sql-based on jd

⭐ SQL — HIGH-PROBABILITY INTERVIEW QUESTIONS (1–2 YOE)

1. SQL Basics

✅ 1. What is SQL? Difference between SQL and MySQL

■ Explanation (simple, conceptual)

SQL (Structured Query Language) is a standard language used to store, retrieve, and manipulate data in relational databases.

It is *just a language*, not software.

MySQL is a **Relational Database Management System (RDBMS)** that uses SQL to manage data.

■ When/Why is it used

- SQL → used when you want to query, insert, update, or delete data.
- MySQL → used when you want a database server to store data for applications.

■ Example

```
SELECT * FROM employees;
```

This SQL query can run on MySQL, PostgreSQL, Oracle, etc.

■ Short summary (interview line)

"SQL is a language for querying relational databases, whereas MySQL is a database system that uses SQL."

✅ 2. Difference between DDL, DML, DQL, DCL, TCL

■ Explanation (conceptual)

SQL statements are divided into logical categories based on their purpose:

Category	Full Form	Purpose
DDL	Data Definition Language	Defines database structure (tables, schema)
DML	Data Manipulation Language	Manipulates data (insert/update/delete)
DQL	Data Query Language	Fetches data from database

Category	Full Form	Purpose
DCL	Data Control Language	Controls access and permissions
TCL	Transaction Control Language	Manages transactions

■ When/Why is it used

- DDL → when creating/altering database objects
- DML → used in application CRUD operations
- DQL → used in reports, queries
- DCL → used by admins to control permissions
- TCL → used for maintaining data consistency in transactions

■ Examples

DDL

```
CREATE TABLE users(id INT, name VARCHAR(50));
```

DML

```
INSERT INTO users VALUES (1, 'Mohini');
```

DQL

```
SELECT name FROM users;
```

DCL

```
GRANT SELECT ON users TO app_user;
```

TCL

```
COMMIT;
ROLLBACK;
```

■ Short summary (interview line)

“DDL defines structure, DML modifies data, DQL reads data, DCL controls permissions, and TCL manages transactions.”

✅ 3. Difference between a Database, Schema, and Table

■ Explanation (simple, conceptual)

- **Database** → A collection of organized data. It contains schemas and tables.

- **Schema** → A logical container inside a database. Helps group related tables.
- **Table** → A structured set of rows and columns where actual data is stored.

■ When/Why is it used

- Database → entire application dataset
- Schema → separates modules logically (e.g., `banking`, `fraud`, `analytics`)
- Table → stores records (customers, transactions)

■ Example

```
Database: BankDB
   └── Schema: FraudService
       └── Table: suspicious_transactions
```

■ Short summary (interview line)

"A database contains schemas, schemas group related tables, and tables store the actual data."

✅ 4. What are constraints? Types of constraints?

■ Explanation (simple, conceptual)

Constraints are **rules applied to table columns** to ensure **data accuracy, validity, and consistency**.

■ When/Why is it used

- Prevents invalid data (duplicate IDs, null values, invalid relations)
- Ensures data integrity in enterprise systems like banking & fraud detection
- Enforces business rules at database level

■ Types of Constraints

1. **PRIMARY KEY** → Uniquely identifies each row
2. **FOREIGN KEY** → Maintains relationship between tables
3. **UNIQUE** → Ensures no duplicate values in a column
4. **NOT NULL** → Prevents null values
5. **CHECK** → Enforces a condition (e.g., age > 18)
6. **DEFAULT** → Sets a default value if none is provided

■ Example

```
CREATE TABLE accounts (
   account_id INT PRIMARY KEY,
   balance DECIMAL CHECK (balance >= 0),
```

```
    status VARCHAR(10) DEFAULT 'ACTIVE'
);
```

■ Short summary (interview line)

“Constraints are rules that enforce data integrity, like Primary Key, Foreign Key, Unique, Not Null, Check, and Default.”

---

2. Joins (MOST IMPORTANT)

✅ 1. What are Joins?

■ Explanation (simple, conceptual)

A **JOIN** is used in SQL to combine rows from two or more tables based on a related column between them.

It allows us to fetch meaningful combined information—for example, users **with** their expenses, orders **with** customers, etc.

■ When/Why is it used

- When data is split across multiple tables.
- To avoid data duplication (normalized database).
- To retrieve combined or related information.

■ Example

Joining **users** and **expenses** on user_id:

```
SELECT u.name, e.amount
FROM users u
JOIN expenses e ON u.id = e.user_id;
```

■ Short interview line

“Joins help combine data from multiple tables based on a common key.”

✅ 2. Explain INNER JOIN with an example

■ Explanation (simple, conceptual)

INNER JOIN returns only the matching records between two tables.

If there is no match, the row is excluded.

■ When/Why is it used

- When you need only the **common** or **valid** data between tables.

- Example: Fetch users who **have made** an expense.

■ Example

```
SELECT u.name, e.amount
FROM users u
INNER JOIN expenses e
ON u.id = e.user_id;
```

→ Returns only users who have at least one expense.

■ Short interview line

"INNER JOIN gives only matching rows from both tables."

✅ 3. LEFT JOIN vs RIGHT JOIN

■ Explanation (simple)

Join Type	What it Returns
LEFT JOIN	All rows from left table + matching rows from right table
RIGHT JOIN	All rows from right table + matching rows from left table

■ When/Why used

- **LEFT JOIN** → When you want all records from left table **even if no match exists**.

  Example: All users, even if no expense.
- **RIGHT JOIN** → Opposite scenario; rarely used because same result can be done using LEFT JOIN by switching tables.

■ Example

LEFT JOIN

```
SELECT u.name, e.amount
FROM users u
LEFT JOIN expenses e
ON u.id = e.user_id;
```

RIGHT JOIN

```
SELECT u.name, e.amount
FROM users u
RIGHT JOIN expenses e
ON u.id = e.user_id;
```

■ Short interview line

"LEFT JOIN keeps all left-table rows; RIGHT JOIN keeps all right-table rows."

✅ 4. FULL JOIN vs UNION

■ Explanation (simple)

- **FULL JOIN** → Returns all rows from both tables — matched + unmatched.
- **UNION** → Combines results of **two SELECT queries**, removing duplicates (UNION ALL keeps duplicates).

