

35 Essential Questions & Answers
with Code Examples for Learners

Fundamentals · Joins · Window Functions · Performance · Design

35 Questions	7 Categories	3 Difficulty Levels
-----------------	-----------------	------------------------

Q01

What is SQL and what are its main sublanguages?

● Easy | Fundamentals

■ Answer

SQL (Structured Query Language) is the standard language for managing relational databases. It is divided into four sublanguages: DDL (Data Definition Language) for schema operations like CREATE, ALTER, DROP; DML (Data Manipulation Language) for data operations like SELECT, INSERT, UPDATE, DELETE; DCL (Data Control Language) for permissions like GRANT and REVOKE; and TCL (Transaction Control Language) for transactions like COMMIT and ROLLBACK.

■ SQL Example

```
SQL
1  -- DDL: Create a table
2  CREATE TABLE employees (
3      id      INT PRIMARY KEY,
4      name    VARCHAR(100) NOT NULL,
5      salary  DECIMAL(10,2),
6      dept    VARCHAR(50)
7  );
8
9  -- DML: Insert and Query
10 INSERT INTO employees VALUES (1, 'Alice', 75000, 'Engineering');
11 SELECT * FROM employees WHERE dept = 'Engineering';
```

■ Tip: DDL auto-commits, but DML statements can be rolled back within a transaction.

Q02

What is the difference between WHERE and HAVING?

● Easy | Filtering

■ Answer

WHERE filters rows BEFORE aggregation — it operates on individual rows and cannot use aggregate functions. HAVING filters groups AFTER the GROUP BY aggregation — it is designed to work with aggregate functions like COUNT, SUM, AVG.

■ SQL Example

```
SQL
1  SELECT dept, COUNT(*) AS headcount, AVG(salary) AS avg_salary
2  FROM employees
3  WHERE salary > 40000           -- Row-level filter (before grouping)
4  GROUP BY dept
5  HAVING COUNT(*) >= 3;         -- Group-level filter (after aggregation)
```

■ Tip: A common interview trick — if you need to filter on a COUNT or SUM, you MUST use HAVING, not WHERE.

Q03

Explain the different types of JOINS in SQL.

◆ Medium | Joins

■ Answer

INNER JOIN returns only rows with matching values in both tables. LEFT JOIN returns all rows from the left table plus matched rows from the right (NULLs for no match). RIGHT JOIN is the mirror of LEFT JOIN. FULL OUTER JOIN returns all rows from both tables with NULLs where there is no match. CROSS JOIN returns the Cartesian product — every combination of rows.

■ SQL Example

```
SQL
1  -- INNER JOIN: only matched rows
2  SELECT e.name, d.department_name
3  FROM   employees e
4  INNER JOIN departments d ON e.dept_id = d.id;
5
6  -- LEFT JOIN: all employees, even without a dept
7  SELECT e.name, d.department_name
8  FROM   employees e
9  LEFT JOIN departments d ON e.dept_id = d.id;
10
11 -- FULL OUTER JOIN: all rows from both sides
12 SELECT e.name, d.department_name
13 FROM   employees e
14 FULL OUTER JOIN departments d ON e.dept_id = d.id;
```

■ Tip: To find employees with NO department, use LEFT JOIN ... WHERE d.id IS NULL.

Q04

What are Window Functions and how do they differ from GROUP BY?

▲ Hard | Window Functions

■ Answer

Window functions perform calculations across a set of rows (a 'window') related to the current row WITHOUT collapsing rows like GROUP BY does. Each row retains its identity while also receiving a computed value. They use the OVER() clause with optional PARTITION BY and ORDER BY.

■ SQL Example

```
SQL

1  SELECT
2      name,
3      dept,
4      salary,
5      AVG(salary) OVER (PARTITION BY dept) AS dept_avg,
6      RANK() OVER (PARTITION BY dept ORDER BY salary DESC) AS rank_in_dept,
7      ROW_NUMBER() OVER (ORDER BY salary DESC) AS overall_rank
8  FROM employees;
```

■ Tip: ROW_NUMBER() gives unique ranks, RANK() ties share rank and skip next, DENSE_RANK() ties share rank but does NOT skip.

Q05

What is a CTE (Common Table Expression) and when should you use it?

◆ Medium | Advanced SQL

■ Answer

A CTE is a named temporary result set defined with the WITH clause. It exists only for the duration of the query. CTEs improve readability, allow recursive queries, and can be referenced multiple times in the same query — unlike subqueries which must be repeated.

