02 SQL Interview Questions & Answers

SQL Interview Questions & Answers

Tags: #SQL #InterviewPrep #Database #TechnicalInterview

Last Updated: August 23, 2025

Author: Nikhil Sharma

Fundamentals & Basic Queries

Q1: What's the difference between the `WHERE` and `HAVING` clauses?

Answer

The primary difference relates to **when** they filter data in a query's order of execution.

Feature	`WHERE` Clause	`HAVING` Clause
Purpose	Filters individual rows **before** any grouping or aggregation occurs.	Filters groups of rows **after** aggregation has been performed.
Works With	Individual row data.	Aggregate functions (`COUNT()` , `SUM()` , `AVG()` , etc.).
Placement	Comes before `GROUP BY` .	Comes after `GROUP BY` .

Mnemonic: You can't `HAVING` cake (`aggregate`) until you `GROUP` the ingredients.

```sql
-- Select departments where the total salary is over 20,000,
-- but only include employees with salaries over 8,000 in the calculation.
SELECT
    department_id,
    SUM(salary) AS total_salary
FROM
    emp
WHERE
    salary > 8000  -- Filters rows BEFORE grouping
GROUP BY
    department_id
HAVING
    SUM(salary) > 20000; -- Filters groups AFTER grouping
```

Q2: What's the difference between `DELETE` , `TRUNCATE` , and `DROP` ?

Answer

These commands are used to remove data or objects, but they operate very differently.

Feature	`DELETE`	`TRUNCATE`	`DROP`
Command Type	DML (Data Manipulation)	DDL (Data Definition)	DDL (Data Definition)
Action	Removes rows from a table one by one.	Deallocates all pages in a table to remove all rows instantly.	Removes the entire table object, including its structure, indexes, and constraints.
`WHERE` **Clause**	Can be used to remove specific rows.	Cannot be used. Removes all rows.	Not applicable. Removes the whole table.
Transaction Log	Logs each row deletion. Can be slow for large tables.	Logs the page deallocation. Very fast.	Logs the removal of the object. Very fast.
Rollback	Can be rolled back.	Generally cannot be rolled back (some DBMS might allow it).	Cannot be rolled back.
Triggers	Fires `ON DELETE` triggers for each row.	Does not fire triggers.	Does not fire triggers.

Important

- Use `DELETE` when you need to remove specific rows.
- Use `TRUNCATE` when you need to quickly empty a table completely.
- Use `DROP` only when you want to permanently remove the table and its structure.

Q3: What's the difference between `UNION` and `UNION ALL`?

Answer

Both operators combine the result sets of two or more `SELECT` statements. The key difference is how they handle duplicate rows.

- `UNION` : Scans the combined result set and **removes duplicate rows**. This operation can be slower because of the overhead of sorting and comparing rows to identify duplicates.
- `UNION ALL` : Includes **all rows** from all queries, including any duplicates. It is significantly faster because it doesn't perform the extra work of removing duplicates.

Tip

Always use `UNION ALL` unless you have a specific reason to remove duplicate rows. It offers better performance.

Q4: How do you handle `NULL` values in aggregate functions?

Answer

Most aggregate functions **ignore `NULL` values** in their calculations. This is a critical concept to remember.

- **`COUNT(column_name)`** : Counts the number of non- `NULL` values in the specified column.
- **`COUNT(*)`** : Counts the total number of rows, including those with `NULL` values (as it counts the row itself, not a specific column's value).
- **`SUM(column_name)`** : Sums all non- `NULL` values. `NULL` s are ignored.
- **`AVG(column_name)`** : Calculates the average by summing non- `NULL` values and dividing by the count of non- `NULL` values (`SUM(col) / COUNT(col)`).

To treat `NULL` s as a specific value (like 0) in a calculation, you must use a function like `COALESCE()` or `ISNULL()` .

```
-- Suppose a 'commission' column has NULLs. To include them as 0 in the average:
SELECT AVG(COALESCE(commission, 0)) FROM sales;
```

Q5: How do you detect and delete duplicate rows?

Answer

This is a two-step process: first find the duplicates, then delete them. The best way uses a Common Table Expression (CTE) and the `ROW_NUMBER()` window function.

