1. What is a database ,tuple & Records ?

Database: Collection of organized data.
Tuple: Single row in a table.
Record: Another term for a tuple (row of data).

2. What is the difference between a database and a DBMS?

Database: Stores data.
DBMS: Manages and controls access to the database.

3. What are the different types of database models?
Types of database models: Hierarchical, Network, Relational, Object-oriented.

Hierarchical: Data organized in a tree structure (parent-child).
Network: Data connected through multiple relationships.
Relational: Data stored in tables with rows and columns.
Object-oriented: Data stored as objects like in programming.

4. What is normalization? Why is it important?
Normalization: Process of organizing data to reduce redundancy.
Importance: Ensures data consistency and efficient storage.

5. Explain the different normal forms (1NF, 2NF, 3NF, BCNF).

1NF: No repeating groups; each cell has single value.
2NF: 1NF + no partial dependency on primary key.
3NF: 2NF + no transitive dependency.
BCNF: Stronger 3NF; every determinant is a candidate key.

6. What is denormalization and when should we use it?

Denormalization: Combining tables to reduce joins.
Use: When faster data retrieval is needed.

7. What is a primary key and a foreign key?

Primary Key: Uniquely identifies each record in a table.
Foreign Key: Links one table to another using the primary key

8. What are constraints in SQL?
Constraints are rules applied to table columns to ensure valid and consistent data.

Common types:

PRIMARY KEY: Ensures each record is unique.

FOREIGN KEY: Maintains referential integrity between tables.

UNIQUE: Prevents duplicate values in a column.

NOT NULL: Ensures a column cannot have empty values.

CHECK: Validates data based on a condition.

DEFAULT: Assigns a default value when no value is provided.

9. What is an index? How does it improve performance?

Index: A database structure that speeds up data retrieval.
Performance: Reduces search time by allowing quick lookups instead of scanning the whole table.

10.What is a view in SQL?
View: A virtual table in SQL created using a query; it does not store data physically.

More information:

Shows data from one or more tables.

Can simplify complex queries.

Can provide security by restricting access to specific columns or rows.

Can be **updatable** (if based on a single table) or **read-only** (if complex).

11. What are DDL, DML, DCL, and TCL in SQL?

Constraints are rules applied to table columns to ensure valid and consistent data.

Common types:

PRIMARY KEY: Ensures each record is unique.

FOREIGN KEY: Maintains referential integrity between tables.

UNIQUE: Prevents duplicate values in a column.

NOT NULL: Ensures a column cannot have empty values.

CHECK: Validates data based on a condition.

DEFAULT: Assigns a default value when no value is provided.

12. Write a SQL query to create a table

```
CREATE TABLE Students (
   StudentID INT PRIMARY KEY,
   Name VARCHAR(50) NOT NULL,
```

```
    Age INT,
    Email VARCHAR(50) UNIQUE
);
```

13. How do you insert data into a table?

```
INSERT INTO Students (StudentID, Name, Age, Email)
VALUES (1, 'Rahul', 20, 'rahul@example.com');
```

14. How do you update data in a table?

```
UPDATE Students
SET Age = 21, Email = 'rahul21@example.com'
WHERE StudentID = 1;
```

15. How do you delete data from a table?

```
DELETE FROM Students
WHERE StudentID = 1;
```

16. What is the difference between DELETE and TRUNCATE & DROP ?

DELETE: Removes specific rows; can use WHERE; logs changes; slower.
TRUNCATE: Removes all rows; cannot use WHERE; faster; resets identity counters.
DROP: Deletes the entire table structure and data permanently.

17. Explain the difference between WHERE and HAVING.

WHERE: Filters rows **before** grouping; works on individual records.
HAVING: Filters groups **after** aggregation; works on summarized data.

18. How do you select distinct records from a table?

```
SELECT DISTINCT column_name
FROM table_name;
```

19. Write a query to find the maximum salary from an employee table.

```
SELECT MAX(Salary) AS MaxSalary
FROM Employee;
```

20. What is the Types of Joins & difference between INNER JOIN, LEFT JOIN, RIGHT JOIN,
and FULL OUTER JOIN?

Differences:

Join Type	Description
INNER JOIN	Returns only matching rows from both tables.
LEFT JOIN	Returns all rows from the left table + matching rows from the right table.
RIGHT JOIN	Returns all rows from the right table + matching rows from the left table.
FULL OUTER JOIN	Returns all rows from both tables; fills NULL where there is no match.

21. What is a subquery?
Subquery: A query nested inside another query, used to return data for the main query.

```
SELECT Name
FROM Students
WHERE Age = (SELECT MAX(Age) FROM Students);
```

22. Explain correlated vs. non-correlated subqueries.

Non-Correlated Subquery:

Independent of the outer query.

Executes **once** and returns a result for the outer query.

