VANSHIKA MISHRA SQL INTERVIEW QUESTIONS -THEORY

## 1. What is the SQL query execution sequence?

SQL logically processes query clauses in this order:
FROM → JOIN --> WHERE →GROUP BY → HAVING → SELECT --> DISTINCT--> ORDER BY --> LIMIT/TOP
Example :

```
SELECT dept, COUNT(*)
FROM Employee
WHERE salary > 50000
GROUP BY dept
HAVING COUNT(*) > 5
ORDER BY dept;
```

## 2. What is Normalization?

Normalization is the systematic process of reducing data redundancy by dividing a large table into smaller related tables.
It prevents insert/update/delete anomalies and improves DML performance, but SELECT queries may slow due to joins.
Example:
Split one EmployeeDetails table into Employee and Department tables.

## 3. Degrees of Normalization

1NF: Values must be atomic, no repeating groups, and primary key must exist.
Example: Phone numbers in one cell are split into separate rows.

**Original Table:**

| CustomerID | Name | Phone no. |
|---|---|---|
| 1 | Jia ria | 8978847383 |
| 2 | John Doe | 7899748899, 8899278299 |
| 3 | Smith tie | 9877382892 |

**1NF Table:**

| CustomerID | Name | Phone no. |
|---|---|---|
| 1 | Jia ria | 8978847383 |
| 2 | John Doe | 7899748899 |
| 3 | Smith tie | 9877382892 |
| 4 | John Doe | 8899278299 |

2NF:- Must be already in 1NF and remove partial dependency from a composite key.

**Original Table:**

| OrderID | ProductID | ProductName | Category | Price |
|---|---|---|---|---|
| 1 | 1 | Laptop | Electronics | 1000 |
| 2 | 2 | Smartphone | Electronics | 800 |
| 3 | 3 | Laptop | Electronics | 900 |

**2NF Tables:**

**Table 1: Products**

| ProductID | ProductName | Category |
|---|---|---|
| 1 | Laptop | Electronics |
| 2 | Smartphone | Electronics |

**Table 2: Orders**

| OrderID | ProductID | Price |
|---|---|---|
| 1 | 1 | 1000 |
| 2 | 2 | 800 |
| 3 | 1 | 900 |

VANSHIKA MISHRA SQL INTERVIEW QUESTIONS -THEORY

3NF:- Must be in 2NF and remove transitive dependency (non-key → non-key).

**Original Table:**

| CustomerID | OrderID | ProductID | Price | CustomerName | CustomerEmail |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1000 | John Doe | john@example.com |
| 2 | 2 | 2 | 800 | Jane Smith | jane@example.com |
| 3 | 3 | 1 | 900 | John Doe | johndoe@example.com |

**3NF Tables:**

**Table 1: Customers**

| CustomerID | CustomerName | CustomerEmail |
|---|---|---|
| 1 | John Doe | john@example.com |
| 2 | Jane Smith | jane@example.com |
| 3 | John Doe | johndoe@example.com |

**Table 2: Product**

| ProductID | ProductName |
|---|---|
| 1 | Laptop |
| 2 | Smartphone |

BCNF:- Stronger form of 3NF where every determinant must be a candidate key.

### 4. What are database objects?

Permanent: Tables, Views, Stored Procedures, Functions, Triggers, Indexes
Temporary: Cursors

### 5. What is Collation?

Collation defines the rules for comparing and sorting character data, considering language, case sensitivity, and accents.
Example:

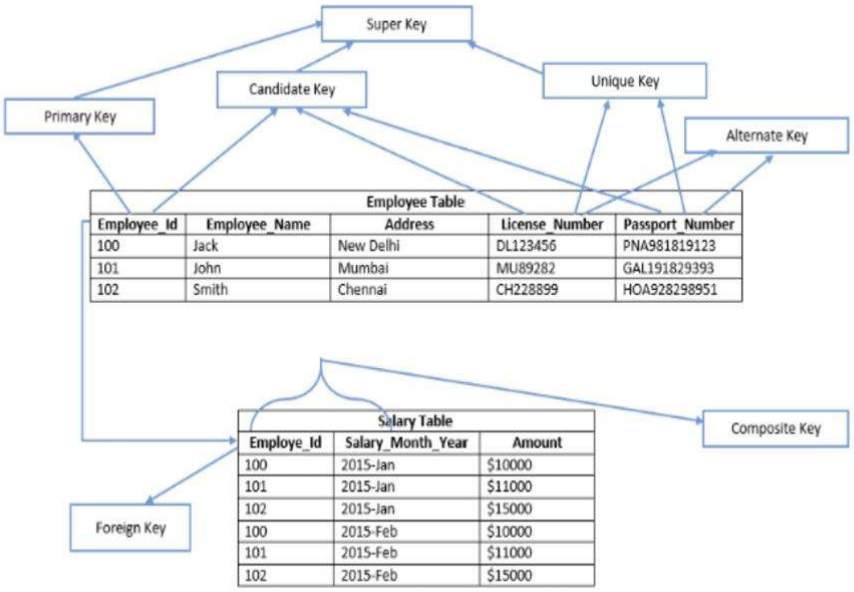
```
SELECT 'A' = 'a' COLLATE Latin1_General_CS_AS;  -- False
```

### 6. What is a Constraint?

Constraints enforce rules on tables.

**NOT NULL:** A column cannot have a NULL value by using the NOT NULL flag.
**UNIQUE:** A unique value makes sure that each value in a column is distinct.
**PRIMARY KEY:** A NOT NULL and UNIQUE combination. Identifies each table row in a unique way.
**FOREIGN KEY:** Prevent acts that would break linkages between tables.
**CHECK -** Verifies if the values in a column meet a certain requirement.
**DEFAULT:** If no value is specified, DEFAULT sets a default value for the column.
**CREATE INDEX -** Used to easily create and access data from the database.

## 7. What are different type of keys?




Types of keys

| Key Type | Definition | Key Properties | Example |
|---|---|---|---|
| Primary Key | Uniquely identifies each record in a table | Unique, Not Null, Only one per table | CustomerID |
| Candidate Key | A field(s) that *can* be a primary key | Unique + Not Null, Multiple per table | Email, Phone |
| Alternate Key | Candidate key not selected as primary key | Unique + Not Null | PAN Number |
| Super Key | Attribute(s) that uniquely identify a record (may include extra columns) | Unique (can contain unnecessary fields) | (Email, Phone) |
| Composite Key | Combination of attributes that together uniquely identify a record | Two or more fields, Unique as group | (OrderID + ProductID) |
| Compound/Composite Key | Similar to composite; combination of columns | Multi-column unique identifier | (StudentID, CourseID) |
| Foreign Key | Key that references a primary key in another table | Enforces referential integrity | CustomerID in Orders table |
| Unique Key | Ensures uniqueness of data in a field | Allows NULL (one), Multiple per table | Email |
| Surrogate Key | System-generated artificial identifier | No business meaning | AUTO_INCREMENT ID |
| Natural Key | Real-world attribute used as identifier | Comes from actual business data | Passport No |

| | | | |
|---|---|---|---|
| Secondary Key | Field used for searching & indexing | Not unique, may allow duplicates | Department, City |

### 8. What is a Derived Column?
A computed output column created dynamically in the SELECT query.
Example: SELECT FirstName + ' ' + LastName AS FullName FROM Employee;

### 9. What is a Transaction?
A set of SQL statements executed as one unit with ACID properties.
Example:
BEGIN TRAN
UPDATE Employee SET salary = 60000;
COMMIT;

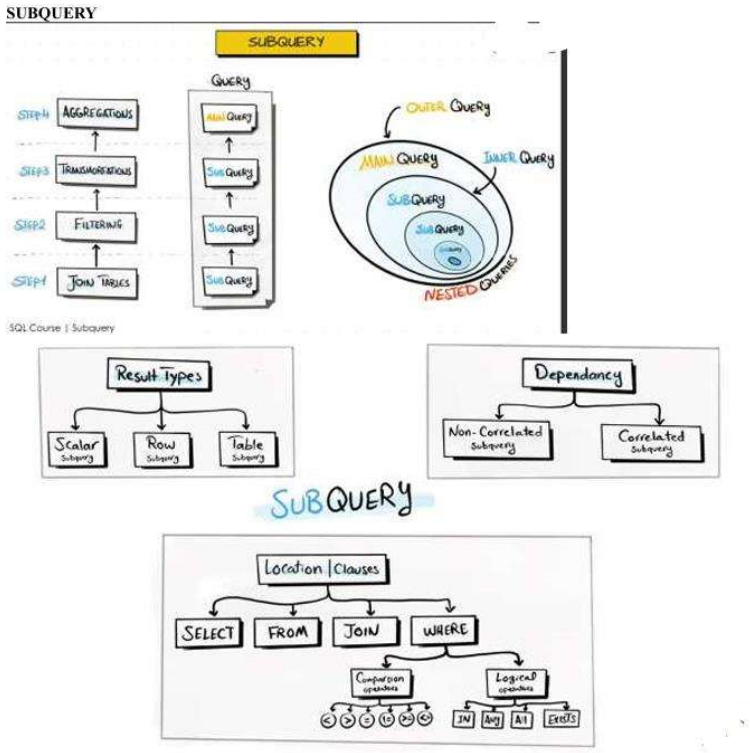
```
-- Start a transaction
BEGIN;
-- Attempt to insert a new record
INSERT INTO accounts (user_id, balance) VALUES (1, 100);
-- Something goes wrong, so we roll back the transaction
ROLLBACK;
-- Start another transaction
BEGIN;
-- Insert a new record
INSERT INTO accounts (user_id, balance) VALUES (1, 100);
-- Commit the transaction to save the changes
COMMIT;
```