■ SQL Example

```
1  WITH high_earners AS (  
2      SELECT dept, name, salary  
3      FROM    employees  
4      WHERE   salary > 80000  
5  ),  
6  dept_summary AS (  
7      SELECT dept, COUNT(*) AS count, AVG(salary) AS avg_sal  
8      FROM    high_earners  
9      GROUP BY dept  
10 )  
11 SELECT h.name, h.salary, d.avg_sal  
12 FROM    high_earners h  
13 JOIN    dept_summary d ON h.dept = d.dept  
14 ORDER BY h.salary DESC;
```

■ Tip: Recursive CTEs are perfect for hierarchical data like org charts or category trees.

Q06

What is the difference between UNION and UNION ALL?

● Easy | Set Operations

■ Answer

UNION combines results of two queries and removes duplicate rows. UNION ALL combines results and keeps ALL rows including duplicates. UNION ALL is significantly faster because it skips the deduplication step. Both queries must have the same number of columns with compatible data types.

■ SQL Example

```
SQL

1  -- UNION: removes duplicates
2  SELECT name FROM employees_2023
3  UNION
4  SELECT name FROM employees_2024;
5
6  -- UNION ALL: keeps all rows (faster)
7  SELECT name, salary FROM employees_2023
8  UNION ALL
9  SELECT name, salary FROM employees_2024
10 ORDER BY salary DESC;
```

■ *Tip: Always prefer UNION ALL when you know duplicates won't exist or don't matter — it's much faster on large datasets.*

Q07

What is a Subquery and what types exist?

◆ Medium | Subqueries

■ Answer

A subquery is a query nested inside another query. Types include: Scalar subquery (returns one value), Row subquery (returns one row), Table subquery (returns multiple rows and columns), Correlated subquery (references outer query — re-executes per row), and Non-correlated subquery (independent of the outer query).

■ SQL Example

```
SQL
1  -- Scalar subquery: single value
2  SELECT name, salary,
3         (SELECT AVG(salary) FROM employees) AS company_avg
4  FROM employees;
5
6  -- Correlated subquery: find employees earning above dept avg
7  SELECT name, dept, salary
8  FROM employees e
9  WHERE salary > (
10     SELECT AVG(salary)
11     FROM employees
12     WHERE dept = e.dept  -- references outer query
13 );
```

■ Tip: Correlated subqueries can be slow on large tables — consider rewriting as a JOIN with a CTE for performance.

Q08

Explain Primary Key vs Foreign Key.

● Easy | Constraints

■ Answer

A Primary Key uniquely identifies each row in a table — it cannot be NULL and must be unique. A Foreign Key is a column in one table that references the Primary Key of another table, enforcing referential integrity. A table can have only one Primary Key but multiple Foreign Keys.

■ SQL Example

```
1 CREATE TABLE departments (  
2     dept_id    INT PRIMARY KEY,  
3     dept_name  VARCHAR(100) NOT NULL UNIQUE  
4 );  
5  
6 CREATE TABLE employees (  
7     emp_id     INT PRIMARY KEY,  
8     name       VARCHAR(100) NOT NULL,  
9     dept_id    INT,  
10    FOREIGN KEY (dept_id)  
11        REFERENCES departments(dept_id)  
12        ON DELETE SET NULL  
13        ON UPDATE CASCADE  
14 );
```

■ Tip: ON DELETE CASCADE will remove child rows when the parent is deleted. ON DELETE SET NULL sets the FK to NULL instead.

Q09

What is Normalization? Explain 1NF, 2NF, 3NF.

◆ Medium | Database Design

■ Answer

Normalization organizes database structure to reduce redundancy. 1NF: Each column holds atomic values; no repeating groups. 2NF: In 1NF + every non-key attribute is fully dependent on the ENTIRE primary key (eliminates partial dependency). 3NF: In 2NF + no transitive dependencies — non-key columns must depend ONLY on the primary key.

■ SQL Example

```
SQL
1  -- Violation of 1NF: multiple values in one column
2  -- BAD: skills = 'Python, SQL, Java'
3
4  -- Proper 1NF design:
5  CREATE TABLE employee_skills (
6      emp_id    INT,
7      skill     VARCHAR(50),
8      PRIMARY KEY (emp_id, skill)
9  );
10
11 -- 3NF: separate city info (transitive dependency removed)
12 CREATE TABLE cities (
13     zip_code   CHAR(5) PRIMARY KEY,
14     city       VARCHAR(100),
15     state      VARCHAR(50)
16 );
```

■ Tip: In interviews, always mention denormalization — sometimes intentionally breaking normal forms improves read performance.

Q10

What is an Index and how does it improve performance?

◆ Medium | Performance

■ Answer

An index is a data structure (typically a B-tree) that allows the database to find rows without scanning the entire table. It speeds up SELECT queries but slightly slows INSERT/UPDATE/DELETE because the index must be maintained. Columns frequently used in WHERE, JOIN, or ORDER BY clauses are good candidates.