Example:

```
SELECT Name

FROM Students

WHERE Age = (SELECT MAX(Age) FROM Students);
```

Correlated Subquery:

Depends on the outer query.

Executes **for each row** of the outer query.

```
SELECT S1.Name

FROM Students S1

WHERE S1.Age > (SELECT AVG(S2.Age)

FROM Students S2

WHERE S1.ClassID = S2.ClassID);
```

23. How does the GROUP BY clause work?
GROUP BY: Groups rows that have the same values in specified columns and allows aggregate functions to be applied on each group.

SELECT ClassID, COUNT(*) AS StudentCount

FROM Students

GROUP BY ClassID;

24. How can you prevent duplicate records from being inserted into a table?

You can prevent duplicates by using:

PRIMARY KEY or **UNIQUE** constraints on the column(s).

CREATE TABLE Students (

StudentID INT PRIMARY KEY,

Email VARCHAR(50) UNIQUE

);

INSERT IGNORE or **ON DUPLICATE KEY UPDATE** (in MySQL) when inserting data.

INSERT IGNORE INTO Students (StudentID, Email) VALUES (1, 'a@example.com');

25. What is a transaction in SQL?

Transaction: A sequence of SQL operations treated as a single unit of work; either **all succeed** or **all fail**.

Properties (ACID):

Atomicity: All or nothing.

Consistency: Database remains valid.

Isolation: Transactions do not interfere.

Durability: Changes are permanent once committed.

BEGIN TRANSACTION;

UPDATE Account SET Balance = Balance - 100 WHERE ID = 1;

```
UPDATE Account SET Balance = Balance + 100 WHERE ID = 2;
COMMIT;
```

26. Explain ACID properties in the context of a transaction

ACID properties ensure reliable database transactions:

Atomicity: All operations in a transaction succeed or none do.

Consistency: Database remains in a valid state before and after the transaction.

Isolation: Concurrent transactions do not affect each other's execution.

Durability: Once committed, changes are permanent even after a system crash.

27. What is the difference between UNION and UNION ALL?

UNION: Combines results of two queries **removing duplicates**.
UNION ALL: Combines results of two queries **including duplicates**.

```
SELECT Name FROM Students
UNION
SELECT Name FROM Teachers;

SELECT Name FROM Students
UNION ALL
SELECT Name FROM Teachers;
```

28. How do you perform a self-join? Provide an example use case.

Self-Join: A table is joined with itself using aliases to compare rows within the same table.

Example Use Case: Find employees and their managers in the same table.

```
SELECT E1.Name AS Employee, E2.Name AS Manager
FROM Employees E1
LEFT JOIN Employees E2 ON E1.ManagerID = E2.EmployeeID;
```

29. What is the use of the CASE statement in SQL?
CASE Statement: Provides conditional logic in SQL queries to return values based on conditions.

```
SELECT Name,
    CASE
        WHEN Age < 18 THEN 'Minor'
        WHEN Age >= 18 AND Age < 60 THEN 'Adult'
        ELSE 'Senior'
```

```
    END AS AgeGroup
FROM Students;
```

30. How do you handle NULL values in queries?

Handling NULL values:

Check for NULL: Use IS NULL or IS NOT NULL
SELECT * FROM Students WHERE Email IS NULL;

Replace NULL: Use COALESCE() or IFNULL()

```
SELECT Name, COALESCE(Email, 'No Email') AS EmailAddress
FROM Students;
```

Ignore NULL in aggregates: Functions like COUNT, SUM automatically skip NULLs (except COUNT(*)).

31. What is a stored procedure?

Stored Procedure: A precompiled SQL program saved in the database that can be executed repeatedly.

Benefits:

Reusability

Faster execution

Improved security and reduced client-server traffic

```
CREATE PROCEDURE GetStudentByID(IN studentID INT)

BEGIN

   SELECT * FROM Students WHERE StudentID = studentID;

END;
```

32. What is a trigger? Give an example use case

Trigger: A special stored procedure that automatically executes in response to

INSERT, UPDATE, or DELETE events on a table.

Example Use Case: Automatically log changes to an Employee table.

```
CREATE TRIGGER LogSalaryChange
```

```
AFTER UPDATE ON Employee

FOR EACH ROW

BEGIN

   INSERT INTO SalaryLog(EmployeeID, OldSalary, NewSalary, ChangeDate)

   VALUES (OLD.EmployeeID, OLD.Salary, NEW.Salary, NOW());

END;
```

33. What is indexing? What are the different types of indexes?

Indexing: A database technique to speed up data retrieval by creating a data structure (like a pointer) for quick search.

Why Use Indexing:

Improves query performance.

Reduces search time in large tables.

Helps in sorting and enforcing uniqueness.

Types of Indexes:

Single-Column Index: Index on one column.

Composite (Multi-Column) Index: Index on multiple columns.

Unique Index: Ensures all values in the column(s) are unique.

Full-Text Index: Optimized for text search.