### 10. OLTP vs OLAP
OLTP: normalized tables; optimized for DML; handles current data
OLAP: denormalized tables; optimized for SELECT; uses historical data
Example: Banking transactions (OLTP) vs sales reporting (OLAP)

### 11. Copy Only Structure of a Table
Use SELECT INTO with a false condition.
Example: SELECT * INTO EmpCopy FROM Employee WHERE 1=0;

### 12. What is a Subquery?
Query within another query; inner feeds outer; supports nesting.
Example:
SELECT name FROM Emp
WHERE deptid IN (SELECT id FROM Dept);

VANSHIKA MISHRA SQL INTERVIEW QUESTIONS -THEORY




| Usage Context | Operator / Function | SQL Example (MySQL) | Explanation | Subquery Return Type |
|---|---|---|---|---|
| SELECT clause | Scalar subquery | SELECT sale_id, (SELECT MAX(revenue) FROM amazon_sales) AS max_rev FROM amazon_sales; | Adds a single computed value to every row | Single scalar value |
| SELECT clause | Scalar subquery (LIMIT) | SELECT sale_id, (SELECT revenue FROM amazon_sales ORDER BY revenue DESC LIMIT 1) AS top_rev FROM amazon_sales; | Ensures subquery returns exactly one highest value | Single scalar value |
| WHERE clause | Comparison with scalar | SELECT * FROM amazon_sales WHERE revenue > (SELECT AVG(revenue) FROM amazon_sales); | Filters rows by comparing to a calculated single number | Single scalar value |
| WHERE clause | IN / NOT IN | SELECT * FROM amazon_sales WHERE customer_id IN (SELECT customer_id FROM amazon_customers WHERE country = 'US'); | Filters using a list of values returned by subquery | Multiple values |
| WHERE clause | EXISTS / NOT EXISTS | SELECT * FROM amazon_sales s WHERE | Checks if matching rows | Any number of rows |

| | | | | |
|---|---|---|---|---|
| | | EXISTS (SELECT 1 FROM returns r WHERE r.sale_id = s.sale_id); | exist; returns true/false, not data | |
| WHERE clause | ANY / SOME | SELECT * FROM amazon_sales WHERE revenue > ANY (SELECT revenue FROM amazon_sales WHERE department = 'Books'); | True if condition matches at least one row | Multiple values |
| WHERE clause | ALL | SELECT * FROM amazon_sales WHERE revenue > ALL (SELECT revenue FROM amazon_sales WHERE department = 'Toys'); | True only if condition matches every returned row | Multiple values |
| FROM clause | Derived table / Inline View | SELECT dept, AVG(rev) FROM (SELECT department AS dept, revenue AS rev FROM amazon_sales) AS dept_sales GROUP BY dept; | Treats subquery as a temporary table for further grouping/filtering | Multiple rows + columns |

### 13. Set Operators

Combine results of multiple SELECTs: UNION, UNION ALL, INTERSECT, EXCEPT
Columns count and datatypes must match.

| Command | Usage / What it Does | Example (Swiggy context) |
|---|---|---|
| UNION | Combines results and removes duplicates | SELECT Name FROM customers_signedup UNION SELECT Name FROM orders; |
| UNION ALL | Combines results and keeps duplicates | SELECT Name FROM customers_signedup UNION ALL SELECT Name FROM orders; |
| INTERSECT (Simulated) | Returns common rows (MySQL uses JOIN) | SELECT c.Name FROM customers_signedup c INNER JOIN orders o ON c.Name = o.Name; |
| EXCEPT (Simulated) | Returns rows in first query not in second | SELECT c.Name FROM customers_signedup c LEFT JOIN orders o ON c.Name = o.Name WHERE o.Name IS NULL; |

### 15. What is a Derived Table?

A subquery used as a virtual table inside the main query.
Example:
SELECT * FROM
(SELECT TOP 3 id FROM Sales ORDER BY amount DESC) AS T;

### 16. What is a View?

Virtual table defined by a SELECT; hides complexity; provides security; allows DML on exposed columns.
Example: CREATE VIEW vEmp AS SELECT name, dept FROM Employee;

| Topic | Details |
|---|---|
| **Definition** | A view is a **virtual table** based on an SQL query; it does **not store data** itself. |
| **Purpose** | - Simplify complex queries- Restrict access (security)- Show aggregated/calculated data- Provide backward compatibility |
| **Create Syntax** | CREATE VIEW view_name AS SELECT column1, column2 FROM table WHERE condition; |
| **Create Example** | CREATE VIEW SalesEmployees AS SELECT employee_id, name, salary FROM employees WHERE department = 'Sales'; |
| **Update Syntax** | UPDATE view_name SET column_name = value WHERE condition; |
| **Update Example** | UPDATE SalesEmployees SET salary = 60000 WHERE employee_id = 101; |
| **Drop Syntax** | DROP VIEW view_name; |
| **Drop Example** | DROP VIEW SalesEmployees; |

| Type | Description | Updatable? |
|---|---|---|
| **Simple View** | Based on one table; no joins or aggregations | ✅ Yes |
| **Complex View** | Involves joins, groupings, or aggregate functions | ❌ No (usually) |
| **Inline View** | Temporary view used as a subquery in the FROM clause | N/A |
| **Materialized View** | Physically stores query result; used for performance optimization; needs refresh | ❌ No (needs refresh) |

## 17. Types of Views

-

| Type | What It Is / Usage | Key Example |
|---|---|---|
| **Regular View** | A virtual table that stores a *saved SQL query*. Does **not store data**, just reads from base tables. | CREATE VIEW customer_orders AS SELECT c.name, o.order_total FROM customers c JOIN orders o ON c.id = o.customer_id; |
| **Schema-Bound View** | A view that is **tied to underlying tables** (WITH SCHEMABINDING). Base table structure **cannot be changed** without dropping/modifying the view. Prevents accidental schema changes. | CREATE VIEW customer_orders_sb WITH SCHEMABINDING AS SELECT c.name, o.order_total FROM dbo.customers c JOIN dbo.orders o ON c.id = o.customer_id; |
| **Indexed View** | A schema-bound view with a **physical index created on it**. The index stores data, making queries faster. Used for performance on big joins/aggregations. | CREATE UNIQUE CLUSTERED INDEX idx_order_view ON customer_orders_sb (name); |

## 18. What is an Indexed View?

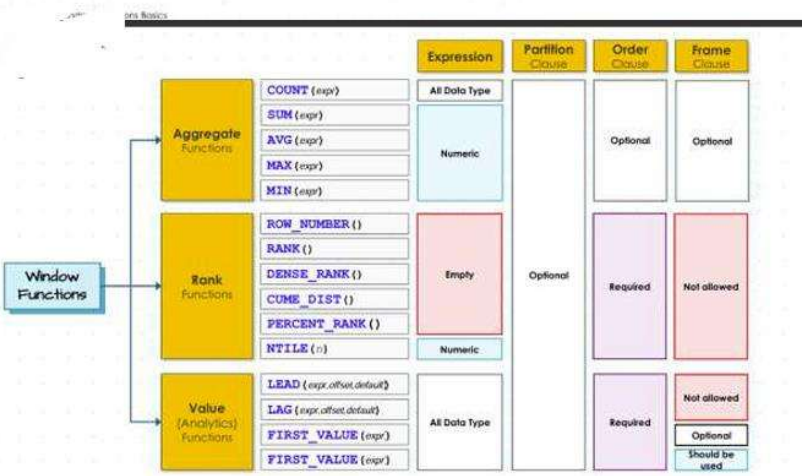
A view materialized on disk with a clustered index. Base table and view are kept synchronized.
Example: Create a view and index to aggregate sales.

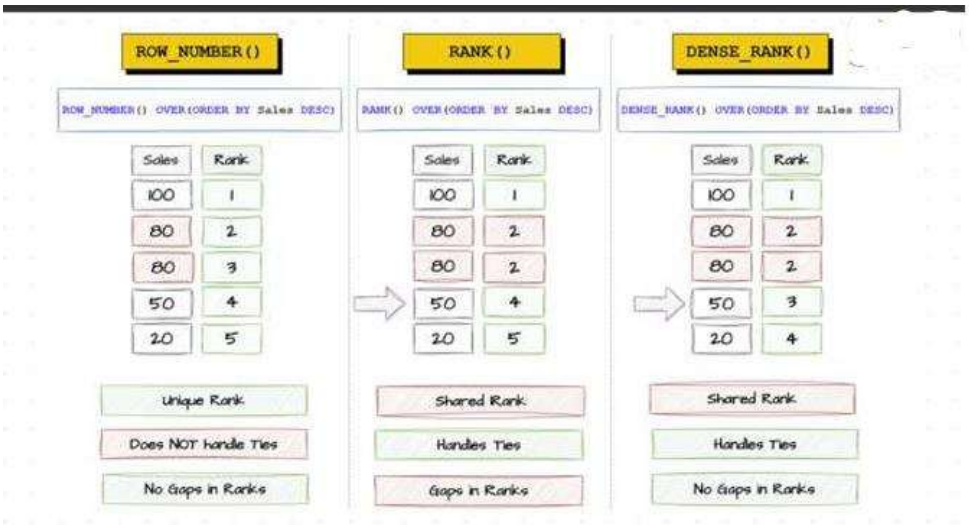
## 19. WITH CHECK OPTION

Prevents insert/update through the view that violates its filter condition.
Example:
CREATE VIEW vHR AS
SELECT * FROM Emp WHERE Dept = 'HR'
WITH CHECK OPTION;
- now if you insert any other dept data then it will not be allowed.