Step 1: Detect Duplicates
Identify rows that have the same values across a set of columns that should be unique.

```
-- Detect employees with the same name and department
SELECT
    emp_name,
    department_id,
    COUNT(*)
FROM
    emp
GROUP BY
    emp_name, department_id
HAVING
    COUNT(*) > 1;
```

Step 2: Delete Duplicates
Use a CTE to assign a unique row number to each duplicate row, then delete any row where the number is greater than 1.

```
-- Delete duplicate employees, keeping only one instance
WITH DuplicatesCTE AS (
    SELECT *,
```

```
        ROW_NUMBER() OVER(
            PARTITION BY emp_name, department_id -- Columns that define a
duplicate
            ORDER BY emp_id -- Arbitrary order to pick which one to keep
        ) AS RowNum
    FROM
        emp
)
DELETE FROM DuplicatesCTE WHERE RowNum > 1;
```

Q6: How do you write an `UPDATE` query to swap values in a column?

Answer

The most robust way to swap values is using a `CASE` statement. This ensures the operation is atomic and avoids intermediate states where all values might temporarily become the same.

```
-- Swap 'Male' and 'Female' values in a customer_gender column
UPDATE orders
SET customer_gender = CASE
    WHEN customer_gender = 'Male' THEN 'Female'
    WHEN customer_gender = 'Female' THEN 'Male'
    ELSE customer_gender -- Good practice to handle other potential values
END;
```

This works because the `CASE` statement evaluates the original value for each row before any updates are actually committed for that row.

Joins, Subqueries, & Relationships

Q7: What's the difference between `INNER`, `LEFT`, `RIGHT`, and `FULL OUTER` joins?

Answer

Joins are used to combine rows from two or more tables based on a related column.

- **`INNER JOIN`**: Returns only the rows where the join condition is met in **both** tables. It's the intersection of the two tables.
- **`LEFT JOIN`** (or `LEFT OUTER JOIN`): Returns **all** rows from the **left** table, and the matched rows from the right table. If there is no match, the columns from the right table will contain `NULL`.
- **`RIGHT JOIN`** (or `RIGHT OUTER JOIN`): Returns **all** rows from the **right** table, and the matched rows from the left table. If there is no match, the columns from the left table will contain `NULL`.

- `FULL OUTER JOIN` : Returns all rows when there is a match in either the left or the right table. It's the union of both tables; it returns matched rows and also includes `NULL` s for non-matching rows from both sides.

Q8: What's the difference between correlated and uncorrelated subqueries?

Answer

The difference lies in their dependency on the outer query.

Uncorrelated Subquery:

- Executes **once** and is independent of the outer query.
- The result of the subquery is "fed" back to the outer query.
- Think of it as a separate, self-contained query.

```
-- Find employees who work in the 'Analytics' department
SELECT * FROM emp
WHERE department_id IN (SELECT dept_id FROM department WHERE dept_name =
'Analytics');
-- The subquery runs once to get the dept_id for 'Analytics'.
```

Correlated Subquery:

- Depends on the outer query for its values. It is evaluated **once for each row** processed by the outer query.
- Can be inefficient and slow if not used carefully.

```
-- Find employees whose salary is the maximum in their department
SELECT e1.emp_name, e1.salary, e1.department_id
FROM emp e1
WHERE e1.salary = (
    SELECT MAX(e2.salary)
    FROM emp e2
    WHERE e2.department_id = e1.department_id -- Correlation: inner query
depends on outer query's row
);
```

Q9: How do you write a query for employees with salaries greater than their manager's?

Answer

This is a classic **self-join** problem, where you join a table to itself. You treat the table as two separate entities: one for the employees and one for the managers.

```sql
SELECT
    e.emp_name AS employee_name,
    e.salary AS employee_salary,
    m.emp_name AS manager_name,
    m.salary AS manager_salary
FROM
    emp e -- Alias for the employee
INNER JOIN
    emp m ON e.manager_id = m.emp_id -- Alias for the manager
WHERE
    e.salary > m.salary;
```

Explanation:

1. We alias the `emp` table as `e` (for employee) and `m` (for manager).
2. We join them on the condition that the employee's `manager_id` is equal to the manager's `emp_id`.
3. The `WHERE` clause then filters these pairs to find only those where the employee's salary is higher.