■ SQL Example

```
SQL
1  -- Create a single-column index
2  CREATE INDEX idx_employee_dept ON employees(dept);
3
4  -- Composite index (order matters!)
5  CREATE INDEX idx_dept_salary ON employees(dept, salary);
6
7  -- Unique index
8  CREATE UNIQUE INDEX idx_emp_email ON employees(email);
9
10 -- Check index usage with EXPLAIN
11 EXPLAIN SELECT * FROM employees WHERE dept = 'Engineering';
```

■ Tip: Composite index follows 'left-prefix' rule — `idx(dept, salary)` helps queries on `dept` alone but NOT `salary` alone.

Q11

What is the difference between DELETE, TRUNCATE, and DROP?

● Easy | DML/DDL

■ Answer

DELETE removes specific rows using WHERE — it's DML, can be rolled back, and fires triggers. TRUNCATE removes ALL rows instantly — it's DDL, is faster, cannot be rolled back in most DBs, and resets auto-increment. DROP removes the entire table structure and data — it's DDL and cannot be rolled back.

■ SQL Example

```
SQL
1  -- DELETE: removes specific rows, can be rolled back
2  BEGIN;
3  DELETE FROM employees WHERE dept = 'HR';
4  ROLLBACK; -- rows restored
5
6  -- TRUNCATE: removes all rows instantly, resets identity
7  TRUNCATE TABLE temp_log;
8
9  -- DROP: removes the entire table permanently
10 DROP TABLE IF EXISTS old_archive;
```

■ Tip: In PostgreSQL, TRUNCATE can actually be rolled back inside a transaction — unlike MySQL.

Q12

What are ACID properties in databases?

◆ Medium | Transactions

■ Answer

ACID stands for: Atomicity — a transaction completes fully or not at all. Consistency — data moves from one valid state to another. Isolation — concurrent transactions don't interfere with each other. Durability — committed transactions survive system failures and are permanently saved.

■ SQL Example

```
SQL

1 BEGIN TRANSACTION;
2
3 -- Atomicity: both must succeed or both fail
4 UPDATE accounts SET balance = balance - 1000 WHERE id = 1;
5 UPDATE accounts SET balance = balance + 1000 WHERE id = 2;
6
7 -- If anything fails:
8 -- ROLLBACK; -- reverts both changes
9
10 -- If all succeeds:
11 COMMIT; -- both changes are permanently saved
```

■ Tip: Isolation levels (READ UNCOMMITTED → SERIALIZABLE) trade performance for data consistency. Know the tradeoffs!

Q13

Explain GROUP BY with aggregate functions.

● Easy | Aggregation

■ Answer

GROUP BY groups rows sharing the same value in specified columns, then aggregate functions (COUNT, SUM, AVG, MAX, MIN) compute a result for each group. Every column in SELECT that is not an aggregate must appear in GROUP BY.

■ SQL Example

```
1  SELECT
2      dept,
3      COUNT(*)           AS total_employees,
4      SUM(salary)        AS total_payroll,
5      AVG(salary)        AS avg_salary,
6      MAX(salary)        AS highest_salary,
7      MIN(salary)        AS lowest_salary
8  FROM employees
9  GROUP BY dept
10 HAVING COUNT(*) > 2
11 ORDER BY avg_salary DESC;
```

■ Tip: COUNT(*) counts all rows including NULLs, while COUNT(column) ignores NULLs in that column.

Q14

What is a View and when would you use it?

● Easy | Views

■ Answer

A View is a stored SQL query that acts like a virtual table. It doesn't store data itself but presents a specific perspective on underlying tables. Views are used for security (hiding sensitive columns), simplifying complex queries, and providing consistent data access.

■ SQL Example

```
SQL

1  -- Create a view hiding sensitive salary info
2  CREATE VIEW public_employee_info AS
3  SELECT emp_id, name, dept, hire_date
4  FROM employees;
5
6  -- Updatable view
7  CREATE VIEW engineering_team AS
8  SELECT * FROM employees
9  WHERE dept = 'Engineering'
10 WITH CHECK OPTION; -- prevents inserting rows outside the view
11
12 SELECT * FROM public_employee_info WHERE dept = 'Marketing';
```

■ Tip: Materialized Views (in PostgreSQL, Oracle) actually store the data and require refresh — they're faster but not always current.

Q15

What is the difference between RANK(), DENSE_RANK(), and ROW_NUMBER()?

