Data Engineering Interview: Questions & Answers.

Data Engineering roles are in high demand globally, with hundreds of thousands of open positions. Employers consistently seek a mix of strong SQL and programming skills, big data pipeline experience, and the ability to design scalable systems. For example, an analysis of 2025 job postings found SQL mentioned in 61% of listings, Python in 56%, and cloud data warehouse or lake technologies in 78%. Core responsibilities include building reliable **ETL/ELT pipelines**, optimizing data storage (e.g. in **warehouses/lakehouses**), ensuring data quality, and collaborating across teams to deliver data for analytics and machine learning.

SQL & Query Optimization

Focus: Testing your ability to write correct and efficient SQL. Questions span from SQL basics (joins, aggregation, subqueries) to performance tuning (indexes, query plans, partitioning). Interviewers want to see strong SQL fundamentals and an understanding of how to optimize queries for large datasets.

Why It Matters: SQL remains the **#1 skill for data engineering** roles. Data engineers routinely write SQL for data warehousing, analytics, and ETL jobs. Efficient SQL ensures pipelines and dashboards run fast and at low cost.

1. **(Easy)** What are the different types of SQL JOINs, and when would you use each?

Answer:

- **INNER JOIN:** Returns only matching rows between tables (e.g. customers with orders). Use when you need records present in both datasets.
- **LEFT JOIN (or LEFT OUTER):** Returns all rows from the left table and matching ones from the right table. Use when you need all records from primary table even if no match (e.g. all customers, with orders if any).
- **RIGHT JOIN:** Opposite of left join; includes all rows from right table. Rarely used (you can usually swap table order and use left join).
- **FULL JOIN (FULL OUTER):** Returns all rows when there is a match in one of the tables. Use when you need complete set of records from both sides (e.g. combine two lists of users, including those unique to each list).
- **CROSS JOIN:** Cartesian product of two tables (every combination). Use sparingly (e.g. to generate all combinations, or if intentionally duplicating every row) due to potentially large output.

What interviewers look for: Understanding of join logic and results. The ability to choose the correct join type for a given scenario (ensuring no unintended data loss or duplicates). Clarity on inner vs outer join differences.

Common pitfalls:

- Confusing **INNER vs OUTER** join results (e.g. thinking a left join will drop non-matches from left table – it does not).
- Using a **FULL JOIN** when an inner or left join is intended (which can introduce NULLs and extra rows unexpectedly).
- Forgetting join condition(s) leading to an accidental **CROSS JOIN** (huge result set).

Follow-up: How would you handle a situation where you need to join tables but also include non-matching records with default values? (Hint: use outer joins and possibly COALESCE for default values).

---

2. **(Easy)** What is the difference between **WHERE** and **HAVING** clauses in SQL?

Answer:

- The **WHERE** clause filters rows **before** grouping/aggregation (i.e. it acts on individual rows from the source tables). It cannot use aggregate functions directly.
- The **HAVING** clause filters groups **after** aggregation (it acts on results of GROUP BY). HAVING can filter based on aggregate calculations (e.g. HAVING COUNT(*) > 1 to keep

groups with more than one row).
- Use **WHERE** for conditions on raw data columns, and **HAVING** for conditions on aggregated values.
- Example: To find departments with more than 5 employees, you would GROUP BY department and then HAVING COUNT(employee_id) > 5 (you can't put that condition in WHERE because count is an aggregate calculated after grouping).

What interviewers look for: Knowledge of query execution order (WHERE -> GROUP BY -> HAVING -> SELECT, etc.). Understanding that HAVING is essentially a post-aggregation filter.

Common pitfalls:

- Trying to use **HAVING without GROUP BY** (works in some databases as an implicit grouping, but not standard; or using it to filter non-aggregated columns).
- Misusing **WHERE** to filter on an aggregate expression (e.g. WHERE COUNT(*) > 5 is invalid; it must be in HAVING).
- Not realizing that **HAVING** is often more expensive (since it filters after aggregation over potentially large intermediate results).

Follow-up: If an interviewer asks, *"Can we use HAVING without GROUP BY?"* – The answer: in standard SQL, HAVING implies grouping (some DBMS allow it to act like a WHERE if no GROUP BY, but it's non-standard). Always better to use WHERE when no grouping is needed.

---

3. **(Easy)** Explain the difference between **UNION** and **UNION ALL** in SQL.