■ When/Why used

- FULL JOIN → When you need a complete view from both tables.
- UNION → When you want to append two independent result sets.

■ Example

FULL JOIN

```
SELECT *
FROM users u
FULL JOIN expenses e
ON u.id = e.user_id;
```

UNION

```
SELECT name FROM users
UNION
SELECT category FROM expenses;
```

These are unrelated queries—UNION doesn't match rows based on keys.

■ Short interview line

"FULL JOIN merges tables side-by-side; UNION stacks results top-to-bottom."

✅ 5. What is a CROSS JOIN?

■ Explanation (simple)

CROSS JOIN returns the **Cartesian product** of two tables — every row from table A combined with every row from table B.

If table A has 5 rows and table B has 4 rows → result = **20 rows**.

■ When/Why used

- When generating combinations (e.g., all product × color pairs).
- When no join condition exists.

■ Example

```
SELECT *
FROM products
CROSS JOIN colors;
```

■ Short interview line

"CROSS JOIN gives all possible row combinations from both tables."

✅ Required Practical Example (asked in interviews)

Question:

"Write an SQL query to fetch all users and their expenses, even if no expenses are recorded."

■ Explanation

We need **all users**, whether expenses exist or not → **LEFT JOIN**.

■ Query

```
SELECT u.id, u.name, e.amount
FROM users u
LEFT JOIN expenses e
ON u.id = e.user_id;
```

■ Short interview line

"To get all users even without expenses, we use LEFT JOIN."

---

3. Filtering & Conditions

✅ 1. Difference between WHERE and HAVING

■ Explanation (simple, conceptual)

- **WHERE** filters rows **before** grouping happens.
- **HAVING** filters groups **after** GROUP BY and aggregates are applied.

WHERE → works on **row-level data**

HAVING → works on **group-level/aggregate data**

■ When/Why is it used

- Use **WHERE** when filtering individual rows (e.g., amount > 100).
- Use **HAVING** when filtering results after aggregation (e.g., total_expense > 1000).

■ Example

```
SELECT user_id, SUM(amount)
FROM expenses
WHERE amount > 100       -- filters rows
GROUP BY user_id
HAVING SUM(amount) > 1000;   -- filters groups
```

■ Short interview line

“WHERE filters rows before grouping; HAVING filters aggregated groups after GROUP BY.”

✅ 2. GROUP BY rules — why must non-aggregated columns appear in GROUP BY?

■ Explanation (simple, conceptual)

SQL must know **how to group** rows.

If you select a column that is *not aggregated*, SQL needs instructions → GROUP BY.

Example mistake:

```
SELECT user_id, amount   -- amount is not aggregated
FROM expenses
GROUP BY user_id;
```

Here SQL doesn’t know **which amount** to show for each user (multiple rows).

■ When/Why

- GROUP BY ensures **deterministic** output.
- Prevents ambiguous results.
- Enforces proper aggregation rules.

■ Correct Example

```
SELECT user_id, SUM(amount)
FROM expenses
GROUP BY user_id;
```

■ Short interview line

“Non-aggregated columns must be in GROUP BY because SQL needs a clear rule on how to group them.”

✅ 3. BETWEEN vs IN Operators

■ Explanation (simple)

BETWEEN → checks if a value lies in a **continuous range** (inclusive).

IN → checks if a value matches **discrete, specific values**.

■ When/Why used

- Use **BETWEEN** for ranges like dates, numeric intervals.
- Use **IN** for fixed sets like categories, IDs.

■ Examples

BETWEEN

```
SELECT * FROM transactions
WHERE amount BETWEEN 100 AND 500;
```

IN

```
SELECT * FROM users
WHERE role IN ('admin', 'manager');
```

■ Short interview line

“BETWEEN is for ranges; IN is for specific listed values.”

✅ 4. LIKE vs ILIKE

■ Explanation (simple)

Both used for pattern matching.

Operator	Case Sensitivity
LIKE	Case-sensitive (MySQL depends on collation)
ILIKE	Case-insensitive (mainly PostgreSQL)

■ When/Why used

- LIKE → when case matters OR database collation is case-insensitive by default.
- ILIKE → when you need consistent **case-insensitive** matching (PostgreSQL).

■ Example

LIKE

```
SELECT * FROM users
WHERE name LIKE 'Moh%';     -- Matches Mohini, Mohamed
```

ILIKE

```
SELECT * FROM users
```

```
WHERE name ILIKE 'moh%';    -- Matches mohini, MOHINI, MohIni
```

■ Short interview line

"LIKE is case-sensitive; ILIKE is case-insensitive pattern matching (mainly in PostgreSQL)."

---

4. Aggregations

✅ 1. COUNT vs COUNT(*) vs COUNT(1)

■ Explanation (simple, conceptual)

All three are used to count rows, but they behave slightly differently.

Expression	What it Counts
COUNT(column)	Counts non-NULL values in that column
COUNT(*)	Counts all rows (including rows with NULLs)
COUNT(1)	Also counts all rows (1 is a constant), same result as COUNT(*)

■ When/Why is it used

- Use **COUNT(column)** when you want to ignore NULLs.
- Use **COUNT(*)** most commonly — fastest & optimized by database engine.
- COUNT(1) behaves the same as COUNT(*) in modern databases.

■ Example

```
SELECT COUNT(salary)     -- counts only rows where salary is not NULL
FROM employees;

SELECT COUNT(*)          -- counts all rows
FROM employees;

SELECT COUNT(1)          -- counts all rows
FROM employees;
```

■ Short interview line

"COUNT() *counts all rows, COUNT(column) ignores NULLs, and COUNT(1) is functionally same as COUNT()*."

✅ 2. Difference between SUM() and COUNT()

■ Explanation (simple)

- **SUM()** adds values of a numeric column.

- **COUNT()** counts rows.

■ When/Why is it used

- SUM → used in financial, analytical operations (total expenses, total salary).
- COUNT → used to find number of rows (number of users, number of transactions).

■ Example

```
SELECT SUM(amount), COUNT(amount)
FROM expenses;
```

■ Short interview line

"SUM totals numeric values, COUNT counts how many values/rows exist."