▲ Hard | Window Functions

■ Answer

ROW_NUMBER() assigns a unique sequential integer to each row — no ties. RANK() gives tied rows the same rank but SKIPS subsequent ranks (1,1,3). DENSE_RANK() gives tied rows the same rank and does NOT skip (1,1,2). This distinction is critical for top-N-per-group queries.

■ SQL Example

```
1 SELECT
2     name, dept, salary,
3     ROW_NUMBER() OVER (ORDER BY salary DESC) AS row_num,
4     RANK() OVER (ORDER BY salary DESC) AS rnk,
5     DENSE_RANK() OVER (ORDER BY salary DESC) AS dense_rnk
6 FROM employees;
7
8 -- Practical: get top 3 earners per department
9 SELECT * FROM (
10     SELECT *, DENSE_RANK() OVER
11         (PARTITION BY dept ORDER BY salary DESC) AS dr
12     FROM employees
13 ) ranked WHERE dr <= 3;
```

■ Tip: ROW_NUMBER is most common for pagination; DENSE_RANK is best for 'top N per group' problems in interviews.

Q16

What is a Stored Procedure and how does it differ from a Function?

◆ Medium | Programmability

■ Answer

A Stored Procedure is a precompiled set of SQL statements stored in the database. Functions must return a value and can be used in SELECT statements. Procedures don't have to return a value, can return multiple result sets, and can use DML with COMMIT/ROLLBACK. Functions cannot have side effects in most databases.

■ SQL Example

```
1  -- Stored Procedure
2  CREATE PROCEDURE give_raise(
3      IN dept_name VARCHAR(50),
4      IN raise_pct  DECIMAL(5,2)
5  )
6  BEGIN
7      UPDATE employees
8      SET salary = salary * (1 + raise_pct / 100)
9      WHERE dept = dept_name;
10 END;
11
12 CALL give_raise('Engineering', 10.0);
13
14 -- Scalar Function
15 CREATE FUNCTION get_annual_salary(monthly DECIMAL)
16 RETURNS DECIMAL
17 RETURN monthly * 12;
```

■ Tip: Procedures are for actions/business logic; Functions are for computations you embed in queries.

Q17

How does SQL handle NULL values? What are the common pitfalls?

◆ Medium | NULL Handling

■ Answer

NULL represents the absence of a value — it is NOT zero, empty string, or false. NULL comparisons always return UNKNOWN (not TRUE or FALSE), which is why `= NULL` never works. Use `IS NULL` or `IS NOT NULL`. Aggregate functions ignore NULLs except `COUNT(*)`. Use `COALESCE` or `NULLIF` to handle NULLs gracefully.

■ SQL Example

```
SQL
1  -- WRONG: this finds nothing!
2  SELECT * FROM employees WHERE manager_id = NULL;
3
4  -- CORRECT:
5  SELECT * FROM employees WHERE manager_id IS NULL;
6
7  -- COALESCE: return first non-NULL value
8  SELECT name, COALESCE(phone, email, 'No contact') AS contact
9  FROM employees;
10
11 -- NULLIF: return NULL if two values are equal (avoid div/0)
12 SELECT total_sales / NULLIF(num_transactions, 0) AS avg_sale
13 FROM sales_summary;
```

■ Tip: NULL in a JOIN condition means the rows won't match — always account for NULLs when debugging unexpected empty results.

Q18

What is a Self-Join and when is it useful?

◆ Medium | Joins

■ Answer

A Self-Join joins a table to itself using aliases. It's useful for hierarchical or comparative queries within the same table, such as finding an employee's manager (both stored in the same employees table) or comparing rows within the same dataset.

■ SQL Example

```
SQL
1  -- Employee-Manager hierarchy (same table)
2  SELECT
3      e.name AS employee,
4      m.name AS manager
5  FROM employees e
6  LEFT JOIN employees m ON e.manager_id = m.emp_id;
7
8  -- Find employees earning more than their manager
9  SELECT e.name AS employee, e.salary,
10         m.name AS manager, m.salary AS mgr_salary
11  FROM employees e
12  JOIN employees m ON e.manager_id = m.emp_id
13  WHERE e.salary > m.salary;
```

■ Tip: Always use meaningful aliases (e for employee, m for manager) to keep self-joins readable.

Q19

What is the CASE statement in SQL?

● Easy | Conditional Logic

■ Answer

CASE is SQL's conditional expression — similar to if-else in programming. It comes in two forms: Simple CASE (compares one expression to multiple values) and Searched CASE (evaluates multiple boolean conditions). It can be used in SELECT, WHERE, ORDER BY, and GROUP BY.