Answer:

- **UNION** combines the result sets of two queries and removes any duplicate rows in the combined result. It performs an implicit **DISTINCT** on the results, which involves sorting or hashing – so it's slightly slower.
- **UNION ALL** simply appends the result sets one after another **without removing duplicates**. It is faster since it doesn't require the duplicate check.
- Use **UNION** when you need a **set** of distinct results from two queries (e.g. merging two lists of IDs where duplicates should appear only once). Use **UNION ALL** when you want to keep all occurrences (e.g. truly concatenating datasets, or when you know duplicates can't occur or don't matter).
- Example: Combining two tables of transactions from different years. If there is no overlap in data, UNION ALL is appropriate (faster, preserves all rows). If overlapping data is possible and you want unique records, use UNION.

What interviewers look for: Awareness of performance implications of de-duplication. Understanding of use cases for each (distinct vs all results).

Common pitfalls:

- Using **UNION** unnecessarily, thus incurring overhead – e.g. when you know data sets are disjoint or don't mind duplicates, but you still use UNION (wastes time by sorting to remove nonexistent dups).
- Expecting **UNION ALL** to remove duplicates (it doesn't – leading to double-counting data in reports if used incorrectly).
- Not realizing that both UNION and UNION ALL also eliminate **NULL vs NULL** duplicates the same way (NULL is treated as a value in comparing duplicates).

Follow-up: How might the performance of UNION vs UNION ALL differ on a large dataset? (Expected answer: UNION ALL is faster due to no duplicate elimination; UNION might involve an extra sort or hash step to remove duplicates).

---

4. **(Medium)** How would you **find the second-highest salary** from an Employee table using SQL?

Answer:

- **Solution 1: Subquery approach:** Select the maximum salary that is less than the absolute maximum. For example:
- sql
- CopyEdit

```
SELECT MAX(salary)
FROM Employee
WHERE salary < (SELECT MAX(salary) FROM Employee);
```

This finds the largest salary that is lower than the top salary.

Solution 2: ORDER BY with OFFSET/LIMIT: If the SQL dialect supports it, e.g.:

```
SELECT salary
FROM Employee
ORDER BY salary DESC
LIMIT 1 OFFSET 1;
```

This orders salaries descending and skips the top one, returning the second.

Solution 3: Using ROW_NUMBER() window function:

```
SELECT salary FROM (
  SELECT salary, ROW_NUMBER() OVER (ORDER BY salary DESC) as rn
  FROM Employee
) sub
WHERE rn = 2;
```

- This assigns rank 1 to highest salary, 2 to second highest, etc., then filters.
- All assume you want the second distinct highest. If duplicates (ties) are present and you want the second *distinct* value, using DENSE_RANK() instead of ROW_NUMBER would handle ties appropriately (or use DISTINCT in subquery approach).

What interviewers look for: Correctness and handling of edge cases (like all salaries equal, or whether to consider ties as one rank or skip them). Also they check if you know modern SQL features (window functions) versus older methods.

Common pitfalls:

- If the highest salary occurs multiple times, some solutions might pick the same highest again as "second". For example, the subquery method works for distinct second highest, but if asked for "second highest employee salary" you must clarify if that means distinct or second row after ordering.
- Using **TOP 1 twice** (in SQL Server) incorrectly or other non-scalable methods (like selecting all salaries into app code).
- Not handling the case where the table might have only one salary value (should ideally return NULL or no result for second highest).

Follow-up: How would your query change to get the **third** highest salary or the **Nth** highest? (They expect mention of adjusting the subquery or using window functions with ROW_NUMBER() or DENSE_RANK() to generalize).

---

5. **(Medium)** What is an **execution plan** in SQL and how do you use it for query optimization?

Answer:

- An **execution plan** is a detailed breakdown of how the database will execute a SQL query – the sequence of operations (scans, joins, sorts, etc.) and their cost estimates. It's generated by the query optimizer.
- Viewing the plan (using commands like EXPLAIN or visual tools) shows whether indexes are used or if full table scans occur, join algorithms chosen (nested loop, hash join, etc.), and the estimated rows processed at each step.
- For optimization, you look for **bottlenecks** in the plan: e.g. a step with a high cost like a **full table scan** on a large table or a **sort** of millions of rows. Then you consider adding an index, rewriting the query, or updating statistics so the optimizer can choose a better plan.
- Example: If EXPLAIN shows a query doing a **sequential scan** on a 10M-row table for a certain WHERE filter, adding an index on that filtered column could switch the plan to an **index scan**, drastically improving speed.
- Execution plans also help decide join order or join type issues – e.g. if a join is causing a large data shuffle, maybe you can restructure the query or add an index on the join keys.