✅ 3. How does GROUP BY work internally?

■ Explanation (simple, conceptual)

GROUP BY groups rows with **same values** into logical buckets, and then aggregate functions are applied to each bucket.

Internal steps:

1. **Scan table rows**
2. **Group rows** based on GROUP BY column(s)
3. **Apply aggregations** like SUM, COUNT, AVG
4. **Generate one output row per group**

Example:

If user_id appears 10 times → GROUP BY collects those 10 rows → produces 1 aggregated row.

■ When/Why is it used

- To summarize or aggregate data by category.
- Used in reporting, dashboards, analytics, budget insights.

■ Example

```
SELECT user_id, SUM(amount)
FROM expenses
GROUP BY user_id;
```

■ Short interview line

"GROUP BY collects identical values into groups and applies aggregate functions on each group."

✅ 4. Can we use WHERE with aggregate functions? Why not?

■ Explanation (simple)

No — WHERE cannot use aggregates like SUM(), COUNT() because:

WHERE filters **rows before grouping**,

but aggregate functions are computed **after grouping**.

Hence, the correct place to filter aggregated results is **HAVING**.

■ When/Why

- Use WHERE → filter raw rows
- Use HAVING → filter aggregated groups

■ Incorrect

```
SELECT user_id, SUM(amount)
FROM expenses
WHERE SUM(amount) > 1000;   -- ❌ Not allowed
```

■ Correct

```
SELECT user_id, SUM(amount)
FROM expenses
GROUP BY user_id
HAVING SUM(amount) > 1000;  -- ✔ Allowed
```

■ Short interview line

"WHERE cannot use aggregates because it runs before grouping; HAVING filters after aggregates are computed."

---

5. Subqueries

✅ 1. What is a subquery?

■ Explanation (simple, conceptual)

A **subquery** is a query *inside another query*.

It is executed first, and its result is used by the outer query.

Example:

Fetch users whose salary is above the average salary.

The inner query finds the average → outer query compares it.

■ When/Why is it used

- To perform comparisons with aggregated values
- When results depend on another query
- To filter or compute values dynamically
- Useful when JOINs are not convenient

■ Example

```
SELECT name
FROM employees
WHERE salary > (
    SELECT AVG(salary) FROM employees
);
```

■ Short interview line

"A subquery is a query inside another query used to compute results dynamically."

✅ 2. Correlated vs Non-Correlated Subquery

■ Explanation (simple, conceptual)

Non-Correlated Subquery

- Executes **independently**, does NOT depend on outer query.
- Runs **once**, and result is reused.

Example:

```
SELECT name
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

Correlated Subquery

- Depends on the **outer query** for values.
- Executes **once for each row** in the outer query.
- Slower but powerful.

Example:

```
SELECT e1.name
FROM employees e1
WHERE salary > (
    SELECT AVG(salary)
    FROM employees e2
```

```
    WHERE e2.department = e1.department
);
```

Here, the inner query uses `e1.department` from the outer row → correlated.

■ When/Why used

- Non-correlated → when comparing with a single static value (average, max, etc.)
- Correlated → when filtering rows based on their own row-specific context

■ Short interview line

"Non-correlated subqueries run once independently; correlated subqueries run once per row and depend on the outer query."

✅ 3. Can subqueries return multiple rows? When?

■ Explanation (simple)

Yes, subqueries **can return multiple rows**, but only when used with operators that support multiple values.

■ When/Why this happens

- When subquery is in a **WHERE IN**, **ANY**, or **ALL** clause
- When the inner query matches multiple rows
- Useful for filtering based on a list of IDs, categories, or foreign key relationships

■ Example — Subquery returns multiple rows using IN

```
SELECT name
FROM employees
WHERE department_id IN (
    SELECT id FROM departments WHERE location = 'Hyderabad'
);
```

■ Invalid case (multiple rows not allowed)

```
SELECT name
FROM employees
WHERE salary = (SELECT salary FROM employees);   -- ❌ multiple rows error
```

Single-value operators (=, <, >) expect exactly **one value**.

■ Short interview line

"Yes, subqueries can return multiple rows when used with IN/ANY/ALL; single-value operators require a single-row result."

---

6. Indexing (VERY IMPORTANT for backend developers)

✅ 1. What is an index and why is it used?

■ Explanation (simple, conceptual)

An **index** is a data structure (usually a **B-tree**) created on a column to make searching faster.

It works like an index in a book — instead of scanning every page, the database jumps directly to the required location.

■ When/Why is it used

- To speed up **SELECT queries**
- To optimize **WHERE, JOIN, ORDER BY** operations
- To avoid **full table scans**, especially on large tables
- For high-traffic applications (banking, fraud detection, e-commerce)

■ Example

```
CREATE INDEX idx_userid ON expenses(user_id);
```

Now queries filtering by `user_id` become faster.

■ Short interview line

"An index is a data structure that speeds up search operations by avoiding full table scans."

✅ 2. Clustered vs Non-Clustered Index

■ Explanation (simple, conceptual)

Clustered Index

- Determines the **physical order** of rows in a table.
- Only **one clustered index** per table.
- The table data itself is stored in indexed order.
- Primary keys are often clustered indexes.

Non-Clustered Index

- A separate structure containing (index column + pointer to actual rows).
- You can create **multiple** non-clustered indexes.
- Like a searchable map pointing to real data.

■ When/Why used

- Clustered → best for range queries (dates, numeric ranges).

- Non-clustered → best for frequent lookups on non-primary key columns.

■ Example

```
-- Clustered index (usually default on primary key)
PRIMARY KEY (transaction_id)

-- Non-clustered index
CREATE INDEX idx_amount ON expenses(amount);
```

■ Short interview line

"A clustered index sorts the actual table rows; a non-clustered index stores a separate lookup structure."

✅ 3. When NOT to use an index?

■ Explanation (simple)

Indexes have overhead. They are not useful in all cases.

■ Avoid indexing when:

1. **Table is very small**

   → Full scan is faster than using an index.

2. **Column has low cardinality (few unique values)**

   Example: gender (M/F), status (active/inactive).

3. **Column is heavily updated**

   → Index will slow down updates.

4. **You rarely use the column in WHERE/JOIN conditions**

5. **For columns used only for reporting occasionally**

■ Short interview line

"Avoid indexes on small tables, low-uniqueness columns, or columns with heavy write operations."

✅ 4. How indexing impacts INSERT/UPDATE performance?

■ Explanation (simple)

Indexes speed up **read operations**, but they add overhead to **write operations**.

■ Why?

Each time you:

- INSERT a row

- UPDATE an indexed column
- DELETE a row

→ The database must **update the index structure** as well.

This means:

- Insert becomes slower
- Update becomes slower
- Delete becomes slower

■ Example

Updating an indexed column:

```
UPDATE users SET email = 'new@mail.com' WHERE id = 1;
```

If `email` is indexed → index tree must be reorganized.

■ Short interview line

"Indexes speed up reads but slow down writes because the index structure must also be updated."

✅ 5. Why indexing was important in your real-time transaction backend?

■ Explanation (simple, contextual to your resume)

In banking and fraud-detection systems, queries often need to fetch:

- Transactions of a user
- Suspicious patterns
- High-value transactions
- Real-time validations

Without indexing, each query performs a **full table scan**, causing delays.

■ When/Why indexing was needed in your project

- To ensure **millisecond-level lookups** in fraud detection
- To handle **large transaction tables** efficiently
- To speed up **JOINs across users, accounts, and transactions**
- To ensure dashboards retrieve analytics quickly
- To maintain **high availability and low latency** in APIs

■ Example from your experience

In your "Fraud Services" project, you can say:

> “We indexed columns like user_id, transaction_time, and amount to quickly detect anomalies and fetch transaction history. This reduced query time drastically and improved API performance in real time.”

■ Short interview line

“Indexing was critical to achieve low-latency queries in real-time fraud detection and transaction APIs.”

---

7. Keys

✅ 1. Primary Key vs Unique Key

■ Explanation (simple, conceptual)

Feature	Primary Key	Unique Key
Uniqueness	Must be unique	Must be unique
NULL Allowed?	**No** (cannot be NULL)	**Yes** (can have 1 NULL)
Number per table	Only **one**	Can have **multiple**
Purpose	Identifies each row uniquely	Ensures data uniqueness

■ When/Why used

- **Primary key** → for uniquely identifying records (user_id, transaction_id).
- **Unique key** → to avoid duplicate values (email, phone number).

■ Example

```
CREATE TABLE users (
    user_id INT PRIMARY KEY,
    email   VARCHAR(100) UNIQUE
);
```

■ Short interview line

“Primary key uniquely identifies a row and cannot be NULL; unique key enforces uniqueness but allows one NULL.”

✅ 2. Can a table have multiple primary keys?

■ Explanation (simple)

No.

A table can have **only one primary key**, but it may consist of **multiple columns** (called a composite primary key).

■ When/Why

- A primary key defines the table's unique identity.
- Only one such identity can exist.

■ Example — NOT allowed

```
-- Cannot have two primary keys
PRIMARY KEY (id)
PRIMARY KEY (email)   -- ❌ Not allowed
```

■ Short interview line

"A table can have only one primary key, but that key may contain multiple columns."

✅ 3. What is a foreign key?

■ Explanation (simple, conceptual)

A **foreign key** is a column that creates a relationship between two tables.

It ensures **referential integrity** — meaning you cannot insert invalid references.

Example:

An expense must belong to a valid user.

■ When/Why is it used

- To maintain relationships (user → expenses, customer → orders).
- To prevent inconsistent data (cannot store an expense for a non-existent user).
- To enforce cascading updates/deletes if configured.

■ Example

```
CREATE TABLE expenses (
    id INT PRIMARY KEY,
    user_id INT,
    amount DECIMAL,
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```

■ Short interview line

"A foreign key links two tables and ensures valid relationships by enforcing referential integrity."

✅ 4. What is a Composite Key? Where is it used?

■ Explanation (simple)

A **composite key** is a primary key made up of **two or more columns**.

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

■ When/Why used

- When a single column **cannot uniquely identify a row**.
- In many-to-many relationship tables.
- In log/audit tables where uniqueness depends on multiple attributes.

■ Example

For an `attendance` table where the same student can attend multiple classes:

```
CREATE TABLE attendance (
    student_id INT,
    date DATE,
    PRIMARY KEY (student_id, date)
);
```

Here:

- `student_id` alone is not unique
- `date` alone is not unique
- But **(student_id + date)** together is unique → composite key.

■ Short interview line

"A composite key uses multiple columns together to uniquely identify a record, useful when no single column is unique."

---

8. ACID & Transactions (Banking & Fraud Detection relevance)

✅ 1. What are ACID properties?

■ Explanation (simple, conceptual)

ACID properties ensure **reliable, consistent, and safe database transactions**, especially important in banking systems.

ACID stands for:

1. **Atomicity** → All steps of a transaction succeed or none do.
2. **Consistency** → Database moves from one valid state to another.
3. **Isolation** → Transactions do not interfere with each other.
4. **Durability** → Once committed, data is permanently saved even if system crashes.

■ When/Why used

- Banking: money transfer must be atomic.
- Fraud detection: consistent and reliable reads.

- Real-time systems: avoid partial updates or corrupt data.

■ Example

Money transfer steps: debit + credit.

If credit fails → debit must rollback → **atomicity**.

■ Short interview line

"ACID ensures transactions are all-or-nothing, consistent, isolated, and permanently saved."

✅ 2. What is a transaction?

■ Explanation (simple)

A **transaction** is a group of SQL operations executed as a single logical unit of work.