■ SQL Example

```
SQL
1  -- Searched CASE: categorize salaries
2  SELECT
3      name, salary,
4      CASE
5          WHEN salary >= 100000 THEN 'Senior'
6          WHEN salary >= 60000  THEN 'Mid-level'
7          WHEN salary >= 30000  THEN 'Junior'
8          ELSE                    'Intern'
9      END AS level
10 FROM employees;
11
12 -- Simple CASE: map department codes
13 SELECT name,
14        CASE dept
15            WHEN 'ENG' THEN 'Engineering'
16            WHEN 'MKT' THEN 'Marketing'
17            ELSE        'Other'
18        END AS department_full
19 FROM employees;
```

■ Tip: CASE can be used inside aggregate functions: `SUM(CASE WHEN dept='ENG' THEN salary END)` for conditional aggregation.

Q20

What is the difference between INNER JOIN and WHERE with multiple tables?

● Easy | Joins

■ Answer

Both can produce the same result for INNER JOINS, but explicit JOIN syntax is preferred. Implicit JOIN (commas in FROM with WHERE conditions) is harder to read, easier to forget conditions (causing accidental Cartesian products), and can't express OUTER JOINS. Modern SQL always uses explicit JOIN syntax.

■ SQL Example

```
SQL

1  -- Implicit JOIN (old style – avoid)
2  SELECT e.name, d.dept_name
3  FROM employees e, departments d
4  WHERE e.dept_id = d.id;
5
6  -- Explicit INNER JOIN (preferred)
7  SELECT e.name, d.dept_name
8  FROM   employees e
9  INNER JOIN departments d ON e.dept_id = d.id;
10
11 -- Accidental Cartesian product if you forget WHERE:
12 -- SELECT * FROM employees e, departments d;
13 -- Returns e_rows × d_rows rows!
```

■ *Tip: Always use explicit JOIN syntax in production and interviews — it demonstrates awareness of best practices.*

Q21

What is a Recursive CTE and how do you query hierarchical data?

▲ Hard | Advanced SQL

■ Answer

A Recursive CTE calls itself repeatedly until a termination condition is met. It has two parts: the anchor member (base case) and the recursive member (the part that references the CTE itself). It's ideal for tree structures like org charts, file systems, or category hierarchies.

■ SQL Example

```
SQL

1  WITH RECURSIVE org_tree AS (
2      -- Anchor: start with top-level managers
3      SELECT emp_id, name, manager_id, 1 AS level
4      FROM   employees
5      WHERE  manager_id IS NULL
6
7      UNION ALL
8
9      -- Recursive: find direct reports
10     SELECT e.emp_id, e.name, e.manager_id, ot.level + 1
11     FROM   employees e
12     INNER JOIN org_tree ot ON e.manager_id = ot.emp_id
13 )
14 SELECT level, name FROM org_tree ORDER BY level, name;
```

■ Tip: Always include a termination condition to prevent infinite recursion. Most DBs also have MAXRECURSION limits.

Q22

Explain LEAD() and LAG() window functions.

▲ Hard | Window Functions

■ Answer

LAG() accesses the value of a previous row in the result set without a self-join. LEAD() accesses a future row's value. Both take a column name and optional offset (default 1) and a default value for when there is no previous/next row. They're perfect for period-over-period comparisons.

■ SQL Example

```
1 SELECT
2     month,
3     revenue,
4     LAG(revenue, 1, 0) OVER (ORDER BY month) AS prev_month,
5     LEAD(revenue, 1, 0) OVER (ORDER BY month) AS next_month,
6     revenue - LAG(revenue) OVER (ORDER BY month) AS mom_change,
7     ROUND(
8         100.0 * (revenue - LAG(revenue) OVER (ORDER BY month))
9         / NULLIF(LAG(revenue) OVER (ORDER BY month), 0)
10    , 2) AS pct_change
11 FROM monthly_sales
12 ORDER BY month;
```

■ Tip: LAG/LEAD is one of the most common interview questions for analytics/data engineering roles.

Q23

What are transaction isolation levels?

▲ Hard | Transactions

■ Answer

Isolation levels control how transactions interact concurrently. READ UNCOMMITTED allows dirty reads (seeing uncommitted data). READ COMMITTED (default in most DBs) prevents dirty reads. REPEATABLE READ prevents dirty and non-repeatable reads. SERIALIZABLE (strictest) prevents all anomalies including phantom reads but is slowest.

■ SQL Example

```
SQL
1  -- Set isolation level (PostgreSQL / MySQL syntax)
2  SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
3  BEGIN;
4
5  -- Phantom read example: SERIALIZABLE prevents this
6  -- T1: SELECT COUNT(*) FROM orders WHERE total > 100
7  -- T2: INSERT INTO orders VALUES(...) -- total = 150
8  -- T1 (again): SELECT COUNT(*) -- gets different result!
9
10 SELECT name, salary FROM employees WHERE dept = 'HR';
11 COMMIT;
```

■ Tip: Higher isolation = more consistency but lower concurrency. Most apps use READ COMMITTED or REPEATABLE READ.