Clustered Index: Alters the physical order of data to match the index (only one per table).

Non-Clustered Index: Separate structure; does not change physical order (multiple per table).

34. What is the difference between clustered and non-clustered index

Feature	Clustered Index	Non-Clustered Index
Data Storage	Alters the physical order of rows in the table.	Separate structure; data remains unordered.
Number per Table	Only one allowed per table.	Multiple indexes allowed per table.

Feature	Clustered Index	Non-Clustered Index
Speed	Faster for range queries.	Slightly slower; uses pointers to data.
Use Case	Primary key or frequently searched column.	Columns used in joins or search conditions.

35. Explain the concept of views.

View: A virtual table in SQL that displays data from one or more tables based on a query. It **does not store data physically**.

Key Points:

Simplifies complex queries.

Provides data security by restricting access to certain columns or rows.

Can be **updatable** (if based on a single table) or **read-only** (if complex).

Acts like a stored query that can be reused in other queries.

CREATE VIEW AdultStudents AS

SELECT Name, Age

FROM Students

WHERE Age >= 18;

36. What is a materialized view? How is it different from a regular view?

Materialized View: A database object that **stores the result of a query physically** and can be refreshed periodically.

Difference from Regular View:

Feature	Regular View	Materialized View
Storage	No physical storage; virtual only	Stores query results physically
Performance	Computed on-demand each time	Faster retrieval as data is precomputed
Freshness	Always reflects current data	May be stale until refreshed
Use Case	Simplify queries, security	Improve performance for complex/large queries

37. What is the difference between OLAP and OLTP databases?

Feature	OLTP (Online Transaction Processing)	OLAP (Online Analytical Processing)
Purpose	Manages day-to-day transactional data	Used for data analysis and reporting
Data Volume	Small, current data	Large, historical data
Operations	Insert, Update, Delete	Complex queries, aggregations
Speed	Fast for transactions	Optimized for read-heavy queries
Schema	Normalized tables	Often denormalized, star/snowflake schema
Users	Clerks, operational staff	Analysts, management

38. How do you optimize a slow-running SQL quer

To optimize a slow-running SQL query, you can follow these strategies:

Use Indexes: Create indexes on columns used in WHERE, JOIN, and ORDER BY.

Avoid SELECT *: Select only required columns.

Use Joins Efficiently: Prefer INNER JOIN over OUTER JOIN if possible.

Filter Early: Apply WHERE conditions before joins or aggregations.

Use LIMIT: Restrict number of rows returned if possible.

Analyze Execution Plan: Use EXPLAIN to identify bottlenecks.

Avoid Functions on Indexed Columns: Functions like UPPER(col) prevent index usage.

Consider Denormalization: For complex queries, sometimes combining

tables improves speed.

Partition Large Tables: Break large tables into smaller partitions.

Optimize Subqueries: Replace correlated subqueries with joins if possible.

These steps help reduce query execution time and improve database performance.

39. What is a deadlock in database systems? How do you prevent it?

Deadlock: A situation where two or more transactions **wait indefinitely** for each other to release locks, causing a standstill.

Prevention Methods:

Lock Ordering: Access resources in a consistent order to avoid circular waits.

Timeouts: Automatically abort transactions that wait too long.

Use Smaller Transactions: Keep transactions short to reduce lock duration.

Avoid Unnecessary Locks: Only lock the data that is required.

Deadlock Detection: Let the database detect deadlocks and roll back one of the transactions.

40. What are foreign key constraints, and how do they help maintain referential integrity?
Foreign Key Constraint: A rule that ensures a column (or set of columns) in one table matches a primary key in another table.

How it maintains referential integrity:

Prevents inserting values in the child table that don't exist in the parent table.

Prevents deleting or updating parent rows if child rows depend on them (unless ON DELETE/UPDATE CASCADE is used).

```
CREATE TABLE Orders (
   OrderID INT PRIMARY KEY,
   CustomerID INT,
   FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

Here, every CustomerID in Orders must exist in Customers.

41. How would you design a database for an online bookstore?

Here's a simple design for an **online bookstore database**:

Entities and Attributes:

Books

BookID (PK), Title, AuthorID (FK), Genre, Price, Stock

Authors

AuthorID (PK), Name, Bio

Customers

CustomerID (PK), Name, Email, Phone, Address

Orders

OrderID (PK), CustomerID (FK), OrderDate, TotalAmount

OrderDetails

OrderDetailID (PK), OrderID (FK), BookID (FK), Quantity, Price

Categories/Genres *(optional)*

CategoryID (PK), CategoryName

Relationships:

Books ↔ Authors: Many-to-One (many books can have one author).

Orders ↔ Customers: Many-to-One (many orders per customer).

Orders ↔ Books: Many-to-Many via OrderDetails.

Books ↔ Categories: Many-to-One (optional).

Additional Notes:

Use **primary keys** for uniqueness.

Use **foreign keys** for referential integrity.

Consider **indexes** on frequently searched columns like Title or AuthorID.

42. How do you enforce uniqueness in a column?

To **enforce uniqueness in a column** in a database, you use a **UNIQUE constraint**.

Explanation:

A **UNIQUE constraint** ensures that all values in a column (or a combination of columns) are **distinct** — no two rows can have the same value in that column.