Either:

- All operations succeed → COMMIT
- Or any failure → ROLLBACK

■ When/Why used

- Banking transfers
- Fraud checks
- Updating multiple related tables
- Ensuring data integrity

■ Example

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;
```

■ Short interview line

"A transaction groups multiple SQL statements so they commit or rollback together."

✅ 3. COMMIT vs ROLLBACK

■ Explanation (simple)

- **COMMIT** → Permanently saves all changes.
- **ROLLBACK** → Undoes all uncommitted changes.

■ When/Why used

- COMMIT → after successful operations

- ROLLBACK → on errors, system failures

■ Example

```
COMMIT;    -- save changes
ROLLBACK;  -- undo changes
```

■ Short interview line

"COMMIT saves the transaction; ROLLBACK undoes it."

✅ 4. What is an isolation level?

■ Explanation (simple)

Isolation levels define **how much one transaction is allowed to see changes from another transaction**.

Higher isolation → safer data but slower performance.

■ Common SQL Isolation Levels

1. **Read Uncommitted**
2. **Read Committed**
3. **Repeatable Read**
4. **Serializable** (highest)

■ When/Why used

- Lower levels → better performance
- Higher levels → avoid inconsistencies in banking & financial systems

■ Short interview line

"Isolation levels control how transactions interact and prevent anomalies."

✅ 5. Dirty Read vs Non-Repeatable Read vs Phantom Read

■ Explanation (clear and simple)

1. Dirty Read

Reads **uncommitted** data from another transaction.

Example:

Txn A updates balance but has not committed → Txn B reads temporary value.

---

2. Non-Repeatable Read

Same query gives **different results** within the same transaction because another transaction updated the data.

Example:

Txn A reads balance = 500

Txn B updates balance to 600

Txn A reads again → 600

---

3. Phantom Read

New rows appear when the same query is executed again.

Example:

Txn A runs `SELECT * FROM transactions WHERE amount > 1000`

Txn B inserts a new matching row

Txn A runs again → sees an extra row

■ When/Why it matters

- Banking systems must avoid dirty and non-repeatable reads.
- Analytics dashboards may tolerate phantom reads.

■ Short interview line

"Dirty = uncommitted read; Non-repeatable = updated data; Phantom = new rows appear."

✅ 6. How to prevent dirty reads?

■ Explanation (simple)

Dirty reads happen at the lowest isolation level — **Read Uncommitted**.

To prevent them:

- Use **READ COMMITTED** or higher isolation
- Use locks (database-managed)
- Ensure transactions commit quickly

■ SQL Example

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

■ Short interview line

"Set isolation level to READ COMMITTED or higher to prevent dirty reads."

---

9. SQL Performance Optimisation

✅ 1. How to optimize slow queries?

■ Explanation (simple, conceptual)

Query optimization means making SQL queries run **faster** by reducing full table scans, unnecessary processing, and improving filtering efficiency.

■ When/Why is it used

- When queries take long due to large data
- When APIs slow down because SQL is inefficient
- When dashboards and analytics lag
- Necessary in **real-time banking, fraud detection, and transaction systems**

■ Optimization Techniques

1. **Add proper indexes**
   - On WHERE, JOIN, ORDER BY columns
2. *Avoid SELECT ***
   - Fetch only required columns
3. **Rewrite subqueries using JOINs**
4. **Use LIMIT** when applicable
5. **Avoid functions on indexed columns**

   Example: `WHERE LOWER(name) = 'mohini'` breaks index
6. **Use appropriate data types**
7. **Use EXISTS instead of IN for large datasets**
8. **Partition large tables**
9. **Use caching when possible**

■ Short interview line

“I optimize slow queries by indexing, avoiding SELECT *, rewriting inefficient subqueries, and using EXPLAIN to analyze performance.”

✅ 2. How did you optimize SQL in your Expense Tracker project?

■ Explanation (simple, contextual)

Your Expense Tracker had category-based reports and monthly summaries. These queries became slow when data increased.

■ When/Why optimization was needed

- Reports fetched yearly/monthly totals

- JOINs between users and expenses
- Dashboard analytics required fast responses

■ Your Optimization Actions

You can confidently say:

✔ **Added indexes** on `user_id`, `category`, and `transaction_date`

✔ Replaced subqueries with **JOINs**

✔ Used **GROUP BY + proper indexing** to speed up aggregations

✔ Removed unnecessary columns (avoided SELECT *)

✔ Optimized SQL queries for computing budgets by using:

```
SUM(amount) OVER (PARTITION BY user_id)
```

✔ Used **EXPLAIN ANALYZE** to analyze slow queries

■ Short interview line

"I improved performance by indexing key columns, rewriting subqueries as JOINs, and using EXPLAIN ANALYZE to remove bottlenecks."

✅ 3. What is EXPLAIN / EXPLAIN ANALYZE?

■ Explanation (simple, conceptual)

EXPLAIN shows how the database plans to execute a query.

EXPLAIN ANALYZE actually executes the query and shows the **real performance stats**.

These tools help identify:

- Full table scans
- Inefficient JOINs
- Missing indexes
- Cost of operations

■ When/Why used

- To debug slow queries
- To verify if indexes are being used
- To understand the query execution path

■ Example

```
EXPLAIN ANALYZE
SELECT * FROM expenses WHERE user_id = 10;
```

■ Short interview line

“EXPLAIN shows the query plan; EXPLAIN ANALYZE executes it and shows real performance details.”

✅ 4. Why should we avoid SELECT * ?

■ Explanation (simple)

`SELECT *` returns **all columns**, even if you need only a few.

■ When/Why it becomes a problem

- Wastes I/O and network bandwidth
- Slows down performance for large tables
- Breaks caching since column order can change
- Prevents efficient use of indexes (covering indexes)
- Adds unnecessary load in microservices and dashboards

■ Better Practice

Specify exact columns:

```
SELECT name, email FROM users;
```

■ Short interview line

“We avoid SELECT * because it increases I/O, prevents index optimization, and impacts performance.”

✅ 5. How indexes improved your query performance?

■ Explanation (simple, conceptual)

Indexes reduce the number of rows the database scans.

Instead of scanning 1 million rows → index narrows it down to a few matches.

■ When/Why used

- To speed up filtering (`WHERE user_id = ?`)
- For faster JOINs
- For sorting and grouping
- Essential for large transaction tables

■ Your real-time experience example

In your Expense Tracker + Banking/Fraud systems, you can say:

> “Indexes on user_id, transaction_date, and category reduced query time from seconds to milliseconds. This was crucial for real-time dashboards and fraud detection APIs.”

■ Short interview line

“Indexes drastically reduced full table scans and improved lookup and JOIN performance in real-time systems.”

---

10. Stored Procedures & Views

✅ 1. What is a view? Can we insert into a view?

■ Explanation (simple, conceptual)

A **view** is a **virtual table** created from a SQL query.

It does NOT store data; it displays data from underlying tables.

Think of it like a saved SELECT query that behaves like a table.

■ When/Why is it used

- To simplify complex SQL queries
- To restrict access to sensitive columns
- To provide a clean interface to underlying tables
- To support BI dashboards and reports

■ Can we insert into a view?

Yes, but only under specific conditions:

- View must be **updatable**
- View should be based on **one table**
- Should not contain:
  - Aggregations (SUM, AVG)
  - GROUP BY, HAVING
  - DISTINCT
  - Joins
  - Subqueries

If the view is complex → insert/update is **not allowed**.

■ Example