Q24

How do you find duplicate records in a table?

◆ Medium | Data Quality

■ Answer

Finding duplicates is a very common SQL task. Use GROUP BY on the columns that define uniqueness, then filter with HAVING COUNT(*) > 1. To see the full duplicate rows, join this result back to the original table or use a window function like ROW_NUMBER().

■ SQL Example

```
1  -- Method 1: GROUP BY + HAVING
2  SELECT email, COUNT(*) AS occurrences
3  FROM employees
4  GROUP BY email
5  HAVING COUNT(*) > 1;
6
7  -- Method 2: ROW_NUMBER (keeps full row details)
8  SELECT * FROM (
9      SELECT *,
10         ROW_NUMBER() OVER (
11             PARTITION BY email
12             ORDER BY created_at DESC
13         ) AS rn
14     FROM employees
15 ) t WHERE rn > 1; -- returns only duplicates
16
17 -- Method 3: Delete duplicates, keep latest
18 DELETE FROM employees WHERE emp_id NOT IN (
19     SELECT MIN(emp_id) FROM employees GROUP BY email
20 );
```

■ Tip: Always verify your duplicate-finding logic on a small sample before running DELETE on production data!

Q25

What is the difference between a Clustered and Non-Clustered Index?

▲ Hard | Performance

■ Answer

A Clustered Index determines the physical sort order of data in the table — there can be only ONE per table (usually the primary key). A Non-Clustered Index is a separate structure that points to the actual data rows — a table can have many. Clustered indexes are faster for range queries; non-clustered require an extra lookup.

■ SQL Example

```
SQL
1  -- Primary key automatically creates clustered index
2  CREATE TABLE orders (
3      order_id INT PRIMARY KEY, -- clustered index
4      customer_id INT,
5      order_date DATE,
6      total DECIMAL(10,2)
7  );
8
9  -- Non-clustered indexes on frequently queried columns
10 CREATE INDEX idx_customer ON orders(customer_id);
11 CREATE INDEX idx_date ON orders(order_date);
12
13 -- Covering index: include extra columns to avoid lookup
14 CREATE INDEX idx_cover ON orders(customer_id)
15     INCLUDE (order_date, total);
```

■ Tip: A 'covering index' includes all columns a query needs — the engine never has to touch the main table, making it very fast.

Q26

How do you pivot data in SQL (rows to columns)?

▲ Hard | Data Transformation

■ Answer

Pivoting transforms row-based data into a column-per-category format. In standard SQL, use conditional aggregation with CASE + GROUP BY. SQL Server supports a native PIVOT operator. This technique is common in reporting queries.

■ SQL Example

```
1  -- Standard SQL pivot using conditional aggregation
2  SELECT
3      dept,
4      SUM(CASE WHEN YEAR(hire_date) = 2022 THEN 1 ELSE 0 END) AS hired_2022,
5      SUM(CASE WHEN YEAR(hire_date) = 2023 THEN 1 ELSE 0 END) AS hired_2023,
6      SUM(CASE WHEN YEAR(hire_date) = 2024 THEN 1 ELSE 0 END) AS hired_2024
7  FROM employees
8  GROUP BY dept;
9
10 -- SQL Server native PIVOT syntax
11 SELECT dept, [2022], [2023], [2024]
12 FROM employees
13 PIVOT (COUNT(emp_id) FOR YEAR(hire_date)
14       IN ([2022],[2023],[2024])) AS pvt;
```

■ Tip: For dynamic column names, you'll need dynamic SQL. In practice, conditional aggregation is more portable across databases.

Q27

What is the N+1 query problem and how do you solve it?

▲ Hard | Performance

■ Answer

The N+1 problem occurs when you run 1 query to get N records, then N additional queries to fetch related data for each record (1+N queries total). This is extremely inefficient. The solution is to use JOINS or batch loading to retrieve all needed data in one or two queries.

■ SQL Example

```
SQL

1  -- N+1 Problem (inefficient — avoid)
2  -- Query 1: SELECT * FROM departments; -- returns 10 depts
3  -- Then for each dept (10 more queries):
4  -- SELECT * FROM employees WHERE dept_id = ?;
5
6  -- Solution: One JOIN fetches everything at once
7  SELECT
8      d.dept_name,
9      e.name AS employee,
10     e.salary
11  FROM departments d
12  LEFT JOIN employees e ON e.dept_id = d.dept_id
13  ORDER BY d.dept_name, e.name;
```

■ Tip: This is a classic ORM anti-pattern — ORMs like Hibernate or SQLAlchemy can suffer from this without eager loading.