```
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    email VARCHAR(100) UNIQUE
```

);

Example 2: Add Unique Constraint Later

ALTER TABLE Students

ADD CONSTRAINT unique_email UNIQUE (email);

Example 3: Unique Across Multiple Columns (Composite Uniqueness)

CREATE TABLE Enrollment (

student_id INT,

course_id INT,

UNIQUE (student_id, course_id)

);

A **PRIMARY KEY** also enforces uniqueness, but it allows only **one** per table and **cannot be NULL**.

A **UNIQUE constraint** can be used on **multiple columns** and **can allow NULL values** (though behavior may vary slightly by DBMS).

43. What are the implications of using a composite primary key?

Using a **composite primary key** — a primary key made up of **two or more columns** — has several **implications** in database design.

A **composite primary key** is used when no single column can uniquely identify a row, but the **combination** of multiple columns can.

CREATE TABLE Enrollment (

student_id INT,

course_id INT,

PRIMARY KEY (student_id, course_id)

);

1. **Uniqueness Across Multiple Columns**

The **combination** of the columns must be unique, not each individual column.

Example: The same student_id can appear multiple times (for different courses).

2. Foreign Key Relationships Become More Complex

When another table references this table, the **foreign key** must include **all** columns of the composite key.

```
CREATE TABLE Grades (

    student_id INT,

    course_id INT,

    grade CHAR(2),

    FOREIGN KEY (student_id, course_id) REFERENCES Enrollment(student_id, course_id)