Q10: What are `CROSS APPLY` and `OUTER APPLY`?

Answer >

`APPLY` operators, primarily found in SQL Server, are similar to `JOIN`s but allow you to invoke a table-valued function for each row from an outer table.

- **`CROSS APPLY`**: Acts like an `INNER JOIN`. It returns only the rows from the left table where the table-valued function on the right returns a result set. If the function returns an empty set for a given row, that row is excluded.
- **`OUTER APPLY`**: Acts like a `LEFT JOIN`. It returns **all** rows from the left table, regardless of whether the table-valued function on the right returns a result. If the function returns an empty set, the columns from the function's result will be `NULL`.

Use Case: They are essential when you have a function that takes parameters from each row of an outer table. For example, getting the top 3 orders for *each* customer.

Q11: How do you identify and remove orphan records?

Answer >

An orphan record is a row in a child table that references a primary key in a parent table that no longer exists. You can find them using a `LEFT JOIN` or `NOT EXISTS`.

Identify Orphan Records:
Find employees whose `department_id` does not exist in the `department` table.

```sql
-- Using LEFT JOIN
SELECT e.*
FROM emp e
LEFT JOIN department d ON e.department_id = d.dept_id
WHERE d.dept_id IS NULL;

-- Using NOT EXISTS (often more performant)
SELECT e.*
FROM emp e
WHERE NOT EXISTS (
    SELECT 1
    FROM department d
    WHERE d.dept_id = e.department_id
);
```

Remove Orphan Records:
Once identified, you can delete them using the same logic.

```sql
DELETE e
FROM emp e
LEFT JOIN department d ON e.department_id = d.dept_id
WHERE d.dept_id IS NULL;
```

Q12: How do you find records in one table without a corresponding record in another?

Answer

This is the same logic as identifying orphan records. The best methods are `LEFT JOIN` with a `WHERE ... IS NULL` check or using `NOT EXISTS`.

Example: Find departments that have no employees.

```sql
-- Using LEFT JOIN
SELECT d.*
FROM department d
LEFT JOIN emp e ON d.dept_id = e.department_id
WHERE e.emp_id IS NULL;

-- Using NOT EXISTS
SELECT d.*
FROM department d
WHERE NOT EXISTS (
    SELECT 1
    FROM emp e
```

```sql
    WHERE e.department_id = d.dept_id
);
```

Advanced Functions & Window Functions

Q13: What's the difference between `RANK()`, `DENSE_RANK()`, and `ROW_NUMBER()`?

Answer

All are window functions used for ranking, but they handle ties differently.

- `ROW_NUMBER()`: Assigns a unique, sequential number to each row. It never repeats numbers, even for ties.
- `RANK()`: Assigns the same rank to rows with tied values. It then skips the next rank(s). (e.g., 1, 2, 2, 4).
- `DENSE_RANK()`: Assigns the same rank to tied rows but does **not** skip any ranks in the sequence. (e.g., 1, 2, 2, 3).

Example Output (Ranking salaries):

Salary	`ROW_NUMBER()`	`RANK()`	`DENSE_RANK()`
15000	1	1	1
12000	2	2	2
12000	3	2	2
10000	4	4	3
10000	5	4	3
9000	6	6	4

Q14: How do you find the second highest salary without `TOP`, `LIMIT`, or `RANK`?

Answer

You can solve this using a subquery. The logic is to find the maximum salary that is less than the overall maximum salary.

```sql
-- Using a subquery
SELECT MAX(salary)
FROM emp
WHERE salary < (SELECT MAX(salary) FROM emp);
```

A more flexible approach for the Nth salary uses a correlated subquery.

```sql
-- Find the 2nd highest salary using a correlated subquery
SELECT salary
FROM emp e1
WHERE 1 = ( -- N-1, so 2-1=1
    SELECT COUNT(DISTINCT salary)
    FROM emp e2
    WHERE e2.salary > e1.salary
);
```

Q15: How do you find the Nth highest value without window functions?

Answer

This builds on the previous question. The two main approaches are using `OFFSET` (if the DBMS supports it) or a correlated subquery.