```
CREATE VIEW active_users AS
SELECT id, name FROM users WHERE status = 'ACTIVE';
```

Insert (only if allowed):

```
INSERT INTO active_users (id, name) VALUES (10, 'Mohini');
```

■ Short interview line

“A view is a virtual table based on a query; inserts are allowed only for simple, updatable views.”

✅ 2. What are stored procedures?

■ Explanation (simple, conceptual)

A **stored procedure** is a precompiled set of SQL statements stored in the database and executed as a program.

■ When/Why is it used

- To perform repeated operations
- To reduce network calls
- To improve performance with precompiled SQL
- To apply business logic inside the database
- Useful in backend systems needing validations, logics, and multi-step transactions

■ Example

```
CREATE PROCEDURE GetUserExpenses(IN uid INT)
BEGIN
   SELECT * FROM expenses WHERE user_id = uid;
END;
```

■ Short interview line

“A stored procedure is a precompiled SQL program that performs a defined task inside the database.”

✅ 3. Stored Procedure vs Function

■ Explanation (comparison)

Feature	Stored Procedure	Function
Purpose	Perform operations	Return a value
Returns	Can return multiple values	Must return exactly one value
Use in SELECT	Cannot be used in SELECT	Can be used in SELECT
Transactions	Supports COMMIT/ROLLBACK	Does NOT allow transactions
Parameters	IN, OUT, INOUT	Only IN

Feature	Stored Procedure	Function
Use-case	Business logic, complex operations	Calculations, transformations

■ When/Why used

- **Procedure** → multi-step operations, updates, inserts, complex logic
- **Function** → reusable calculations inside queries

■ Example Function

```
CREATE FUNCTION GetExpenseTotal(uid INT)
RETURNS DECIMAL
BEGIN
   RETURN (SELECT SUM(amount) FROM expenses WHERE user_id = uid);
END;
```

■ Short interview line

"Procedures perform operations and can return multiple results; functions return a single value and can be used in SELECT."

✅ 4. When did you use stored procedures in your backend? (Your real experience)

■ Explanation (contextual to your resume)

You worked on **real-time banking and fraud detection**, where certain operations required consistent, optimized, and secure DB logic.

■ When/Why you used stored procedures

You can confidently say:

> "In the Fraud Services and Expense Tracker projects, I used stored procedures for multi-step DB logic, such as validating transactions, aggregating expenses, and performing secure updates. Since these operations were repeated often and required consistency, stored procedures helped improve performance and reduce backend code complexity."

■ Real examples you can mention

- **Expense Tracker**
  - Calculating monthly total expenses
  - Auto-categorizing expenses
  - Reducing repeated SQL logic
- **Fraud Detection Service**
  - Fetching last N transactions
  - Running rule-based checks

- Logging suspicious activity

■ Short interview line

“I used stored procedures to encapsulate repeated financial logic, optimize performance, and ensure consistent multi-step operations.”

⭐ Practical Query Questions (They will ask you to write)

1. Write a query to find the 2nd highest salary

■ Explanation (simple, conceptual)

We want the second-largest distinct salary value. Approaches: use window functions (preferred, clear), or subquery that picks the max of salaries less than the overall max.

■ When/Why is it used

Common interview question to test knowledge of window functions, subqueries, and handling duplicates (ties).

■ Example / Code snippet

Using window functions (Postgres / MySQL 8+ / SQL Server):

```
-- returns 2nd highest distinct salary
SELECT salary
FROM (
  SELECT DISTINCT salary,
        DENSE_RANK() OVER (ORDER BY salary DESC) AS rnk
  FROM employees
) t
WHERE rnk = 2;
```

Using subquery (works on older MySQL too):

```
-- assumes there are at least 2 distinct salaries
SELECT MAX(salary) AS second_highest
FROM employees
WHERE salary < (SELECT MAX(salary) FROM employees);
```

Edge case (ties) — if you want "second row" even with same values, use ROW_NUMBER instead of DENSE_RANK.

■ Short summary line to speak in interview

“Use DENSE_RANK() over salary DESC to get the 2nd highest distinct salary, or MAX(...) WHERE salary < MAX(...) as a subquery alternative.”

---

2. Write a query to find duplicate records in a table

■ Explanation (simple, conceptual)

Find groups of rows that share the same values on the columns that define duplicates, using `GROUP BY` and `HAVING COUNT(*) > 1`.

■ When/Why is it used

Detecting data quality issues, removing duplicates, or auditing ingestion pipelines.

■ Example / Code snippet

Find duplicate keys/columns (for example on columns `col1, col2`):

```
SELECT col1, col2, COUNT(*) AS cnt
FROM my_table
GROUP BY col1, col2
HAVING COUNT(*) > 1;
```

If you want the full duplicate rows (including IDs):

```
SELECT t.*
FROM my_table t
JOIN (
  SELECT col1, col2
  FROM my_table
  GROUP BY col1, col2
  HAVING COUNT(*) > 1
) d ON t.col1 = d.col1 AND t.col2 = d.col2
ORDER BY t.col1, t.col2;
```

■ Short summary line to speak in interview

“Group by the duplicate-defining columns and use `HAVING COUNT(*) > 1` to find duplicates.”

---

3. Write a query to delete duplicates but keep only one

■ Explanation (simple, conceptual)

Identify duplicate rows with a row-number per group and delete rows where row_number > 1. Modern databases support `ROW_NUMBER()` in a CTE for a safe delete.

■ When/Why is it used

Clean up data while preserving one canonical row for each duplicate group.

■ Example / Code snippet

Using CTE + ROW_NUMBER() (Postgres / MySQL 8+ / SQL Server):