);
```

3. Indexing and Performance

A composite primary key automatically creates a **composite index** on the key columns.

Query performance can depend on the **order** of columns in the composite key.

Example: An index on (student_id, course_id) helps queries filtering by both, or by student_id first — but not necessarily by course_id alone.

4. Data Redundancy Reduction

It avoids creating a separate surrogate key (like an auto-increment ID) when natural keys already exist.

Keeps data more meaningful.

5. **Increased Join Complexity**

Joins on tables with composite keys require **multiple column matches**, making SQL queries more verbose.

Potential Maintenance Issues

If one of the composite columns changes frequently, updating all related foreign key tables can become cumbersome.

Use a **composite key** when:
The combination of natural attributes uniquely identifies a record.

You want to enforce real-world constraints (e.g., one student per course).

Avoid it when:

It complicates relationships excessively.

You can introduce a simple **surrogate key (like** id **INT AUTO_INCREMENT)** for simplicity.

A composite primary key ensures uniqueness using multiple columns, useful in many-to-many relationships, but can make foreign key references and joins more complex.

44. Explain how you would back up and restore a database

Backing up and restoring a database are critical tasks for data protection and recovery in case of data loss, corruption, or system failure.

1. What Is a Database Backup?

A **backup** is a copy of the entire database (or parts of it) stored safely, so it can be restored if needed.
It can include data files, schemas, and transaction logs.

2. Types of Database Backups

Type	Description
Full Backup	Copies the entire database — all tables, schemas, and data.
Differential Backup	Copies only data changed since the **last full backup**.
Incremental Backup	Copies only data changed since the **last backup** (full or incremental).
Transaction Log Backup	Captures all transactions since the last log backup — useful for point-in-time recovery.

mysqldump -u root -p mydatabase > mydatabase_backup.sql

mysqldump -u root -p mydatabase customers orders > selected_backup.sql

mysqldump --routines --triggers --events -u root -p mydatabase > full_backup.sql

pg_dump -U postgres -F c -b -v -f "backup_file.backup" mydatabase

4. How to Restore a Database

mysql -u root -p mydatabase < mydatabase_backup.sql

pg_restore -U postgres -d mydatabase -v "backup_file.backup"

RESTORE DATABASE mydatabase

FROM DISK = 'C:\Backup\mydatabase_full.bak'

WITH REPLACE;

Schedule **regular automated backups** (daily or hourly for critical systems).

Store backups in **multiple locations** — on-site and off-site/cloud.

Test **restore procedures** periodically to ensure reliability.

Encrypt backups for security.

Monitor backup logs to detect failures early.

Summary

Task	MySQL Command	SQL Server Command
Backup	mysqldump -u root -p dbname > backup.sql	BACKUP DATABASE dbname TO DISK = 'file.bak'
Restore	mysql -u root -p dbname < backup.sql	RESTORE DATABASE dbname FROM DISK = 'file.bak'

To **back up** a database, create a copy of its data and structure using built-in tools or commands.
To **restore**, use those backup files to recreate the database in the same or another environment.

45. How do you handle concurrent database access?

Handling concurrent database access is crucial to ensure data consistency, accuracy, and integrity when multiple users or transactions try to access or modify data at the same time.

1. What Is Concurrent Access?

Concurrent access occurs when **multiple transactions** (users or processes) interact with the same data **simultaneously** — for example:

Two users trying to **update** the same record.

One user **reading** data while another is **writing** to it.

2. Problems Caused by Concurrency

Without proper handling, concurrency can lead to **data anomalies**, such as:

Problem	Description	Example
Dirty Read	A transaction reads uncommitted data from another transaction.	Transaction A updates a value; B reads it before A commits.
Non-Repeatable Read	Same query returns different results within one transaction.	A reads data twice; another transaction modifies it between reads.
Phantom	New rows are added or deleted by	A queries all “employees >

Problem	Description	Example
Read	another transaction during execution.	50000" — B adds one matching employee later.
Lost Update	Two transactions update the same record; one overwrites the other's change.	A and B both update salary; B's change overwrites A's.

3. Techniques to Handle Concurrent Access

(a) Transactions

Wrap multiple database operations in a **transaction** so they are executed as a single unit.

BEGIN TRANSACTION;

UPDATE accounts SET balance = balance - 500 WHERE id = 1;

UPDATE accounts SET balance = balance + 500 WHERE id = 2;

COMMIT;

Ensures **atomicity** — either all operations succeed or none do.(b) Isolation Levels

Databases provide **isolation levels** to control visibility of data between concurrent transactions.

Isolation Level	Prevents	Allows	Performance
Read Uncommitted	None	Dirty reads	Fastest
Read Committed	Dirty reads	Non-repeatable & phantom reads	Moderate
Repeatable Read	Dirty & non-repeatable reads	Phantom reads	Slower
Serializable	All anomalies	None	Slowest (strictest)

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRANSACTION;

-- queries

COMMIT;

(c) Locking MechanismsLocks prevent multiple transactions from modifying the same data simultaneously.

Type	Description
Shared Lock (Read Lock)	Allows multiple reads but no writes.
Exclusive Lock (Write Lock)	Prevents other reads/writes on that data.
Row-level Locking	Locks only the row being modified.
Table-level Locking	Locks the entire table (less concurrent).

SELECT * FROM accounts WHERE id = 1 FOR UPDATE;

(d) Optimistic Concurrency ControlUsed when conflicts are **rare**.
Each row has a **version number** or **timestamp**.
Before updating, the system checks if the record's version matches the one initially read.

UPDATE products

SET price = 120, version = version + 1

WHERE product_id = 10 AND version = 3;

If no rows are affected → another transaction modified it → retry.(e) Pessimistic Concurrency Control

Used when conflicts are **frequent**.
Locks data before accessing it to prevent others from changing it until the operation completes.

(f) Connection Pooling & Queueing

Use **connection pools** to manage limited database connections efficiently.

Use **queueing systems** (like Kafka, RabbitMQ) for ordered, asynchronous updates.

4. Best Practices

Choose the **appropriate isolation level** (not always the strictest).

Keep transactions **short** to reduce lock contention.

Avoid manual locks unless necessary.

Use **indexes** to make read locks more granular.

Regularly monitor for **deadlocks** and handle retries gracefully.

Concept	Purpose
Transactions	Ensure atomic operations
Isolation Levels	Control visibility between transactions
Locks	Prevent conflicting updates

Concept	Purpose
Optimistic Concurrency	Detect conflicts using versioning
Pessimistic Concurrency	Prevent conflicts using locks

46. What is a database trigger, and when would you use one?

A **database trigger** is a **special stored procedure** that **automatically executes** (or "fires") in response to a **specific event** on a table or view — such as an **INSERT**, **UPDATE**, or **DELETE** operation.

```
CREATE TRIGGER update_timestamp

BEFORE UPDATE ON employees

FOR EACH ROW

SET NEW.last_modified = NOW();
```

3. Types of Triggers

Type	Description
BEFORE Trigger	Executes **before** the triggering action (e.g., before inserting data). Useful for validation or modification of data.
AFTER Trigger	Executes **after** the triggering action. Useful for logging or auditing.
INSTEAD OF Trigger	(Mainly in views) Executes **in place** of the triggering action — often used to modify data in underlying base tables.
ROW-Level Trigger	Executes once **per row** affected.
STATEMENT-Level Trigger	Executes once **per SQL statement**, regardless of rows affected.

4. Example Use Cases

a. Enforcing Business Rules
Automatically validate or modify data before insertion.

```
CREATE TRIGGER check_salary

BEFORE INSERT ON employees

FOR EACH ROW

BEGIN

  IF NEW.salary < 0 THEN
```

```
   SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Salary cannot be
negative';

  END IF;

END;
```

b. Maintaining Audit Logs

Track who changed what and when.