What interviewers look for: Comfort with reading and interpreting execution plans. Awareness that query tuning is often about guiding the optimizer (through indexes, rewriting queries, etc.) to a more efficient plan.

Common pitfalls:

- Ignoring the execution plan entirely and guessing at optimizations – good candidates rely on the plan to guide changes.
- Over-indexing or applying hints prematurely without evidence from the plan.
- Not updating or analyzing statistics – sometimes the plan is suboptimal due to outdated stats (this could be a deeper follow-up).

Follow-up: Can you give an example of a scenario where the execution plan led you to a specific optimization? (e.g. "I saw a hash join with big data shuffle, so I introduced a partitioning or used a different join key distribution to reduce data movement").

---

6. **(Medium)** Your SQL query is performing slowly. What steps would you take to **improve its performance**?

Answer:

- **Check the execution plan** first: identify bottlenecks like full table scans, missing indexes, or large sorts/temp table usage. This tells you where to focus.
- **Index tuning:** Ensure appropriate indexes on filtering/join columns. If the query is scanning too many rows, a well-chosen index (or composite index) can speed up lookups. But avoid over-indexing tables with heavy inserts/updates (trade-off write cost).
- **Query rewrite:** Simplify complex subqueries or OR conditions. For example, replace subqueries with JOINs or use EXISTS instead of IN for better performance in some case. Remove SELECT * and only select needed columns (reduces I/O).

- **Adjust joins/aggregations:** Ensure joins are on indexed columns and of appropriate type (e.g., if one table is small, maybe driving the join differently). Use **JOINs instead of subselects** when possible for clarity and possibly performance.
- **Use partitioning** if dealing with huge tables: Partition large fact tables by date or category so queries can scan only relevant partitions. This avoids full table scans of old data.
- **Optimize aggregations:** Use summary tables or materialized views for expensive aggregations over large data. Or ensure you have proper **GROUP BY** keys indexes or use window functions smartly if scanning too much.
- **Increase resources or adjust engine settings:** e.g., for cloud warehouses, scale up the warehouse size (more compute slots) or enable adaptive query execution if available. In something like BigQuery, ensure using clustering/partitioning to prune data.
- **Test incremental changes**: apply one change at a time and measure improvement. For instance, add an index and see the plan and timing, ensuring it actually got used.

What interviewers look for: A structured approach (analysis -> tuning -> verification). Knowledge of various techniques: indexing, rewriting, partitioning, resource scaling. Also awareness not to blindly add hints or hardware without understanding the query.

Common pitfalls:

- Focusing only on adding indexes without considering query refactor (not all slow queries are fixed by indexes, e.g. complex subquery logic might need rewrite).
- Neglecting the effect of data volume on chosen approach (e.g., a nested loop join might be fine on small tables but not on huge ones – plan may need change).
- Overlooking database-specific features (like query caching, result caching in Snowflake/BigQuery) that might be leveraged if consistent queries run repeatedly.

Follow-up: How would you approach performance tuning differently for an OLTP (transactional) query vs. an OLAP (analytical) query? (Expect: OLTP – focus on indexes, simple queries; OLAP – focus on partitioning, aggregation pushdown, MPP engine optimizations).

---

7. **(Medium)** Explain the concept of a **window function** in SQL. Can you give an example of how it's used?

Answer:

- A **window function** performs a calculation across a set of table rows that are somehow related to the current row. Unlike GROUP BY, it doesn't collapse rows; each row retains its detail while having an "aggregated" value computed over a window of rows.
- The **OVER() clause** defines the window (the partitioning and ordering of rows). Common window functions include: ROW_NUMBER(), RANK(), DENSE_RANK(), LAG()/LEAD() (for accessing other row values), and aggregate functions like SUM() or AVG() used as windowed versions.
- Example: **Running total** of sales by date:

```
SELECT date, sales,
    SUM(sales) OVER (ORDER BY date
                     ROWS BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW)
        AS running_total
FROM DailySales;
```

This computes a cumulative sum of sales up to each date.