```
WITH ranked AS (
  SELECT *,
         ROW_NUMBER() OVER (PARTITION BY col1, col2 ORDER BY id) AS rn
  FROM my_table
)
DELETE FROM my_table
WHERE id IN (
  SELECT id FROM ranked WHERE rn > 1
);
```

(If your DB supports `DELETE FROM ranked WHERE rn > 1` *directly, you can use that syntax — e.g., Postgres:* `WITH ranked AS (...) DELETE FROM my_table USING ranked WHERE my_table.id = ranked.id AND ranked.rn > 1;` *)*

Alternative for older MySQL (no window functions):

```
DELETE t1 FROM my_table t1
INNER JOIN my_table t2
  ON t1.col1 = t2.col1
 AND t1.col2 = t2.col2
 AND t1.id > t2.id;
```

(This keeps the row with the smallest `id`.)

■ Short summary line to speak in interview

"Use ROW_NUMBER() partitioned by duplicate columns and delete rows with rn > 1; for older MySQL use a self-join delete keeping the smallest id."

---

4. Write a query to count expenses per category (GROUP BY)

■ Explanation (simple, conceptual)

Use `GROUP BY` on the category column and `COUNT()` or `SUM()` to get totals per category.

■ When/Why is it used

Common for reports, dashboards, category-wise analytics.

■ Example / Code snippet

```
SELECT category,
       COUNT(*)      AS expense_count,
       SUM(amount)   AS total_amount,
       AVG(amount)   AS avg_amount
FROM expenses
GROUP BY category
ORDER BY total_amount DESC;
```

■ Short summary line to speak in interview

“Group by category and use COUNT()/SUM() to get counts and totals per category.”

---

5. Write a query to return users who have not logged an expense (LEFT JOIN + WHERE IS NULL)

■ Explanation (simple, conceptual)

Left join users to expenses and filter rows where the expense side is null — gives users without matching expenses.

■ When/Why is it used

Find inactive users or missing activity records.

■ Example / Code snippet

```
SELECT u.id, u.name
FROM users u
LEFT JOIN expenses e ON u.id = e.user_id
WHERE e.user_id IS NULL;
```

Alternative using NOT EXISTS (often efficient):

```
SELECT u.id, u.name
FROM users u
WHERE NOT EXISTS (
  SELECT 1 FROM expenses e WHERE e.user_id = u.id
);
```

■ Short summary line to speak in interview

“LEFT JOIN users to expenses and filter `WHERE e.user_id IS NULL` to find users with no expenses; `NOT EXISTS` is a good alternative.”

---

6. Write a query to fetch top 3 highest transactions for each user (window functions)

■ Explanation (simple, conceptual)

Partition transactions by user and rank by amount (or timestamp). Use `ROW_NUMBER()` or `RANK()` depending on tie behavior, then filter rows with rank ≤ 3.

■ When/Why is it used

Per-user leaderboards, recent/high-value transaction lists, anomaly detection.

■ Example / Code snippet

```
SELECT user_id, transaction_id, amount, txn_time
FROM (
  SELECT *,
       ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY amount DESC) AS rn
  FROM transactions
) t
WHERE rn <= 3
ORDER BY user_id, rn;
```

If you want ties to allow more than 3 when amounts tie, use `RANK()` or `DENSE_RANK()` instead of `ROW_NUMBER()`.

■ Short summary line to speak in interview

"Use `ROW_NUMBER()` over `PARTITION BY user_id ORDER BY amount DESC` and select rows where rn ≤ 3 to get top 3 transactions per user."

⭐ Scenario-Based Questions (Based on Her Resume)

1. In real-time fraud detection, how did you optimize database performance?

■ Explanation (simple, conceptual)

I optimized DB performance by reducing work the DB must do at request-time and by making lookups targeted and indexable. Key techniques: selective indexing, partitioning, read replicas, caching, query tuning, pre-aggregation, and moving heavy computations out of the hot path.

■ When / Why it is used

Used when APIs must respond in milliseconds for fraud checks (e.g., checking last N transactions, user velocity). These techniques prevent full-table scans and keep latency predictable under high throughput.

■ Example / Code snippet (practical actions I took)

- **Indexes** on `user_id`, `transaction_time`, (`user_id`, `transaction_time`) and composite indexes for common queries:

```
CREATE INDEX idx_txn_user_time ON transactions(user_id, transaction_time DESC);
```

- **Partitioning** by transaction date (daily/monthly) to limit scanned partitions:

```
-- Postgres example (conceptual)
CREATE TABLE transactions (
  id BIGSERIAL PRIMARY KEY,
  user_id INT,
```

```
  amount NUMERIC,
  transaction_time TIMESTAMP
) PARTITION BY RANGE (transaction_time);
```

- **Read replicas** for analytics/long-running reads so OLTP is unaffected.
- **Caching** recent N transactions in Redis for each user (TTL short) to answer velocity checks without hitting DB.
- **Pre-aggregated tables / materialized views** for common dashboard metrics:

```
CREATE MATERIALIZED VIEW mv_user_daily_totals AS
SELECT user_id, date_trunc('day', transaction_time) AS day, SUM(amount) total
FROM transactions GROUP BY user_id, day;
```

- **EXPLAIN ANALYZE** used to find slow plans and rewrite queries (turn subqueries into joins / avoid functions on indexed columns).

■ Short summary line to speak in interview

“I reduced latency by targeted indexing, partitioning, using read replicas + Redis caching, and pre-aggregating common metrics so fraud checks run in milliseconds.”

---

2. How did you ensure transaction consistency in your banking APIs?

■ Explanation (simple, conceptual)

I ensured consistency by using DB transactions (ACID), proper isolation levels, idempotency, optimistic concurrency (versioning) where applicable, and compensating flows for distributed operations. For multi-step operations spanning services, I relied on patterns that avoid inconsistent partial states.

■ When / Why it is used

Used for money movement, ledger updates, and any multi-row or multi-table updates where partial success would corrupt financial state. Critical in banking/fraud to avoid double-spend and inconsistent balances.

■ Example / Code snippet (practical patterns)

- **Single DB transaction** for related writes:

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = 1 AND balance >= 100;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
INSERT INTO transfers (from_acc, to_acc, amount, status) VALUES (1,2,100,'COMPLETED');
COMMIT;
-- if any step fails → ROLLBACK
```

- **Optimistic locking** (version column) to prevent lost updates:

```
UPDATE accounts
SET balance = balance + :delta, version = version + 1
WHERE id = :id AND version = :expected_version;
-- check rows affected == 1 else retry
```

- **Idempotency keys** on API level to avoid double posting on retries:
  - store `idempotency_key` with operation and ignore duplicates.
- **For cross-service flows** use **sagas / compensating transactions** instead of distributed 2PC:
  - Reserve funds in service A, confirm in service B, if failure then compensate (release reservation).
- **Appropriate isolation** like `READ COMMITTED` or `REPEATABLE READ` depending on anomaly tolerance; avoid `READ UNCOMMITTED` for money ops.