Q28

What is COALESCE and how is it different from ISNULL/IFNULL?

● Easy | NULL Handling

■ Answer

COALESCE is standard SQL and accepts any number of arguments, returning the first non-NULL value. ISNULL (SQL Server) and IFNULL (MySQL) accept only two arguments and are database-specific. COALESCE is preferred for portability and its ability to chain multiple fallback values.

■ SQL Example

```
SQL
1  -- COALESCE: first non-NULL among many options
2  SELECT
3      name,
4      COALESCE(mobile, home_phone, work_phone, 'No phone') AS contact
5  FROM customers;
6
7  -- Use in calculations to avoid NULL propagation
8  SELECT
9      name,
10     COALESCE(bonus, 0) + COALESCE(commission, 0) AS total_extra
11  FROM employees;
12
13  -- NULLIF: returns NULL if values are equal (avoids div/0)
14  SELECT total / NULLIF(units, 0) AS unit_price FROM sales;
```

■ Tip: COALESCE short-circuits — it stops evaluating once it finds a non-NULL value, making it efficient.

Q29

Explain EXPLAIN / Query Execution Plan.

▲ Hard | Performance

■ Answer

EXPLAIN (or EXPLAIN ANALYZE in PostgreSQL) shows how the database engine will execute a query — which indexes it uses, join methods, estimated row counts, and costs. It's the primary tool for diagnosing and optimizing slow queries. Key things to watch for: full table scans, high cost nodes, and poor row estimates.

■ SQL Example

```
SQL
1  -- PostgreSQL: EXPLAIN ANALYZE runs the query too
2  EXPLAIN ANALYZE
3  SELECT e.name, d.dept_name
4  FROM employees e
5  JOIN departments d ON e.dept_id = d.dept_id
6  WHERE e.salary > 80000;
7
8  -- MySQL EXPLAIN
9  EXPLAIN
10 SELECT * FROM employees
11 WHERE dept = 'Engineering' AND salary > 70000;
12 -- Look for: type=ALL (bad, full scan)
13 -- Look for: type=ref or range (good, uses index)
```

■ Tip: In EXPLAIN output, 'Seq Scan' = full table scan. Adding an index should change it to 'Index Scan' or 'Index Only Scan'.

Q30

What is a Trigger and when would you use one?

◆ Medium | Programmability

■ Answer

A Trigger is a stored procedure that automatically executes in response to INSERT, UPDATE, or DELETE events on a table. Triggers are used for audit logging, enforcing complex business rules, maintaining derived data, and preventing invalid operations. They can be BEFORE or AFTER the triggering event.

■ SQL Example

```
SQL

1  -- Audit trigger: log every salary change
2  CREATE TABLE salary_audit (
3      audit_id    SERIAL PRIMARY KEY,
4      emp_id      INT,
5      old_salary  DECIMAL,
6      new_salary  DECIMAL,
7      changed_at  TIMESTAMP DEFAULT NOW()
8  );
9
10 CREATE TRIGGER trg_salary_audit
11 AFTER UPDATE OF salary ON employees
12 FOR EACH ROW
13 BEGIN
14     INSERT INTO salary_audit(emp_id, old_salary, new_salary)
15     VALUES (OLD.emp_id, OLD.salary, NEW.salary);
16 END;
```

■ *Tip: Use triggers sparingly — they execute invisibly and can make debugging difficult. Prefer application-level logic when possible.*

Q31

How do you calculate a running total in SQL?

◆ Medium | Window Functions

■ Answer

Use a window function with SUM() and a cumulative frame. The ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW clause tells SQL to sum from the first row up to the current row. This is much cleaner than self-joins, which were used before window functions existed.

■ SQL Example

```
1 SELECT
2     order_date,
3     daily_revenue,
4     SUM(daily_revenue) OVER (
5         ORDER BY order_date
6         ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
7     ) AS running_total,
8     AVG(daily_revenue) OVER (
9         ORDER BY order_date
10        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
11    ) AS rolling_7day_avg
12 FROM daily_sales
13 ORDER BY order_date;
```

■ *Tip: RANGE vs ROWS: RANGE includes all tied values in the window, ROWS is strictly positional. ROWS is usually what you want.*

Q32

What is a Materialized View?

▲ Hard | Views

■ Answer

A Materialized View is like a regular view but physically stores the result of the query on disk. It must be refreshed to reflect underlying data changes. It dramatically improves read performance for complex, expensive queries at the cost of storage and potential staleness. Supported in PostgreSQL, Oracle, and SQL Server (as indexed views).