VANSHIKA MISHRA SQL INTERVIEW QUESTIONS -THEORY

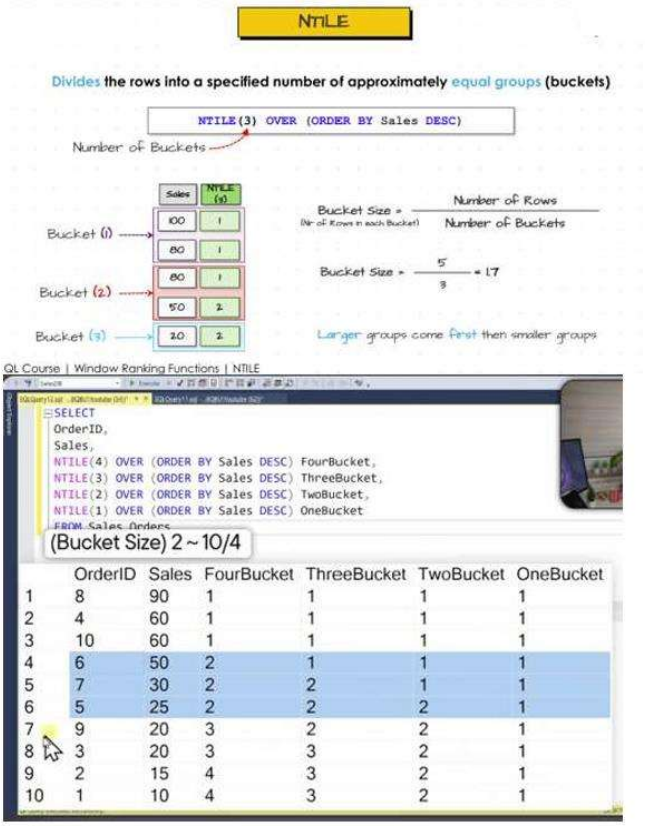
## 20. What are Ranking functions in MySQL?



VANSHIKA MISHRA SQL INTERVIEW QUESTIONS -THEORY

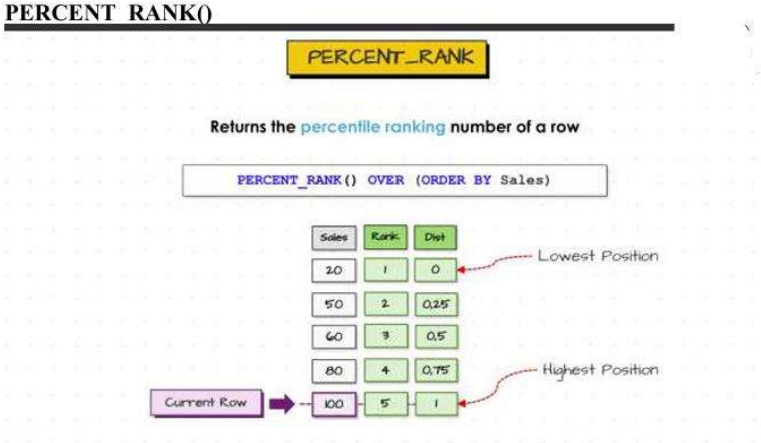
**NTILE**



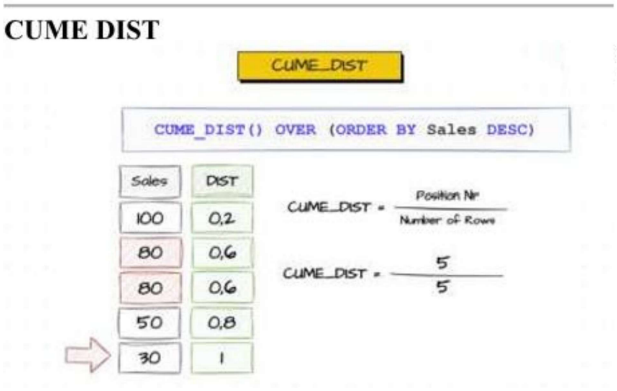

| | OrderID | Sales | FourBucket | ThreeBucket | TwoBucket | OneBucket |
|---|---|---|---|---|---|---|
| 1 | 8 | 90 | 1 | 1 | 1 | 1 |
| 2 | 4 | 60 | 1 | 1 | 1 | 1 |
| 3 | 10 | 60 | 1 | 1 | 1 | 1 |
| 4 | 6 | 50 | 2 | 1 | 1 | 1 |
| 5 | 7 | 30 | 2 | 2 | 1 | 1 |
| 6 | 5 | 25 | 2 | 2 | 2 | 1 |
| 7 | 9 | 20 | 3 | 2 | 2 | 1 |
| 8 | 3 | 20 | 3 | 3 | 2 | 1 |
| 9 | 2 | 15 | 4 | 3 | 2 | 1 |
| 10 | 1 | 10 | 4 | 3 | 2 | 1 |

PERCENT RANK :-
SELECT sale_id, department, revenue,
PERCENT_RANK() OVER (ORDER BY revenue) AS percent_rank
FROM amazon_sales;
Explanation:
• Formula: (rank - 1) / (total_rows - 1)
• Normalized rank between 0 and 1
• First row has 0, last row has 1 if no ties

**PERCENT_RANK()**



CUME_DIST():-




SELECT sale_id, department, revenue,
CUME_DIST() OVER (ORDER BY revenue) AS cume_dist
FROM amazon_sales;
• Formula: (number of rows with value ≤ current row) / total rows
• Value ranges from >0 to 1
• It answers: “What fraction of rows have this value or less?”

### 21. What is PARTITION BY?

PARTITION BY divides a result set into logical groups where ranking or aggregate restarts for each group.
Example:
SELECT Country, Sales,
  DENSE_RANK() OVER(PARTITION BY Country ORDER BY Sales DESC) AS drnk
FROM SalesInfo;

### 22. What is a temporary table ?

Temporary tables store session-scoped data and are automatically dropped at session close.
Example:
CREATE TEMPORARY TABLE temp_sales AS
SELECT * FROM SalesOrder WHERE YEAR(order_date)=2025;

### 23. Explain variables in MySQL

Variables store single session values.
User variables: @name
Session/system variables: SET/SHOW variables.
Example:
SET @target = 10000;
SELECT * FROM SalesOrder WHERE TotalDue > @target;

### 24. Explain Dynamic SQL

Dynamic SQL is SQL constructed and executed at runtime, commonly using prepared statements.
Example:
SET @tbl = 'SalesOrder';
SET @sql = CONCAT('SELECT COUNT(*) FROM ', @tbl);
PREPARE stmt FROM @sql;
EXECUTE stmt;
DEALLOCATE PREPARE stmt;

### 25. What is SQL Injection?
SQL injection is when user input manipulates a query to run destructive commands.
Mitigation in MySQL: prepared statements, parameter binding, whitelist validation.
Example attack input:
' OR 1=1 --

### 26. What is SELF JOIN?
Self join is joining a table to itself, usually where a row relates to another row in the same table.
Example:
```
SELECT e.name AS employee,
       m.name AS manager
FROM employees e
JOIN employees m ON e.manager_id = m.id;
```

### 27. What is a correlated subquery?
A correlated subquery references columns from the outer query and executes once per outer row.
Example:
```
SELECT e.*
FROM employees e
WHERE salary > (
  SELECT AVG(salary)
  FROM employees s
  WHERE s.department_id = e.department_id
);
```

### 28. Difference between subquery and correlated subquery
Regular subquery runs once and is independent.
Correlated runs repeatedly and depends on values from the outer query.

### 29. Difference between DELETE and TRUNCATE and DROP
DELETE removes selected rows, logs each row and can filter with WHERE.
TRUNCATE removes all rows, resets auto_increment, minimal logging, is faster.
DROP: Removes the table structure entirely.
Examples:
```
DELETE FROM orders WHERE status='cancelled';
TRUNCATE TABLE orders;
```

### 30. Loop constructs in stored programs (WHILE, REPEAT, LOOP)
Example:
```
SET @x = 5;
IF @x > 3
THEN SELECT 'Big';
END IF;
```

### 31. What is a stored procedure?
A stored procedure is a named block of SQL logic stored in the database that can accept parameters and return results.

```sql
-- Create a stored procedure to get employee details by ID
CREATE PROCEDURE GetEmployeeDetails(IN emp_id INT)
BEGIN
  SELECT * FROM employees WHERE id = emp_id;
END;
-- Call the stored procedure with an employee ID
CALL GetEmployeeDetails(101);
```

Stored procedures improve performance by reducing the number of compilations needed for repeated SQL operations.
They also enhance security by allowing users to execute complex operations without needing direct access to the underlying tables.
Additionally, they help maintain consistency in operations by centralizing logic that can be reused across applications.
Stored procedures can accept input parameters, return output parameters, and even handle complex logic with control-of-flow statements like IF, WHILE, and CASE.