Example 2: **Ranking:** Suppose you want to label each row with a rank by score:

```
SELECT name, score,
    RANK() OVER (ORDER BY score DESC) as rank
FROM Players;
```

- This will give highest score rank 1, ties get same rank, etc., without losing any rows.
- Window functions are powerful for queries like "find each employee's salary relative to department average" (use AVG(...) OVER (PARTITION BY department)), or finding gaps, trends with LAG/LEAD.

What interviewers look for: Understanding that window functions **do not reduce result row count** like GROUP BY does. They want to see that you know use cases (running totals, ranking, comparisons between rows) and syntax basics (PARTITION BY, ORDER BY inside OVER).

Common pitfalls:

- Confusing window functions with GROUP BY. E.g. trying to use HAVING with a window function (not allowed) or expecting window function to filter groups (it doesn't filter, it just adds a column).
- Using ORDER BY incorrectly in window (ORDER BY in the window function's OVER clause is not the same as global query ORDER BY; the former defines calculation order).
- Not realizing performance implications: e.g. a window with no PARTITION (whole table) might require sorting the entire dataset.

Follow-up: Provide a scenario: "Each customer's order is flagged if it's the largest order they made that week." How would you do that? (Expect them to use a window function like MAX(amount) OVER (PARTITION BY customer, week) to compare each order to the max for that customer in that week).

---

8. **(Hard)** When might a **denormalized** schema be preferred over a fully **normalized** schema? Could you give a scenario and the reasoning?

Answer:

- A denormalized schema (fewer tables, some redundancy) is often preferred in **analytics and read-heavy environments** like data warehouses or OLAP scenarios. This is because denormalization (e.g. star schema) avoids expensive joins by pre-joining or duplicating data, thus speeding up read queries at the expense of write/update complexity.
- **Scenario:** A dashboard requires fast querying of sales metrics by product and region. In a fully normalized model, getting all metrics might involve joining sales fact table with multiple dimension tables (product, region, date, etc.). If performance is critical, a denormalized wide table (or materialized view) that already contains product name, region, and precomputed aggregates can be used. The queries become simpler (no joins) and faster because all data is in one place.
- Denormalization is also used when **joining itself is too slow** due to huge data volume or lack of efficient keys. E.g., logging or IoT data might just store some repeated reference values directly to avoid join overhead in large distributed systems.
- Another example: In Apache Cassandra or DynamoDB (NoSQL), data is often denormalized because those systems favor data that's queried by a single key – you might duplicate some fields in multiple places to retrieve them without multi-key joins (since join is not natively supported).
- You'd choose denormalization knowingly when **read performance and simplicity outweigh storage cost and update anomalies**. Often accompanied by strategies to handle the redundancy (like updates through pipelines rather than user transactions).

What interviewers look for: Recognition that normalization vs denormalization is a trade-off. They expect mention of **query performance** and **use-case context** (OLTP vs OLAP). Also awareness of how modern big data systems often denormalize (data lakes, NoSQL) for speed at scale.

Common pitfalls:

- Saying denormalization is "always better for performance" without caveats – they want nuance: it helps reads but hurts updates and storage.
- Not mentioning the downsides: data anomalies, more complex ETL to keep data in sync, higher storage usage.
- Failing to consider intermediate approaches (like indexing or caching) – sometimes normalization can be retained with other performance tweaks, but if not, then denormalize.

Follow-up: How does a **Star Schema** relate to this? (Answer: Star schema is a form of denormalization for analytics – fact table and de-normalized dimension tables (one-hop join), optimized for read performance in BI queries.

---

9. **(Hard)** Given two tables A and B, how can you **delete** all rows from A that have no matching key in B, using SQL?

Answer:

- **Using a subquery with NOT IN / NOT EXISTS:**

```
DELETE FROM A
WHERE A.key_column NOT IN (SELECT key_column FROM B);
```

Or better (to avoid issues with NULLs):

```
DELETE FROM A
WHERE NOT EXISTS (
  SELECT 1 FROM B WHERE B.key_column = A.key_column
);
```

This will delete rows in A that don't have a corresponding key in B.

