

* Simplifies repetitive and complex queries.
* Enhances security by exposing only required data.
* Provides abstraction, making schema changes less impactful.

**Disadvantages:**

* Views do not store data, so performance may be slower for large or complex queries.
* Cannot always perform INSERT, UPDATE, or DELETE on views, depending on the underlying query.

**EXPLAIN Keyword**

The **EXPLAIN** keyword is used to understand how queries are executed. It provides a detailed execution plan, showing how the database retrieves data, processes joins, and uses indexes.

**Key Points:**

**Purpose**:

* Analyze and optimize query performance.
* Identify inefficiencies like full table scans or missing indexes.

**Syntax**:

EXPLAIN SELECT \* FROM table\_name;

**Output Columns** (common fields in the execution plan):

* **id**: The identifier of the query step.
* **select\_type**: The type of query (e.g., SIMPLE, PRIMARY, SUBQUERY).
* **table**: The table being accessed.
* **type**: The join type (e.g., ALL, INDEX, REF).
* **possible\_keys**: Indexes that may be used for the query.
* **key**: The index actually used.
* **rows**: The estimated number of rows to be scanned.
* **extra**: Additional details like Using where, Using index, etc.

**Backup and Restore in MySQL Workbench**

MySQL Workbench provides a user-friendly interface to back up and restore databases without using complex command-line operations.

**Backup a Database**

**Steps**:

1. Open **MySQL Workbench** and connect to your database server.
2. Go to the **Server** menu and select **Data Export**.
3. In the **Data Export** tab:
   1. Choose the database(s) you want to back up.
   2. Select tables if you want a partial backup or keep all selected for a full backup.
   3. Choose the output format:
      * **Dump Structure and Data**: Includes both schema and data.
      * **Dump Data Only**: Only data, no schema.
      * **Dump Structure Only**: Only schema, no data.
4. Click **Start Export** to generate the backup file (usually a .sql file).

**Result**: The .sql file contains SQL statements to recreate the database structure and populate it with data.

**Restore a Database**

**Steps**:

1. Open **MySQL Workbench** and connect to your database server.
2. Go to the **Server** menu and select **Data Import**.
3. In the **Data Import** tab:
   1. Select **Import from Self-Contained File** and choose your backup file (e.g., backup.sql).
   2. Choose a database to restore into, or create a new database if needed.
   3. If the file contains multiple databases, use the option **Dump Structure and Data** to import them all.
4. Click **Start Import** to begin the restoration process.

**Result**: The database is recreated, including its structure and data, based on the contents of the backup file.

**Advantages of Backup and Restore:**

* Prevents data loss by creating a copy of critical data.
* Easy migration of databases between servers or environments.
* Useful for testing by restoring databases in development environments.

Let me know if you'd like to dive deeper into any of these steps!

**Stored Procedure :**

A **stored procedure** in MySQL is a set of SQL statements that can be stored and executed on the database server. It allows for reusability, modular code, and improved performance by reducing client-server communication. You can create, execute, and manage stored procedures using SQL commands.

**Key Benefits:**

* **Modular code**: Group related SQL operations.
* **Performance**: Reduces the number of queries sent to the server.
* **Security**: Limits direct access to underlying tables.

Example syntax :   
CREATE PROCEDURE GetEmployeeInfo(IN emp\_id INT)

BEGIN

SELECT \* FROM employees WHERE id = emp\_id;

END;

Example Call :

CALL GetEmployeeInfo(101);

**Cursor :**

A **cursor** in MySQL is a database object used to retrieve, manipulate, and iterate through a result set row by row. It is particularly useful when you need to process individual rows of a query result in a stored procedure.

**Steps to use a cursor in MySQL:**

1. **Declare** the cursor.
2. **Open** the cursor.
3. **Fetch** data from the cursor.
4. **Process** each fetched row.
5. **Close** the cursor.

Example :

DELIMITER $$

CREATE PROCEDURE ReadCursor()

BEGIN

DECLARE stuID INT;

DECLARE stuCity VARCHAR(50);

DECLARE readNullCursor CURSOR FOR SELECT id, city FROM student;

OPEN readNullCursor;

FETCH readNullCursor INTO stuID, stuCity;

CLOSE readNullCursor;

END $$

DELIMITER ;

**When to use:**

* When you need to process each row of a result set individually (e.g., in complex business logic).
* Avoid using cursors in simple queries as they may degrade performance.

**Trigger :**

A trigger in MySQL is a special kind of stored procedure that automatically executes or "fires" when a specific event occurs on a table or view. Triggers are useful for automatically enforcing business rules, validating data, or keeping audit logs.

Key Components of a Trigger in MySQL:

**Event:** The type of operation that activates the trigger. Common events include:

INSERT: Trigger fires when a new row is inserted into the table.

UPDATE: Trigger fires when a row is updated.

DELETE: Trigger fires when a row is deleted.

**Timing:** The point in time when the trigger should be fired relative to the event:

BEFORE: The trigger is fired before the actual event (INSERT, UPDATE, DELETE).

AFTER: The trigger is fired after the event has occurred.

**Trigger Action:** The operation that should be performed when the trigger is fired. This can be an SQL statement such as updating another table, logging changes, etc.

Ex :

CREATE TRIGGER trigger\_name

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

ON table\_name

FOR EACH ROW

trigger\_action;

* trigger\_name: The name of the trigger.
* BEFORE or AFTER: Specifies when the trigger will fire.
* INSERT, UPDATE, or DELETE: Specifies the event that will trigger the action.
* table\_name: The table on which the trigger will be applied.
* FOR EACH ROW: Ensures the trigger is applied to each row affected by the operation.
* trigger\_action: The action you want to perform when the trigger is fired, which can include modifying other tables or generating messages

For Drop trigger :   
DROP TRIGGER trigger\_name;

**Key Points:**

* **NEW**: In a BEFORE or AFTER INSERT or UPDATE trigger, NEW refers to the new values being inserted or updated in the row.
* **OLD**: In a BEFORE or AFTER DELETE or UPDATE trigger, OLD refers to the old values of the row before it was modified or deleted.
* increase complexity .
* MySQL allows a maximum of 6 triggers per table (one for each combination of BEFORE/AFTER and INSERT/UPDATE/DELETE).
* Triggers cannot be invoked manually