### 32. Advantages of stored procedures

• Improve performance after caching
• Reduce network round trips
• Centralize business logic
• Improve security by restricting direct table access

### 33. What is a User Defined Function (UDF)?

A UDF returns a single scalar value or a table (in MySQL: scalar or stored program via RETURNS TABLE in MySQL 8.0.21+)
Example:

```
CREATE FUNCTION AddTax(p DECIMAL(10,2))
RETURNS DECIMAL(10,2)
DETERMINISTIC
RETURN p * 1.18;
```

### 34. Difference between stored procedure and UDF

Stored procedure: may return multiple result sets, can call dynamic SQL, can modify data.
UDF: must return a value, cannot modify data, deterministic logic preferred.

### 35. What is a trigger?

A trigger is stored logic that runs automatically when a table-level INSERT, UPDATE, or DELETE occurs.
Example:

```sql
-- Assume we have a table named 'employees' with columns 'id', 'name'
and 'last_modified'

-- Create Trigger
CREATE TRIGGER update_last_modified
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
  -- Update the 'last_modified' column to the current timestamp
  SET NEW.last_modified = CURRENT_TIMESTAMP;
END;
-- Example UPDATE operation that triggers the above trigger
UPDATE employees SET name = 'John Doe' WHERE id = 1;
```

## 36. Types of triggers in MySQL

• BEFORE INSERT/UPDATE/DELETE
• AFTER INSERT/UPDATE/DELETE

1. DML Trigger

DML Triggers are invoked when a DML statement such as INSERT, UPDATE, or DELETE occur which modify data in a specified TABLE or VIEW.

A DML trigger can query other tables and can include complex TSQL statements. They can cascade changes through related tables in the database.

They provide security against malicious or incorrect DML operations and enforce restrictions that are more complex than those defined with constraints.

2. DDL Trigger

Pretty much the same as DML Triggers but DDL Triggers are for DDL operations. DDL Triggers are at the database or server level (or scope).

DDL Trigger only has AFTER. It does not have INSTEAD OF.

3. Logon Trigger

Logon triggers fire in response to a logon event.

This event is raised when a user session is established with an instance of SQL server. Logon TRIGGER has server scope.

## 37. What are old and new triggers?

OLD refers to pre-change row; NEW refers to post-change row.

- **OLD** stores the row's value before the change
- **NEW** stores the row's value after the change
  Used to validate, compare, or log data in trigger bodies.

Example:

```
CREATE TRIGGER audit_update
BEFORE UPDATE ON products
FOR EACH ROW
```

```
INSERT INTO audit(product_id, old_price, new_price)
VALUES (OLD.id, OLD.price, NEW.price);
```

### 38. What are the three types of error handling in SQL?

Error handling allows a program to control what happens when a runtime failure occurs instead of crashing.

**DECLARE HANDLER**

Errors are handled inside stored procedures using DECLARE HANDLER.

**@@ERROR**

SQL Server only, not in MySQL.

**Custom error:-**

MySQL uses SIGNAL SQLSTATE to raise custom errors.

**Example**

```
CREATE PROCEDURE demo()
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
  SET @msg = 'Error occurred';

  INSERT INTO invalid_table VALUES (1);
END;
```

### 39. Explain Cursors

A cursor is a control structure used to fetch and process **rows one at a time**.

Useful only when set-based operations cannot solve a problem.

- SQL operates best on sets.
- Cursors force procedural row-by-row logic, which is slower.
- Used in stored routines for tasks like aggregation or sequential calculations.

**Example**

```
DECLARE done INT DEFAULT 0;
DECLARE uid INT;
DECLARE cur CURSOR FOR SELECT id FROM users;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

OPEN cur;
read_loop: LOOP
  FETCH cur INTO uid;
  IF done = 1 THEN LEAVE read_loop; END IF;
  UPDATE users SET status='active' WHERE id=uid;
END LOOP;
CLOSE cur;
```

### 40. Difference between Table Scan and Index Seek

**Table Scan:-** Full scan of all pages of a table.

Occurs when:

- No index exists
- A function is applied to a column
- Search predicate is non-selective (LIKE '%x')

**Index Seek :-** Optimizer navigates through the B-Tree to jump directly to the matching key.

Fastest lookup method.

**Example**
SELECT * FROM users WHERE id = 10;  -- Seek
SELECT * FROM users WHERE name LIKE '%a%'; -- Scan

### 41. Why are DML operations slower with indexes?
Indexes must be maintained every time data changes.
**Theory**
- Insert → add entry in index + possible page split
- Update → update index keys if indexed column changed
- Delete → remove index entry, leaves empty space
- More indexes = more overhead

**Example**
UPDATE users SET email='a@x.com' WHERE id=5;

### 42. Architecture:-
MySQL (InnoDB) stores data at multiple layers.
- InnoDB tables exist in tablespaces
- **Pages** are the smallest unit of I/O (default 16KB)
- 64 contiguous pages = **Extent**
- Multiple extents form segments → table

Efficient page allocation helps performance.
**Example**
SHOW VARIABLES LIKE 'innodb_page_size';

### 43. What are the nine different types of Indexes?
Indexing improves lookup performance by creating searchable structures.

| Index Type | Description / Theory | Best Use Case |
|---|---|---|
| **Clustered Index (via Primary Key)** | Table data is physically stored in primary key order; leaf nodes contain actual row data; one per table. | Fast PK lookups, range scans, ordered queries. |
| **Secondary / Non-Clustered Index** | Separate index containing key + pointer to clustered index; requires extra lookup for full row. | Searching/filtering/joining on non-PK columns. |
| **Covering Index** | Index contains all columns required by a query so engine can answer without accessing table storage. | Performance-critical SELECT on limited columns. |
| **Full Text Index** | Designed for natural language keyword searching using MATCH AGAINST; optimized for long text fields. | Searching text, articles, comments, logs. |
| **Spatial Index** | Index on GIS geometry types using R-Tree structure; supports spatial calculations and constraints. | Maps, coordinate lookups, location-based filtering. |
| **Unique Index** | Guarantees no duplicate values in indexed columns; improves performance while enforcing data integrity. | Emails, usernames, natural keys. |
| **Partial / Filtered Index (via Generated Column)** | Emulates filtered index by indexing only rows mapped through a generated column; reduces index size. | Filtering over a subset (active rows, non-null values). |

| | | |
|---|---|---|
| **Functional Index (Expression-Based)** | Index built on expressions (functions) instead of raw columns; makes expression search SARGable. | Case-insensitive search, computed filters in WHERE. |
| **Computed/Persisted Value Index** | Index on STORED generated columns holding pre-computed data; avoids recalculation at query time. | Analytics fields, derived calculations used frequently. |

🧠 **When to Use Which Index**

| Scenario | Recommended Index Type |
|---|---|
| Fast access via primary key | Primary / Clustered Index |
| Prevent duplicate values | Unique Index |
| Searching/filtering on non-key columns | Non-Clustered (Secondary) Index |
| Frequent filters on multiple columns together | Composite Index |
| Text search (e.g., blog posts, product info) | Fulltext Index |
| Geolocation or map-based filtering | Spatial Index |
| Read-only queries using same column sets | Covering Index |
| Range queries and sorting (e.g., price between X and Y) | BTREE Index |

### 44. What happens when a Clustered Index is created?

Steps:

1. InnoDB builds a B-Tree
2. Data is physically reorganized by PK order
3. Rows are placed into leaf nodes of the B-tree
4. Heap structure disappears

This is why altering PK on large tables is expensive.

**Example**

ALTER TABLE users ADD PRIMARY KEY(id);

### 45. Types of Searching

1. Table Scan – full table read
2. Index Scan – scan full index tree
3. Index Seek – jump directly to matched key

MySQL resolves table-seek through index seek + row lookup.

**Example**

EXPLAIN SELECT * FROM users WHERE id = 25;

### 46. What is Fragmentation?

Inefficient use of storage caused by DML operations.

- Inserts cause page splits
- Deletes leave empty holes
- Page order stops matching logical order
- Bad fragmentation → slower I/O

### 47. Types of Fragmentation

**Internal:-** Free space inside pages not fully used

**External:-** Page order no longer matches logical B-tree order

Can cause increased disk I/O

### 48. What are Statistics?

Metadata describing how data is distributed.