```
CREATE TRIGGER check_salary

BEFORE INSERT ON employees

FOR EACH ROW

BEGIN

  IF NEW.salary < 0 THEN

   SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Salary cannot be
negative';

  END IF;

END;
```

b. Maintaining Audit Logs

Track who changed what and when.

```
CREATE TRIGGER log_update

AFTER UPDATE ON employees

FOR EACH ROW

INSERT INTO audit_log(emp_id, action, changed_at)

VALUES (NEW.emp_id, 'UPDATE', NOW());
```

c. Cascading Changes

Automatically update related tables when a parent table changes.

```
CREATE TRIGGER update_order_total
```

```
AFTER INSERT ON order_items

FOR EACH ROW

UPDATE orders

SET total = total + NEW.price

WHERE order_id = NEW.order_id;
```

d. Preventing Unauthorized Actions

Restrict deletion of certain records.

```
CREATE TRIGGER prevent_admin_delete

BEFORE DELETE ON users

FOR EACH ROW

BEGIN

  IF OLD.role = 'Admin' THEN

    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Cannot delete admin
users';

  END IF;

END;
```

6. Summary

Aspect	Description
Definition	SQL code that runs automatically when a table/view changes
Purpose	Enforce rules, maintain logs, cascade updates, validate data
Trigger Timing	BEFORE, AFTER, or INSTEAD OF an operation
Common Uses	Auditing, validation, auto-updates, restricting actions
Caution	Use carefully to avoid hidden logic and performance issues

5. When *Not* to Use Triggers

When they make logic **too complex or hidden** (hard to debug).

When performance may be affected by too many triggers firing.

When the same effect can be achieved with **constraints** or **application logic**.

A **database trigger** is an automated action executed in response to data changes (INSERT, UPDATE, DELETE).
You use triggers to **enforce business rules**, **maintain audit trails**, or **automate cascading updates** without manual intervention.

47. How do you manage user permissions in a database?

Managing user permissions in a database is essential for maintaining security, data integrity, and controlled access to sensitive information.1. What Are User Permissions?

User permissions define **what actions** a user or role can perform on database objects (tables, views, procedures, etc.).
Examples of actions include:

SELECT → Read data

INSERT → Add data

UPDATE → Modify data

DELETE → Remove data

EXECUTE → Run stored procedures

2. Steps to Manage User Permissions

Step 1: Create a User

Each database system has commands to create users.

MySQL:

```
CREATE USER 'john'@'localhost' IDENTIFIED BY 'password123';
```

PostgreSQL:

```
CREATE USER john WITH PASSWORD 'password123';
```

SQL Server:

```
CREATE LOGIN john WITH PASSWORD = 'password123';

CREATE USER john FOR LOGIN john;
```

Step 2: Grant Permissions

Assign specific privileges to the user.

Example (MySQL):

GRANT SELECT, INSERT ON bookstore.* TO 'john'@'localhost';

PostgreSQL:

GRANT SELECT, UPDATE ON TABLE students TO john;

SQL Server:

GRANT SELECT, UPDATE ON students TO john;

Step 3: Revoke Permissions

Remove access if no longer needed.

REVOKE INSERT ON bookstore.* FROM 'john'@'localhost';

Step 4: Use Roles for Easier Management
Instead of managing each user individually, assign permissions to **roles** and then assign users to those roles.

CREATE ROLE librarian;

GRANT SELECT, INSERT, UPDATE ON books TO librarian;

GRANT librarian TO john;

Step 5: Apply the Principle of Least Privilege

Always grant users **only the permissions they actually need** to do their job.

Role	Typical Permissions
Admin	ALL PRIVILEGES
Analyst	SELECT only
Editor	SELECT, INSERT, UPDATE
Viewer	SELECT only

3. Common Privilege Types

Privilege	Description
SELECT	Read data from tables or views
INSERT	Add new records
UPDATE	Modify existing records
DELETE	Remove records
CREATE	Create new database objects
DROP	Delete database objects
EXECUTE	Run stored procedures or functions
GRANT OPTION	Allow user to grant permissions to others

4. Security Best Practices

Use **roles** instead of granting privileges directly to users.

Enforce **strong passwords** and **authentication**.

Regularly **audit user privileges** and revoke unnecessary ones.

Separate **admin** and **user** accounts.

Use **views** to restrict data visibility instead of giving table-level access.

Implement **row-level security** for sensitive records (e.g., in PostgreSQL).