■ SQL Example

```
1  -- PostgreSQL Materialized View
2  CREATE MATERIALIZED VIEW mv_dept_stats AS
3  SELECT
4      dept,
5      COUNT(*) AS headcount,
6      AVG(salary) AS avg_salary,
7      SUM(salary) AS payroll
8  FROM employees
9  GROUP BY dept;
10
11 -- Refresh (non-concurrent: locks the view)
12 REFRESH MATERIALIZED VIEW mv_dept_stats;
13
14 -- Concurrent refresh (requires unique index)
15 CREATE UNIQUE INDEX ON mv_dept_stats(dept);
16 REFRESH MATERIALIZED VIEW CONCURRENTLY mv_dept_stats;
```

■ Tip: Materialized views are ideal for dashboards and reports that run frequently on rarely-changing aggregated data.

Q33

Write a query to find the second highest salary.

◆ Medium | Classic Problems

■ Answer

This is one of the most classic SQL interview questions. There are several approaches: using OFFSET, using MAX with a subquery excluding the max, using DENSE_RANK(), or using LIMIT. The DENSE_RANK approach is most robust and handles ties correctly.

■ SQL Example

```
SQL

1  -- Method 1: DENSE_RANK (most robust – handles ties)
2  SELECT salary FROM (
3      SELECT salary, DENSE_RANK() OVER
4          (ORDER BY salary DESC) AS dr
5      FROM employees
6  ) ranked WHERE dr = 2;
7
8  -- Method 2: Subquery with MAX
9  SELECT MAX(salary) AS second_highest
10 FROM employees
11 WHERE salary < (SELECT MAX(salary) FROM employees);
12
13 -- Method 3: LIMIT / OFFSET
14 SELECT DISTINCT salary
15 FROM employees
16 ORDER BY salary DESC
17 LIMIT 1 OFFSET 1;
```

■ Tip: For 'Nth highest salary', parameterize DENSE_RANK: change WHERE dr = 2 to WHERE dr = N.

Q34

What are NATURAL JOIN and CROSS JOIN?

◆ Medium | Joins

■ Answer

CROSS JOIN produces the Cartesian product of two tables — every combination of rows ($M \times N$ rows). It's used for generating combinations or test data. NATURAL JOIN automatically joins on all columns with the same name and compatible types — convenient but dangerous if column names change unexpectedly.

■ SQL Example

```
SQL
1  -- CROSS JOIN: all combinations
2  SELECT s.size, c.color
3  FROM shirt_sizes s
4  CROSS JOIN shirt_colors c;
5  -- Returns: S-Red, S-Blue, M-Red, M-Blue, L-Red, L-Blue
6
7  -- NATURAL JOIN: auto-match same-named columns
8  SELECT e.name, d.dept_name
9  FROM employees e
10 NATURAL JOIN departments d;
11 -- Joins on any columns with the same name (e.g., dept_id)
12
13 -- Safer: always use explicit ON clause instead
14 SELECT e.name, d.dept_name
15 FROM employees e
16 JOIN departments d ON e.dept_id = d.dept_id;
```

■ *Tip: Avoid NATURAL JOIN in production — it's fragile. Adding a new column with the same name in both tables can silently break queries.*

Q35

Explain the SQL ORDER OF EXECUTION.

◆ Medium | Fundamentals

■ Answer

SQL clauses are written in a different order than they execute. The logical execution order is: FROM → JOIN → WHERE → GROUP BY → HAVING → SELECT → DISTINCT → ORDER BY → LIMIT. This is why you cannot reference a SELECT alias in WHERE (WHERE runs before SELECT), but you can reference it in ORDER BY.

■ SQL Example

```
1  -- This query illustrates execution order:
2  -- 1. FROM employees           (identify the table)
3  -- 2. JOIN departments ON dept_id (combine tables)
4  -- 3. WHERE e.salary > 40000    (filter rows)
5  -- 4. GROUP BY d.dept_name      (group result)
6  -- 5. HAVING COUNT(*) > 2       (filter groups)
7  -- 6. SELECT dept_name, COUNT(*) (compute output)
8  -- 7. ORDER BY COUNT(*) DESC    (sort)
9  -- 8. LIMIT 5                  (restrict output)
10
11 SELECT d.dept_name, COUNT(*) AS cnt
12 FROM employees e
13 JOIN departments d ON e.dept_id = d.dept_id
14 WHERE e.salary > 40000
15 GROUP BY d.dept_name
16 HAVING COUNT(*) > 2
17 ORDER BY cnt DESC
18 LIMIT 5;
```

■ *Tip: Knowing execution order explains why SELECT aliases can't be used in WHERE — WHERE executes BEFORE SELECT!*