**Helps optimizer decide**

- Which index to use
- How many rows will match
- Cost of joins, scans, and sorting

MySQL updates stats automatically but can be manually refreshed.
**Example**
ANALYZE TABLE users;

### 49. SQL Optimization Techniques

Core principles:

- Make indexes that support where + join predicates
- Avoid SELECT *
- Use EXPLAIN to understand the plan
- Avoid functions on indexed columns
- Prefer JOIN over subqueries when possible
- Partition large tables
- Keep statistics fresh
- Reduce scanned data using LIMIT
- Avoid cursors & unnecessary triggers
- Avoid non-SARGable predicates (WHERE YEAR(date)=2023)

**Example**
EXPLAIN SELECT id FROM users WHERE email='x@y.com';

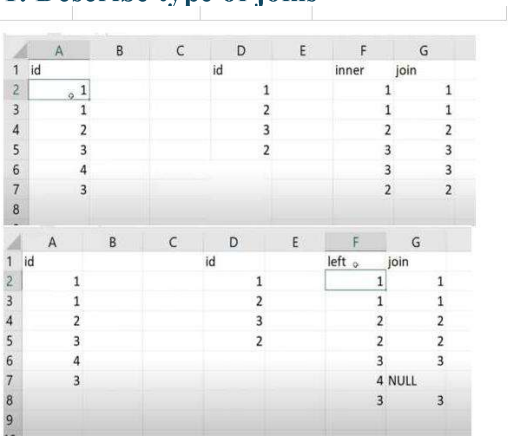
### 50. Explain Execution Plan

Execution Plan explains **how** MySQL will run a query.
Optimizer decides:

- Index vs full scan
- Join algorithms
- Sorting, temporary tables, filesort usage
- Estimated cost

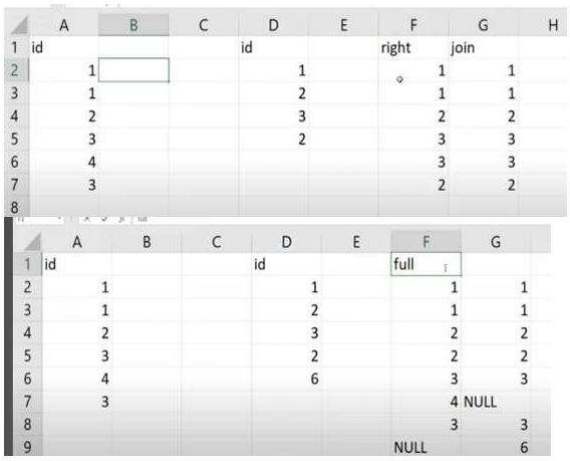
Plan quality determines real-world performance.
**Example**
EXPLAIN FORMAT=JSON
SELECT * FROM users WHERE email='abc@xyz.com';

### 51. Describe type of joins



| id | id | inner join |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 2 | 1 |
| 2 | 3 | 2 |
| 3 | 2 | 3 |
| 4 | | 3 |
| 3 | | 2 |

| id | id | left join |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 2 | 1 |
| 2 | 3 | 2 |
| 3 | 2 | 2 |
| 4 | | 3 |
| 3 | | 4 NULL |
| | | 3 |



| | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| 1 | id | | | id | | right | join | |
| 2 | 1 | | | 1 | | 1 | 1 | |
| 3 | 1 | | | 2 | | 1 | 1 | |
| 4 | 2 | | | 3 | | 2 | 2 | |
| 5 | 3 | | | 2 | | 3 | 3 | |
| 6 | 4 | | | | | 3 | 3 | |
| 7 | 3 | | | | | 2 | 2 | |
| 8 | | | | | | | | |

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 1 | id | | | id | | full | |
| 2 | 1 | | | 1 | | 1 | 1 |
| 3 | 1 | | | 2 | | 1 | 1 |
| 4 | 2 | | | 3 | | 2 | 2 |
| 5 | 3 | | | 2 | | 2 | 2 |
| 6 | 4 | | | 6 | | 3 | 3 |
| 7 | 3 | | | | | 4 | NULL |
| 8 | | | | | | 3 | 3 |
| 9 | | | | | | NULL | 6 |

## 52. Date time functions in MySQL

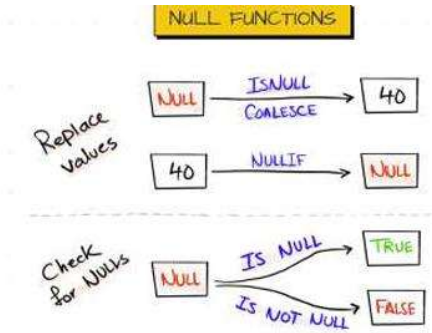
| Function | Description | Example Usage | Output |
|---|---|---|---|
| DAY() | Day of month (1–31) | SELECT DAY('2025-06-12 14:23:10'); | 12 |
| MONTH() | Month number (1–12) | SELECT MONTH('2025-06-12 14:23:10'); | 6 |
| YEAR() | 4-digit year | SELECT YEAR('2025-06-12 14:23:10'); | 2025 |
| DAYNAME() | Name of weekday | SELECT DAYNAME('2025-06-12 14:23:10'); | Thursday |
| MONTHNAME() | Full month name | SELECT MONTHNAME('2025-06-12 14:23:10'); | June |
| QUARTER() | Quarter of year | SELECT QUARTER('2025-06-12 14:23:10'); | 2 |
| WEEK() | Week of year | SELECT WEEK('2025-06-12 14:23:10'); | 24 |
| WEEKDAY() | Week index (Mon=0) | SELECT WEEKDAY('2025-06-12 14:23:10'); | 3 |
| HOUR() | Hour (0–23) | SELECT HOUR('2025-06-12 14:23:10'); | 14 |
| MINUTE() | Minute (0–59) | SELECT MINUTE('2025-06-12 14:23:10'); | 23 |
| SECOND() | Second (0–59) | SELECT SECOND('2025-06-12 14:23:10'); | 10 |
| DATE() | Extract date | SELECT DATE('2025-06-12 14:23:10'); | 2025-06-12 |
| TIME() | Extract time | SELECT TIME('2025-06-12 14:23:10'); | 14:23:10 |
| NOW() | Current timestamp | SELECT NOW(); | 2025-06-12 14:23:10 |
| CURDATE() | Current date | SELECT CURDATE(); | 2025-06-12 |

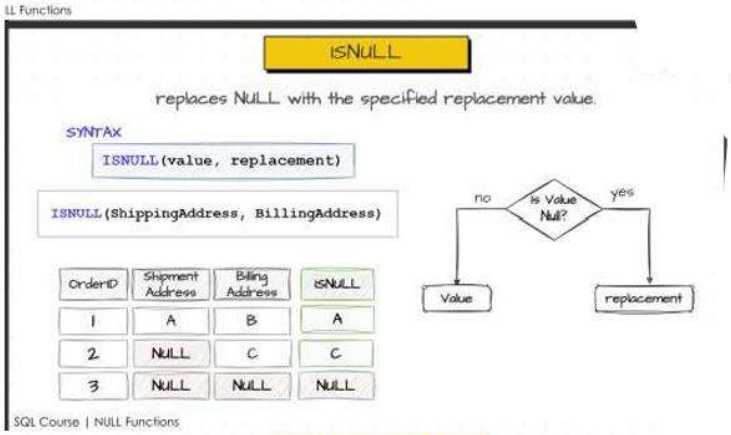
| | | | |
|---|---|---|---|
| CURTIME() | Current time | SELECT CURTIME(); | 14:23:10 |
| SYSDATE() | Time at execution start | SELECT SYSDATE(); | 2025-06-12 14:23:10 |
| UTC_DATE() | Current UTC date | SELECT UTC_DATE(); | 2025-06-12 |
| UTC_TIME() | Current UTC time | SELECT UTC_TIME(); | 08:53:10 |
| UTC_TIMESTAMP() | Current UTC datetime | SELECT UTC_TIMESTAMP(); | 2025-06-12 08:53:10 |
| DATEDIFF() | Difference in days | SELECT DATEDIFF('2025-06-15','2025-06-12'); | 3 |
| TIMEDIFF() | Time difference | SELECT TIMEDIFF('15:00:00','14:23:10'); | 00:36:50 |
| TIMESTAMPDIFF(SECOND) | Diff in seconds | SELECT TIMESTAMPDIFF(SECOND,'2025-06-12 14:00:00','2025-06-12 14:23:10'); | 1390 |
| TIMESTAMPDIFF(MINUTE) | Diff in minutes | SELECT TIMESTAMPDIFF(MINUTE,'2025-06-12 14:00:00','2025-06-12 14:23:10'); | 23 |
| TIMESTAMPDIFF(HOUR) | Diff in hours | SELECT TIMESTAMPDIFF(HOUR,'2025-06-12 10:00:00','2025-06-12 14:23:10'); | 4 |
| TIMESTAMPDIFF(DAY) | Diff in days | SELECT TIMESTAMPDIFF(DAY,'2025-06-10','2025-06-12'); | 2 |
| TIMESTAMPDIFF(WEEK) | Diff in weeks | SELECT TIMESTAMPDIFF(WEEK,'2025-06-01','2025-06-12'); | 1 |
| TIMESTAMPDIFF(MONTH) | Diff in months | SELECT TIMESTAMPDIFF(MONTH,'2025-01-01','2025-06-12'); | 5 |
| TIMESTAMPDIFF(QUARTER) | Diff in quarters | SELECT TIMESTAMPDIFF(QUARTER,'2024-01-01','2025-06-12'); | 6 |
| TIMESTAMPDIFF(YEAR) | Diff in years | SELECT TIMESTAMPDIFF(YEAR,'2020-06-01','2025-06-12'); | 5 |
| ADDDATE() | Add days to date | SELECT ADDDATE('2025-06-12 14:23:10',INTERVAL 5 DAY); | 2025-06-17 14:23:10 |
| SUBDATE() | Subtract days | SELECT SUBDATE('2025-06-12 14:23:10',INTERVAL 3 DAY); | 2025-06-09 14:23:10 |
| DATE_ADD() | Add interval | SELECT DATE_ADD('2025-06-12 14:23:10',INTERVAL 2 HOUR); | 2025-06-12 16:23:10 |