■ Short summary line to speak in interview

"I used DB transactions + optimistic locking, idempotency keys, and sagas for distributed flows to guarantee consistency without blocking scale."

---

3. Why did you choose MySQL over NoSQL for Expense Tracker?

■ Explanation (simple, conceptual)

Expense Tracker required **structured, relational** data (transactions, users, categories), strong ACID guarantees for monetary data, complex queries (aggregations, GROUP BY, joins), and easy ad-hoc reporting — all strengths of relational DBs like MySQL.

■ When / Why it is used

Choose relational when data relationships, joins, transactions, and precise querying matter — especially with financial and audit requirements (ACID, constraints, schema). NoSQL is better when you need extreme horizontal scale with flexible schema and relaxed consistency.

■ Example / Code snippet (reasons mapped to features)

- **Transactions & constraints**:

```
-- enforce referential integrity
ALTER TABLE expenses ADD CONSTRAINT fk_user FOREIGN KEY (user_id) REFERENCES users (id);
```

- **Analytics with SQL** (GROUP BY, window functions) and ease of indexing:

```
SELECT category, SUM(amount) FROM expenses WHERE user_id = 1 GROUP BY category;
```

■ Short summary line to speak in interview

"I chose MySQL for the Expense Tracker because it provides ACID transactions, strong referential integrity, and rich SQL for analytics — essential for financial data and reporting."

---

4. How did you handle millions of transaction records efficiently?

■ Explanation (simple, conceptual)

I made reads fast and writes sustainable through **partitioning/sharding, indexing, archiving**, pre-aggregation, and using appropriate storage/retention policies — plus background jobs for heavy analytics.

■ When / Why it is used

When the transaction table grows into millions (or billions), naive queries and backups become slow; these patterns keep production queries fast and storage manageable.

■ Example / Code snippet (practical strategies)

- **Range partitioning** (by date) so queries touching recent data hit few partitions:

```
-- conceptual example
ALTER TABLE transactions PARTITION BY RANGE (YEAR(transaction_time)) (
  PARTITION p2024 VALUES LESS THAN (2025),
  PARTITION p2025 VALUES LESS THAN (2026)
);
```

- **Archival pipeline**: move older data to analytical storage (data lake / cold DB) and keep a summarized history in OLTP.
- **Sharding** by `user_id` when single-node scale limit reached.
- **Materialized summaries** for dashboard queries (daily/hourly aggregates) to avoid scanning raw transactions:

```
INSERT INTO daily_user_summary (user_id, day, total)
SELECT user_id, date_trunc('day', transaction_time), SUM(amount)
FROM transactions WHERE transaction_time >= current_date - 1
GROUP BY user_id, date_trunc('day', transaction_time);
```

- **Bulk-loading & batch jobs**: use bulk insert for ETL, and async processing for heavy enrichments.
- **Monitoring & maintenance**: regular index maintenance, vacuum/optimize, partition pruning, and tuned connection pools.

■ Short summary line to speak in interview

“To handle millions of rows I used partitioning/sharding, archival for cold data, materialized summaries, and background ETL so real-time queries stay fast.”

---

5. How do you design tables for analytics dashboards (budget insights)?

■ Explanation (simple, conceptual)

I design analytics with a **star-like model**: small dimension tables (users, categories) and a fact table (transactions). I also create pre-aggregated fact tables (daily/monthly summaries) and materialized views for common queries to serve dashboards quickly.

■ When / Why it is used

Dashboards need fast aggregation across time and categories with predictable latency. Star schemas and pre-aggregation reduce JOIN work and scanning of raw transactional data.

■ Example / Code snippet (table design + sample materialized view)

Star schema

```
-- Dimension tables
CREATE TABLE dim_user (user_id INT PRIMARY KEY, name TEXT, region TEXT);
CREATE TABLE dim_category (category_id INT PRIMARY KEY, name TEXT);

-- Fact table
CREATE TABLE fact_transactions (
  txn_id BIGSERIAL PRIMARY KEY,
  user_id INT REFERENCES dim_user(user_id),
  category_id INT REFERENCES dim_category(category_id),
  amount NUMERIC,
  txn_time TIMESTAMP,
  is_fraud BOOLEAN DEFAULT FALSE
);
```

Pre-aggregated daily summary (fact_rollup):

```
CREATE TABLE fact_daily_user_category (
  day DATE,
  user_id INT,
  category_id INT,
  total_amount NUMERIC,
  txn_count BIGINT,
  PRIMARY KEY (day, user_id, category_id)
);
-- populate nightly/near-real-time via ETL
```

Materialized view for dashboard:

```
CREATE MATERIALIZED VIEW mv_user_monthly_summary AS
SELECT user_id, date_trunc('month', txn_time) AS month,
      SUM(amount) total, COUNT(*) cnt
FROM fact_transactions
GROUP BY user_id, date_trunc('month', txn_time);
```

Other design considerations

- Use **appropriate data types** and timezone-aware timestamps.

- **Partition** rollup tables by date for fast pruning.
- Add indexes on `(user_id, day)` and `(category_id, day)`.
- Maintain **retention** policy (hot months in OLTP, older in data warehouse).
- Provide API endpoints that read from rollups/materialized views, not raw transactions.

■ Short summary line to speak in interview

“I design a star schema with fact tables and pre-aggregated rollups/materialized views so dashboards query small, indexed tables and return results instantly.”