Method 1: `OFFSET` (MySQL, PostgreSQL, SQL Server)
This is the most straightforward modern approach. For the 3rd highest salary:

```sql
SELECT salary
FROM emp
ORDER BY salary DESC
LIMIT 1 OFFSET 2; -- OFFSET is N-1 (3-1=2)
```

Method 2: Correlated Subquery (Universal)
This works on almost any SQL database. For the 3rd highest salary (N=3):

```sql
SELECT salary
FROM emp e1
WHERE 2 = ( -- N-1, so 3-1=2
    SELECT COUNT(DISTINCT e2.salary)
    FROM emp e2
    WHERE e2.salary > e1.salary
);
```

Q16: How do you calculate running totals or moving averages?

Answer

This is a perfect use case for **window functions**.

Running Total:
A running total sums the values of a column up to the current row, within a specified order.

```
SELECT
    order_date,
    order_amount,
    SUM(order_amount) OVER (ORDER BY order_date) AS running_total
FROM
    orders;
```

Moving Average:
A moving average calculates the average of a certain number of preceding rows. For a 3-day moving average:

```
SELECT
    order_date,
    order_amount,
    AVG(order_amount) OVER (
        ORDER BY order_date
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS moving_avg_3_days
FROM
    orders;
```

Q17: How do you pivot rows into columns?

Answer

Pivoting transforms data from a row-level format to a columnar format. The standard SQL way is to use `CASE` statements with an aggregate function. Some databases (like SQL Server) have a built-in `PIVOT` operator.

Method: `CASE` with Aggregation (Universal)
Suppose you have sales data and want to see total sales for each year in separate columns.

```
-- Assume a 'sales' table with columns: product, sale_year, amount
SELECT
    product,
    SUM(CASE WHEN sale_year = 2023 THEN amount ELSE 0 END) AS sales_2023,
    SUM(CASE WHEN sale_year = 2024 THEN amount ELSE 0 END) AS sales_2024,
    SUM(CASE WHEN sale_year = 2025 THEN amount ELSE 0 END) AS sales_2025
FROM
    sales
```

```
GROUP BY
    product;
```

Q18: How do you perform recursive queries (hierarchical data)?

Answer

Recursive queries are handled using **Recursive Common Table Expressions (CTEs).** This is ideal for organizational charts, bill of materials, or folder structures.

A recursive CTE has two parts:

1. **Anchor Member:** The initial `SELECT` statement that provides the starting point for the recursion.
2. **Recursive Member:** A `SELECT` statement that references the CTE itself, joined with a `UNION ALL`.

Example: Find the entire management chain for employee 'Agam' (emp_id = 6).

```
WITH EmployeeHierarchy AS (
    -- 1. Anchor Member: Start with the employee
    SELECT emp_id, emp_name, manager_id, 0 AS level
    FROM emp
    WHERE emp_id = 6

    UNION ALL

    -- 2. Recursive Member: Join back to find the manager
    SELECT e.emp_id, e.emp_name, e.manager_id, eh.level + 1
    FROM emp e
    INNER JOIN EmployeeHierarchy eh ON e.emp_id = eh.manager_id
)
SELECT * FROM EmployeeHierarchy;
```

Q19: How do you find gaps and islands in a sequence?

Answer

This is an advanced problem. "Islands" are contiguous ranges of data, and "gaps" are the missing values between them. A common method is to use window functions to identify the start of each island.

Logic:

1. Use `ROW_NUMBER()` to create a sequence.

2. Create another sequence based on the value itself (e.g., date or ID).
3. The difference between these two sequences will be constant for all rows within a single "island".
4. Group by this constant difference to identify the start and end of each island.

Example: Find consecutive login dates for users.

```
-- Table: logins (user_id, login_date)
WITH DateGroups AS (
    SELECT
        user_id,
        login_date,
        DATE_SUB(login_date, INTERVAL ROW_NUMBER() OVER(PARTITION BY user_id
ORDER BY login_date) DAY) AS grp
    FROM
        logins
)
SELECT
    user_id,
    MIN(login_date) AS island_start,
    MAX(login_date) AS island_end,
    COUNT(*) AS consecutive_days
FROM
    DateGroups
GROUP BY
    user_id, grp
ORDER BY
    user_id, island_start;
```

Q20: How do you calculate the median without a built-in function?

Answer

The median is the middle value in a sorted dataset. The logic involves sorting the data and picking the middle row(s). A common way is using window functions `ROW_NUMBER()` and `COUNT()`.