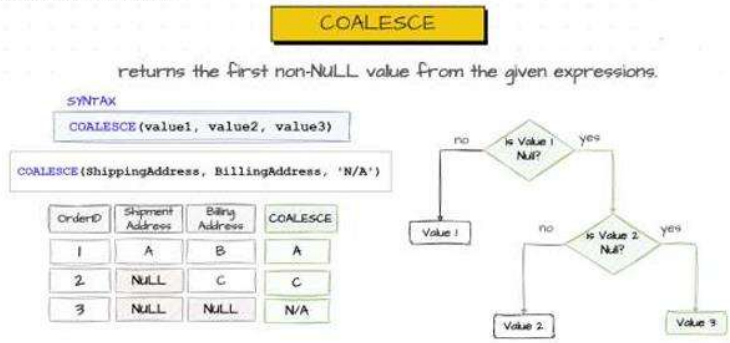
| | | | |
|---|---|---|---|
| DATE_SUB() | Subtract interval | SELECT DATE_SUB('2025-06-12 14:23:10',INTERVAL 30 MINUTE); | 2025-06-12 13:53:10 |
| LAST_DAY() | Last day of month | SELECT LAST_DAY('2025-06-12'); | 2025-06-30 |
| EXTRACT() | Extract part of date | SELECT EXTRACT(YEAR FROM '2025-06-12'); | 2025 |
| DATE_FORMAT() | Format date using placeholders | SELECT DATE_FORMAT('2025-06-12','%W, %M %d %Y'); | Thursday, June 12 2025 |
| STR_TO_DATE() | Convert string to date | SELECT STR_TO_DATE('12/06/2025','%d/%m/%Y'); | 2025-06-12 |
| TO_SECONDS() | Datetime to seconds since year 0 | SELECT TO_SECONDS('2025-06-12 14:23:10'); | 638316743 0 |
| SEC_TO_TIME() | Seconds → time | SELECT SEC_TO_TIME(3661); | 01:01:01 |
| TIME_TO_SEC() | Time → total seconds | SELECT TIME_TO_SEC('01:01:01'); | 3661 |
| MAKEDATE() | Build date from year + day | SELECT MAKEDATE(2025,163); | 2025-06-11 |
| MAKETIME() | Build time from H-M-S | SELECT MAKETIME(12,30,45); | 12:30:45 |
| PERIOD_ADD() | Add months to YYYYMM | SELECT PERIOD_ADD(202406,2); | 202408 |
| PERIOD_DIFF() | Months difference between periods | SELECT PERIOD_DIFF(202406,202401); | 5 |

### 53. How do you handle Nulls in MySQL?

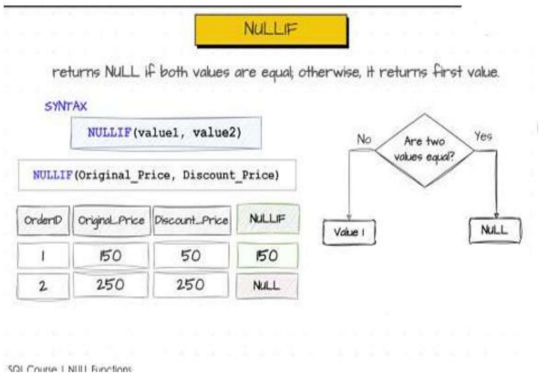
VANSHIKA MISHRA SQL INTERVIEW QUESTIONS -THEORY

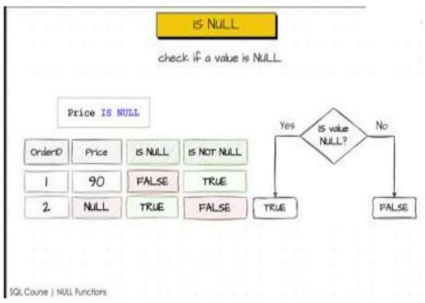


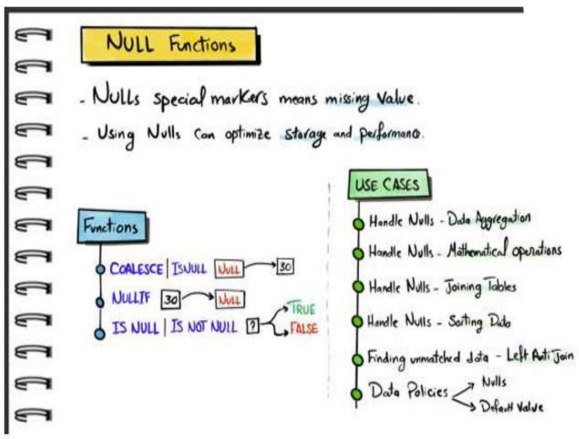



VANSHIKA MISHRA SQL INTERVIEW QUESTIONS -THEORY



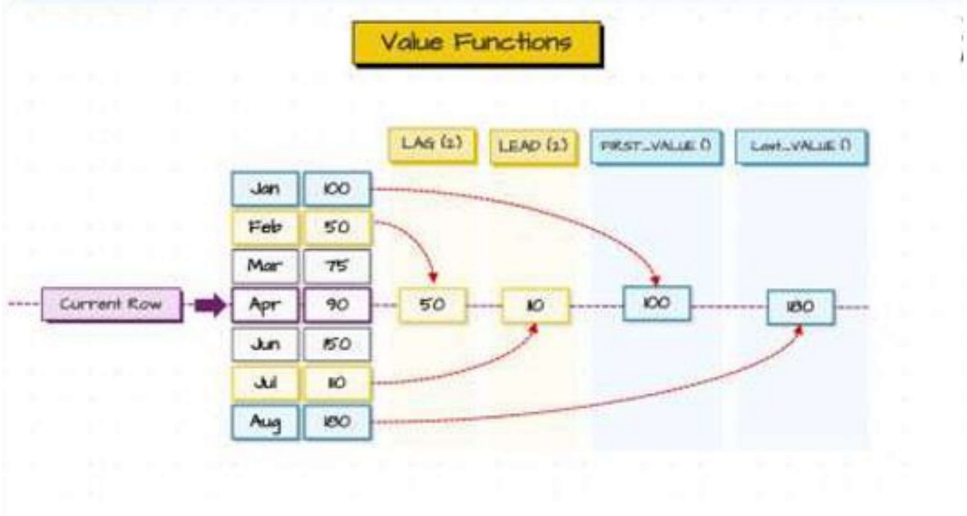



VANSHIKA MISHRA SQL INTERVIEW QUESTIONS -THEORY

## 54. What happens if we don't deal with Nulls?

| Context | Use Case Name | Zomato Real-life Use Case | 3-Row Data Example | How to Handle NULLs | SQL Example |
|---|---|---|---|---|---|
| Data Aggregation | Skewed Averages | A restaurant with many `NULL` ratings may appear to have a better average. | `ratings = [5, 4, NULL]` → `AVG = 4.5`, but really might be less. | Use `COUNT(rating)` alongside `AVG()`; fill with 0 if needed | `SELECT AVG(rating), COUNT(rating) FROM Reviews;` |
| Mathematical Ops | NULL Propagation | `NULL` discount makes `price * discount = NULL` | `price = [100, 150, 200], discount = [0.1, NULL, 0.2]` → result `[10, NULL, 40]` | Use `COALESCE(discount, 0)` to avoid null multiplication | `SELECT price * COALESCE(discount, 0) AS discounted_price FROM Orders;` |
| Joining Tables | Join Incompleteness | `orders.restaurant_id` is NULL → disappears in `INNER JOIN` | `Orders = [(1, 1001), (2, NULL), (3, 1002)]` + `Restaurants = [1001, 1002]` → only 2 join | Use `LEFT JOIN` to preserve unmatched rows | `SELECT * FROM Orders LEFT JOIN Restaurants ON Orders.restaurant_id = Restaurants.id;` |
| Sorting | Custom NULL Order | Sort restaurants by `last_order_time`, put those with no orders at bottom | `last_order_time = ['2024-01-01', NULL, '2024-04-05']` | Use `ORDER BY last_order_time NULLS LAST` | `SELECT * FROM Restaurants ORDER BY last_order_time NULLS LAST;` |
| Finding Unmatched | Null-Aware Filtering | Find users with no saved address or unassigned delivery agents | `Users = [(1, 'delhi'), (2, NULL), (3, 'pune')]` | Use `WHERE address IS NULL` | `SELECT * FROM Users WHERE address IS NULL;` |
| Data Policies | Enforce Completeness | `user_id`, `order_total`, etc. must not be NULL | `Orders = [(1, 500), (2, NULL), (NULL, 300)]` | Add `NOT NULL` constraint or validate in app layer | `ALTER TABLE Orders MODIFY order_total DECIMAL(10,2) NOT NULL;` |
| Reporting | Show Meaningful Defaults | Missing `delivery_time` may confuse reporting dashboards | `delivery_time = ['10:00', NULL, '12:00']` → show `'Pending'` instead | Use `COALESCE()` | `SELECT COALESCE(delivery_time, 'Pending') AS delivery_time FROM Orders;` |
| Filtering | Include/Exclude NULLs | Exclude unrated restaurants in some views, include in others | `rating = [5, NULL, 3]` | Use `IS NULL`, `IS NOT NULL`, or `CASE` | `SELECT * FROM Restaurants WHERE rating IS NOT NULL;` |
| Auditing & Monitoring | NULL Completeness Check | Count `NULL` values in key fields (e.g., payment mode) | `payment_mode = ['Card', NULL, 'UPI']` | Use `CASE WHEN ... IS NULL` | `SELECT COUNT(*) AS total, COUNT(payment_mode) AS non_nulls, SUM(CASE WHEN payment_mode IS NULL THEN 1 ELSE 0 END) AS nulls FROM Orders;` |