```
-- 1. Create user

CREATE USER 'john'@'localhost' IDENTIFIED BY 'password123';

-- 2. Grant limited permissions

GRANT SELECT, UPDATE ON company.employees TO 'john'@'localhost';

-- 3. Revoke permission later if needed

REVOKE UPDATE ON company.employees FROM 'john'@'localhost';
```

You manage user permissions by **creating users**, **granting appropriate privileges**, and **revoking or modifying them** as needed — often using **roles** to simplify management.
Always follow the **principle of least privilege** to keep your database secure.

48. What is sharding in database systems?

Sharding is a database architecture technique used to split large databases into smaller, faster, and more manageable pieces called shards — each shard holds a subset of the data.

It's mainly used to improve performance, scalability, and availability for large-scale applications.1. Definition

Sharding is the process of **horizontally partitioning** a database — that is, dividing rows of a table across multiple databases or servers.

Each shard:

Contains its **own independent subset** of data.

Has the **same schema** as the original database.

Is hosted on a **different database server**.

2. Example

Imagine a database of 100 million users.

Instead of keeping all users in one table, we **split** them:

Shard	Data Stored
Shard 1	Users with IDs 1–10 million
Shard 2	Users with IDs 10–20 million
Shard 3	Users with IDs 20–30 million

Each shard handles only part of the data — reducing load and increasing performance.

3. How Sharding Works

Sharding is usually done based on a **shard key**, a column used to decide which shard a particular record belongs to.

Example: Shard Key = user_id

user_id % 4

If result = 0 → Store in Shard 1

If result = 1 → Store in Shard 2

If result = 2 → Store in Shard 3

If result = 3 → Store in Shard 4

4. Types of Sharding

Type	Description
Horizontal Sharding	Split data **by rows** across multiple databases (most common).
Vertical Sharding	Split data **by columns** — e.g., store user profile info in one DB and transaction data in another.
Directory-Based Sharding	Use a **lookup service** to find which shard holds which data (flexible but adds lookup overhead).

5. Advantages of Sharding

Benefit	Explanation
Improved Performance	Each shard handles fewer queries → faster responses.
Better Scalability	You can add new shards (servers) as data grows.
High Availability	If one shard fails, others remain operational.
Reduced Load	Balances workload across multiple servers.

6. Disadvantages / Challenges

Challenge	Description
Complexity	Application must know where data lives (routing logic).
Rebalancing	Adding/removing shards requires redistributing data.
Cross-Shard Queries	Joins across shards are difficult and slow.
Backup/Restore	Each shard must be backed up separately.

7. When to Use Sharding

Use sharding when:

Your database has **massive data volume** (hundreds of millions of rows).

Read/write load exceeds the capacity of a single server.

You need **horizontal scalability** — adding more servers instead of upgrading one.

8. Real-World Examples

Facebook: User data is sharded by user ID.

YouTube: Video metadata and comments stored in separate shards.

E-commerce platforms: Orders distributed across shards based on region or customer ID.

Summary

Aspect	Description
Definition	Splitting a large database into smaller parts (shards).
Purpose	Improve performance, scalability, and availability.
Key Concept	Each shard holds a subset of data with the same schema.
Challenge	Harder joins and data management across shards.

Sharding is a way to **horizontally partition** a large database into smaller, distributed pieces (shards) to improve **performance, scalability, and fault tolerance**, especially for large-scale, high-traffic systems.

49. What is the difference between horizontal scaling and vertical scaling in databases?

The difference between horizontal scaling and vertical scaling in databases lies in how you increase capacity and performance as your data or traffic grows.1. Definition Overview

Scaling Type	Description
Vertical Scaling (Scale-	Add **more power** (CPU, RAM, storage) to a **single**

Scaling Type	Description
Up)	**server**.
Horizontal Scaling (Scale-Out)	Add **more servers/nodes** to distribute the load across multiple machines.

2. Vertical Scaling (Scale-Up)

How it works

You keep a **single database server** but upgrade its **hardware** — e.g.:

Add more **CPU cores**

Increase **RAM**

Use **faster SSDs**

Example

If your MySQL server is slow:

Move it from 8 GB → 64 GB RAM

Upgrade from 4-core → 16-core CPU

Advantages

Simple to implement — no changes to application or database structure.

No data partitioning or synchronization issues.

Ideal for small to medium workloads.

Disadvantages

Hardware limits — there's a maximum CPU/RAM you can add.

Single point of failure — if the server crashes, database goes down.

Expensive — high-end hardware costs more per performance unit.

3. Horizontal Scaling (Scale-Out)

How it works

Instead of one big server, you **add more servers (nodes)** and **distribute** data or queries among them.

This can be done via:

Sharding (splitting data by user ID, region, etc.)

Replication (multiple read replicas)

Distributed databases (like MongoDB, Cassandra)