Using a JOIN in DELETE: Some SQL dialects support:

```
DELETE A
FROM A
LEFT JOIN B ON A.key_column = B.key_column
WHERE B.key_column IS NULL;
```

- This joins and identifies A's rows with no match (NULL from B side) and deletes them.
- Both approaches achieve the same result. **NOT EXISTS** is often preferred for clarity and to avoid pitfalls with NULL logic (since NOT IN can behave unexpectedly if the subquery returns any NULL values).
- It's important to have an index on B.key_column for performance, otherwise the subquery or join could be slow for large tables (each A row would require a lookup in B).

What interviewers look for: Correct logic that deletes only the intended rows (not accidentally all rows or the wrong set). Understanding of different SQL idioms (NOT EXISTS vs JOIN). Also, whether the candidate is aware of NULL handling issues with NOT IN.

Common pitfalls:

- Using A.key_column NOT IN (SELECT ...) without considering NULL: if B.key_column can be NULL, the NOT IN subquery will return no rows because NOT IN returns false if any subquery result is NULL (thus nothing gets deleted or, worse, everything might, depending on DB – results can be counterintuitive).
- Forgetting to alias properly in multi-table DELETE syntax (which is vendor-specific in many cases).
- Not considering referential integrity: If foreign keys exist, a cascading delete might be another approach (but here presumably no FK or it's handled manually).

Follow-up: How would you do the opposite (delete from A all rows *that do* have a match in B)? Simply remove the NOT – e.g., using EXISTS instead of NOT EXISTS.

---

10. **(Hard)** Explain **index cardinality** and why it matters for query performance.

Answer:

- **Index cardinality** refers to the uniqueness of values in the indexed column – essentially, how many distinct values the column has relative to the number of rows. High cardinality means most values are unique (e.g. an ID), low cardinality means many duplicates (e.g. a boolean or category that repeats often).
- It matters because high-cardinality indexes are generally more selective: when you query on a value, the index narrows down to few rows. The optimizer can efficiently use such an index to do index seeks. Low-cardinality indexes (like a gender field with just "M"/"F") might not be used, because filtering on that doesn't reduce the data much – the optimizer might decide to just scan the table rather than use an index that hits 50% of rows.
- For composite (multi-column) indexes, cardinality also involves the combination of fields. If the leading column of an index is low-cardinality, the index might not be very selective until you consider more columns.
- **Example:** If you index a is_active flag (true/false) on a million-row table, cardinality=2 (very low). A query WHERE is_active = true might return ~500k rows – the DB likely will just scan rather than bounce between index and table. But an index on a primary key (unique for each row) has cardinality = number of rows, and will always narrow down to 1 row quickly.
- Some databases have **index statistics** that include cardinality, which the query planner uses to decide if an index is worth using. If cardinality is low (a lot of duplicates), the planner might skip the index because reading via the index yields too many hits.
- **TL;DR:** High-cardinality indexes are more effective for selective queries; low-cardinality indexes may not provide benefit (and can be overhead on writes).

What interviewers look for: Knowledge of how indexes work beyond just "they make it faster". The candidate should demonstrate understanding of selectivity and why some indexes aren't used. Possibly expecting mention of terms like **selectivity** or **statistics**.

Common pitfalls:

- Saying “cardinality means number of rows” (confusing with cardinality of result set vs. distinct values).
- Not understanding that an index with very few distinct values can be nearly useless – or not relating that to a real scenario.
- Not mentioning that cardinality influences the optimizer’s choice (i.e., it’s not a manual decision, but it’s something you consider when designing indexes).

Follow-up: If you have a column with only a few distinct values (say “status” with values like NEW, PROCESSING, DONE), how might you still improve query performance? (Expect: possibly partitioning the table by that column if distributions skew and queries target one partition; or combine with other filter criteria in a composite index).

Sources (SQL & Optimization):

- Satyam Sahu, *“SQL Query Optimization Best Practices”* – Tips on indexing, SELECT * pitfalls, limiting data
- **DataLemur** – *Data Engineer Interview Guide*, SQL performance Q&A
- **Oracle/MySQL Docs** – Optimization and Index guidelines (index where clauses, avoid full scans)
- **Mode Blog** – Window Functions explained (difference vs GROUP BY)
- **StackOverflow** – Discussion on NOT IN vs NOT EXISTS (NULL handling in subqueries)
- **Use The Index, Luke** – Guidance on index selectivity & cardinality for SQL tuning

---