## 55. What are lead, lag window functions?



## 56. What are logical operators in MySQL?


## 57. String functions in MYSQL

| Function | Description | SQL Example (MySQL) |
|---|---|---|
| CHAR_LENGTH(str) | Returns number of characters in a string | SELECT CHAR_LENGTH('Hello'); |
| LENGTH(str) | Returns number of bytes in a string | SELECT LENGTH('Hello'); |
| LOWER(str) | Converts string to lowercase | SELECT LOWER('HELLO'); |
| UPPER(str) | Converts string to uppercase | SELECT UPPER('hello'); |
| LTRIM(str) | Removes leading spaces | SELECT LTRIM(' Hello'); |
| RTRIM(str) | Removes trailing spaces | SELECT RTRIM('Hello '); |
| TRIM(str) | Removes spaces from both ends | SELECT TRIM(' Hello '); |
| TRIM(BOTH 'x' FROM str) | Removes specific character from both ends | SELECT TRIM(BOTH 'x' FROM 'xxxHelloxxx'); |
| SUBSTRING(str, pos, len) | Extract substring from starting position for length | SELECT SUBSTRING('Hello World', 7, 5); |
| LEFT(str, len) | Returns leftmost characters | SELECT LEFT('Hello', 2); |
| RIGHT(str, len) | Returns rightmost characters | SELECT RIGHT('Hello', 2); |
| INSTR(str, substr) | Returns position of substring; 0 if not found | SELECT INSTR('Hello World', 'World'); |
| LOCATE(substr, str) | Same as INSTR | SELECT LOCATE('World', 'Hello World'); |
| REPLACE(str, from, to) | Replace all occurrences in a string | SELECT REPLACE('Hello World', 'World', 'MySQL'); |
| REVERSE(str) | Reverses the characters | SELECT REVERSE('Hello'); |
| CONCAT(str1, str2, ...) | Concatenates strings | SELECT CONCAT('Hello', ' ', 'World'); |
| CONCAT_WS(sep, str1, ...) | Concatenates using a separator | SELECT CONCAT_WS('-', '2024', '09', '10'); |
| REPEAT(str, count) | Repeats a string n times | SELECT REPEAT('Hi', 3); |
| SPACE(n) | Returns a string of n spaces | SELECT CONCAT('Hello', SPACE(3), 'World'); |

| | | |
|---|---|---|
| ELT(index, str1, ...) | Returns argument at given index | SELECT ELT(2, 'A', 'B', 'C'); |
| FIELD(str, str1, ...) | Returns index position of str in list | SELECT FIELD('B', 'A', 'B', 'C'); |
| FIND_IN_SET(str, set) | Position of str in comma-separated values | SELECT FIND_IN_SET('b', 'a,b,c'); |
| FORMAT(num, decimal) | Formats number with commas and decimals | SELECT FORMAT(1234567.89, 2); |
| ASCII(str) | ASCII code of first character | SELECT ASCII('A'); |
| HEX(str) | Converts input to hex representation | SELECT HEX('abc'); |
| UNHEX(str) | Converts hex to original string | SELECT UNHEX('616263'); |

## 58. What are DDL, DML, TCL, DQL, DCL Commands?

| Category | Command | Function | Example |
|---|---|---|---|
| **Data Definition Language (DDL)** | CREATE | Creates database objects (tables, views, etc.) | CREATE TABLE employees (id INT, name VARCHAR(100), ...); |
| | DROP | Deletes database objects | DROP TABLE employees; |
| | ALTER | Modifies existing database structures | ALTER TABLE employees ADD COLUMN department VARCHAR(50); |
| | TRUNCATE | Deletes all rows but retains table structure | TRUNCATE TABLE employees; |
| **Data Manipulation Language (DML)** | INSERT | Adds new data into a table | INSERT INTO employees VALUES (1, 'John', 'Manager', 60000.00); |
| | UPDATE | Updates existing data | UPDATE employees SET salary = 65000 WHERE id = 1; |
| | DELETE | Deletes rows from a table | DELETE FROM employees WHERE id = 1; |
| **Data Query Language (DQL)** | SELECT | Retrieves data from the database | SELECT id, name FROM employees; |
| **Data Control Language (DCL)** | GRANT | Grants permissions to users | GRANT SELECT ON employees TO user_name; |
| | REVOKE | Revokes permissions from users | REVOKE SELECT ON employees FROM user_name; |
| **Transaction Control Language (TCL)** | COMMIT | Saves all changes in the current transaction | COMMIT; |
| | ROLLBACK | Reverts changes in the current transaction | ROLLBACK; |
| | SAVEPOINT | Sets a restore point within a transaction | SAVEPOINT savepoint_name; |

## 59. Differentiate between primary key vs foreign key

| PRIMARY KEY | FOREIGN KEY |
|---|---|
| It is used to identify a records Uniquely from the table. | It is used to establish a connection Between the tables |
| It cannot accept Null | It can accept Null |
| It cannot accept duplicate values | It can accept duplicate values |
| It is always a combination of Not Null and Unique constraint | It is not a combination of Not Null and Unique constraint |
| We can have only 1 PK in a table | We can have Multiple FK in a table |

### 60. Describe the use of the MERGE statement in SQL.

The MERGE statement, also known as "upsert," allows conditional update or insertion of data into a target table based on the presence of matching records in a source table. Syntax:

```
MERGE INTO target_table AS target
USING source_table AS source
ON target.id = source.id
WHEN MATCHED THEN UPDATE SET target.column = source.column
WHEN NOT MATCHED THEN INSERT (id, column) VALUES (source.id, source.column);
```

This statement simplifies complex operations by combining them into a single atomic action.

### 61. How do you implement slowly changing dimensions (SCD) in SQL?

Slowly Changing Dimensions track historical changes in dimension data in data warehouses.
Type 2 (most common): Maintain full history by adding a new row for each change. Example: Columns: customer_id, name, valid_from, valid_to, is_current Use triggers

### 62. What are materialized views and how are they different from regular views?

Materialized View: Stores the result of a query physically and can be refreshed periodically. Regular View: Stores the SQL definition only and runs fresh each time.
Use Cases: Use materialized views in reporting systems or when query performance is critical.

### 63. How do you handle schema changes in a production environment?

Use rolling deployments with backward-compatible schema changes.
Add new columns as nullable before populating them.
Create views or temporary tables to support both old and new code paths.
Use blue-green deployments to switch traffic gradually.

### 64. What is a pivot and unpivot in SQL?

PIVOT: Converts row data into column format.
UNPIVOT: Converts columns back into rows.
Example (Pivot):

```
SELECT * FROM (
SELECT department, salary
FROM Employees ) AS src
PIVOT ( AVG(salary) FOR department IN ([Sales], [IT], [HR]) )
AS pvt;
```

### 65. How do you implement auditing in SQL?

Use triggers to log changes on sensitive tables.

```
CREATE TRIGGER AuditEmployeeChanges
```

AFTER INSERT, UPDATE, DELETE ON Employees
FOR EACH ROW BEGIN INSERT INTO AuditLog (change_type, changed_by, change_time) VALUES ('UPDATE', CURRENT_USER, NOW());
END;

### 66. What is the purpose of GROUPING SETS in SQL?

GROUPING SETS allows you to define multiple groupings in a single query.
SELECT department, product,
SUM(sales) FROM Sales
GROUP BY GROUPING SETS (
(department),
(product),
(department, product) );
It's like running multiple GROUP BY queries and combining the results.

### 67. How do you implement row-level security in SQL?

Row-level security restricts access to specific rows in a table based on user identity or role.

[Code Example]

```
-- Create a new user with a strong password and grant limited privileges
CREATE USER 'new_user'@'localhost' IDENTIFIED BY
'StrongP@ssw0rd!';
GRANT SELECT, INSERT ON my_database.* TO
'new_user'@'localhost';
```

[Execution Result]

```
This command creates a new user named new_user with the specified
password and grants them the ability to perform SELECT and INSERT
operations on the my_database.
```

### 68. How do you bulk-load millions of records efficiently?

Disable non-essential indexes and constraints temporarily.
Use native bulk commands ( LOAD DATA in MySQL).
Batch in chunks (e.g., 100k rows) to avoid log saturation.
Commit after each batch; monitor log growth.
For cloud warehouses (Snowflake, BigQuery), stage files in object storage (S3, GCS) and use parallel load.

### 69. Describe how JSON data can be stored and queried in modern RDBMSs.

Data Type: JSON (MySQL/MariaDB)
Functions: to extract; JSON_VALUE, JSON_QUERY.
Indexing: virtual computed columns (MySQL) for key paths.
Use JSON for semi-structured attributes (e.g., marketplace product specs) while keeping core dimensions in relational columns.