Example

Split your users table across multiple servers:

Shard 1: Users A–M

Shard 2: Users N–Z

Advantages

Virtually unlimited scalability — just add more nodes.

High availability — one node can fail without total downtime.

Load balancing — traffic is shared across servers.

Disadvantages

More complex architecture (data sharding, routing, synchronization).

Cross-node queries (joins across shards) are difficult.

Requires application logic changes to route queries to the right node.

---

4. Quick Comparison Table

Feature	Vertical Scaling	Horizontal Scaling
Approach	Add power to existing server	Add more servers
Complexity	Simple	Complex
Cost	Expensive hardware	Cheaper commodity hardware
Scalability Limit	Limited by hardware	Nearly unlimited
Downtime Required	Often yes	Usually no
Failure Impact	Single point of failure	Fault-tolerant
Examples	Upgrading MySQL server	Sharding MySQL across nodes

---

5. Real-World Examples

Company	Approach	Description
Small businesses	Vertical	Use one strong SQL server

Company	Approach	Description
Facebook / Google / Amazon	Horizontal	Use clusters of distributed databases for billions of users

---

6. Summary

Vertical Scaling: “Bigger machine” — add more power to one server.

Horizontal Scaling: “More machines” — distribute data and load across multiple servers.

In short:

Vertical scaling = Upgrading your existing database server’s hardware.

Horizontal scaling = Adding more servers and spreading the data or traffic among them for better scalability and fault tolerance.

50. How would you approach migrating a database from MySQL to PostgreSQL?

Migrating a database from **MySQL → PostgreSQL** involves moving both the **schema (structure)** and the **data (records)** while ensuring that the **application continues to work correctly** with the new system.

Here’s a structured step-by-step approach

---

1. Understand the Differences

Before starting, know the key differences between MySQL and PostgreSQL:

Feature	MySQL	PostgreSQL
Storage Engine	InnoDB	Native MVCC
Data Types	Limited JSON support	Rich JSONB, Array, ENUM, UUID
Case Sensitivity	Not case-sensitive by default	Case-sensitive by default
Functions/Syntax	Different string & date functions	Slightly different syntax
Auto Increment	AUTO_INCREMENT	SERIAL or IDENTITY

Knowing these helps plan conversions properly.

---

2. Pre-Migration Planning

Identify all **databases, tables, views, triggers, and stored procedures**.

Review **data types** and **incompatible features** (like ENUMs, unsigned ints, etc.).

Plan **downtime or minimal downtime** if it's a production migration.

Take **backups** of your MySQL database before migration.

---

3. Schema Conversion

You can do this manually or using tools.

Option 1: Use AWS Schema Conversion Tool (SCT)

Converts MySQL schema and most objects automatically to PostgreSQL-compatible syntax.

Option 2: Use pgloader

A powerful open-source tool that automates schema and data migration.

pgloader mysql://user:password@localhost/mydb
postgresql://user:password@localhost/mydb

This command connects to MySQL, creates equivalent PostgreSQL tables, and copies the data.

Option 3: Manual Conversion Example

MySQL	PostgreSQL Equivalent
INT AUTO_INCREMENT	SERIAL
TINYINT(1)	BOOLEAN
DATETIME	TIMESTAMP
DOUBLE	DOUBLE PRECISION
ENUM('A','B')	TEXT CHECK (value IN ('A','B'))

---

4. Data Migration

If you don't use pgloader, you can use export/import:

Step 1: Export data from MySQL

mysqldump --compatible=postgresql --no-create-info mydb > data.sql

Step 2: Import data into PostgreSQL

psql -U postgres -d mydb -f data.sql

Tip: Disable foreign keys and indexes during bulk import to improve performance, then re-enable them afterward.

---

5. Application Compatibility

Update your **connection strings** to point to PostgreSQL.

Check SQL queries for **syntax differences**:

LIMIT clause works, but AUTO_INCREMENT and date functions differ.

Replace MySQL-specific functions (like NOW(), CONCAT_WS(), etc.) with PostgreSQL equivalents.

Test **transactions**, **joins**, and **stored procedures**.

---

6. Testing Phase

Verify:

Schema structure (tables, constraints, indexes).

Data integrity (record counts match).

Application functionality (queries return expected results).

Use scripts to compare record counts between databases:

SELECT COUNT(*) FROM table_name;

---

7. Go Live

Plan a **final sync** (migrate recent changes since last backup).

Redirect the application to PostgreSQL.

Monitor performance, query execution, and logs.

---

8. Post-Migration Optimization

Analyze and **create indexes** where needed (ANALYZE, VACUUM).

Adjust **PostgreSQL configuration parameters** (work_mem, shared_buffers, etc.).

Monitor queries using **pg_stat_statements**.

Summary

Step	Description
1.	Analyze MySQL and PostgreSQL differences
2.	Plan and back up MySQL database
3.	Convert schema (pgloader / AWS SCT / manual)
4.	Migrate data (pgloader / dump & restore)
5.	Update and test application
6.	Verify data integrity and performance
7.	Final sync and go live
8.	Optimize and monitor PostgreSQL system

In short: To migrate from **MySQL to PostgreSQL**, you convert the schema, transfer data, adapt queries, and validate application behavior — using tools like **pgloader** or **AWS SCT** to simplify and automate most of the process.