Logic:

1. Count the total number of rows (`total_rows`).
2. Assign a row number to each row in ascending order of the value (`rn_asc`).
3. Assign a row number in descending order of the value (`rn_desc`).
4. The median is the row where `rn_asc` is close to `rn_desc`. Specifically, where `rn_asc` is `(total_rows+1)/2` or `(total_rows+2)/2`.

```
WITH OrderedSalaries AS (
    SELECT
```

```
        salary,
        ROW_NUMBER() OVER(ORDER BY salary ASC) as rn,
        COUNT(*) OVER() as total_rows
    FROM
        emp
)
SELECT AVG(salary)
FROM OrderedSalaries
WHERE rn IN (FLOOR((total_rows + 1) / 2.0), CEILING((total_rows + 1) / 2.0));
```

This handles both odd and even numbers of rows correctly by averaging the middle one or two values.

Performance & Data Warehousing

Q21: What's the difference between a clustered and non-clustered index?

Answer

Indexes are used to speed up data retrieval. The main difference is how they store data.

Clustered Index:

- **Physically orders** the data rows in the table based on the indexed column(s).
- Think of it like a dictionary or a phone book, where the words/names are already sorted alphabetically.
- A table can have **only one** clustered index.
- The leaf nodes of a clustered index contain the actual data pages.

Non-Clustered Index:

- Creates a separate structure that contains the indexed column(s) and a **pointer** (row locator) to the actual data row.
- The data rows in the table remain in whatever order they were inserted (a "heap") or are sorted by the clustered index.
- Think of it like the index at the back of a book; it points you to the page where the information is located.
- A table can have **multiple** non-clustered indexes.

Q22: How do you optimize queries with large datasets?

Answer

This is a broad topic, but key strategies include:

1. **Proper Indexing:** This is the most important factor. Ensure indexes exist on columns used in `WHERE` clauses, `JOIN` conditions, and `ORDER BY` clauses. Use composite indexes for multi-

column filters.
2. **Analyze the Execution Plan:** Use `EXPLAIN PLAN` (or similar tools) to see how the database is executing your query. Look for full table scans on large tables, as they are a major performance killer.
3. **Write Efficient SQL (`SARGable` queries):** Ensure your `WHERE` clauses are "Search Argument-able". Avoid applying functions to the column side of the condition (e.g., `WHERE YEAR(order_date) = 2025` is bad; `WHERE order_date >= '2025-01-01' AND order_date < '2026-01-01'` is good because it can use an index on `order_date`).
4. **Select Only Necessary Columns:** Avoid `SELECT *`. Retrieving unnecessary data increases I/O and network traffic.
5. **Minimize Large Joins:** Join only the tables you need. Use appropriate join types and ensure join columns are indexed.
6. **Use `UNION ALL` Instead of `UNION`:** If you don't need to remove duplicates, `UNION ALL` is much faster.

Q23: How do you implement Slowly Changing Dimensions (SCD) Type 2?

Answer

SCD Type 2 is a data warehousing technique used to track historical data by creating new records for changes in dimensional data, rather than overwriting them. This preserves the full history.

Implementation:
You add specific columns to your dimension table to manage the history:
- `StartDate` or `EffectiveDate`: The date from which the record is valid.
- `EndDate`: The date until which the record was valid. The current record often has a future date (e.g., '9999-12-31').
- `IsCurrent` Flag: A boolean or character flag (`Y`/`N`, `1`/`0`) to easily identify the currently active record.

Example `DimEmployee` Table:

EmployeeKey	EmpID	EmpName	Department	StartDate	EndDate	IsCurrent
101	1	Ankit	Analytics	2022-01-15	2024-06-30	N
102	1	Ankit	IT	2024-07-01	9999-12-31	Y

Update Process:
When an employee's department changes:
1. The `EndDate` of the current record (`IsCurrent = Y`) is updated to the day before the change. The `IsCurrent` flag is set to `N`.
2. A **new record** is inserted with the updated information, a new `StartDate` (the date of the change), a far-future `EndDate`, and `IsCurrent = Y`.

This allows you to join fact tables (like sales) to the employee dimension and accurately report on what the employee's department was *at the time of the sale*.