### 70. How does sharding differ from simple horizontal partitioning?

Partitioning: Transparent to application; database engine manages partitions within one logical database instance.
Sharding: Each shard is an independent database (maybe on separate servers). Application code or middleware routes queries to the correct shard. Sharding is used when a single server cannot handle dataset size or traffic, such as billions of ride records for an Indian mobility app.

**71. Describe the LISTAGG / STRING_AGG function and a common use case.**
STRING_AGG(col, ',') WITHIN GROUP (ORDER BY col) concatenates values into a comma-separated list.
Use case: Show all product categories a seller operates in as a single field in a report.

**72. How does ON DELETE CASCADE differ from ON DELETE SET NULL in foreign keys?**
CASCADE: Deleting parent row automatically deletes child rows.
SET NULL: Child foreign key value becomes NULL, preserving the child row.
Choose based on whether orphaned children make business sense (e.g., invoice_lines after invoice delete).

**73. Can you perform a JOIN operation without specifying the JOIN condition?**
- No, a join condition is required
- Without it, DB generates a Cartesian product (every row with every row)
- Always specify ON or USING for meaningful joins

**74. How does a LEFT JOIN differ from a RIGHT JOIN?**
- LEFT JOIN → all rows from left table + matching rows from right
- RIGHT JOIN → all rows from right table + matching rows from left
- Nulls fill missing sides in unmatched rows
-

**75. What is the result of joining a table with itself using a CROSS JOIN?**
- Produces Cartesian product
- Combines every row with every other row (including itself)
- Output size = n × n rows → becomes very large quickly

**76. Can you JOIN more than two tables in a single SQL query?**
- Yes, SQL allows joining multiple tables
- Common when retrieving related information from several entities
- Syntax: JOIN table2 ... JOIN table3 ...

**77. How can you simulate a FULL OUTER JOIN if it's not supported?**
- Combine LEFT JOIN and RIGHT JOIN using UNION
- Removes duplicates using UNION (not UNION ALL)
- Covers rows appearing in only one table

**78. What is the difference between a natural join and an equijoin?**
- Natural join: matches columns with same name automatically
- Equijoin: explicitly defines join condition using =
- Natural join risks incorrect matches if naming overlaps unintentionally

**79. Can you use the WHERE clause to perform a JOIN operation?**
- Yes historically, using FROM A,B WHERE A.id=B.id

- Best practice: specify ON in JOIN clause for clarity and safety
- WHERE is mainly for filtering after join

**80. How can you exclude rows that match in a JOIN operation?**

- Use LEFT or RIGHT JOIN and filter NULLs
- Return rows that exist in only one table
- Example: LEFT JOIN ... WHERE right.col IS NULL

**81. Can you join tables with different data types?**

- Yes, if values are logically comparable
- DB automatically performs type conversion when possible
- Must avoid incompatible types (e.g., int vs text without cast)

**82. How can you optimize JOIN performance?**

- Create indexes on join key columns
- Use correct join type (INNER JOIN faster than OUTER)
- Filter early using WHERE to reduce result size
- Avoid SELECT * if unnecessary

**83. Is it possible to JOIN tables based on non-matching columns?**

- Yes, using non-equality conditions
- Examples: > , <, BETWEEN, or inequality
- Called non-equijoins

**84. What are the implications of using a CROSS JOIN?**

- Produces huge result sets (Cartesian product)
- Can degrade performance significantly
- Often used intentionally in analytics or matrix generation

**85. Can you JOIN tables that have different column names?**

- Yes
- Must explicitly specify join keys using ON
- Aliases help readability

**86. Difference between WHERE and HAVING clauses**

- WHERE filters rows *before* grouping
- HAVING filters groups *after* aggregation
- HAVING supports aggregate functions; WHERE does not

**87. Can you use HAVING without GROUP BY?**

- No—HAVING logically relies on grouped results
- Some DBs accept HAVING with no GROUP BY if entire result is treated as one group

**88. What happens if you select a column not in GROUP BY?**

- SQL error in strict engines
- Some DBs return an arbitrary value (not reliable)
- Always group or aggregate non-grouped columns

**89. Can you have multiple HAVING clauses in one query?**

- Only one HAVING clause allowed
- Can contain multiple logical conditions using AND/OR
- Equivalent to multiple HAVING statements combined

### 90. How is ORDER BY different from HAVING?

- ORDER BY sorts final output
- HAVING filters grouped results prior to ordering
- ORDER BY executes after HAVING

### 91. Can you use aggregate functions in HAVING?

- Yes—primary purpose of HAVING
- Used to filter based on SUM, COUNT, MAX, AVG, etc.

### 92. Can you use GROUP BY without HAVING?

- Yes—HAVING is optional
- GROUP BY simply organizes data into groups
- HAVING only needed for group filtering

### 93. Execution order of GROUP BY and HAVING

- GROUP BY executes first
- Aggregates are computed
- HAVING filters the aggregated groups
- Then ORDER BY if present

### 94. What happens if GROUP BY and HAVING are swapped?

- Query fails; invalid syntax
- GROUP BY must come before HAVING
- SQL grammar enforces correct order

### 95. Can you use non-aggregated columns in HAVING?

- No
- HAVING conditions must reference aggregated results or grouped columns
- Using raw columns causes errors

### 96. What is table partitioning in databases, and how does it improve performance? Give an example.

- **Definition:**
  - Partitioning divides a large table into smaller logical segments while still behaving like one table.
- **Why it improves performance:**
  - Queries scan only relevant partitions instead of the full table.
  - Speeds up reads, writes, and maintenance tasks on large datasets.
  - Helps manage data growth more efficiently.
- **Types of partitioning:**
  - Common methods include range, list, hash, and key partitioning.
  - Range partitioning works well for date-based data.
- **Example: Range partition by year in MySQL:**

```
CREATE TABLE sales (
    id INT,
    amount DECIMAL(10, 2),
    sale_date DATE
)
PARTITION BY RANGE (YEAR(sale_date)) (
    PARTITION p2022 VALUES LESS THAN (2023),
    PARTITION p2023 VALUES LESS THAN (2024),
    PARTITION p2024 VALUES LESS THAN (2025)
);
```

- **Inserting data into partitions:**

```
INSERT INTO sales (id, amount, sale_date) VALUES (1, 100.00, '2022-05-01');
INSERT INTO sales (id, amount, sale_date) VALUES (2, 150.00, '2023-06-15');
```

- **Key takeaway:**
  - Partitioning helps optimize performance and manage very large tables by splitting data into smaller chunks based on logical rules, such as date ranges.

### 97. How do you back up and restore a database?

***Backup Example (MySQL using mysqldump):***

mysqldump -u username -p database_name > backup.sql

- Exports all data + schema into a file
- Replace username and database_name accordingly

***Restore Example:***

mysql -u username -p database_name < backup.sql

- Recreates tables and inserts all data

📌 Most DBMS (MySQL/PostgreSQL/SQL Server) offer:

- Command-line utilities
- Scheduled automated backups
- Full, incremental, and differential backups

### 98. What is point-in-time recovery (PITR)?

- A technique that restores the database **to a specific moment**, not just to the last full backup
- Useful if:
  - Someone deletes crucial records at 10:05 AM
  - You restore to 10:04 AM and avoid the mistake
- Requires:
  - Full backup + transaction logs

### 99. Why use materialized views in OLAP?

• Materialized views physically store the query result instead of recalculating it every time
• Ideal for OLAP workloads involving heavy aggregation and joins

• Greatly improves performance for complex analytical queries
• Reduces load on base tables by avoiding repeated expensive computation
• Frequently used in reporting systems, dashboards, and data warehousing
• Can be refreshed on schedule (daily/hourly) based on business need

### 100. Why can too many indexes hurt performance?
• Indexes speed up reads but slow down writes (INSERT, UPDATE, DELETE)
• Every time data changes, all related indexes must be updated
• Too many indexes increase storage and memory usage
• Query optimizer has more index choices which may lead to inefficient plans
• Unused or redundant indexes waste system resources
• Best practice is to index only columns used frequently in filtering, joining, or sorting

### 101. Can you update data through a view? When allowed/not allowed?
Allowed when:
• The view is based on a single table
• It does not perform aggregation functions (SUM, AVG, COUNT, MIN/MAX)
• It does not include DISTINCT, GROUP BY, UNION, JOIN, or TOP
• All required columns for the base table are available
• WITH CHECK OPTION ensures only valid data is inserted or updated
Not allowed when:
• View uses multiple tables (joins)
• It contains computed fields or expressions
• It includes GROUP BY, aggregation, UNION, DISTINCT
• Underlying columns are read-only or restricted
• Indexed/materialized views may impose additional rules depending on DBMS

### 102. How to choose the right index?
• Identify columns frequently used in WHERE filters
• Prioritize columns used in JOIN and ORDER BY clauses
• Use indexes on high-selectivity columns (values vary a lot)
• Avoid indexing low-selectivity columns (example: boolean, gender)
• Consider composite indexes when multiple columns are filtered together
• Put the most selective column first in a composite index
• Regularly monitor usage using index statistics and execution plans
• Remove unused, overlapping, or redundant indexes to keep writes fast